Meteor WebApp Cordova plugin

Cordova apps don't load web content over the network, but rely on locally stored HTML, CSS, JavaScript code and other assets.

Plain Cordova uses `file://` URLs to serve assets from the app bundle, but this has some drawbacks:

- We need the ability to reliably switch over to new versions of the app
- file:// triggers some non-standard browser behaviors and bugs
- We want to serve index.html when a file is not found, to better support client-side routing
- We need to respect the URL-path mappings from the manifest

The Meteor Cordova integration therefore uses a plugin to serve assets from the local filesystem and to support hot code push.

The old version of the plugin has some problems:

- It controls the download of assets from JavaScript using the Cordova file transfer plugin, which has performance issues and isn't very reliable
- It redownloads all assets on every update
- It doesn't validate the assets it downloads, often leaving versions in an inconsistent state (that happens if the server updates while we are downloading for instance)
- It has no way to recover from a faulty downloaded version, which means the only way to get the app working again is to remove it from the device
- On iOS, it intercepts URL loading using `NSURLProtocol`, which is not supported on `WKWebView` (and also doesn't support streaming video).

The goals of the plugin rewrite are to improve performance and reliability, to ensure compatibility with `WKWebView`, and to fix some other issues in the process.

The new design has two major parts:

Switching to a real embedded web server (iOS only)

Instead of using a URL loading interception mechanism, we now use an embedded web server. Besides compatibility with `WKWebView`, this allows us to more closely replicate the behavior of `webapp_server.js`. We respect the URL mappings in the asset manifest and set the appropriate headers for caching and source map support for instance.

This should give us very efficient reloading, because assets marked as `cacheable` in the manifest (which get served with a long `max-age`) should not have to be re-requested from the local web server at all. Other files can still use conditional requests and often result in a `304 Not Modified` response because we set the ETag header to the asset hash. However, it turns out browsers (as well as the web view) don't respect `max-age` when `window.location.reload()` is used, and always perform at least a conditional request. The tweak required is to use `window.location.replace(window.location.href)` instead, which seems to have no adverse consequences.

Assigning a port for the local web server

The local web server needs a port to bind to, and we can't simply use a fixed port because that might lead to conflicts when running multiple Meteor Cordova apps on the same device.

Initially, the easiest solution seemed to be to let the OS assign a randomized port in the ephemeral port range (49152–65535). This works, but has a serious drawback: if the port changes each time you run the app, web features that depend on the origin (like caching, localStorage, IndexedDB) won't persist between runs.

So instead we now pick a port from a predetermined range (12000-13000), calculated based on the appId. That ensures the same app will always use the same port, but it hopefully avoids collisions betweens apps as much as possible.

There is still a theoretical possibility of the selected port being in use. Currently, starting the local server will fail in that case.

Moving downloading updates to native code

Coordinating the download of files from JavaScript isn't ideal, because calls over the JavaScript-native bridge are expensive and block the main thread. In addition, the Cordova file transfer plugin for iOS is based on an older networking mechanism (`NSURLConnection`) that has some efficiency issues of itself. As a result, downloads often fail, and the way this is dealt with in the existing code is to try again after a 30 second timeout. Therefore, downloads are very unpredictable and can sometimes take minutes even on a local network.

The new plugin moves downloads to native code. The JavaScript code is only responsible for detecting new versions (by subscribing to `meteor_autoupdate_clientVersions` , the normal autoupdating behavior) and notifying the plugin (using `WebAppLocalServer.checkForUpdates()`). Besides avoiding a series of calls from JavaScript, this gives us more control over downloading mechanism. On iOS, we can now use `NSURLSession` , which allows for more customization and efficiently supports parallel downloads without blocking (it also supports SPDY and HTTP/2, which could make this even more efficient in the future).

Originally, we were hoping to take advantage of `NSURLSession` s support for background file transfers, which allows downloads to continue even if the app is not active. But it turns out these out of process transfers are much slower when downloading small files (in one test, 600ms vs 6000ms), so especially during development that would make updates unnecessarily slow. Instead, we're now creating a background task so downloads get to continue for another 3 minutes after the app goes into the background. That should be enough in most cases, but resuming is also pretty efficient (see below), so it doesn't seem not having real background transfers is a big loss.

The downloading design is based on two important principles:

- We should make sure we always end up with a valid asset bundle, even if downloads take a while or are resumed later
- We should not download or copy files unnecessarily

Storing and serving asset bundles

The initial asset bundle is stored as part of the app bundle, in a read-only location. Downloaded asset bundles are stored in a writeable part of the file system (`Library/NoCloud/meteor` on iOS), with one subdirectory per version. Versions are cleaned up when they are no longer needed, so most of the time there should be only one version, but cleanup only happens after a successful startup (we wait until then so we can revert to a last known good version if startup isn't successful, see below).

The local web server serves files both from the `www` directory in the app bundle (for Cordova files, including files that are part of plugins) and from the `currentAssetBundle` . An `AssetBundle` creates `Asset` s based on the entries in an `AssetManifest` (for entries that also specify a source map, a separate asset is created). It knows which assets correspond to which URL path, and it also knows its `indexFile` .

We never switch the `currentAssetBundle` immediately, because that would mean the app could have loaded some files from the old version and now loads others from the new one. Instead, we keep a `pendingAssetBundle` and switch over when a reload occurs (Cordova plugins get a call to their `onReset` method when this happens). Reloads are under control of the `reload` package as usual, so they respects the result of `Meteor._reload.onMigrate()` callbacks for instance. So especially with `reload-on-resume`, it may be a while before the switch happens.

With a freshly installed app, the `currentAssetBundle` will be initialized to the `initialAssetBundle`. For apps that have already downloaded updates, `lastDownloadedVersion` will be set (as a persistent configuration value, `NSUserDefaults` on iOS) and we use that to locate the appropriate downloaded asset bundle and make it current instead.

Downloading updates

- When checking for updates, we first download the asset manifest. Then:
    - If we're already serving that version or if it is pending, do nothing (this could happen if we invoke `checkForUpdates()` manually, or on startup)
    - If we have an existing downloaded version, use that (this could only happen if we revert to a version that we downloaded before but aren't currently serving or have set as pending)
    - Else, we create a new asset bundle from the asset manifest and point it to a fresh `Downloading` directory. Then:
        - For every asset in the bundle, based on the URL path and hash, we try to find an existing file:
            - For the initial bundle, which as part of the app bundle is read-only, we just use the existing file path (an alternative would be to copy it, but we'd like to avoid that).
            - Even though downloaded bundles are writeable, we can't move files because we are likely still serving from that version. And we'd like to avoid copying because of the performance implications, so hard links offer a really nice solution here. Hard links have the advantage that they make the file system responsible for keeping track of what files are in use by what version. When we remove a version directory, all files that are not in use by other versions will be removed automatically, but the ones that are shared won't.
    - Before the above, if there is an existing `Downloading` directory, we rename it to `PartialDownload` and take the assets already downloaded into account when trying to find existing files (again, based on the URL path and the hash).
    - We then download just the missing assets, checking every asset against the hash in the manifest to make sure we downloaded the right version (see below)
    - Once we have a complete new version, we rename `Downloading` to a directory name based on its version, we make it the `pendingAssetBundle`, and we invoke the `onNewVersionReady()` callback (this callback is normally handled as part of the `autoupdating` package and calls into the `reload` package).

Resuming downloads follows naturally from this model. We always download the asset manifest first (because there may be a newer version available since the last time we started the app), and use that to decide what files we can reuse and which ones we still need to download. So even if the version in `PartialDownload` is different from the one we are `Downloading` now, we never download files twice. We remove `PartialDownload` after initializing the downloading asset bundle.

Because the app on the server may be updated at any time, it is not guaranteed the files we download are all from the same version. Especially on slower mobile connections, there may be a delay between individual file downloads. To check whether the file we receive is the one we expected, we check against the hash in the manifest. One option would be to calculate the SHA1 of the file locally, but this could slow updates down, especially with larger files and older devices. What we've done now is to make the server set the ETag header to the hash, so we can simply compare the header value in the response against the manifest (it only does this if the ETag value matches the format of a SHA1 hash).

When the app is updated on the App Store (which we detect by checking `lastSeenInitialVersion`), we remove the entire versions directory and clear `lastDownloadedVersion` and `lastKnownGoodVersion`. This ensures we will use the new initial asset bundle and avoids problems with downloaded versions (because these may depend on files in the old initial asset bundle).

Recovering from faulty versions

As mentioned, a major problem of the old plugin is that faulty versions may require a reinstall of the app.

To counteract this, we invoke `WebAppLocalServer.startupDidComplete()` after a successful startup. Currently, we only invoke this after all `Meteor.startup()` callbacks have been invoked, which seems like a safe point.

If `startupDidComplete()` is not invoked within a certain time period, we consider the version faulty and will rollback to an earlier version. Alternatively, we may be able to detect faulty versions faster by hooking into `window.onError`, but I'm not sure if that won't generate false alarms.

We roll back to the `initialAssetBundle`, or `lastKnownGoodVersion` if it has been set. Receiving `startupDidComplete()` sets `lastKnownGoodVersion` and also cleans up the downloaded asset bundles.

A problem with this recovery mechanism is that unless the server has been updated, the server will hot code push the faulty version again. Therefore we blacklist faulty versions on the device so we know not to retry.