Safari Extensions: A Modern Guide

1 Introduction

Safari extensions enhance browser functionality by interacting with web pages or Safari itself. They are built using web technologies like HTML, CSS, and JavaScript. Apple supports two types of extensions:

Information

- **Safari Web Extensions: Based on the WebExtensions API, compatible with Chrome, Firefox, and Edge. Recommended for most use cases due to cross-browser compatibility.**
- **Safari App Extensions: Built into macOS apps and integrate deeply with macOS and Safari.**

2 Languages and Technologies

Safari extensions use a variety of languages depending on their purpose. Some common languages include:

2.1 Core Languages (Web Extensions)

- **HTML**: Defines the structure of the extension's UI, such as popups and options pages.
- **CSS**: Styles the visual elements of the extension.
- **JavaScript**: Implements logic, interactivity, and communication between the extension's components.
- **JSON**: Used in the `manifest.json` file to define metadata, permissions, and configuration.

2.2 Advanced and Supporting Languages

- **TypeScript**: Adds static typing to JavaScript, improving maintainability and reducing runtime errors.
- **WebAssembly (Wasm)**: For performance-critical tasks like image processing or cryptographic computations. Typically written in:
  - **C**
  - **C++**
  - **Rust**
- **Swift**: Preferred for Safari App Extensions due to its deep integration with macOS and iOS.
- **Python**: Used for scripting, automation, and backend tasks supporting the extension.

2.3 Can Extensions Be Written Entirely in Python or C++?

Safari and Chrome extensions cannot be written entirely in Python or C++. These languages are not directly supported for building the core extension functionality, which requires HTML, CSS, JavaScript, and JSON. However, Python and C++ can complement extensions by:

- **Python**: Automating workflows, generating configuration files like `manifest.json`, or serving as a backend for advanced data processing.
- **C++**: Handling performance-intensive tasks by compiling into WebAssembly and integrating with JavaScript.

While these languages are invaluable for enhancing extensions, JavaScript remains essential for interacting with browsers.

3 Pros and Cons of Key Languages

Table 1: Comparison of Languages Used in Safari Extensions

Language/Tool	**Pros**	**Cons**
HTML	Universal and simple for defining UI structure	Limited to static content
CSS	Flexible for styling; works seamlessly with HTML	Complex for dynamic designs
JavaScript	Powerful for logic and browser APIs; widely supported	No static typing; prone to runtime errors

Language/Tool	Pros	Cons
TypeScript	Enhances JavaScript with static typing	Requires compilation to JavaScript
WebAssembly	Native-like performance; ideal for heavy computation	Steep learning curve; requires JavaScript integration
Swift	Optimized for macOS/iOS; modern and safe	Limited to Apple platforms
Python	Excellent for scripting and backend services	Not directly usable in Safari extensions

4 Accessing Websites and Data

Safari extensions can access websites and retrieve data using content scripts and browser APIs. Here are the key techniques:

4.1 Content Scripts

Content scripts allow your extension to interact with web pages. Use them to manipulate DOM elements or extract data.

Example: Extracting Page Title

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

Add this to `manifest.json`:

Example: Updating `manifest.json`

```
{
  "manifest_version": 2,
  "name": "Website Data Access",
  "version": "1.0",
  "content_scripts": [
    {
      "matches": ["<all_urls>"],
      "js": ["content.js"]
    }
  ]
}
```

4.2 Browser APIs

Browser APIs provide advanced methods for accessing and managing data.

- **`fetch()`**: Retrieve data from APIs or websites.

- **tabs**: Access and manage browser tabs.

Example: Fetching Data from an API

```
fetch("https://api.example.com/data")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error("Error fetching data:", error));
```

5 Integrating Python, Qt, and C++ with Extensions

While Safari and Chrome extensions do not directly support languages like Python, Qt, or C++, they can complement extension development in these ways:

5.1 Python

Python is ideal for automating tasks like generating files, testing extensions, or serving as a backend for advanced features.

Example: Using Python to Fetch Data

```
import requests

response = requests.get("https://api.example.com/data")
if response.status_code == 200:
    print(response.json())
else:
    print("Failed to fetch data")
```

5.2 Qt

Qt can create companion apps to manage extension configurations or display advanced data visualizations.

Example: Qt GUI for Managing Extension Settings

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

app = QApplication([])
window = QWidget()
layout = QVBoxLayout()

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

window.setLayout(layout)
window.setWindowTitle("Extension Manager")
window.show()
app.exec()
```

5.3 C++

C++ can optimize performance-critical tasks by compiling into WebAssembly and integrating with JavaScript.

Example: C++ to WebAssembly Integration

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

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

Compile the code:

Compiling C++ to WebAssembly

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

Use it in JavaScript:

Example: Using C++ in JavaScript

```
fetch("mymodule.wasm").then(response =>
    response.arrayBuffer()
).then(bytes =>
    WebAssembly.instantiate(bytes)
).then(result => {
    const add = result.instance.exports.add;
    console.log(add(2, 3)); // Outputs: 5
});
```

6 Conclusion

Safari extensions and Chrome extensions share the WebExtensions API foundation. While Chrome extensions are cross-platform, Safari extensions emphasize macOS/iOS integration and privacy. Python, Qt, and C++ can greatly enhance Safari extension development by automating workflows, creating companion apps, or optimizing performance.