Cordova integration

The Cordova integration allows running, building and testing of Cordova projects from the CLI.

It hooks into the following commands:

- `cli/commands.js`: `meteor run/test-packages/build`
- `cli/commands-cordova.js`: `meteor add-platform/remove-platform/list-platforms`
- `cli/commands-packages.js`: `meteor add/remove cordova:<plugin>`

These commands call into functionality provided by the classes under `cordova/` (`CordovaProject`, `CordovaBuilder`, `CordovaRunner`, and `CordovaRunTarget`), as well as some utility functions in `cordova/index.js` (generally imported under the `cordova` namespace).

Project

`CordovaProject` (in `project.js`) represents the generated Cordova project (in the `.meteor/local/cordova-build` directory), and is the only class with a direct dependency on `cordova-lib`. Its methods allow you to call into the prepare, build and run stages, and manage platforms and plugins.

In this version, we've switched from relying on Cordova CLI to using `cordova-lib` directly. This avoids multiple levels of script invocations, which leads to difficult to diagnose failures and makes it harder to support Windows.

Old situation: Meteor CLI → Meteor Cordova wrapper scripts → Cordova CLI → Cordova platform-specific scripts

New situation: Meteor CLI + `cordova-lib` (in-process) → Cordova platform-specific scripts

Commands in `cordova-lib` are invoked asynchronously (they return promises) but for the most part we wait with `Promise.await`. This is abstracted away in `runCommands`. There is some other code in `CordovaProject` that helps with setting the `cwd` and `env` and catching `CordovaError`s.

Stages

- `CordovaProject#createIfNeeded()`

Invoked automatically from the constructor. Makes sure the project has been created if the `cordova-build` directory does not currently exist.

- `CordovaProject#prepareFromAppBundle(bundlePath, pluginVersions, options = {})`

Uses a builder (see `CordovaBuilder`) to generate the `www` directory, resources, and `config.xml` based on the app bundle result and `mobile-config.js`.

- `CordovaProject#prepareForPlatform(platform)`

Similar to `cordova prepare <platform>`. Synchronizes the project contents (`www` directory and resources) with a platform directory and installs or updates plugins in a platform-specific manner (modifying an Xcode project for instance).

- `CordovaProject#buildForPlatform(platform, options = {}, extraPaths)`

Similar to `cordova build <platform>`. Uses platform-specific build mechanisms to compile app. Includes everything done in the prepare stage. It is used for `meteor build`.

- `async CordovaProject#run(platform, isDevice, options = [], extraPaths)`

Similar to `cordova run <target>`, except that it doesn't include prepare, so you'll have to make sure `CordovaProject#prepareForPlatform()` is called before. Uses platform-specific mechanisms to run the built app on a device or emulator/simulator. It is used for `meteor run/test-packages`.

Managing platforms

- `CordovaProject#ensurePlatformsAreSynchronized(platforms = this.cordovaPlatformsInApp)`

Ensures the platforms installed in the Cordova project are synchronized with the app-level platforms (in `.meteor/platforms`). This gets invoked as part of `meteor add-platform/remove-platform` (from `commands-cordova.js`) and as part of the build process (from `CordovaBuilder`).

Uses methods `CordovaProject#listInstalledPlatforms()`, `CordovaProject#addPlatform(platform)`, and `CordovaProject#removePlatform(platform)` to call into Cordova.

`CordovaProject#checkPlatformRequirements(platform)` uses the Cordova platform-specific tools to get a list of installation requirements, whether they are satisfied, and if not, why. The latter can be very useful and is pretty detailed (telling you for instance that `ANDROID_HOME` has been set to a non-existent directory, or that the right platform tools cannot be found).

We massage the results a little and print them (only if not all requirements for a platform are satisfied) in a list with checkmarks and crosses. This may need some work. We probably want to simplify the results a little and only show the full list in a verbose mode. And we definitely want to add a link to a Wiki page with installation instructions.

This gets invoked from `meteor add-platform` (so we can immediately give feedback about the state of the installation) and before running (from `runner.checkPlatformsForRunTargets()`).

Managing plugins

- `CordovaProject#ensurePluginsAreSynchronized(pluginVersions, pluginsConfiguration = {})`

Ensures the plugins installed in the Cordova project are synchronized with the app-level Cordova plugins. This gets invoked as part of the build process (from `CordovaBuilder`) where it is passed the plugins from the star manifest in the app bundle (`pluginVersionsFromStarManifest`, a combination of `.meteor/cordova-plugins` for stand-alone plugin installs and the plugins added as dependencies of packages through `Cordova.depends`). The `pluginsConfiguration` comes from `App.configurePlugin` calls in `mobile-config.js`.

Uses methods `CordovaProject#listInstalledPluginVersions()`, `CordovaProject#addPlugin(name, version, config)`, `CordovaProject#removePlugins(plugins)` to call into Cordova.

Running

`CordovaRunner` (in `runner.js`) represents running the app on a set of run targets. It holds a reference to a `cordovaProject` and an array of `runTargets`. It is created in `meteor run/test-packages` in `cli/commands.js` and gets passed to `AppRunner`.

This is a change from the previous behavior, where the Cordova build and run had to be completed before we could start the proxy, MongoDB, and bundle and run the server app.

One of the main benefits of this is that we can now use the app bundle generated in `AppRunner`. Previously, we were actually doing a whole separate `bundler.bundle()` for Cordova, even though the one in `AppRunner` also generated the `web.cordova` architecture (which is needed to serve it to clients as part of Hot Code Push). We now invoke `CordovaRunner#prepareProject()` with the existing bundle result right before starting the server app.

Moving building/running of Cordova apps to `AppRunner` should also allow Cordova apps to participate in the reload/fix cycle. (We will need to figure out how to best do this. This may also tie in with detecting changed platforms/plugins, automatically restarting the Cordova app when we add a plugin for instance).

`CordovaRunTarget` (in `run-targets.js`) represents a target to run the app on. It has subclasses `iOSRunTarget` and `AndroidRunTarget` that contain some platform-specific behavior (this will be more when we add log tailing again). Right now, we only differentiate between running on a device or an emulator/simulator. We may want to allow specifying a specific device ID in the future (so we can select between different connected devices for instance).

`CordovaRunner#startRunTargets()` is responsible for starting the app on the associated run targets (while displaying build messages and a run log entry). It currently starts the targets asynchronously (in parallel). This means running the app on multiple targets is a lot faster than before. Because we don't currently have a way of showing multiple progress bars, it may be confusing to see 'Started app on Android emulator` while the build/run is still in progress.

Building

`CordovaBuilder` (in `builder.js`) represents the process of building a Cordova project from an app bundle. It contains state (`metadata`, `additionalConfiguration`, `accessRules`, `imagePaths`, `pluginsConfiguration`) that is built-up through different stages (`initalizeDefaults()`, `processControlFile()`) and can be used to generate and copy the `www` directory, resources, and `config.xml`.

`CordovaBuilder#processControlFile()` is responsible for running the `mobile-config.js` control file (passing itself as a context to `App`).