A Simple Guide to Safari Extensions

1 What Are Safari Extensions?

A Safari extension adds new features to Safari or modifies its behavior. It uses HTML, CSS, JavaScript, and JSON at its core. There are two main types:

1.1 Safari Web Extensions

These rely on the WebExtensions API, also used by Chrome, Firefox, and Edge, making them generally cross-browser.

1.2 Safari App Extensions

These live inside a macOS app and integrate deeply with Safari and macOS. Ideal if you need features that interact with the system at a lower level.

2 Common Languages and Tools

While Safari extensions are primarily written in HTML, CSS, JavaScript, and JSON, other languages can provide additional power:

- **TypeScript:** Adds type safety to JavaScript.
- **WebAssembly:** Run C/C++/Rust code in the browser with near-native speed.
- **Swift:** Especially for Safari App Extensions on macOS/iOS.
- **Python:** Handy for build scripts, automation, or backend tasks.
- **C++:** Can be compiled to WebAssembly for computationally intensive tasks.

2.1 Using Just Python or C++?

You cant build the entire extension in these languages alone. Browsers expect HTML, CSS, JavaScript, and JSON. Python or C++ is best used on the side: generating config files or doing heavy lifting compiled to WebAssembly.

3 Pros and Cons of Key Languages

Language	Pros	Cons
HTML	Universal layout	Static by itself
CSS	Powerful styling	Complex for large-scale design
JavaScript	Widely supported, flexible	No static typing by default
TypeScript	Type safety, large project scaling	Compiles to JavaScript
WebAssembly	Near-native speed	Requires JS integration
Swift	Deep Apple integration	Apple-only
Python	Great for scripting	Not for the browser core

4 Accessing Websites and Data

4.1 Content Scripts

Content scripts run in the context of a webpages DOM.

```
{%
  "manifest_version": 2,
  "name": "Content Script Example",
  "version": "1.0",
  "content_scripts": [
    {%
      "matches": ["<all_urls>"],
      "js": ["content.js"]
    }
  ]
}
```

```
// content.js
```

```
console.log("Page title is:", document.title);
```

4.2 Browser APIs

Use built-in APIs for data fetching or tab management:

```
fetch("https://api.example.com/data")
  .then(r => r.json())
  .then(data => console.log(data))
  .catch(err => console.error("Fetch error:", err));
```

5 Integrating Python, Qt, and C++

5.1 Python for Scripting or Backend

You might fetch data or manipulate files:

```
import requests

res = requests.get("https://api.example.com/data")
if res.status_code == 200:
    print(res.json())
else:
    print("Error fetching data")
```

5.2 Qt for a Companion GUI

If you need a desktop app to configure your extension:

```
from PySide6.QtWidgets import (
    QApplication, QWidget, QVBoxLayout, QPushButton
)

app = QApplication([])
window = QWidget()
```

```
layout = QVBoxLayout()

button = QPushButton("Save Settings")
layout.addWidget(button)

window.setLayout(layout)
window.show()
app.exec()
```

5.3 C++ via WebAssembly for Performance

Compile C++ to Wasm and call it from JavaScript.

```
// mymodule.cpp
#include <emscripten/emscripten.h>

extern "C" {%
  EMSCRIPTEN_KEEPALIVE
  int add(int a, int b) {%
    return a + b;
  }
}
```

Compile:

```
emcc mymodule.cpp -o mymodule.wasm -s EXPORTED_FUNCTIONS='["_add"]'
```

Then use from JavaScript:

```
fetch("mymodule.wasm")
  .then(r => r.arrayBuffer())
  .then(bytes => WebAssembly.instantiate(bytes))
  .then(result => {%
    console.log(result.instance.exports.add(2, 3));
  });
```

6 Conclusion

Safari extensions rely on standard web technologies, but you can use other languages to streamline workflows or speed up certain tasks. Whether you automate files with Python or bring in C++ via WebAssembly, you can combine these tools to build powerful Safari extensions that go beyond simple front-end scripting.