PyScript Collective

Paul Everitt

Aug 02, 2022

CONTENTS

1 Features 3

2 Requirements 5

3 Installation 7

4 Usage 9

5 Contributing 11

6 License 13

7 Issues 15

8 Credits 17

Python Module Index 41

Index 43

A prototype for a new PyScript Collective repo.

Want to easily get and run high-quality PyScript examples? Want to contribute to the PyScript community by helping others?

The PyScript Collective is a compendium of curated and maintained PyScript sample applications, along with the people behind it.

CHAPTER

ONE

FEATURES

This Collective repo is a bunch of contributors who help each other maintain good examples, plus onboarding enthusiastic new contributors.

But it's also a software project, providing:

- Easy to find, run, and tinker with high quality, maintained PyScript examples
- Easy to contribute new examples that become co-owned by the Collective

CHAPTER

TWO

REQUIREMENTS

To view the examples locally:

```
$ pipx psc
```

CHAPTER

THREE

INSTALLATION

You can install *PyScript Collective* via pip from PyPI:

```
$ pip install psc
```

CHAPTER

FOUR

USAGE

Please see the [Command-line Reference] for details.

CHAPTER

FIVE

CONTRIBUTING

Contributions are very welcome. To learn more, see the *Contributor Guide*.

CHAPTER

SIX

LICENSE

Distributed under the terms of the *Apache 2.0 license*, *PyScript Collective* is free and open source software.

CHAPTER

SEVEN

ISSUES

If you encounter any problems, please file an issue along with a detailed description.

CHAPTER

EIGHT

CREDITS

This project was generated from @cjolowicz's Hypermodern Python Cookiecutter template.

8.1 Building PSC

This PSC package is a proof-of-concept of a possible future PyScript Collective. To make it easy to evaluate, it's written diary-style. At each step:

- Make a branch
- Write docs for the problem being solved
- Notes on the solution
- Open questions
- Merge

This section covers each step in the prototype.

8.1.1 Proper Package

This PSC isn't just prototyping examples to list on a webpage. It's actually an installable Python package, making it easy for people to *consume and tinker with* the examples.

As such, PSC starts with a clean-slate:

- A new repo
- Built from (controversially) the Hypermodern Python cookiecutter
- Switch from conda to Poetry/pip
- An installable package on PyPI
- A full Collective website based on Sphinx

Let's look at each decision in detail.

A New Repo

It's just a disposable prototype, but PSC was written as if it was the new repo. I'm using a Release Drafter style GitHub workflow to allow release notes to be generated.

Installable Package

As part of the repo reboot, it's written as if it is a Python package, meant to be installed. This will potentially make it dead-simple for people to play with the examples, even to edit them and see results. They will just `pip install our-package` and get everything needed. Or even simpler, use `pipx` to just run the examples.

:::{attention} Package name? The PyScript ecosystem should adopt a prefix for add-on packages, as done in Django, Flask, Pyramid, etc. Presumably this package will be named `pyscript-collective`.

That's pretty long, though. Perhaps PyScript should adopt `ps-` as the prefix? :::

Hypermodern Python

Our repo will have some of "our" software:

- `pytest` fixtures
- Command runners, e.g. a CLI to run the examples

For our stuff, we'll want *some* automation and quality control tools, such as linters. For the examples themselves, we might want a small subset of that (low bar for them to *give* contribution) or large subset (high bar for us to *take* contribution.)

This stuff is hard to wire together and keep working. I used the Hypermodern Python cookiecutter as the starting point:

- I have experience with it
- I know it works
- I know how to turn things off that are too pedantic

Some of my Collective brethren will certainly vomit when confronted with all that Hypermodern does. We can dial it back until we get the right balance of "best practice and long-term maintenance" vs. "easy of contribution."

Conda -> Poetry

The existing `pyscript-collective` repo starts with a `Makefile`. It also presumes Conda for everything.

This PSC prototype switches over to Poetry for contributors and pip for consumers. The use of Poetry isn't important – we could easily switch to `virtualenv` and `requirements.txt`. But the switch away from Conda is more intentional: it's less used per the PSF survey, and we want to show that PyScript isn't tied to Anaconda no Conda.

Website in Sphinx

The writing of this prototype will ultimately result in a full website, based on Sphinx, with a highly-custom and attractive landing page. Since it is in Sphinx, and since the main PyScript docs are in Sphinx, it should be straightword for PSC to be included into the main site.

8.1.2 Command Runner

We want to make it easy for Viewers to see the examples locally. In fact, to easily tinker with them and see changes.

Let's add a CLI that fires up a web server.

CLI

I'll use Typer to manage the CLI, with `typer[all]` as the installation to get Rich. I then add a single entry point in `__main__.py` and ensure it is registered in `pyproject.toml`. I then add a test in `tests/test_main.py` to ensure it works as expected.

This was a little tricky. I don't want to actually run the server in the CLI and Typer spawns a subprocess, so I can't mock (I think?) Thus, I added a CLI option I could pass in to prevent the server from starting.

With this, users can run, in their virtualenv once PSC is installed:

```
$ python -m psc
```

If you have `pipx` installed (and if PSC were actuall uploaded to PyPI):

```
$ pipx psc
```

I can run in my editable install:

```
$ poetry run python -m psc
```

Web App

I'll use Starlette and uvicorn. In this step, it's just one route which returns static HTML. I can test this easily using `TestClient` which requires an installation of `requests`.

:::{note} Nice way to run tests `TestClient` is nice because it doesn't start an actual server. That's a big part of the complaint on the current testing strategy for PyScript. The `SimpleHTTPServer` running in a thread is kind of fragile for getting hung. :::

Wrapup

I run pre-commit, mypy, and nox. Everything is good, onward.

8.1.3 Playwright Interceptors

PyScript testing needs a real browser – web components which load Pyodide and execute Python, then change the DOM. Playwright provides this, but we'd like more convenience in the testing:

- Don't actually launch a web server to fetch the examples
- Make it easier to write examples and tests by having some automation

In this step we bring in Playwright, but don't yet use PyScript. Here's the big idea: we do *not* run a web server. Instead, we write a Playwright interceptor as a pytest fixture.

Install Playwright

We need to add Playwright to PSC. In the current repo, this is done as part of a Makefile rule, which also copies the examples to a relative directory (bleh).

Instead, we'll just make it a development dependency. If a Contributor wants to write an example, they just need to clone the repo and install in editable mode with dev dependencies. Kind of normal Python dev workflow. To make it work in CI using Nox, we added this dependency to the `noxfile.py`.

This still requires running `playwright install` manually, to get the Playwright browsers globally installed. That has to be added to PSC Contributor documentation.

Fixture

With Playwright installed, now it is time to make it easier to write/run the tests for examples. In the previous step, we did "shallow" testing of an example, using `TestClient` to ensure the HTML was returned. We didn't actually load the HTML into a DOM, certainly didn't evaluate the PyScript web components, and *definitely* didn't run some Python in Pyodide.

The current Collective uses Playwright's `page` fixture directly: you provide a URL, it tells the browser to make an HTTP request. This means it needs an HTTP server running. The repo fires up and shuts down a Python `SimpleHTTPServer` running in a thread, as part of test running.

If something gets hung…ouch. You have to wait for the thread to time out.

PSC changes this by not running an HTTP server for testing the examples. Instead, we use Playwright interceptors. When the URL comes in, our Python code runs and returns a response…quite like `TestClient` pretends to run an ASGI server. Our "interceptor" looks at the URL, and if it is to the "fake" server, it reads/returns the path from disk.

This fixture is software, so we'll make a file at `src/psc/fixtures.py` and a test at `test_fixtures.py`. The test file has dummy objects for the Playwright request/response/page/route etc. The tests exercise the main code paths we need for the interceptor:

- A request but *not* to the fake server URL should just be passed-through to an HTTP request
- A request to the fake server URL should extract the path
  - If that path exists in the project, read the file and return it
  - If not, raise a value error

With that in place, we write a `fixtures.fake_page` fixture function. It asks `pytest` to inject the real `page`. It then installs the interceptor by calling a helper function. This helper function is what we actually write the fixture test for.

Serve Up Examples

We aren't going to test by fetching examples from an HTTP server. But our Viewers will look at examples from the HTTP server we made in the previous step. Let's add that to `app.py` with another `Mount`, this time pointing `/examples` at `src/psc/examples`. Also, add `first.html` with some dummy text as an "example".

Before the implementation, we add `test_app.test_first_example` as a failing test. Then, once `app.py` is fixed, the test will pass.

First Test

Our fixture is now in place, with a test that has good coverage. We have a dummy example in `first.html`. Let's write a test that uses Playwright and the interceptor.

We just added a `TestClient` test – a kind of "shallow" test – for `first.html`. In `test_app.py` we add `test_first_example_full` as a Playwright test.

When we first run it, we see `fixture 'fake_page' not found`. This is because `conftest.py` needs to load the `psc.fixtures`. With that line added, the tests pass.

Shallow vs. Full Markers

These Playwright tests are SLOW. When we get a bunch of examples, it's going to be a pain. As such, we'll want to emphasize unit tests and the shallow `TestClient` tests.

To make this first-class, we'll add 3 pytest markers to the project: unit, shallow, and full. We do so in `pyproject.toml` along with the option to warn if someone uses an undefined customer marker.

With this in place, we add decorators such as `@pytest.mark.full` to our tests. Later, we can run `pytest -m "not full"` to skip the Playwright tests.

QA

Cleaned up everything for pre-commit, mypy, nox, etc.

Along the way, Typeguard got mad at the introduction of the marker. I skipped investigation and just disabled Typeguard from the noxfile for now.

8.1.4 First PyScript Example

Well, that was certainly a lot of prep.

Let's get into PyScript and examples. In this step we'll add the "Hello World" example along with unit/shallow/full tests. We will *not* though go further into how this example gets listed. We also won't do any automation across examples: each example gets its own tests.

Big ideas: tests run offline and faster, no quirks for threaded server, much simpler "wait" for DOM.

Re-Organize Tests

In the previous step, we made an `src/psc/examples` directory with `first.html` in it. Let's remove `first.html` and instead, have a `hello_world` directory with `index.html` in it. For now, it will be the same content as `first.html`, though we need to change the CSS path to `../static/psc.css`.

We also have our "first example" tests in `test_app.py`. Let's leave that test file to test the application itself, not each individual test. Thus, let's start `tests/examples/test_hello_world.py` and move `test_first_example*` into it. We'll finish with `test_hello_world` and `test_hello_world_full` in that file.

With these changes, the tests pass. Let's change the example to be the actual PyScript `Hello World` HTML file.

Download PyScript/Pyodide Into Static

Using `curl`, I grabbed the latest `pyscript.css`, `pyscript.js`, and `pyscript.py`, plus the '.map` etc. This brings up an interesting question about versions. Should the Collective examples all use the same PyScript/Pyodide versions, or do we need to support variations?

:::{note} Git LFS Support

These Pyodide WASM distributions are…big. Putting them in the repo, then updating them frequently, will make cloning slow. OTOH, we don't want to lose "run everything locally".

This might mean enabling Git LFS on the repo. As an alternative, an extra install step to fetch the latest Pyodide WASM and put in a non-versioned directory. For now, punting on this. As final note…it appears to be around 23 MB to include all the WASM, wheels, etc. :::

Next up, Pyodide. I got the `.bz2` from the releases and uncompressed/untarred into a release directory. Bit by bit, I copied over pieces until "Hello World" loaded:

- The `.mjs` and `.asm*`
- `packages.json`
- The distutils.tar and pyodide_py.tar files
- `.whl` directories for micropip, packaging, and pyparsing

Hello World Example

Back to `src/psc/examples/hello_world/index.html`. Before starting, we should ensure the shallow test – `TestClient` – in `test_hello_world.py` works.

To set up PyScript, first, in `head`:

```
<link rel="icon" type="image/png" href="../../favicon.png" />
<link rel="stylesheet" href="../../static/pyscript.css" />
<script defer src="../../static/pyscript.js"></script>
```

That gets PyScript stuff. The JS requests `pyscript.py` which is also in `static`.

To get Pyodide from local installation instead of remove, I added `<py-config>`:

```
<py-config>
- autoclose_loader: true
  runtimes:
    - src: "../../static/pyodide.js"
      name: pyodide-0.20
```

(continues on next page)

(continued from previous page)

```
      lang: python
</py-config>
```

This was complicated by a few factors:

- The PyScript docs page is broken
- There are no examples in PyScript (and thus no tests) that show a working version of <py-config>
- The default value on autoclose_loader appears to be false so if you use <py-config> you need to explicitly turn it off.

At this point, the page loaded correctly in a browser, going to http://127.0.0.1:3000/examples/hello_world/index.html. Now, on to Playwright.

Playwright Interceptor

We're going to be handling more types of files now, so we change the Content-Type sniffing. Instead of looking at the extension, we use Python's mimetypes library.

For the test, we want to check that our PyScript output is written into the DOM. This doesn't happen immediately. In the PyScript repo, they sniff at console messages and do a backoff to wait for Pyodide execution.

Playwright has help for this. The page can wait for a selector to be satisfied.

This is *so much nicer*. Tests run a LOT faster:

- Our assets (HTML, CSS, pyscript.js, pyscript.css) are served without an HTTP server
- Pyodide itself isn't loaded from CDN nor even HTTP Also, if something goes wrong, you aren't stuck with a hung thread in SimpleHTTPServer. Finally, as I noticed when working on vacation with terrible Internet – everything can run offline…the examples and their tests.

It was *very* hard to get to this point, as I ran into a number of obscure bugs:

- The <py-config> YAML bug above was a multi-hour waste
- Reading files as strings failed obscurely on Pyodide's .asm.* files
- Ditto for MIME type, which needs to be application/wasm (though the interwebs are confusing on this)
- Any time the flake8/black/prettier stack ran on stuff in static, all hell broke loose

Debugging

It was kind of miserable getting to this point. What debugging techniques did I discover?

Foremost, running the Playwright test in "head-ful" mode and looking at both the Chromium console and the network tab. Playwright made it easy, including with the little controller UI app that launches and lets you step through:

```
$ PWDEBUG=1 poetry run pytest -s tests/examples/test_hello_world.py
```

For this, you need to add a page.pause() after the page.goto().

Next, when running like this, you can use Python print() statements that write to the console which launched the head-ful client. That's useful in the interceptor. You could alternatively do some console logging with Playwright's (cumbersome) syntax for talking to the page from Python. But diving into the Chromium console is a chore.

When things weren't in "fail catastrophically" mode, the most productive debugging was…in the debugger. I set a breakpoint interceptor code, ran the tests, and stopped on each "file" request.

Finally, the most important technique was to… slow down and work methodically with unit tests. I should have done this from the start, hardening the interceptor and its surface area with Playwright. I spent hours on things a decent test (and even `mypy`) would have told me about bytes vs. strings.

QA Tools

When running `pre-commit`, it appears Prettier re-formats the YAML contents of `<py-config>`. I could have spent time to figure it out (e.g. skip those files, or teach Prettier how to handle it.) But it's not urgent, so I disabled Prettier in the `.pre-commit-config.yaml` file.

Flake8 had a lot of complaints with `pyscript.py`. I edited `.flake8` to turn off all the particular problems, to avoid editing the file itself. Probably needs a better solution.

`mypy` found a couple of actual bugs. Thanks, Python type hinting! (Although I did chicken out with a `type: ignore` on a bytes thing in a test.)

Could Be Better

This is very much a prototype and lots could be better.

There are still bunches of failure modes in the interceptor, and when it fails, things get *very mysterious*. A good half-day of hardening and test-writing – primarily unit tests – would largely do it. To go further, using Starlette's `FileResponse` and making an "adapter" to Playwright's `APIResponse` would help. Starlette has likely learned a lot of lessons on file reading/responding.

Speaking of the response, this code does the minimum. Content length? Ha! Again, adopting more of a regular Python web framework like Starlette (or from the old days, `webob`) would be smart.

We could speed up test running with ideas from Pyodide Playwright ticket. It looks fun to poke around on that, but the hours lost in hell discouraged me. It's pretty fast right now, and a great improvement over the status quo. But a 3x speedup seems interesting.

Finally, it's possible that async Playwright is the answer, for both general speedups and `wait_for_selector`. When I first dabbled at this though, it got horrible, quickly (integrating main loops, sprinkling async/await everywhere.) I then read something saying "don't do it unless you have to."

8.1.5 Bulma Styling

Let's start moving towards the goal of providing attractive examples. Each example will appear in several "targets", primarily a website like the existing examples.

In this step, we'll start building the PSC website. We will *not* in this step, though, tackle any concept of a build step, nor anything beyond the homepage.

Big ideas: use off-the-shelf CSS framework, static generation, dead-simple web tech.

Why Bulma?

We're making a website – the PyScript Collective. But we're also making a web app – the PyScript Gallery. As it turns out, we're also shipping a PyPI package – the PyScript Gallery, aka `psga`.

We need a nice-looking web app. Since we're not designers, let's use a popular, off-the-shelf CSS framework. I have experience with (and faith in) Bulma: it's attractive out-of-the-box, mature, and strikes the right balance.

New Test With `beautifulsoup`

Let's start with a failing test in `tests/test_assets.py`. We'll move the `test_app.test_favicon` there as a start.

We don't want to just make sure the favicon is at the URL. We want to parse the HTML, find the target, and test *that*, to make sure the HTML link isn't broken.

Let's add `beautifulsoup` as a parser, grab the response body, and make it easy to go find links. Since I'm using `mypy`, I also do `poetry add -D types-beautifulsoup4`. I'll also install `html5lib` as an HTML parser.

The `test_favicon` test was changed to get the `favicon.png` path from the `<link>`, then fetch it.

:::{note} PyCharm Tip For Resources

I'd like PyCharm to warn me about bad links in HTML. So I marked the `src/psc` directory as a "Resource Root". This gives me autocomplete, warnings, refactoring, etc. :::

Bring In Bulma

The tests are in good shape and having `beautifulsoup` will prove…beautiful. Let's add a `<link rel="stylesheet" href="/static/bulma.min.css"/>` to the home page. We start with a failing test, similar to the favicon one.

To make it pass, I:

- Downloaded the bits into static
- Added a `<link rel="stylesheet" href="/static/bulma.min.css"/>` to the home page.

I also added the other parts of the Bulma starter (doctype, meta.) If we open it up in the browser, it looks a bit different.

Navbar and Footer

We'd like a common navbar on all our pages. Bulma has a navbar. This also means a download of the PyScript SVG logo, which we'll write a test for.

Ditto for a – for now, *very* simple – footer component.

Body

Bulma makes use of `<section>`, `<footer>`, etc. Let's put the "main" part of our page into a `<main class="section">` tag.

For the failing test, we'll simply look to see there is a `<main>`. But, as this is no longer a static asset, we'll put this in `test_app.test_homepage`.

Templating

It sucks to repeat the layout across every static HTML file. Let's make some Jinja2 templates, then setup Starlette to use it the templates.

As precursor, install Jinja2 as a dev dependency.

We'll start by making a templates directory at `src/psc/templates`. In there, we'll make `page.jinja2` and `layout.jinja2`.

:::{note} PyCharm Template Setup

If you use PyCharm, use `Mark This Directory` to make it a template directory. Also, go into preferences and set the project as using Jinja2 for `Template Languages`. :::

In `layout.jinja2`, take everything out that isn't in `<main>`. Provide a slot for title and main. Then change `page.jinja2` to point at `layout.jinja2`, filling those two slots.

In `app.py`, we change the `homepage` route to return a template. The context dictionary for the template will have two pieces of data:

- The title of the current page
- The HTML that should go in the main block.

Let's do three things:

- Change `index.html` only have the `<main>` part
- In the route, read the file content
- Then pass the file contents into `page.jinja2`, using `| safe`

When done correctly, the tests should pass.

Future

This PSC prototype just uses downloaded Bulma CSS and other assets. It doesn't bring in the SASS customizations and software, nor does it look at Bulma forks that bring in CSS Variables.

While we did a little templating, we didn't go far. It's going to get a lot more interesting and intricate, as we have different ways we want to package things.

8.1.6 Examples Template

Having a template is pretty slick. We'll now do the same for each example, but things are gonna get kinda weird: we need just part of the example's HTML.

Example Template and Route

We'll start with a test. We already have a `TestClient` test at `test_hello_world.test_hello_world`. We start by adapting it to the same BeautifulSoup approach we just saw.

Next, an implementation. We add a template at `templates/example.jinja2` then make a new `example` view and route in `app.py`. By copying the existing view, we get something that works and, with a small test change, passes the tests. But it's returning the contents of the home page.

Instead, we:

- Get the route parameter

- Read that file
- Use `beautifulsoup4` to extra the contents of `<main>`
- Shove that into the template as the context value of `main`

Along the way, we also extract this example's title from the `<title>` in the HTML file. We then shove it in as the template context value of `title`.

This leaves out:

- Everything in the `<head>`, such as…loading PyScript
- All the `<py-*>` nodes elsewhere in the example's `<body>`

Uhhh…that's kind of dumb. Why are we doing that?

Standalone vs. Integrated vs. Unmanaged

The HTML for an example might appear in a bunch of places:

1. *Standalone*. People want to cut-and-paste an example and run it from a `file:///` URL. The Contributor might want to start this way. It needs the PyScript JS/CSS and possibly a `<py-config>`.
2. *Integrated Website*. In the website, for the "best" examples, we want everything to fit together well: consistent styling, fast page transitions, using the same PyScript/Pyodide. The Gallery should have control of these things, not the examples. Let's call those the "integrated" examples, vs. others that need their own control.
3. *Unmanaged Website*. These are examples on the website which need to set their own Pyodide, or not use the Gallery CSS.
4. *Integrated App*. These are when the examples are running in the Gallery Python web app, under Starlette. Perhaps the Contributor is browsing the example, perhaps a Coder is running the example via `pipx`. Mostly the same as "Integrated Website".
5. *CI Builds Website*. In this case, the example is compiled into a `public` directory and included into the website. The example isn't really being executed. Rather, it's being assembled into output.

At this point, we're still in "Integrated App". The Starlette process wants an "integrated" example, where the CSS/JS/Pyodide is under the `layout.jinja2` control. All the "integrated" examples will look and feel consistent.

Extra PyScript Stuff in Head

With that said, an "integrated" examples might have other static assets to go in the `<head>`: extra CSS, for example. We'll add that to our example.

Remember, these examples are "standalone". They include the `<link>` and `<script>` pointing to PyScript. We don't want *those* – they come from `layout.jinja2`. We *do* want anything else them put in there, with relative links as the targets.

Let's write a failing first for including `hello_world.css`. For implementation:

- Add a slot in `layout.jinja2`
- Change `example.jinja2` to fill that slot, based on passed in string
- Pass in a string of all the HTML to include
- Build that string from a `beautifulsoup` `select`

Example Template Needs <py-config>

We want the HTML for the examples to get a Gallery-managed <py-config>. But we don't want this in other, non-example pages. We'll add an extra_body slot in layout.jinja2, then fill it from example.jinja2.

Starting, of course, with a test.

Plucking Example Parts

That's good for stuff in the <head>. But we have a problem in the <body>. PyScript only allows <py-script> as a direct child of <body>, so we can't put it in <main>. We need a policy like this:

- Anything in the example's "UI" (the DOM) goes in <main> and gets copied over
- Any <py-*> nodes directly under <body> get copied over
- *Except* <py-config>
- Everything else in <body> is left out

We'll write some tests:

- Ensure only one <py-config> with a runtime pointed to our local Pyodide
- The <py-script> is copied over, in the right spot
- Some tracer <h6> that is *outside* of <main> is *not* copied over

QA

mypy gave us some trouble at the end, because beautifulsoup has some unusual typing. We thus moved the example view's soup filtering into a standalone function which had a cast.

Future

This is actually pretty neat. But the view is doing too much. Later, we'll introduce a "resource" concept, kind of like a model, and move the work there.

8.1.7 Resource Listing

Our navigation needs to list the available examples. Equally, we need a cleaner way to extract the data from an example. In this step, we make "resource" objects for various models: page, example, contributor. We'll leave page and contributor for the next step.

Big ideas: A standard model helps us with all the representations of an example.

What's an Example?

We'll start with, of course, tests. This time in `test_resources`. Let's write `test_example` and see if we can correctly construct an instance. It will need all the bits the template relies on.

Our resource implementation will use dataclasses, with a base `Resource`. We'll use `__post_init__` to fill in the bits by opening the file. Also, as "keys", we'll use a `PurePath` scheme to name/id each example.

To help testing and to keep `__post_init__` dead-simple, we move most of the logic to easily-testable helpers.

Gathering the Resources

We'll make a "resource dB" available at startup with the examples. For now, we'll do a `Resources` dataclass with an examples field as `dict[PurePath, Example]`. Later, we'll add fields for pages and contributors.

First a test, of course, to see if `get_resources` returns a `Resources.examples` with `PurePath("hello_world")` mapping to an `Example`.

We then write the `resources.get_resources` function that generates a populated `Resources`.

Listing Examples

Now that we have the listing of examples, we head back to `app.py` and put it to work. We'll use Starlette's "lifespan" support to:

- Register a startup function which…
- Runs `get_resources` and…
- Assigns to `app.state.resources`

We'll then change the `example` view to get the resource from `request.app.state.resources`.

When we do this, though, `TestClient` breaks. It doesn't ordinarily run the lifecycle methods. Instead, we need to use the context manager. We do this, and along the way, refactor the tests to inject the test_client. In fact, we also make a `get_soup` fixture that further eases test writing.

We then add a `/examples` view, starting with a test of course. This uses an `examples.jinja2` template. We wire this up into the navbar and have the test ensure that it is there via navbar.

The listings use a Bulma tile 3 column layout. Lots that can be done here.

Cleanup

- Arrange for `/example/hello_world` and `/example/hello_world/index.html` to both resolve
- Fix the silly BeautifulSoup "allow str or List[str]" in upstream so tests don't have to cast all the time
- Get a "subtitle" from `/examples/hello_world/index.html` and a `<meta name="subtitle" content="...">` tag
- Don't extract `<main>` itself from the example, as we want to control it in the layout…just the children

Future

- Have the rows in the tiles be yielded by a generator, allowing multi-column title
- Include contributor information at bottom of each tile

8.1.8 Content Pages

We have a home page, but we'll need some other content pages, such as "About". Add a "Page" resource, with Markdown+frontmatter-driven content under `/pages`.

Pages In `/pages`

- All pages will be Markdown-driven, e.g. `src/psc/pages/about.md` and `/pages/about.html`
- Install `python-frontmatter` as regular dependency
- Write tests for a new `Page` resource type and views
- Implement them
- Home page stays as a Jinja2 template, as it is heavily-Bulma (not a good candidate for Markdown)

Navbar

- Wire `About` into the navbar

Home Page

- Better formatting
- Content for: homepage, about, contributing, vision, homepage hero

Future

- Images
- Static content
- Contributors
- Build command

8.1.9 Build Step

Lots of Paths

There's a `<py-script>` in an example. What are the ways someone might consume it? Turns out – a LOT!

1. *Viewer From Website*. This is the most normal one. A Viewer goes to the Collective site, clicks the link for Examples, and sees the listing. The Viewer then clicks on the example, gets it loaded up, and looks at it. This one has a later variation – what if we let Viewers edit in the browser and see the updated example?

2. *Coder From Package*. A Coder wants the examples locally. They use `pipx` to directly run the examples or `pip` to install into a virtual environment. Either way, a Python process starts that runs a web server to show the examples. Later, we make it easy to edit the example sources and see the changes.
3. *Contributor Writes Example*. In-place vs. built.
4. *Collaborator Reviews Example*.
5. *Test Runs Example*.

8.2 Reference

8.2.1 psc

PyScript Collective.

8.3 Contributor Guide

Thank you for your interest in improving this project. This project is open-source under the Apache 2.0 license and welcomes contributions in the form of bug reports, feature requests, and pull requests.

Here is a list of important resources for contributors:

- Source Code
- Documentation
- Issue Tracker
- *Code of Conduct*

8.3.1 How to report a bug

Report bugs on the Issue Tracker.

When filing an issue, make sure to answer these questions:

- Which operating system and Python version are you using?
- Which version of this project are you using?
- What did you do?
- What did you expect to see?
- What did you see instead?

The best way to get your bug fixed is to provide a test case, and/or steps to reproduce the issue.

8.3.2 How to request a feature

Request features on the Issue Tracker.

8.3.3 How to set up your development environment

You need Python 3.7+ and the following tools:

- Poetry
- Nox
- nox-poetry

Install the package with development requirements:

```
$ poetry install
```

You can now run an interactive Python session, or the command-line interface:

```
$ poetry run python
$ poetry run psc
```

8.3.4 How to test the project

Run the full test suite:

```
$ nox
```

List the available Nox sessions:

```
$ nox --list-sessions
```

You can also run a specific Nox session. For example, invoke the unit test suite like this:

```
$ nox --session=tests
```

Unit tests are located in the *tests* directory, and are written using the pytest testing framework.

8.3.5 How to submit changes

Open a pull request to submit changes to this project.

Your pull request needs to meet the following guidelines for acceptance:

- The Nox test suite must pass without errors and warnings.
- Include unit tests. This project maintains 100% code coverage.
- If your changes add functionality, update the documentation accordingly.

Feel free to submit early, though—we can always iterate on this.

To run linting and code formatting checks before committing your change, you can install pre-commit as a Git hook by running the following command:

```
$ nox --session=pre-commit -- install
```

It is recommended to open an issue before starting work on anything. This will allow a chance to talk it over with the owners and validate your approach.

8.4 Contributor Covenant Code of Conduct

8.4.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, caste, color, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

8.4.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

8.4.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

8.4.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

8.4.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at pauleveritt@me.com. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

8.4.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

8.4.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.1, available at https://www.contributor-covenant.org/version/2/1/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

8.5 License

```
Copyright 2022 Paul Everitt

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

-------------------------------------------------------------------------

                              Apache License
                        Version 2.0, January 2004
                     http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

  "License" shall mean the terms and conditions for use, reproduction,
  and distribution as defined by Sections 1 through 9 of this document.

  "Licensor" shall mean the copyright owner or entity authorized by
  the copyright owner that is granting the License.

  "Legal Entity" shall mean the union of the acting entity and all
  other entities that control, are controlled by, or are under common
  control with that entity. For the purposes of this definition,
```

(continues on next page)

(continued from previous page)

```
  "control" means (i) the power, direct or indirect, to cause the
  direction or management of such entity, whether by contract or
  otherwise, or (ii) ownership of fifty percent (50%) or more of the
  outstanding shares, or (iii) beneficial ownership of such entity.

  "You" (or "Your") shall mean an individual or Legal Entity
  exercising permissions granted by this License.

  "Source" form shall mean the preferred form for making modifications,
  including but not limited to software source code, documentation
  source, and configuration files.

  "Object" form shall mean any form resulting from mechanical
  transformation or translation of a Source form, including but
  not limited to compiled object code, generated documentation,
  and conversions to other media types.

  "Work" shall mean the work of authorship, whether in Source or
  Object form, made available under the License, as indicated by a
  copyright notice that is included in or attached to the work
  (an example is provided in the Appendix below).

  "Derivative Works" shall mean any work, whether in Source or Object
  form, that is based on (or derived from) the Work and for which the
  editorial revisions, annotations, elaborations, or other modifications
  represent, as a whole, an original work of authorship. For the purposes
  of this License, Derivative Works shall not include works that remain
  separable from, or merely link (or bind by name) to the interfaces of,
  the Work and Derivative Works thereof.

  "Contribution" shall mean any work of authorship, including
  the original version of the Work and any modifications or additions
  to that Work or Derivative Works thereof, that is intentionally
  submitted to Licensor for inclusion in the Work by the copyright owner
  or by an individual or Legal Entity authorized to submit on behalf of
  the copyright owner. For the purposes of this definition, "submitted"
  means any form of electronic, verbal, or written communication sent
  to the Licensor or its representatives, including but not limited to
  communication on electronic mailing lists, source code control systems,
  and issue tracking systems that are managed by, or on behalf of, the
  Licensor for the purpose of discussing and improving the Work, but
  excluding communication that is conspicuously marked or otherwise
  designated in writing by the copyright owner as "Not a Contribution."

  "Contributor" shall mean Licensor and any individual or Legal Entity
  on behalf of whom a Contribution has been received by Licensor and
  subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
  this License, each Contributor hereby grants to You a perpetual,
  worldwide, non-exclusive, no-charge, royalty-free, irrevocable
  copyright license to reproduce, prepare Derivative Works of,
```

(continues on next page)

(continued from previous page)

```
  publicly display, publicly perform, sublicense, and distribute the
  Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
  this License, each Contributor hereby grants to You a perpetual,
  worldwide, non-exclusive, no-charge, royalty-free, irrevocable
  (except as stated in this section) patent license to make, have made,
  use, offer to sell, sell, import, and otherwise transfer the Work,
  where such license applies only to those patent claims licensable
  by such Contributor that are necessarily infringed by their
  Contribution(s) alone or by combination of their Contribution(s)
  with the Work to which such Contribution(s) was submitted. If You
  institute patent litigation against any entity (including a
  cross-claim or counterclaim in a lawsuit) alleging that the Work
  or a Contribution incorporated within the Work constitutes direct
  or contributory patent infringement, then any patent licenses
  granted to You under this License for that Work shall terminate
  as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
  Work or Derivative Works thereof in any medium, with or without
  modifications, and in Source or Object form, provided that You
  meet the following conditions:

  (a) You must give any other recipients of the Work or
      Derivative Works a copy of this License; and

  (b) You must cause any modified files to carry prominent notices
      stating that You changed the files; and

  (c) You must retain, in the Source form of any Derivative Works
      that You distribute, all copyright, patent, trademark, and
      attribution notices from the Source form of the Work,
      excluding those notices that do not pertain to any part of
      the Derivative Works; and

  (d) If the Work includes a "NOTICE" text file as part of its
      distribution, then any Derivative Works that You distribute must
      include a readable copy of the attribution notices contained
      within such NOTICE file, excluding those notices that do not
      pertain to any part of the Derivative Works, in at least one
      of the following places: within a NOTICE text file distributed
      as part of the Derivative Works; within the Source form or
      documentation, if provided along with the Derivative Works; or,
      within a display generated by the Derivative Works, if and
      wherever such third-party notices normally appear. The contents
      of the NOTICE file are for informational purposes only and
      do not modify the License. You may add Your own attribution
      notices within Derivative Works that You distribute, alongside
      or as an addendum to the NOTICE text from the Work, provided
      that such additional attribution notices cannot be construed
      as modifying the License.
```

(continues on next page)

(continued from previous page)

```
  You may add Your own copyright statement to Your modifications and
  may provide additional or different license terms and conditions
  for use, reproduction, or distribution of Your modifications, or
  for any such Derivative Works as a whole, provided Your use,
  reproduction, and distribution of the Work otherwise complies with
  the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
  any Contribution intentionally submitted for inclusion in the Work
  by You to the Licensor shall be under the terms and conditions of
  this License, without any additional terms or conditions.
  Notwithstanding the above, nothing herein shall supersede or modify
  the terms of any separate license agreement you may have executed
  with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
  names, trademarks, service marks, or product names of the Licensor,
  except as required for reasonable and customary use in describing the
  origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
  agreed to in writing, Licensor provides the Work (and each
  Contributor provides its Contributions) on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
  implied, including, without limitation, any warranties or conditions
  of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
  PARTICULAR PURPOSE. You are solely responsible for determining the
  appropriateness of using or redistributing the Work and assume any
  risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
  whether in tort (including negligence), contract, or otherwise,
  unless required by applicable law (such as deliberate and grossly
  negligent acts) or agreed to in writing, shall any Contributor be
  liable to You for damages, including any direct, indirect, special,
  incidental, or consequential damages of any character arising as a
  result of this License or out of the use or inability to use the
  Work (including but not limited to damages for loss of goodwill,
  work stoppage, computer failure or malfunction, or any and all
  other commercial damages or losses), even if such Contributor
  has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
  the Work or Derivative Works thereof, You may choose to offer,
  and charge a fee for, acceptance of support, warranty, indemnity,
  or other liability obligations and/or rights consistent with this
  License. However, in accepting such obligations, You may act only
  on Your own behalf and on Your sole responsibility, not on behalf
  of any other Contributor, and only if You agree to indemnify,
  defend, and hold each Contributor harmless for any liability
  incurred by, or claims asserted against, such Contributor by reason
```

(continues on next page)

(continued from previous page)

```
  of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

  To apply the Apache License to your work, attach the following
  boilerplate notice, with the fields enclosed by brackets "[]"
  replaced with your own identifying information. (Don't include
  the brackets!)  The text should be enclosed in the appropriate
  comment syntax for the file format. We also recommend that a
  file or class name and description of purpose be included on the
  same "printed page" as the copyright notice for easier
  identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

   http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

PYTHON MODULE INDEX

p
psc, 31

INDEX

M

module
    psc, 31

P

psc
    module, 31