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 Examples of Safari Web Extensions

Below are additional examples that show how you might structure a simple Safari Web Extension. Each example uses standard web technologies, but you can adapt them to TypeScript, WebAssembly, or other languages as needed.

6.1 Dark Mode Toggler

This example toggles a dark mode style on any webpage. It includes a basic `manifest.json`, a background script, and a content script. The variable, function, and constant naming in the newly shown code follows a custom convention for clarity.

```
{
  "manifest_version": 2,
  "name": "Dark Mode Toggler",
  "version": "1.0",
  "description": "Toggle dark mode on any page.",
  "permissions": [
    "activeTab"
  ],
  "browser_action": {
    "default_title": "Toggle Dark Mode",
    "default_icon": "icon.png"
  },
  "background": {
    "scripts": ["background.js"]
  },
  "content_scripts": [
    {
      "matches": ["<all_urls>"],
      "js": ["content.js"]
    }
  ]
}
```

```
// background.js
function ToggleDarkMode() {
  chrome.tabs.executeScript({
    file: "content.js"
  });
}
```

```
chrome.browserAction.onClicked.addListener(ToggleDarkMode);
```

```
// content.js
const DARK_MODE_CLASS = "dark-mode-active";

function TogglePageDarkMode() {
  let bodyElement = document.body;
  if (bodyElement.classList.contains(DARK_MODE_CLASS)) {
    bodyElement.classList.remove(DARK_MODE_CLASS);
  } else {
    bodyElement.classList.add(DARK_MODE_CLASS);
  }
}

TogglePageDarkMode();
```

```
/* content.css: example stylesheet if needed */
.dark-mode-active {
  filter: invert(1) hue-rotate(180deg);
}
```

You can bundle the optional `content.css` if your extension needs more styling control. When the user clicks the extension's icon, it injects `content.js` which toggles a dark mode filter.

6.2 Link Shortener Extension

This extension fetches a short link from an API and inserts it into the current pages DOM. Its useful for quickly sharing long links. The `manifest.json` is similar, so well just focus on the content script.

```
// content.js
const API_URL = "https://api.tinyurl.com/create";
let currentUrl = window.location.href;

function FetchShortLink(urlToShorten) {
  return fetch(API_URL, {
    method: "POST",
```

```
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({
      url: urlToShorten
    })
  })
    .then(response => response.json());
}

function InsertShortLinkIntoPage(shortLink) {
  let messageContainer = document.createElement("div");
  messageContainer.style.position = "fixed";
  messageContainer.style.top = "10px";
  messageContainer.style.right = "10px";
  messageContainer.style.padding = "10px";
  messageContainer.style.backgroundColor = "#1F618D";
  messageContainer.style.color = "#FFFFFF";
  messageContainer.style.zIndex = 999999;
  messageContainer.textContent = "Short link: " + shortLink;
  document.body.appendChild(messageContainer);
}

FetchShortLink(currentUrl)
  .then(data => {
    if (data.data && data.data.tiny_url) {
      InsertShortLinkIntoPage(data.data.tiny_url);
    } else {
      console.error("Shortening failed:", data);
    }
  })
  .catch(error => console.error("Request error:", error));
```

When the content script runs, it posts the pages URL to a link-shortening API, then displays the short link in a fixed-position overlay. You could add a browser action or context menu to trigger it only when requested.

7 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. The examples above demonstrate how you can inject and manage scripts, toggle styling, or integrate external APIs, and you can expand on them to create truly unique experiences in Safari.