Building Reactive Data Apps with Shinylive and WebAssembly

Christoph Scheuch

Founder of Tidy Intelligence

Pain points of Python web apps

- **Traditional web apps** rely on:
  - Server backend (Flask, Django, FastAPI)
  - Frontend in JavaScript or Python frameworks (e.g., Streamlit, Dash)
- Complex **state management** & data caching
- Potential **deployment** complexity

Enter WebAssembly (Wasm)

> A binary instruction format for a stack-based virtual machine

What does this mean?

- Run compiled code (C/C++/Rust/Python) in the browser
- Near-native performance
- No hosted backend server required
- Enables static deployment (e.g., GitHub Pages, Netlify)

Enter Shiny & Shinylive

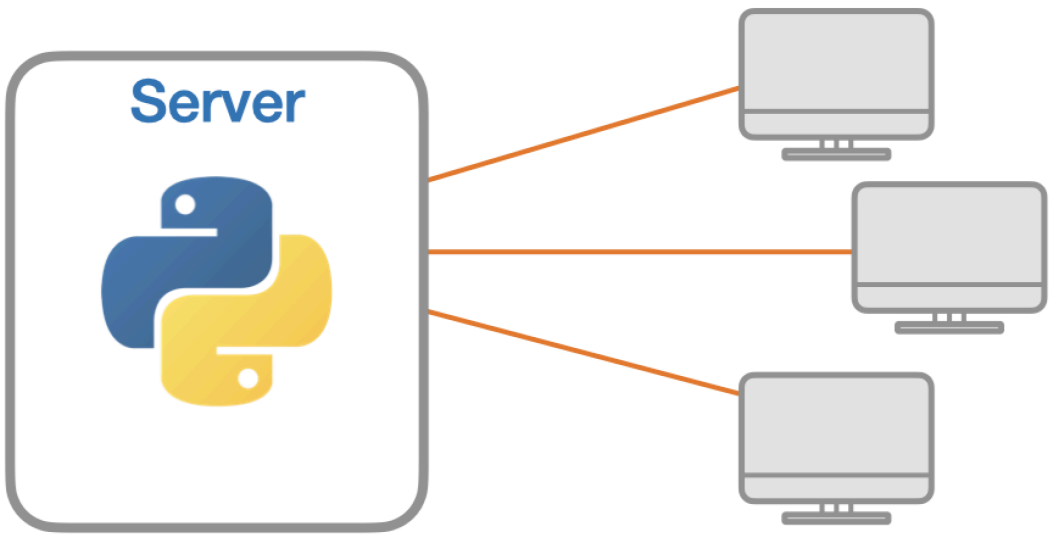
Shiny: reactive data apps in *pure Python*

- No manual state management (e.g., callback functions)
- Automatic reactive execution engine
- Full support for CSS & JavaScrip customization

Shinylive: run Shiny apps entirely in the browser

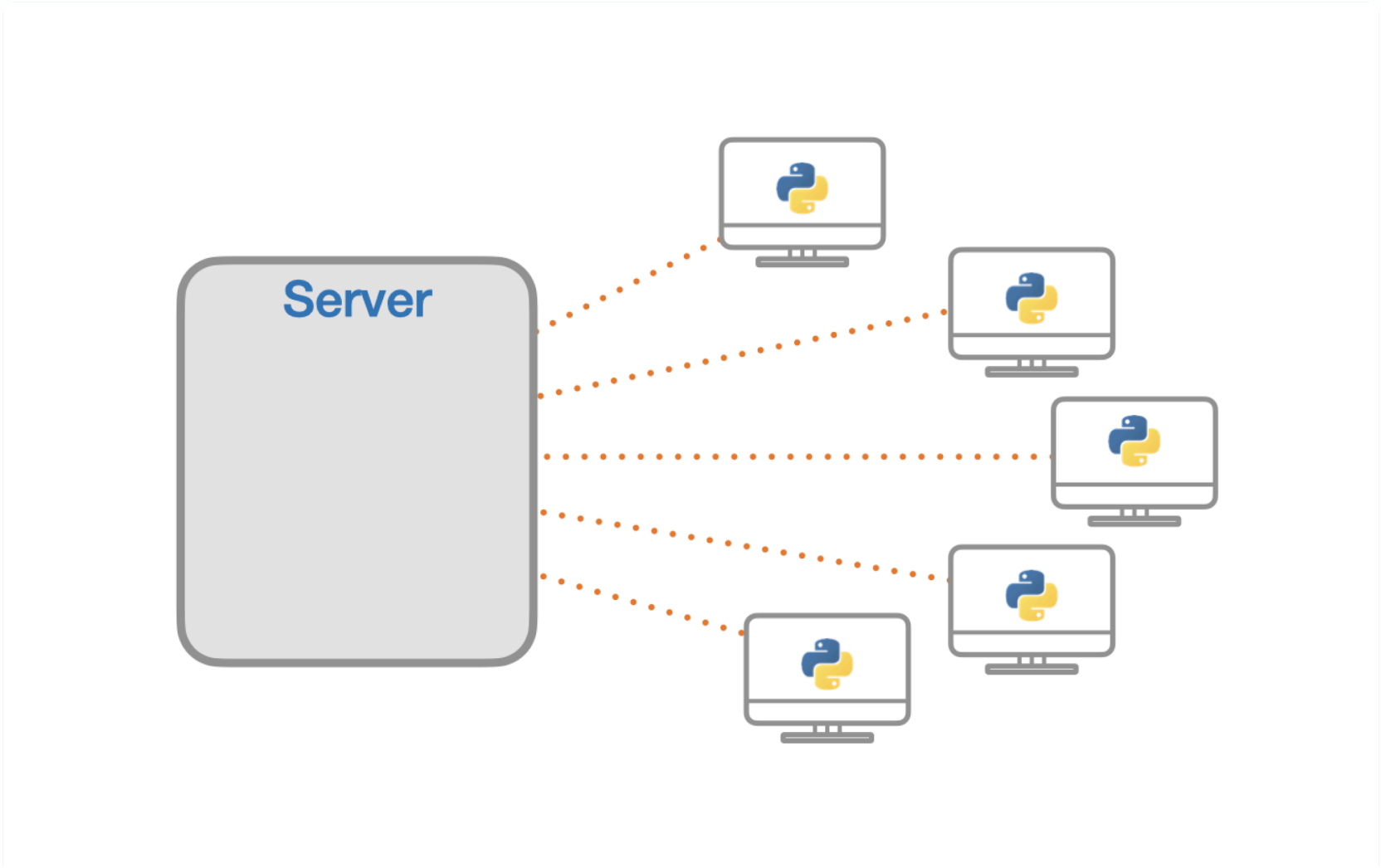
- Built on *Pyodide* (= port of CPython to Wasm)

Traditional Shiny deployment



shiny.posit.co

Shinylive deployment



shiny.posit.co

Example: a simple data app in Shiny

```
import matplotlib.pyplot as plt
import numpy as np
from shiny import App, render, ui

# Define the UI
app_ui = ui.page_sidebar(
    ui.sidebar(ui.input_slider("n", "N", min=0, max=100, value=20)),
    ui.output_plot("histogram"),
    title="Hello, PyData Berlin!",
)


# Define the server logic
def server(input, output, session):
    @output
    @render.plot
    def histogram():
        np.random.seed(1234)
```

Building the Shinylive app

Run `shinylive export app docs` in terminal

```
docs
├── app.json           # The app's files serialized to JSON
├── index.html         # A web page for the application
├── edit
│   └── index.html     # A web page for an editor view of the application
├── shinylive-sw.js    # Shinylive service worker
└── shinylive          # Shinylive content
    └── pyodide        # Pyodide files
```

Script is encoded in `app.json`

```
[
    {
        "name": "app.py",
        "content": "import matplotlib.pyplot as plt\nimport numpy as np\nfrom
        "type": "text"
    }
]
```

Deploying Shinylive apps as static pages

How-To by Rami Krispin

Testing the deployed app

tidy-intelligence.github.io/pydata-berlin-2025

Shipping data with Parquet

- **Parquet** = efficient, compressed columnar data format
- Can be read **in-browser** using PyArrow / Polars
- Ideal for **small datasets** & offline-first apps
- Disadvantage: (potentially public) **static data**

Example: reading Parquet files

```python
import matplotlib.pyplot as plt
from shiny import App, render, ui

# Import polars to load Parquet
import polars as pl

df = pl.read_parquet("app-parquet/data.parquet")

app_ui = ui.page_sidebar(
    ui.sidebar(ui.input_slider("n", "N", min=0, max=100, value=20)),
    ui.output_plot("histogram"),
    title="Hello, PyData Berlin!",
)


def server(input, output, session):
    @output
    @render.plot
```

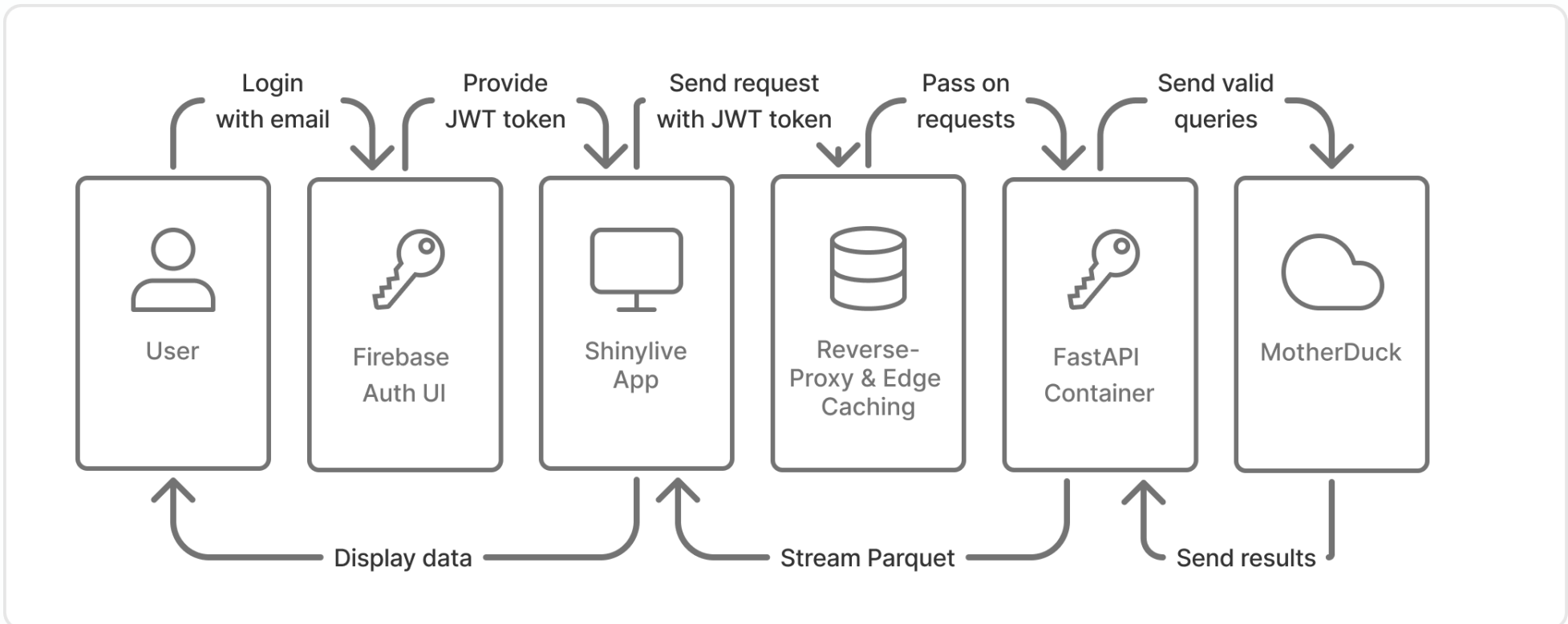
Files are also encoded in `app.json`

```
[
    {
        "name": "app.py",
        "content": "import matplotlib.pyplot as plt\n\n# Import polars to load
        "type": "text"
    },
    {
        "name": "data.parquet",
        "content": "UEFSMRUAFd42FcwzLBXqBhUAFQYVBhw2ACgInJ4YKaWuYUAYCAP+fEoMRkd
        "type": "binary"
    }
]
```

Optional backend with FastAPI

- Offload **compute** or **storage** tasks to dedicated server
- Handle **authentication**, remote database access, heavy lifting
- Build API endpoints with **FastAPI** as needed
- Disadvantage: authentication may be **complex**

Example: complex auth & data backend



Design patterns

- **Pure client-side**: when datasets are small, apps are simple
- **Hybrid mode**: add backend for auth, heavy compute, or write access
- **Modularize**: keep server logic stateless
- Use **reactivivty** to simplify UI logic

Limitations to be aware of

- **Bundle size** (e.g., Pyodide ~3MB, Numpy ~2.5MB, Pandas ~4.1MB)
- Initial **loading time** (but caching helps)
- **Limited packages** (pure Python, no C extensions unless Wasm compiled)
- Browser **memory & performance** constraints

When to use this stack

✅ Great for:

- Dashboards, small data apps (potentially behind firewalls)
- Education / demos / data exploration
- Lightweight deployments (e.g., GitHub Pages)

🚫 Not ideal for:

- Real-time streaming or huge datasets
- Complex authentication or role-based access control

Key takeaways

- Wasm + Shinylive opens new doors for **Python in the browser**
- **Fully reactive apps** without JavaScript or explicit callbacks
- Efficient **local data** access via **Parquet**
- **Optional FastAPI** services for hybrid models

Thank You 🙏

Follow on LinkedIn

GitHub Repo

Resources

- WebAssembly: webassembly.org
- Shinylive: shiny.posit.co/py/get-started/shinylive
- Pyodide: pyodide.org
- Apache Parquet: parquet.apache.org
- FastAPI: fastapi.tiangolo.com
- Repo: tidy-intelligence/pydata-berlin-2025