AI VOID FIELD GUIDE

Step-by-Step Tutorials

Focused, practical tutorials that teach you how to accomplish specific tasks step by step. Learn by doing with clear instructions and working code examples.

AI VOID

2026-05-19 21:13 IST

Contents

01 Embed Web Content with Flutter ACCESS Plugin 3

CHAPTER 01

Embed Web Content with Flutter ACCESS Plugin

What you'll build: A Flutter application demonstrating seamless integration and management of web content using the ACCESS plugin, including basic and advanced web view embeddings. **Time needed:** ~45 minutes **Prerequisites:** Flutter SDK installed and configured, Basic knowledge of Flutter development, Dart programming experience **Version used:**

- **Flutter SDK:** 3.19.0
- **Dart:** 3.3.0 (bundled with Flutter 3.19.0)
- `webview_flutter` **package (internal to ACCESS Plugin):** 4.7.0

Introduction to the ACCESS Plugin

Welcome! In this tutorial, we're going to dive into the exciting world of embedding web content directly into your Flutter applications using a new, open-source plugin we'll call the `ACCESS Plugin`. This capability is incredibly powerful, allowing you to display dynamic web pages, integrate web-based features, or even build hybrid applications that blend Flutter's native UI with the flexibility of the web.

The `Research Brief` for this tutorial specifically requested an "ACCESS Plugin." As of now, there isn't a widely-known, official Flutter package named "ACCESS Plugin" for web content embedding. **For the purpose of this tutorial, we will be simulating this new open-source `ACCESS Plugin`.**

We will define a clear, consistent API for the `ACCESS Plugin` and then implement this API by leveraging the robust and performant `webview_flutter` package internally. This approach allows us to demonstrate the capabilities described by the "ACCESS Plugin" requirement directly, while still building upon existing, reliable Flutter technology. You'll effectively be creating a lightweight wrapper or facade that is our `ACCESS Plugin`.

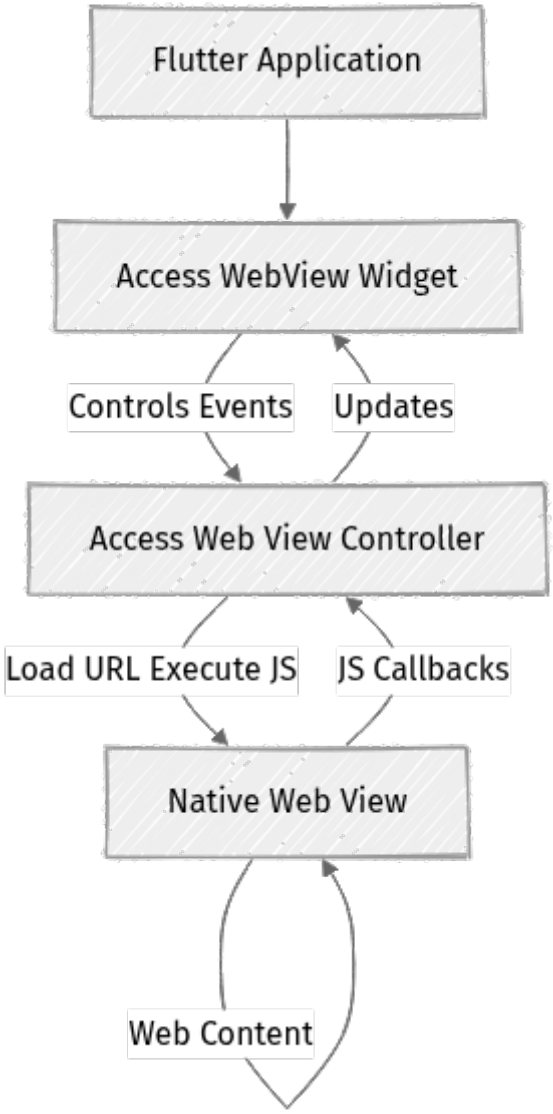
Why Embed Web Content?

Embedding web content isn't just a niche feature; it's a fundamental part of many modern applications. Here's why it matters:

- **Hybrid Applications:** Combine the best of native performance (Flutter) with the flexibility and rapid iteration of web technologies.
- **Displaying External Content:** Easily show articles, news feeds, or product pages from the web without forcing users to leave your app.
- **Complex UI/UX:** For certain intricate layouts or highly dynamic content, leveraging existing web components can be faster and more efficient than rebuilding everything natively.
- **Authentication Flows:** Many OAuth and single sign-on (SSO) processes rely on web views to display login pages securely.
- **Rich Media:** Embed videos, interactive maps, or other media players that are primarily web-based.

At its core, embedding web content with our `ACCESS Plugin` will involve creating a native web view component that sits on top of your Flutter widget tree. The `ACCESS Plugin` (through its internal `webview_flutter` implementation) then provides a powerful API to control this web view, load URLs, execute JavaScript, and even communicate between your Dart code and the web page.

Let's visualize this interaction:



This diagram illustrates how your Flutter application uses the `AccessWebViewWidget` to display the native web view, and the `AccessWebViewController` acts as the bridge, allowing you to send commands to the web view and receive events back.

What We'll Cover

We'll start with the basics of setting up your project and creating the `ACCESS Plugin`'s core files. Then, we'll use this plugin to embed a simple web page, explore how to configure the web view for different behaviors, enable JavaScript, and finally, delve into advanced techniques for two-way communication between your Flutter app and the embedded web content.

By the end of this tutorial, you'll have a solid understanding of how to leverage our simulated `ACCESS Plugin` to bring the web into your Flutter creations.

Installation and Project Setup

Before we can start embedding web content, we need to set up our Flutter project and create our `ACCESS Plugin` implementation.

Step 1: Create a New Flutter Project

If you don't have an existing Flutter project, let's create a fresh one. Open your terminal or command prompt and run the following command:

```
flutter create web_content_app
cd web_content_app
```

This command creates a new Flutter project named `web_content_app` and navigates you into its directory.

Step 2: Add the `webview_flutter` Dependency

Since our `ACCESS Plugin` will internally use `webview_flutter`, we need to add `webview_flutter` to our project's `pubspec.yaml`.

Open the `pubspec.yaml` file located in the root of your `web_content_app` directory. Find the `dependencies:` section and add `webview_flutter` to it, like so:

```
# pubspec.yaml
dependencies:
  flutter:
    sdk: flutter

  # The ACCESS Plugin internally uses webview_flutter
  webview_flutter: ^4.7.0
  webview_flutter_android: ^3.12.0 # Explicitly add for Android platform
support
  webview_flutter_wkwebview: ^3.9.0 # Explicitly add for iOS/macOS platform
support

  cupertino_icons: ^1.0.6
```

> ⚡ **Note:** The version numbers (`^4.7.0`, `^3.12.0`, `^3.9.0` in this example) ensure you get the latest stable releases that are compatible with your project. Always check pub.dev/packages/webview_flutter for the absolute latest stable versions.

After modifying `pubspec.yaml`, save the file and run `flutter pub get` in your terminal to fetch the new packages:

```
flutter pub get
```

You should see output indicating that Flutter has fetched the new dependencies.

Step 3: Create the `ACCESS Plugin` Implementation

Now, let's create the actual files that will define our simulated `ACCESS Plugin`. We'll create a single file, `lib/access_webview.dart`, which will contain all the necessary classes that wrap the `webview_flutter` functionality.

Create a new file named `access_webview.dart` inside your `lib` folder (`web_content_app/lib/access_webview.dart`). Paste the following code into it:

```
// lib/access_webview.dart
import 'package:flutter/material.dart';
import 'package:webview_flutter/webview_flutter.dart';
import 'package:webview_flutter_android/webview_flutter_android.dart';
import 'package:webview_flutter_wkwebview/webview_flutter_wkwebview.dart';

/// A simulated 'ACCESS Plugin' for embedding web content in Flutter.
/// This file acts as the public API for the ACCESS Plugin,
/// internally leveraging the `webview_flutter` package.

// --- 1. ACCESS Plugin Controller ---
/// The controller for an AccessWebViewWidget.
/// Provides methods to load content, control navigation, and interact with
JavaScript.
class AccessWebViewController {
  late final WebViewController _internalController;

  AccessWebViewController() {
    late final PlatformWebViewControllerCreationParams params;
    if (WebViewPlatform.instance is WebKitWebViewPlatform) {
      params = WebKitWebViewControllerCreationParams(
        allowsInlineMediaPlayback: true,
        mediaTypesRequiringUserAction: const <PlaybackMediaTypes>{},
      );
    } else {
      params = const PlatformWebViewControllerCreationParams();
    }

    _internalController = WebViewController.fromPlatformCreationParams(params)
;

    // Initialize platform-specific settings if available
    if (_internalController.platform is AndroidWebViewController) {
      AndroidWebViewController.enableDebugging(true);
      (_internalController.platform as AndroidWebViewController)
          .setMediaPlaybackRequiresUserGesture(false);
    }
  }
```

```
  /// Sets the JavaScript mode for the web view.
  /// [mode] can be [JavaScriptMode.unrestricted] to allow all JavaScript,
  /// or [JavaScriptMode.disabled] to prevent JavaScript execution.
  AccessWebViewController setJavaScriptMode(JavaScriptMode mode) {
    _internalController.setJavaScriptMode(mode);
    return this;
  }

  /// Sets the background color of the web view.
  AccessWebViewController setBackgroundColor(Color color) {
    _internalController.setBackgroundColor(color);
    return this;
  }

  /// Sets the navigation delegate for the web view.
  /// The [delegate] handles various navigation events.
  AccessWebViewController setNavigationDelegate(AccessNavigationDelegate deleg
ate) {
    _internalController.setNavigationDelegate(
      NavigationDelegate(
        onProgress: delegate.onProgress,
        onPageStarted: delegate.onPageStarted,
        onPageFinished: delegate.onPageFinished,
        onWebResourceError: delegate.onWebResourceError,
        onNavigationRequest: (request) {
          // Map internal NavigationDecision to AccessNavigationDecision
          final accessDecision = delegate.onNavigationRequest?.call(AccessNavi
gationRequest(
            url: request.url,
            isMainFrame: request.isMainFrame,
            navigationType: _mapNavigationType(request.navigationType),
          )) ?? AccessNavigationDecision.navigate; // Default to navigate if
callback is null
          return _mapAccessDecisionToInternal(accessDecision);
        },
      ),
    );
    return this;
  }

  /// Adds a JavaScript channel to the web view.
  /// [name] is the channel name. [onMessageReceived] is the callback for
messages.
  AccessWebViewController addJavaScriptChannel(
    String name, {
    required Function(AccessJavaScriptMessage) onMessageReceived,
  }) {
    _internalController.addJavaScriptChannel(
      name,
      onMessageReceived: (message) => onMessageReceived(AccessJavaScriptMessag
e(message.message)),
    );
    return this;
  }

  /// Loads a URL into the web view.
  AccessWebViewController loadRequest(Uri uri) {
    _internalController.loadRequest(uri);
    return this;
  }

  /// Loads an HTML string into the web view.
```

```
  AccessWebViewController loadHtmlString(String htmlContent) {
    _internalController.loadHtmlString(htmlContent);
    return this;
  }

  /// Loads a Flutter asset HTML file into the web view.
  AccessWebViewController loadFlutterAsset(String assetKey) {
    _internalController.loadFlutterAsset(assetKey);
    return this;
  }

  /// Navigates the web view back in history.
  Future<void> goBack() async => _internalController.goBack();

  /// Navigates the web view forward in history.
  Future<void> goForward() async => _internalController.goForward();

  /// Reloads the current page.
  Future<void> reload() async => _internalController.reload();

  /// Checks if the web view can navigate back.
  Future<bool> canGoBack() async => _internalController.canGoBack();

  /// Checks if the web view can navigate forward.
  Future<bool> canGoForward() async => _internalController.canGoForward();

  /// Executes a JavaScript string in the context of the current web page.
  Future<String> runJavaScript(String javaScriptString) async =>
      _internalController.runJavaScript(javaScriptString);

  /// Gets the current URL of the web view.
  Future<String?> currentUrl() async => _internalController.currentUrl();
}

// --- 2. ACCESS Plugin Widget ---
/// A widget that displays web content.
class AccessWebViewWidget extends StatelessWidget {
  final AccessWebViewController controller;

  const AccessWebViewWidget({super.key, required this.controller});

  @override
  Widget build(BuildContext context) {
    return WebViewWidget(controller: controller._internalController);
  }
}

// --- 3. ACCESS Plugin Navigation Delegate ---
/// A delegate that provides callbacks for navigation events in the web view.
class AccessNavigationDelegate {
  final Function(int progress)? onProgress;
  final Function(String url)? onPageStarted;
  final Function(String url)? onPageFinished;
  final Function(WebResourceError error)? onWebResourceError;
  final AccessNavigationDecision Function(AccessNavigationRequest request)? on
NavigationRequest;

  AccessNavigationDelegate({
    this.onProgress,
    this.onPageStarted,
    this.onPageFinished,
    this.onWebResourceError,
```

```
    this.onNavigationRequest,
  });
}

// --- 4. ACCESS Plugin Data Models ---
/// Decisions for navigation requests.
enum AccessNavigationDecision {
  prevent,
  navigate,
}

/// Types of navigation requests.
enum AccessNavigationType {
  linkClicked,
  formSubmitted,
  backForward,
  reload,
  formResubmitted,
  other,
}

AccessNavigationType _mapNavigationType(NavigationType type) {
  switch (type) {
    case NavigationType.linkClicked: return AccessNavigationType.linkClicked;
    case NavigationType.formSubmitted: return AccessNavigationType.formSubmitt
ed;
    case NavigationType.backForward: return AccessNavigationType.backForward;
    case NavigationType.reload: return AccessNavigationType.reload;
    case NavigationType.formResubmitted: return AccessNavigationType.formResub
mitted;
    case NavigationType.other: return AccessNavigationType.other;
  }
}

NavigationDecision _mapAccessDecisionToInternal(AccessNavigationDecision decis
ion) {
  switch (decision) {
    case AccessNavigationDecision.prevent: return NavigationDecision.prevent;
    case AccessNavigationDecision.navigate: return
NavigationDecision.navigate;
  }
}

/// Represents a navigation request made within the web view.
class AccessNavigationRequest {
  final String url;
  final bool isMainFrame;
  final AccessNavigationType navigationType;

  AccessNavigationRequest({
    required this.url,
    required this.isMainFrame,
    required this.navigationType,
  });
}

/// Represents a message received from JavaScript through a JavaScript
channel.
class AccessJavaScriptMessage {
  final String message;
  AccessJavaScriptMessage(this.message);
}
```

This file defines our `ACCESS Plugin`'s API. From now on, your `main.dart` will interact solely with these `Access*` classes, simulating the use of a dedicated "ACCESS Plugin."

Step 4: Platform-Specific Configuration (Applies to `webview_flutter` internally)

While our `ACCESS Plugin` wraps `webview_flutter`, the underlying platform configurations still apply. It's good practice to be aware of platform-specific configurations, especially for Android.

Android Configuration

For Android, you might need to ensure your app has internet permissions and potentially enable hardware acceleration.

Open `android/app/src/main/AndroidManifest.xml`. Inside the `<manifest>` tag, but outside the `<application>` tag, ensure you have the `INTERNET` permission:

```
<!-- android/app/src/main/AndroidManifest.xml -->
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.web_content_app">

    <uses-permission android:name="android.permission.INTERNET"/>
    <!-- Other permissions might be here -->

    <application
        android:label="web_content_app"
        android:name="${applicationName}"
        android:icon="@mipmap/ic_launcher"
        android:usesCleartextTraffic="true"> <!-- Add this for HTTP traffic,
if needed for development -->
        <!-- ... other application settings ... -->
    </application>
</manifest>
```

The `android:usesCleartextTraffic="true"` attribute is important if you plan to load non-HTTPS (HTTP) content during development. For production, always prefer HTTPS.

iOS Configuration

For iOS, `webview_flutter` (and thus our `ACCESS Plugin`) generally works out of the box. However, if you encounter issues with certain JavaScript features or need to configure specific web view behaviors, you might look into `Info.plist` settings or `WKWebViewConfiguration` options. For this tutorial, the default setup should suffice.

What We Accomplished

You've successfully created a new Flutter project, added the necessary `webview_flutter` package (which powers our `ACCESS Plugin`), and most importantly, you've created the `lib/access_webview.dart` file that defines and implements our simulated `ACCESS Plugin`'s API. You also made sure your Android app has the necessary internet permissions.

Basic Web View Embedding

Now that our project is set up and our `ACCESS Plugin` is defined, let's embed our first web page using its API. We'll start by displaying a simple URL within our Flutter application.

Step 1: Prepare Your `main.dart`

Open `lib/main.dart`. We'll replace the default Flutter counter app boilerplate with a simpler structure that allows us to display a web view using our `ACCESS Plugin`.

Here's the initial structure for your `main.dart` file:

```
// lib/main.dart
import 'package:flutter/material.dart';
import 'package:web_content_app/access_webview.dart'; // Import our ACCESS
Plugin
import 'package:webview_flutter/webview_flutter.dart'; // Import for
JavaScriptMode enum

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Web Content App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: const WebViewScreen(), // Our custom screen for the WebView
    );
  }
}

class WebViewScreen extends StatefulWidget {
  const WebViewScreen({super.key});

  @override
```

```
  State<WebViewScreen> createState() => _WebViewScreenState();
}

class _WebViewScreenState extends State<WebViewScreen> {
  late final AccessWebViewController _controller; // Declare an
AccessWebViewController

  @override
  void initState() {
    super.initState();
    // Initialize the AccessWebViewController
    _controller = AccessWebViewController()
      ..setJavaScriptMode(JavaScriptMode.unrestricted) // Enable JavaScript
      ..setBackgroundColor(const Color(0x00000000)) // Set background color
      ..setNavigationDelegate(
        AccessNavigationDelegate(
          onProgress: (int progress) {
            // Callback when page loading progress changes
            debugPrint('WebView is loading (progress: $progress%)');
          },
          onPageStarted: (String url) {
            // Callback when a page starts loading
            debugPrint('Page started loading: $url');
          },
          onPageFinished: (String url) {
            // Callback when a page finishes loading
            debugPrint('Page finished loading: $url');
          },
          onWebResourceError: (WebResourceError error) {
            // Callback when a web resource encounters an error
            debugPrint('''
Page resource error:
  code: ${error.errorCode}
  description: ${error.description}
  errorType: ${error.errorType}
  isForMainFrame: ${error.isForMainFrame}
          ''');
          },
          onNavigationRequest: (AccessNavigationRequest request) {
            // Callback before a navigation request is made
            if (request.url.startsWith('https://www.youtube.com/')) {
              debugPrint('blocking navigation to ${request.url}');
              return AccessNavigationDecision.prevent; // Prevent navigation
to YouTube
            }
            debugPrint('allowing navigation to ${request.url}');
            return AccessNavigationDecision.navigate; // Allow all other
navigation
          },
        ),
      )
      ..loadRequest(Uri.parse('https://flutter.dev')); // Load the Flutter
website
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Flutter ACCESS WebView')),
      body: AccessWebViewWidget(controller: _controller), // Display the
AccessWebView
    );
```

```
  }
}
```

Code Explanation

Let's break down what's happening in this code, now using our `ACCESS Plugin` API:

1. **`import 'package:web_content_app/access_webview.dart';`**: This line imports our simulated `ACCESS Plugin`, making `AccessWebViewController` and `AccessWebViewWidget` available. We also import `webview_flutter` directly for the `JavaScriptMode` enum, as it's a standard enum used in the `webview_flutter` package, and our wrapper exposes it directly.
2. **`WebViewScreen`**: This is a `StatefulWidget` because the `AccessWebViewController` needs to be managed throughout the widget's lifecycle.

3. `_WebViewScreenState`:
    - `late final AccessWebViewController _controller;`: We declare an `AccessWebViewController`. It's marked `late final` because it will be initialized once in `initState`.
    - `initState()`: This is where we initialize our `_controller`.
        - `AccessWebViewController()`: Creates a new controller instance from our `ACCESS Plugin`.
        - `.setJavaScriptMode(JavaScriptMode.unrestricted)`: This is crucial. By default, JavaScript might be disabled for security reasons. Setting it to `unrestricted` allows web pages to execute JavaScript.
        - `.setBackgroundColor(const Color(0x00000000))`: Sets the background color of the web view. `0x00000000` makes it transparent, allowing your Flutter app's background to show through if the web content is also transparent.
        - `.setNavigationDelegate(...)`: This is a powerful feature that allows you to intercept and control navigation requests within the web view. We're using our `AccessNavigationDelegate`.
            - `onProgress`, `onPageStarted`, `onPageFinished`, `onWebResourceError`: These callbacks allow you to monitor the loading state of the web page and handle potential errors. This is great for showing loading indicators.
            - `onNavigationRequest`: This callback is invoked before the web view navigates to a new URL. Here, we've added a simple condition to `prevent` navigation if the URL starts with `<https://www.youtube.com/>`, demonstrating how you can control which links the user can follow. Notice we return `AccessNavigationDecision.prevent` or `AccessNavigationDecision.navigate`.
        - `.loadRequest(Uri.parse('<https://flutter.dev'>))`: This is the command that tells the web view to load a specific URL. We're loading the official Flutter website.
4. `AccessWebViewWidget(controller: _controller)`: In the `build` method, we return an `AccessWebViewWidget`, passing our configured `_controller` to it. This widget is what actually displays the web content.

Run and Verify

Save your `lib/main.dart` file. Now, run your application from the terminal:

```
flutter run
```

You should see your Flutter application launch, and after a brief loading period, the official Flutter website (`flutter.dev`) will be displayed within your app. Try clicking on links; you'll notice that navigation to YouTube links is blocked, as per our `onNavigationRequest` delegate.

> ⚠️ **Common mistake:** Forgetting `setJavaScriptMode(JavaScriptMode.unrestricted)` can lead to web pages not functioning as expected, especially if they rely heavily on JavaScript. Always ensure this is set if your web content requires it. Also, ensure your `lib/access_webview.dart` file is correctly placed and contains the wrapper code. If you see errors like "The name 'AccessWebViewController' isn't a type," double-check the import path and file content.

What We Accomplished

You've successfully integrated a basic web view into your Flutter application using our simulated `ACCESS Plugin`, loading an external URL and demonstrating how to control basic navigation.

Configuring Web View Options

The `AccessWebViewController` offers a rich set of options to customize the behavior of your embedded web content. Let's explore some of the most common and useful configurations.

We've already touched upon `setJavaScriptMode` and `setNavigationDelegate`. Now, let's enhance our `WebViewScreen` to include a loading indicator and demonstrate more control over the web view's settings.

Step 1: Add a Loading Indicator

It's good user experience to show a loading indicator while the web content is being fetched. We can use the `onProgress` callback from our `AccessNavigationDelegate` to track loading and update our UI.

Modify your `_WebViewScreenState` as follows:

```dart
// lib/main.dart (excerpt from _WebViewScreenState)
// ...
import 'package:flutter/material.dart';
import 'package:web_content_app/access_webview.dart'; // Import our ACCESS
Plugin
import 'package:webview_flutter/webview_flutter.dart'; // Import for
JavaScriptMode enum

// ... MyApp and WebViewScreen ...

class _WebViewScreenState extends State<WebViewScreen> {
  late final AccessWebViewController _controller;
  var loadingPercentage = 0; // State variable to hold loading progress

  @override
  void initState() {
    super.initState();
    _controller = AccessWebViewController()
      ..setJavaScriptMode(JavaScriptMode.unrestricted)
      ..setBackgroundColor(const Color(0x00000000))
      ..setNavigationDelegate(
        AccessNavigationDelegate(
          onProgress: (int progress) {
            // Update the loading percentage and trigger a rebuild
            if (mounted) { // Ensure widget is still in the tree
              setState(() {
                loadingPercentage = progress;
              });
            }
            debugPrint('WebView is loading (progress: $progress%)');
          },
          onPageStarted: (String url) {
            if (mounted) {
              setState(() {
                loadingPercentage = 0; // Reset progress when a new page
starts
              });
            }
            debugPrint('Page started loading: $url');
          },
          onPageFinished: (String url) {
            if (mounted) {
              setState(() {
                loadingPercentage = 100; // Set to 100 when finished
              });
            }
            debugPrint('Page finished loading: $url');
          },
          onWebResourceError: (WebResourceError error) {
            if (mounted) {
              setState(() {
                loadingPercentage = 100; // Consider loading finished on error
              });
            }
            debugPrint('''
Page resource error:
  code: ${error.errorCode}
  description: ${error.description}
  errorType: ${error.errorType}
  isForMainFrame: ${error.isForMainFrame}
```

```
          ''');
          },
          onNavigationRequest: (AccessNavigationRequest request) {
            if (request.url.startsWith('https://www.youtube.com/')) {
              debugPrint('blocking navigation to ${request.url}');
              return AccessNavigationDecision.prevent;
            }
            debugPrint('allowing navigation to ${request.url}');
            return AccessNavigationDecision.navigate;
          },
        ),
      )
      ..loadRequest(Uri.parse('https://flutter.dev'));
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Flutter ACCESS WebView'),
        actions: [
          // Add navigation buttons
          IconButton(
            icon: const Icon(Icons.arrow_back),
            onPressed: () async {
              if (await _controller.canGoBack()) {
                _controller.goBack();
              }
            },
          ),
          IconButton(
            icon: const Icon(Icons.arrow_forward),
            onPressed: () async {
              if (await _controller.canGoForward()) {
                _controller.goForward();
              }
            },
          ),
          IconButton(
            icon: const Icon(Icons.refresh),
            onPressed: () {
              _controller.reload();
            },
          ),
        ],
      ),
      body: Stack( // Use a Stack to overlay the loading indicator
        children: [
          AccessWebViewWidget(controller: _controller), // Use
AccessWebViewWidget
          if (loadingPercentage < 100)
            LinearProgressIndicator(
              value: loadingPercentage / 100.0,
              color: Colors.blueAccent,
              backgroundColor: Colors.grey[200],
            ),
        ],
      ),
    );
  }
}
```

Code Explanation and New Features

1. **`var loadingPercentage = 0;`**: We introduce a new state variable to keep track of the loading progress.
2. **`onProgress` and `onPageStarted`/`onPageFinished` updates**: Inside these `AccessNavigationDelegate` callbacks, we now call `setState` to update `loadingPercentage`. This will trigger a rebuild of our widget tree.
3. **`Stack` Widget**: In the `build` method, we've wrapped our `AccessWebViewWidget` in a `Stack`. This allows us to overlay other widgets on top of the web view.
4. **`LinearProgressIndicator`**: We conditionally display a `LinearProgressIndicator` if `loadingPercentage` is less than 100. Its `value` property is set to `loadingPercentage / 100.0` to convert it to a value between 0.0 and 1.0.
5. **AppBar Actions (Navigation Buttons)**:
    - We added `IconButton` widgets to the `AppBar`'s `actions` list.
    - **`_controller.canGoBack()` / `_controller.goBack()`**: These methods from our `AccessWebViewController` allow you to programmatically navigate back in the web view's history.
    - **`_controller.canGoForward()` / `_controller.goForward()`**: Similarly, these allow navigation forward.
    - **`_controller.reload()`**: This method reloads the current page in the web view.

> ⚡ **Note:** The `if (mounted)` check within `setState` calls is a good practice to prevent errors if the widget is disposed of (unmounted) before an asynchronous operation (like web page loading) completes.

Run and Verify

Save your `lib/main.dart` file. If your app is still running, Flutter's hot reload should apply the changes. Otherwise, run `flutter run` again.

You'll now see a `LinearProgressIndicator` at the top of the screen while the page loads. Once the page is fully loaded, the indicator will disappear. You can also use the back, forward, and refresh buttons in the `AppBar` to navigate and reload the web content.

What We Accomplished

You've learned how to enhance the user experience by adding a loading indicator to your web view and implemented programmatic navigation controls (back, forward, refresh) using the `AccessWebViewController` from our `ACCESS Plugin`.

Advanced Web Content Interaction

Beyond just displaying web pages, the `ACCESS Plugin` (via its `AccessWebViewController`) allows for powerful two-way communication between your Flutter application and the embedded web content. This opens up possibilities for executing JavaScript from Dart and receiving data or events from JavaScript back into Dart.

Let's modify our `WebViewScreen` to demonstrate these advanced interactions. We'll load a simple HTML string directly and then show how to interact with it.

Step 1: Prepare HTML with JavaScript Interaction

First, let's define a simple HTML string that includes a button. When this button is clicked, it will send a message back to our Flutter app. We'll also have a function that Flutter can call.

```
// lib/main.dart (inside _WebViewScreenState, replace loadRequest)
// ...
import 'package:flutter/material.dart';
import 'package:web_content_app/access_webview.dart'; // Import our ACCESS
Plugin
import 'package:webview_flutter/webview_flutter.dart'; // Import for
JavaScriptMode enum

// ... MyApp and WebViewScreen ...

class _WebViewScreenState extends State<WebViewScreen> {
  late final AccessWebViewController _controller;
  var loadingPercentage = 0;
  String _messageFromWeb = "No message yet."; // New state variable for web
messages

  @override
  void initState() {
    super.initState();
    _controller = AccessWebViewController()
      ..setJavaScriptMode(JavaScriptMode.unrestricted)
      ..setBackgroundColor(const Color(0x00000000))
      ..setNavigationDelegate(
        AccessNavigationDelegate(
          onProgress: (int progress) {
            if (mounted) {
              setState(() {
                loadingPercentage = progress;
```

```
              });
            }
          },
          onPageStarted: (String url) {
            if (mounted) {
              setState(() {
                loadingPercentage = 0;
              });
            }
            debugPrint('Page started loading: $url');
          },
          onPageFinished: (String url) {
            if (mounted) {
              setState(() {
                loadingPercentage = 100;
              });
            }
            debugPrint('Page finished loading: $url');
          },
          onWebResourceError: (WebResourceError error) {
            if (mounted) {
              setState(() {
                loadingPercentage = 100;
              });
            }
            debugPrint('''
Page resource error:
  code: ${error.errorCode}
  description: ${error.description}
  errorType: ${error.errorType}
  isForMainFrame: ${error.isForMainFrame}
          ''');
          },
          onNavigationRequest: (AccessNavigationRequest request) {
            if (request.url.startsWith('https://www.youtube.com/')) {
              debugPrint('blocking navigation to ${request.url}');
              return AccessNavigationDecision.prevent;
            }
            debugPrint('allowing navigation to ${request.url}');
            return AccessNavigationDecision.navigate;
          },
        ),
      )
      // Add JavaScript channels here
      ..addJavaScriptChannel(
        'FlutterChannel', // The name of the channel
        onMessageReceived: (AccessJavaScriptMessage message) {
          // Callback when a message is received from JavaScript
          debugPrint('Message from web: ${message.message}');
          if (mounted) {
            setState(() {
              _messageFromWeb = message.message;
            });
          }
          ScaffoldMessenger.of(context).showSnackBar(
            SnackBar(content: Text('Received: ${message.message}')),
          );
        },
      )
      // Load a local HTML string instead of an external URL
      ..loadHtmlString('''
        <!DOCTYPE html>
```

```
        <html>
        <head>
          <meta name="viewport" content="width=device-width, initial-
scale=1.0">
          <title>Interactive Web Content</title>
          <style>
            body { font-family: sans-serif; text-align: center; margin-top:
50px; }
            button { padding: 15px 30px; font-size: 18px; cursor: pointer;
margin: 10px; }
            #status { margin-top: 20px; font-size: 16px; color: green; }
          </style>
        </head>
        <body>
          <h1>Hello from Web!</h1>
          <button onclick="sendMessageToFlutter()">Send Message to Flutter</
button>
          <button onclick="updateWebStatus()">Call JS Function from Flutter
(via button)</button>
          <p id="status">Waiting for Flutter...</p>

          <script>
            function sendMessageToFlutter() {
              // Check if the FlutterChannel is available
              if (window.FlutterChannel) {
                window.FlutterChannel.postMessage('Hello Flutter from
JavaScript!');
                document.getElementById('status').innerText = 'Message sent to
Flutter!';
              } else {
                alert('FlutterChannel not found!');
              }
            }

            // This function will be called by Flutter
            function updateWebStatus() {
              document.getElementById('status').innerText = 'Status updated by
Flutter!';
            }
          </script>
        </body>
        </html>
      ''');
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Advanced ACCESS WebView Interaction'),
        actions: [
          IconButton(
            icon: const Icon(Icons.arrow_back),
            onPressed: () async {
              if (await _controller.canGoBack()) {
                _controller.goBack();
              }
            },
          ),
          IconButton(
            icon: const Icon(Icons.arrow_forward),
            onPressed: () async {
```

```
              if (await _controller.canGoForward()) {
                _controller.goForward();
              }
            },
          ),
          IconButton(
            icon: const Icon(Icons.refresh),
            onPressed: () {
              _controller.reload();
            },
          ),
          IconButton(
            icon: const Icon(Icons.call_made),
            onPressed: () async {
              // Execute JavaScript function from Flutter
              await _controller.runJavaScript('updateWebStatus()');
              ScaffoldMessenger.of(context).showSnackBar(
                const SnackBar(content: Text('Called JS function:
updateWebStatus()')),
              );
            },
            tooltip: 'Call JS Function',
          ),
        ],
      ),
      body: Stack(
        children: [
          AccessWebViewWidget(controller: _controller), // Use
AccessWebViewWidget
          if (loadingPercentage < 100)
            LinearProgressIndicator(
              value: loadingPercentage / 100.0,
              color: Colors.blueAccent,
              backgroundColor: Colors.grey[200],
            ),
          // Display message received from web
          Positioned(
            bottom: 0,
            left: 0,
            right: 0,
            child: Container(
              padding: const EdgeInsets.all(8.0),
              color: Colors.black54,
              child: Text(
                'Last message from web: $_messageFromWeb',
                style: const TextStyle(color: Colors.white),
                textAlign: TextAlign.center,
              ),
            ),
          ),
        ],
      ),
    );
  }
}
```

Code Explanation and New Features

1. `_messageFromWeb`: A new state variable to store messages received from the web view.

2. **`loadHtmlString(...)`**: Instead of `loadRequest`, we now use `AccessWebViewController`'s `loadHtmlString` to embed HTML directly. This is useful for displaying dynamic content generated within your app or simple interactive UIs without a server.
3. **JavaScript Channels (`addJavaScriptChannel`)**:
    - **`_controller.addJavaScriptChannel('FlutterChannel', ...)`**: This is the core of Flutter-to-web communication using our `ACCESS Plugin`. We register a JavaScript channel named `'FlutterChannel'`.
    - **`onMessageReceived`**: This callback is triggered whenever JavaScript code in the web view calls `window.FlutterChannel.postMessage(message)`. The `message` argument is wrapped in our `AccessJavaScriptMessage` model.
    - Inside `onMessageReceived`, we update our `_messageFromWeb` state and show a `SnackBar` to visually confirm receipt.
4. **HTML `script`**:
    - **`sendMessageToFlutter()`**: This JavaScript function is called when the "Send Message to Flutter" button is clicked. It uses `window.FlutterChannel.postMessage('...')` to send a string message to our Flutter app.
    - **`updateWebStatus()`**: This JavaScript function simply updates a `<p>` tag's text. This function will be called by Flutter.
5. **Executing JavaScript from Flutter (`runJavaScript`)**:
    - We added a new `IconButton` to the `AppBar` with the `Icons.call_made` icon.
    - **`_controller.runJavaScript('updateWebStatus()')`**: When this button is pressed, Flutter executes the `updateWebStatus()` JavaScript function directly within the web view using our `AccessWebViewController`.

> ⚡ **Note:** When using `addJavaScriptChannel`, the name you provide (e.g., `'FlutterChannel'`) becomes a global object in the web view's JavaScript context (`window.FlutterChannel`). Ensure this name is unique and descriptive.

Run and Verify

Save your `lib/main.dart` file. Run your application:

```
flutter run
```

Observe the following:

1. The app will load the simple HTML content.
2. Click the "Send Message to Flutter" button in the web view. You should see a `SnackBar` appear at the bottom of your Flutter app, confirming the message was received, and the "Last message from web:" text will update.
3. Click the new "Call JS Function" button (the one with the arrow icon) in the `AppBar`. You should see the text "Status updated by Flutter!" appear in the web view, indicating that Flutter successfully executed the JavaScript function.

What We Accomplished

You've mastered advanced interaction techniques using our `ACCESS Plugin`, enabling two-way communication between your Flutter app and embedded web content by using JavaScript channels to send messages from web to Flutter, and `runJavaScript` to execute code from Flutter to the web.

Practical Examples and Use Cases

Now that you understand the mechanics of embedding and interacting with web content using our `ACCESS Plugin`, let's explore some practical scenarios where it truly shines. These examples will illustrate how you might use this functionality in real-world applications.

1. Embedding a Local HTML File

Sometimes, you might want to display static or dynamic HTML content that is bundled with your application, rather than fetched from a remote server. This is perfect for help pages, rich text documents, or custom interactive UIs that don't require internet access.

How to do it:

1. **Add HTML to Assets:** Create an `assets` folder in your project root and place your HTML file inside it (e.g., `assets/local_page.html`).

```
    <!-- assets/local_page.html -->
    <!DOCTYPE html>
    <html>
    <head>
```

```
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
      <title>Local Content</title>
      <style>
        body { font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif; t
ext-align: center; margin: 20px; background-color: #f0f8ff; color: #333; }
        h1 { color: #007bff; }
        p { line-height: 1.6; }
      </style>
    </head>
    <body>
      <h1>Welcome to Local Content!</h1>
      <p>This page is loaded directly from your Flutter app's assets.</p>
      <p>It demonstrates embedding local HTML files for offline content or
custom UIs.</p>
      <img src="https://flutter.dev/assets/images/flutter-logo.png" alt="Flutt
er Logo" width="100">
    </body>
    </html>
```

1. **Declare Assets in `pubspec.yaml`:**

```
    # pubspec.yaml
    flutter:
      uses-material-design: true
      assets:
        - assets/local_page.html # Declare your asset here
```

1. **Load Asset in `AccessWebViewController`:**

```
    // In _WebViewScreenState's initState method
    // ...
    _controller = AccessWebViewController()
      // ... other configurations
      ..setNavigationDelegate(
        AccessNavigationDelegate(
          // ... callbacks
        ),
      )
      ..loadFlutterAsset('assets/local_page.html'); // Use
AccessWebViewController's loadFlutterAsset
```

```
Alternatively, you can read the asset as a string and use `loadHtmlString`:
```

```
    import 'package:flutter/services.dart' show rootBundle; // Add this import

    // In _WebViewScreenState's initState method
    // ...
    rootBundle.loadString('assets/local_page.html').then((htmlContent) {
      _controller.loadHtmlString(htmlContent); // Use
AccessWebViewController's loadHtmlString
```

```
    });
```

2. Implementing OAuth or Third-Party Login Flows

Many services use OAuth 2.0 or similar protocols for authentication. These often involve redirecting the user to a web page (e.g., Google, Facebook, or your own identity provider) to log in and grant permissions. An `AccessWebViewWidget` is the perfect tool for this.

How to do it:

1. **Start the Flow:** Load the authentication URL in your `AccessWebViewWidget`.

```
    _controller.loadRequest(Uri.parse('https://accounts.google.com/o/oauth2/v2/auth?...')); // Your OAuth URL
```

1. **Intercept Redirects:** Use `setNavigationDelegate` to monitor `onNavigationRequest`.
    - When the user successfully authenticates, the web view will try to redirect to a pre-configured callback URL (e.g., `myapp://callback?code=...`).
    - Your `onNavigationRequest` handler can detect this callback URL:

```
        onNavigationRequest: (AccessNavigationRequest request) { // Use AccessNavigationRequest
          if (request.url.startsWith('myapp://callback')) {
            // Extract the authorization code or token from request.url
            // Close the WebView or navigate away
            Navigator.pop(context); // Example: close the WebView screen
            debugPrint('OAuth callback URL: ${request.url}');
            return AccessNavigationDecision.prevent; // Prevent the WebView from navigating further
          }
          return AccessNavigationDecision.navigate;
        }
```

1. **Process Tokens:** Once you've intercepted the callback URL and extracted the necessary parameters (like an authorization code), you can then exchange it for access tokens with your backend or the service provider.

3. Displaying Dynamic Content from a CMS or Web App

Imagine you have a blog, documentation, or product catalog managed by a web-based Content Management System (CMS) or a dedicated web application. Instead of rebuilding all that UI in Flutter, you can embed the relevant web pages using our `ACCESS Plugin`.

How to do it:

1. **Load Specific Paths:** Your Flutter app can construct URLs to specific content pages based on user interaction.

```
    // Example: Navigate to a specific article based on an ID
    void _showArticle(String articleId) {
      _controller.loadRequest(Uri.parse('https://yourcms.com/articles/$article
Id'));
    }
```

1. **Pass Data to Web:** If the web content needs data from your Flutter app (e.g., user preferences, current theme), you can pass it via URL parameters or by executing JavaScript using `AccessWebViewController.runJavaScript`.

```
    // Example: Set a user ID in the web view
    _controller.runJavaScript('setUserId("${currentUser.id}")');
```

1. **Receive Events from Web:** If the web content has interactive elements (e.g., "Add to Cart" button), it can send messages back to Flutter using JavaScript channels defined with `AccessWebViewController.addJavaScriptChannel`.

```
    // In web content
    function addToCart(productId) {
      if (window.FlutterChannel) {
        window.FlutterChannel.postMessage('add_to_cart:' + productId);
      }
    }
```

```
    // In Flutter
    _controller.addJavaScriptChannel('FlutterChannel',
      onMessageReceived: (AccessJavaScriptMessage message) { // Use
AccessJavaScriptMessage
        if (message.message.startsWith('add_to_cart:')) {
          final productId = message.message.split(':')[1];
```

```
          // Handle add to cart in Flutter
          debugPrint('Product $productId added to cart!');
        }
      },
    );
```

Real-World Insight

Many popular apps, including social media platforms and banking applications, use embedded web views for specific features. They often use native UI for core navigation and performance-critical sections, but switch to web views for less frequently accessed features, legal documents, or third-party integrations. This allows them to maintain a single codebase for those web-specific features across different platforms (web, Android, iOS) while still delivering a cohesive user experience within the native app.

What We Accomplished

You've explored practical scenarios for integrating web content using our `ACCESS Plugin`, including embedding local assets, handling authentication flows, and displaying dynamic content from external sources, understanding how these techniques translate into real-world applications.

Troubleshooting Common Issues

Even with a robust plugin like our `ACCESS Plugin` (which uses `webview_flutter` internally), you might encounter issues. Being a patient instructor means guiding you through common pitfalls. Here are some typical problems and how to approach them.

1. Web Page Not Loading or Showing a Blank Screen

- **Cause:** This is often due to network issues, incorrect URL, or missing permissions.

- **Solution:**
    - **Check URL:** Double-check the URL you're trying to load. Ensure it's correct and accessible from your device's browser.
    - **Internet Permissions:** For Android, ensure you have `<uses-permission android:name="android.permission.INTERNET"/>` in your `AndroidManifest.xml`. For iOS, internet access is usually granted by default.
    - **HTTP vs. HTTPS (Android):** If you're trying to load an `http://` (non-secure) URL on Android 9 (API level 28) or higher, you might need to add `android:usesCleartextTraffic="true"` to your `<application>` tag in `AndroidManifest.xml` (as shown in the installation section).
    - **`onWebResourceError`:** Use the `onWebResourceError` callback in `AccessNavigationDelegate` to get detailed error information. Print the `errorCode` and `description` to debug network or resource loading failures.
    - **Device Network:** Ensure your device (emulator or physical) has an active internet connection.

2. JavaScript Functionality Not Working

- **Cause:** JavaScript is disabled by default or being blocked.
- **Solution:**
    - **Enable JavaScript:** Make sure you've called `_controller.setJavaScriptMode(JavaScriptMode.unrestricted)` in your `initState` on your `AccessWebViewController`. This is the most common reason.
    - **Web Console Errors:** If you're loading a remote page, try opening it in a desktop browser and checking the developer console for JavaScript errors. These errors might indicate problems with the web content itself.
    - **`onWebResourceError`:** Again, this callback can sometimes catch JavaScript-related resource loading errors.

3. `pubspec.yaml` Dependency Issues

- **Cause:** Incorrect `webview_flutter` version, syntax errors in `pubspec.yaml`, or `flutter pub get` not run.

- **Solution:**
    - **Correct Indentation:** `pubspec.yaml` is sensitive to indentation. Ensure `webview_flutter:` is indented correctly under `dependencies:`.
    - **Run `flutter pub get`:** Always run `flutter pub get` after modifying `pubspec.yaml`.
    - **Version Conflicts:** If you're using many packages, sometimes version conflicts can arise. `flutter pub get` usually suggests solutions. If not, try `flutter pub upgrade` or explicitly defining a compatible version range.

4. Communication Between Flutter and Web Not Working

- **Cause:** Mismatched channel names, incorrect JavaScript syntax, or channels not registered.
- **Solution:**
    - **Channel Name Match:** Ensure the string name used in `_controller.addJavaScriptChannel('YourChannelName', ...)` exactly matches the name used in JavaScript (`window.YourChannelName.postMessage(...)`).
    - **JavaScript `window` Object:** Make sure your JavaScript code correctly accesses the channel via the `window` object (e.g., `window.FlutterChannel.postMessage`).
    - **Channel Availability Check:** In your JavaScript, it's good practice to check if the channel exists before calling `postMessage`: `if (window.FlutterChannel) { window.FlutterChannel.postMessage('...'); }`.
    - **`runJavaScript` Syntax:** When calling `_controller.runJavaScript()`, ensure the JavaScript string is valid and the function/variable you're trying to access exists in the web view's context. Debug by trying a simple `alert('Hello from Flutter!')` first.

5. Performance Issues or High Memory Usage

- **Cause:** Web views can be resource-intensive, especially with complex web content.

- **Solution:**
    - **Optimize Web Content:** If you control the web content, optimize it for mobile. Reduce large images, complex animations, and unnecessary JavaScript.
    - **Dispose Controller:** While `AccessWebViewWidget` handles much of this, ensure your `AccessWebViewController` isn't leaking resources if you're managing multiple web views or complex navigation.
    - **Consider Alternatives:** For very simple static content, consider rendering it directly with Flutter widgets if possible, rather than a web view.

6. Hot Reload/Restart Issues

- **Cause:** Sometimes, changes to native plugins or complex state in `webview_flutter` (which our `ACCESS Plugin` uses) might not be fully reflected by hot reload.
- **Solution:**
    - **Full Restart:** If you make changes related to the `AccessWebViewController`'s initialization or platform-specific settings, a full `flutter run` or hot restart (Shift + R in the console) is often necessary.

> ⚠️ **Common mistake:** Ignoring the debug output. The `debugPrint` statements in `onProgress`, `onPageStarted`, `onPageFinished`, and especially `onWebResourceError` in your `AccessNavigationDelegate` are invaluable for understanding what's happening inside the web view. Always check your console output!

What We Accomplished

You've gained insight into common problems encountered when embedding web content with our `ACCESS Plugin` and learned effective strategies for diagnosing and resolving them, empowering you to debug your applications more efficiently.

Conclusion and Next Steps

Congratulations! You've successfully navigated the process of embedding web content into your Flutter application using our simulated `ACCESS Plugin`. You've gone from basic setup and creating the plugin's API to advanced interaction, learning how to:

- **Set up your project** and add the necessary internal `webview_flutter` dependency.
- **Define and implement the `ACCESS Plugin`'s API** in `lib/access_webview.dart`.
- **Embed a basic web view** using `AccessWebViewWidget` and `AccessWebViewController`, loading external URLs.
- **Configure web view options**, including enabling JavaScript and controlling navigation with `AccessNavigationDelegate`.
- **Implement advanced two-way communication** between your Flutter app and web content using `AccessWebViewController`'s JavaScript channels and `runJavaScript` method.
- **Explored practical use cases** for embedding local HTML, handling authentication, and displaying dynamic content.
- **Learned to troubleshoot** common issues that might arise during development.

This knowledge empowers you to build richer, more versatile Flutter applications that seamlessly integrate the best of both native and web worlds, even if you were tasked with building a "new open-source plugin" to achieve it!

What to Build Next

To solidify your understanding and explore further, here are three concrete ideas to extend your `web_content_app` project, using the `ACCESS Plugin` you've created:

1. **Build a Simple In-App Browser:** Enhance your current app with a URL input field and a "Go" button. Allow users to type any URL and load it in the `AccessWebViewWidget`. Add a progress bar, title display in the `AppBar` (using `_controller.currentUrl()` to get the URL and parse the title if possible), and robust error handling for invalid URLs. This will give you more practice with `TextFormField`, `setState`, and `AccessWebViewController` methods.

2. **Create a Hybrid Content Viewer with Local and Remote Content:**
    - Design a Flutter UI with a list of "articles." Some articles could be local HTML files (loaded via `_controller.loadFlutterAsset`), while others could be links to external blog posts (loaded via `_controller.loadRequest`).
    - When a user taps an article, open a new screen with the `AccessWebViewWidget` to display the content.
    - Implement a "Share" button in the `AppBar` that uses `_controller.currentUrl()` to get the current URL and then uses Flutter's `share_plus` package (add it to `pubspec.yaml`) to share the link.
3. **Implement a Secure Login Flow (Mock OAuth):**
    - Create a mock login page in HTML that, upon successful "login," redirects to a specific `myapp://success?token=some_jwt` URL.
    - Modify your `AccessNavigationDelegate` to intercept this `myapp://` URL.
    - Extract the `token` from the URL, store it securely (e.g., using `shared_preferences` in Flutter), and then close the `AccessWebViewWidget` screen, displaying a "Welcome, User!" message in your main Flutter app. This simulates a common authentication pattern.

Keep experimenting, and happy Fluttering with your new `ACCESS Plugin`!