Proxying web servers in the Workbench Extension

Table of contents

Proxying web servers

The Posit Workbench Extension includes a Proxied Servers view when selecting the extension from the Activity Bar. This view contains a list of currently running web servers. Each item in this list includes the web server's name and the port the process is running on. Selecting an item opens the server in a new browser tab.

The extension determines which servers to display by searching through your currently running processes for those with open and listening sockets. The extension excludes processes it expects and non-development servers such as R and Jupyter sessions. For R and Python processes, the extension represents the server's name as the directory name.

We have ensured the following server types can be proxied when using the extension in VS Code sessions. Other application types may work as expected.

- Dash
- FastAPI
- Flask
- Quarto
- Shiny for Python
- Shiny for R
- Streamlit

Dash

Requirements

- Minimum dash version of 1.10.0 (earlier versions require the user to manually set the `port` variable in the `run_server` call)
- The server must be run with the debug argument set to `True` e.g. `app.run_server(debug=True)`
- The `python-dotenv` package – can be installed with `pip install python-dotenv`

Environment variables

Workbench proxies dash servers by managing the `PORT` and `DASH_REQUESTS_PREFIX_PATHNAME` environment variable in `.env` files. Users should avoid setting these variables in their own code.

Dash doesn't always respect the port provided by `python-dotenv`, even when it is properly reading other variables from the file, such as `DASH_REQUESTS_PREFIX_PATHNAME`. Posit is looking into a long-term fix for this issue, but there are manual steps that can be taken to force dash to use the provided port.

One way to force dash to use the provided port is to run your code using the Python extension and VS Code's debugger. Alternatively, update your main application file to load the `python-dotenv` file before importing Dash, as in the following:

```
from dotenv import load_dotenv
load_dotenv()
import dash
```

Manually setting a port number

To force Workbench to use a specific port, set the desired port in the `run_server` call or add a comment to the top of the main app file containing the port number in the following format:

```
# .run_server(port = 'your_port_num_here').
```

FastAPI

Because Workbench sessions run remotely, users must properly set the `root_path` variable in their FastAPI applications. FastAPI documents this requirement in more detail in their Behind a Proxy documentation. As described in the URL Prefixes section of this guide, `rserver-url` should be used to determine the root path's value.

Running FastAPI with Uvicorn

Using the default port

When using the Uvicorn ASGI server and the default FastAPI port, 8000, no extra configuration is required by the user. This is because Workbench sets the environment variable `UVICORN_ROOT_PATH` to the necessary value for port 8000 in all VS Code and Jupyter sessions. It is recommended that rather than relying on this default port, you specify the `port` and `root_path` directly.

Using a specific port

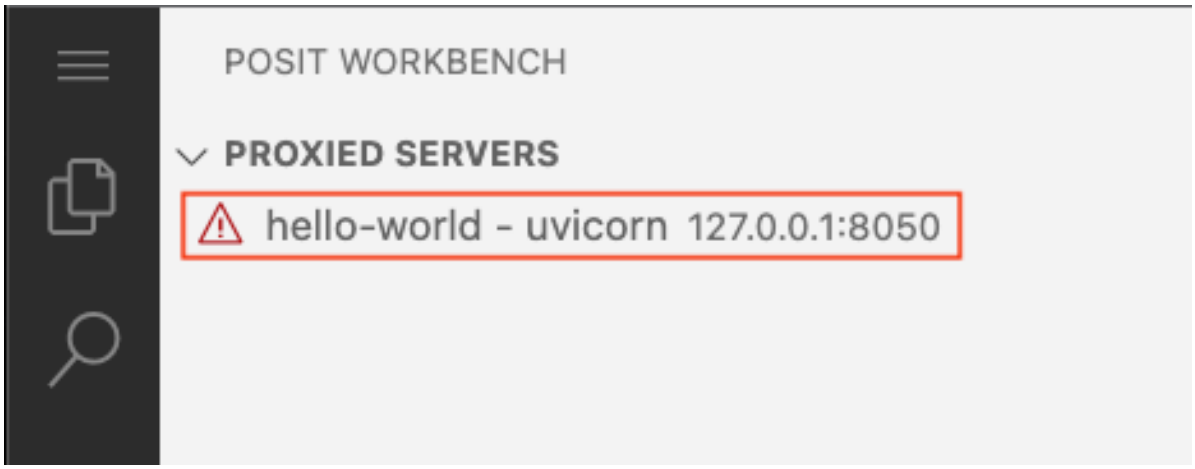
Users can specify a port by starting Uvicorn at the terminal and passing the desired port and root path as a command line argument or by importing the Uvicorn module into their Python code and passing a port to the `uvicorn.run` function. Users can retrieve the URL to pass as the root path by using the `rserver-url` binary as described in the URL Prefixes section of this guide and below.

Starting Uvicorn at the terminal

When specifying a port that is not 8000 at the command line, users must pass the `root-path` argument to Uvicorn. This looks like the following:

```
$ uvicorn main:app --port=8051 --root-path=$(/usr/lib/rstudio-server/bin/rserver-url -l 8050)
```

If this root path is not specified and a port has been passed to Uvicorn, the server will appear in the Proxied Servers view with a warning:



You can access the server by clicking on it as normal, but all features may not work as expected.

Import the Uvicorn module

> **Note**
>
> The examples in this section use port 8001. This value should be replaced with your desired port.

Users can programmatically specify a port to run their application on by passing the port number to the Uvicorn constructor. Typically, this would look like the following code:

```
from fastapi import FastAPI
import uvicorn

app = FastAPI()

if __name__ == '__main__':
    uvicorn.run(app, port = 8001)
```

When running in Workbench, users should call the `rserver-url` binary with the `-l` flag to set the `root_path` variable. The Posit Workbench VS Code Extension provides the **Posit Workbench FastAPI Uvicorn Root Path** code snippet for users to easily retrieve this required code. This snippet can be retrieved by typing `from fastapi import FastAPI`:

```
from fastapi import FastAPI
import uvicorn

app = FastAPI()

if __name__ == '__main__':
    path, port = '', 8001

    if 'RS_SERVER_URL' in os.environ and os.environ['RS_SERVER_URL']:
        path = subprocess.run(f'echo $(/usr/lib/rstudio-server/bin/rserver-url -l {port})'
                              stdout=subprocess.PIPE, shell=True).stdout.decode().strip()
    uvicorn.run(app, port = port, root_path = path)
```

In this code, the `path` variable will be set to the URL prefix for the current Workbench session and the requested port. It will look like the URL that your VS Code session is running at but with an additional `/p/<port-id>` suffix. Because this value will vary per session, it should not be hard-coded and `rserver-url` should always be used to generate the value. Note that the `path` variable is only set when the environment variable `RS_SERVER_URL` is set; this ensures that the `path` variable will not be set when this code is run outside of Workbench. If you want to set `path` to a different value when run outside of Workbench, replace the empty string that

`path` is initialized to with your desired path:

```
    path, port = '<default-path>', 8001
```

Flask

Because Posit Workbench runs an Nginx HTTP proxy server, Flask applications must include additional logic to tell Flask it is running behind a proxy. Flask recommends using the Werkzeug middleware to do this. From Werkzeug:

> Middlewares wrap applications to dispatch between them or provide additional request handling
>
> This module provides a middleware that adjusts the WSGI environ based on `X-Forwarded-` headers that proxies in front of an application may set. When an application is running behind a proxy server, WSGI may see the request as coming from that server rather than the real client. Proxies set various headers to track where the request actually came from. This middleware should only be used if the application is actually behind such a proxy, and should be configured with the number of proxies that are chained in front of it. Not all proxies set all the headers. Since incoming headers can be faked, you must set how many proxies are setting each header so the middleware knows what to trust.

The Posit Workbench Extension provides the Posit Workbench Flask Interface code snippet so that users can quickly access this middleware. This snippet can be triggered by typing `from flask import` within a python file and provides the following code:

```
from flask import Flask
import os
app = Flask(__name__)
if 'RS_SERVER_URL' in os.environ and os.environ['RS_SERVER_URL']:
  from werkzeug.middleware.proxy_fix import ProxyFix
  app.wsgi_app = ProxyFix(app.wsgi_app, x_prefix=1)
```

The line `if 'RS_SERVER_URL' in os.environ and os.environ['RS_SERVER_URL']:` tests if the `RS_SERVER_URL` environment variable is set before implementing the middleware fix and ensures that this middleware is only run when running the application from within Posit Workbench.