Web Runtime

Contents

Definitions

- **web runtime** a set of rules, supporting libraries, App Framework and SDK integration to allow development and integration of web and hybrid apps targeting the Apertis system
- **web app** an application using web technologies such as HTML, CSS and JavaScript, integrated on the system through the web runtime
- **native app** an application targeting the Apertis system using the Apertis UI toolkit for its user interface, developed in either C or one of the other supported languages such as JavaScript
- **hybrid app** a *native app* that includes a web page as a piece of its user interface; it may use the web runtime infrastructure but not necessarily does

Overview

The web runtime will be provided through a library that mediates the interaction between the WebView and the system, and a launcher. There will be two ways of using the web runtime: handing over the whole task to it by using the shared launcher or by picking and choosing from the library APIs. The simple, hand-over mode will be the one adopted by pure web apps (i.e. not hybrid ones) which do not need a custom launcher. It will use the shared launcher executable to start the web app. The web runtime (library and shared launcher) will be a part of the system image, alongside other frameworks such as the App Manager. The web runtime will handle the window and manage any additional WebViews the application ends up using, providing appropriate stacking.

For hybrid apps, a custom launcher will be necessary, since more than just the web runtime is required. In that case, the web runtime library can still be useful to provide integration with the JavaScript bindings for the system APIs or to help manage additional views. The hybrid app's author will need to decide which functions to delegate and which to build based on the requirements, the web runtime library needs to be flexible to cover these use cases.

An important note about the *web app* concept is in order to avoid confusion. Some platforms like Gnome and iOS support a way of turning regular web sites into what they call a web app: an icon for launching a separate instance of the browser that loads that web site as if it were a regular app. This form of web app does not provide any form of special interaction with system APIs and is not the kind of app discussed in this document.

Anatomy of a web app

Web apps will be very much like regular ones. Their bundles will contain any HTML, image, CSS and JS files they require for running, but they may also use remote resources if it makes sense.

The web runtime will ensure any caching and data storage such as the databases for cookies, HTTP cache, and HTML storage APIs will live under the application's storage area. More specifically, data such as those of cookies and HTML storage APIs will live under the *Application User* type while HTTP cache will live under the *Cache* type, see the *Data Storage* section of the Applications design document.

User interface

There will be access to the native UI toolkit through the JavaScript API, but any interface created using it cannot be mixed into the web page. Being able to access those APIs may be useful for using standard dialogs, for instance. Unless the developer is writing a hybrid app, and thus rolling their own launcher, any native UI toolkit usage will only be possible on a separate window.

In the future a web framework may be provided to help create interfaces similar to the ones available natively. For now, the app developer is fully responsible for the look & feel of the web app.

Integration with Apertis App Framework

As with native apps, most interactions with the App Framework are to be done by the apps themselves, but there may be some cases where the web runtime itself needs to handle or mediate that integration. This section lists and addresses those cases.

App Manager (Canterbury)

Canterbury is a service that handles installation and starting applications, along with centralizing several App Framework features (some of which may end up being split in redesigns, though). The current proof of concept implementation of a web runtime includes handling of app state and hardkeys.

The app state handling currently does nothing in web runtime mode and is likely not relevant for the web runtime - the web app can connect to the service and do its own handling should it decide it is important.

Hardkey integration (Canterbury / Compositor)

Hardware keys handling is currently in a state of flux and is likely to move from Caterbury to the compositor, but the basic mechanism to register for notification is probably going to remain the same.

In the current proof of concept runtime, the hardkey handling connects to the *back* feature of the view, similar to the back button in regular browsers. While that may make sense for apps that use a single view and rely on navigation history alone, it may make more sense to let the app decide what to do and default to closing the top-most view if the app does not handle it. The same goes for other hardware keys.

To achieve that and avoid the problem of having to register twice for getting the button press signals, the runtime will be the sole responsible for registering with the relevant service for this. It will then relay hardware key presses to JavaScript callbacks provided by the web app. In the event the web app does not provide a callback or does not handle it (i.e. returns false), the runtime may adopt a default behaviour, such as closing the top-most view for a *back* press, or ignore it.

Inter-App integration (Didcot)

Applications can tell the system to launch other applications by using the content hand-over service (Didcot)[1]. The API used to communicate with the service will be available to the JavaScript context through the bindings provided by *seed*, but regular links in web apps can use the same schemes used by the content hand-over mechanism. The WebView provided by the web runtime library will automatically handle navigation to those schemes and call the content hand-over API.

Security

Web apps security should not be that different from native apps. The same firewall rules may apply, as well as resource limits and AppArmor rules. While

[1] https://sjoerd.pages.apertis.org/apertis-website/concepts/content_hand-over/

native and hybrid apps can easily be contained by their own AppArmor profile, pure web apps will use a shared binary and thus need a different solution.

The AppArmor API provides a *change_profile* function that allows specifying a different profile to switch to. Web apps will follow the convention of prefixing their profile names with *web_*, so that the launcher can be allowed to switch exclusively to web app profiles by means of a wildcard permission. The name of the profile will be the *web_* prefix plus the reverse domain-name notation used in the app's manifest and install path.

As described in the overview, being an web app does not necessarily mean that its data comes from remote web servers. Most apps should actually ship with local HTML, CSS and JS files for rendering and controlling their UIs, only using remote data like any native app would.

Notice however that this is not a rule, web apps may also come in the form of simple shells for content downloaded from a remote web server, in which case it is probably advisable that the system API made available to the app be limited. For this reason, the JavaScript bindings will make it possible to use a blacklist or a whitelist to describe which modules are made or not available to a given app.

It is advisable that no bindings are made available at all for remote content not directly controlled by the web app publisher. This needs to be screened for in the app store approval process.

Potential API

WebRuntimeApplication (GApplication subclass)

Manages WebViews, when the main WebView is closed it will quit the application. Connects to the back hardkey, tells web app about it being pressed and handles it if the app does not.

webruntime_init - WebExtension

In WebKit1, a single process was used for the whole browser, thus the WebView had control over the JavaScript context. With WebKit2 that is no longer the case, since the WebView lives in the UI process and everything web is in the Web process, including JS execution. WebKit2 provides a facility called WebExtension. The Web process loads any loadable modules provided by the application and allows them to interact with Web process functionality such as DOM bindings and JS execution.

The Web Runtime will provide a loadable web extension which will enable the following:

- makes GObject bindings available to the page's JavaScript context

- handles requests for opening new windows/views and tells WebRuntimeApplication about them
- handles navigation requests for non-web, non-file schemes and hands them to Didcot

A few JavaScript methods will also be made available by the WebRuntime itself through a globally available ApertisWebRuntime object:

- popView(): will remove the top-most view from the stack, destroying the related WebView
- onBackPress(): the web app should provide this method by adding it to the ApertisWebRuntime object; if it exists on the top-most view, the runtime will call it whenever the back key is pressed, the boolean return value tells the runtime whether the press has been handled
- onFavoritesPress(): like onBackPress, called for the favorites button
- on…Press(): all hardware keys will have equivalent callbacks