Sample Code

Developing a browser app that uses an alternative browser engine

Create a web browser app and associated extensions.

Download

iOS 17.4+ | Xcode 15.3+

Overview

This sample code project demonstrates how a web browser app uses `BrowserEngineKit` and XPC to communicate with its alternative browser engine over XPC, and implement a browser UI. The workspace contains four targets that define components of the browser app:

- `BrowserExample` contains the main app that presents the UI.
- `RenderingExtension` contains the rendering extension that the app uses to play media and render complex views.
- `NetworkingExtension` contains the networking extension that the app uses to fetch data from remote websites.
- `WebContentExtension` contains the web content extension that the app uses to parse HTML content and compile code just-in-time.

For more information on how these targets work together to support a custom web browser engine, see Designing your browser architecture.

Configure the sample code project

To build a web browser that uses an alternative browser engine, including the one in this sample project, you need to apply for certain entitlements from WWDR. For more information on how to

use these entitlements, see Creating browser extensions in Xcode. For information on applying for the entitlements, see Using alternative browser engines in the European Union.

To build the sample code project for a device, you must target an iPhone that supports the `arm64e` instruction set. `BrowserEngineKit` doesn't support iPadOS, so this project won't build for iPad.

Open a new tab

When someone using the browser app opens a new tab, the browser app's `BrowserPage` creates a `TabContentView` to display the tab's contents, which gets its data from the tab's `TabViewModel`.

```
struct TabContentView: View {

  @ObservedObject var tab: TabViewModel

  var body: some View {
    if let error = tab.error {
      VStack {
        Image(systemName: "exclamationmark.triangle")
        Text(error.localizedDescription)
      }
    } else {
      WebViewRepresentable(webView: tab.webView)
    }
  }
}
```

The `TabViewModel`'s `webView` property is a `WebView` that displays the tab's contents, which the `TabContentView` wraps in a `WebViewRepresentable` to display using SwiftUI.

The `WebView`'s initializer creates a `WebContentView`, which is the view that renders the HTML document in the tab. `WebContentView` calls `launchProcesses(id: PageID)`, which does the following:

- Creates a new web content extension process to render the web content, and set up an XPC connection between the browser app and the web content process.

```
    let contentProcess = try await getOrLaunchContentProcess(pageID: id)
    let contentConnection = try contentProcess.makeLibXPCConnection()
    let contentProxy = WebContentExtensionProxy(connection: contentConnection)
```

```
try contentProxy.lookdown(version: lookdownVersion)
```

- Gets a connection to the single rendering-extension process, launching it if necessary.

```
let renderingProcess = try await getOrLaunchRenderingProcess()
let renderingConnection = try renderingProcess.makeLibXPCConnection()
let renderingProxy = RenderingExtensionProxy(connection: renderingConnection)
```

- Asks the rendering process to create an anonymous XPC connection and sends an endpoint for the connection to the browser app.

```
let renderingEndpoint = try await renderingProxy.getEndpoint()
```

- Repeats steps 2-3 for the single networking extension process, so that the browser app has an anonymous XPC connection endpoint for the networking process.

```
let networkProcess = try await getOrLaunchNetworkProcess()
let networkConnection = try networkProcess.makeLibXPCConnection()
let networkProxy = NetworkingExtensionProxy(connection: networkConnection)
let networkEndpoint = try await networkProxy.getEndpoint()
```

- Sends a bootstrap message to the web-content process, handing it the endpoints to the other two extension processes, so that they can communicate directly.

```
try await contentProxy.bootstrap(renderingExtension: renderingEndpoint, networkE
```

The web content process responds to the bootstrap message by retrieving the anonymous XPC connections for the other extensions from the endpoints it was sent by the host app, and sends them each “ping” messages to ensure the connections are valid.

```
private func perform(bootstrap cmd: WebContentExtensionBootstrapCommand) async thr
  log.log("performing bootstrap")

  if let renderingProxy = self.renderingProxy { // Connect to the Rendering extens
    log.log("already connected to rendering extension: \(String(describing: render
  } else {
    let endpoint = cmd.renderingEndpoint
    log.log("connecting to rendering extension at: \(String(describing: endpoint))
    let connection = xpc_connection_create_from_endpoint(endpoint)
    self.renderingProxy = .init(connection: connection)
```

```
      try await connection.ping()
      log.log("connected to rendering extension: \(String(describing: connection))")
    }

    if let networkProxy = self.networkProxy {  // Connect to the Networking extensio
      log.log("already connected to network extension: \(String(describing: networkP
    } else {
      let endpoint = cmd.networkEndpoint
      log.log("connecting to network extension at: \(String(describing: endpoint))")
      let connection = xpc_connection_create_from_endpoint(endpoint)
      self.networkProxy = .init(connection: connection)
      try await connection.ping()
      log.log("connected to network extension: \(String(describing: connection))")
    }
  }
```

For more information on this process, see the section “Pass anonymous connection endpoints between extensions” in Using XPC to communicate with browser extensions.

Load web content

When someone using the browser app navigates to a new location, the `WebContentView` asks the proxy object that represents the web-content process to load the data at the location’s URL:

```
    let result = try await webContentProxy.load(destination: destination)
```

This method sends a network load message to the web-content process:

```
    let task: WebContentExtensionTask = .load(destination: destination)
    return try await connection.sendWithReply(task, decodingReplyAs: NetworkTaskResu
```

The web-content process checks what kind of data it’s trying to load. If it’s a URL, it asks the networking extension to load the content. Otherwise, it prepares the data itself, either by decoding a string, or loading the contents of a file.

```
    switch destination {
    case .url(let url):
      guard let networkProxy else {
        throw CustomBrowserError("not connected to the network extension")
      }
```

```
      return try await networkProxy.fetchData(from: url)
    case .htmlString(let string):
      guard let data = string.data(using: .utf8) else {
        throw CustomBrowserError("failed to get utf8 data from string")
      }
      return .init(response: nil, data: data, error: nil)
    case .localFile(let file):
      let data = try Data(contentsOf: file)
      return .init(response: nil, data: data, error: nil)
    }
```

> **Note**
>
> To open a file in your browser, send a bookmark with implicit security scope to the web-content process, so that it can extend its sandbox to access the file. For more information, see Accessing files in browser extensions.

In your browser app, you need to handle any networking errors at this point. If the networking extension loads the content, parse it using your alternative browser engine to create rendering commands, that you send to the rendering process. In the rendering process, update the layer that the browser app is hosting.

Finally, the web-content process replies to the browser app, telling the `WebContentView` to update its view.

```
    render(result)
```

See Also

Essentials

- Designing your browser architecture
  Isolate privileged access to operating system resources and private data from untrusted code.
- Preparing your app to be the default web browser
  Configure your browser app so users can set it as the default on their device instead of Safari.