R3 Prototyping Documentation

Release master

R3 CEV

Jan 13, 2019

Project

Graviton is an app browser and cross-platform software manager.

It gives you:

- A **browser-like shell** that downloads and runs desktop apps, written in languages that can run on the JVM.
- **Silent upgrades** via a regular scheduled background task, like in Chrome. Apps, dependencies, the JVM and Graviton itself are all upgraded regularly, whether or not the user is currently running an app.
- First class support for **cross platform command line apps**. They also smoothly upgrade, even if the app is running at the time (good for servers). Graviton enables colour terminal handling on Windows 10, so ANSI escapes can be used on any platform with confidence.
- **Publish software simply** using GitHub, GitLab or BitBucket releases! The repository is cloned on a remote server in a sandbox, compiled, packaged, downloaded and this process repeats in the background on regular intervals to keep your app up to date. Push new versions to your users by simply making a new GitHub release.
- **Apps can be written in any language** with a JVM backend, such as Java, JavaScript, Python, Scala, Kotlin, Haskell, Ruby, with support for sandboxed C, C++ and Rust coming in future via Sulong.
- **Detection of proxy settings** from the OS or browsers, and can handle proxy auto-config files. Apps automatically benefit.
- Use advanced designed-for-apps GUI toolkits like JavaFX with visual designers, or embed WebKit and use HTML.

Learn more by reading the *Introduction*.

CHAPTER 1

Get involved

The project is alpha software and could benefit from your help.

- Talk to us on the graviton-dev mailing list
- Or on the #graviton-browser channel on the Kotlin Slack.
- Or on our GitHub repository.

Graviton needs you! We're looking for people with:

- Web design skills, to make a nicer site for the project (these pages would remain as developer docs).
- Evangelism skills, to find projects that can run on the JVM and convince them to use Graviton as a distribution mechanism.
- **Developer skills:**
  - Journeyman level, to create a beautiful showcase for both GUI and CLI apps. This would be a good student project.
  - Intermediate level, to improve the start page shell GUI.
  - Expert level, to work on improved app streaming formats, JVM isolation, OpenGL, network discovery, OS integration tasks and more. See the *Roadmap* for ideas.

Commercial version? Would you like a Pro/Enterprise level version of Graviton? Let me know!

1.1 Introduction

We would like a competitor to the web browser for JVM application distribution and deployment. This would be an alternative to applets, the now deprecated Java Web Start and javapackager style bundled distribution.

Initial target markets:

- Internal apps in industrial and enterprise scenarios, e.g. finance and trading applications where web apps are often not preferred for productivity/usability/security and other reasons. Slick updates and a deploy-once runtime are useful here.

- Hobbyist and learner apps where users do not want the overhead of the complex web frontend stack or desktop deployment, but do want to publish their apps. They aren't attempting to maximise the user acquisition funnel so don't mind telling users to download an app browser component. These are people who distribute JARs or only source code today.
- The Java gaming community.
- People who want to write cross platform command line apps.
- IoT devices that struggle to provide secure administration interfaces, given the demands of web browser makers to use web PKI SSL and the poor fit of SSL with embedded web servers that do not have domain names.
- People who want to write apps in languages other than JavaScript, although we plan for JS to be a first class citizen along with all other mainstream languages (via the Truffle project).

We feel these market segments are under-served by web browser developers.

The resulting platform should use the JVM for platform independence, sandboxing, code streaming and other services but should otherwise be language and UI toolkit agnostic to the extent possible.

1.1.1 FAQ

Why the JVM? We don't want to repeat the web's mistake of restricting us to a single language. The JVM has the best support for running multiple languages and having them interoperate. It also has some of the best cross platform user interface frameworks, many excellent libraries and modules, good tools and a relatively clean design for its age. Through Graal and Truffle languages we can combine not only scripting languages like JavaScript, Python and Ruby but also C/C++ modules via Sulong. This gives us a direct equivalent to WebAssembly.

Aren't JVMs bloated and sluggish? Historically yes. We aren't worried about this though for three reasons. (1) Our competition isn't expertly written C++ apps but web apps, which are much worse. (2) The bulk of the JVM's reputation for sluggishness comes from startup time and memory usage, not peak runtime performance. All of these are being tackled by the JVM team through recently added features like ahead of time compilation, AppCDS (class data pre-computation and sharing), GCs that don't pause the application and other such features. (3) The JVM's reputation largely dates from the late 1990's and early 2000's when hardware wasn't as good as it is now. Over time hardware got bigger and JVMs got more efficient. So we aren't worried about this so much anymore.

Do apps need to be written specifically for it? No, Graviton can download and run ordinary Java apps with a main method that have been uploaded to a Maven repository or github. There are many such apps already. But with small adaptations, the user experience will get a lot better. As such there is no bright line between a JVM app and a Graviton app. Learn more about *Graviton apps*.

1.2 Newsletters

1.2.1 30th December 2018

The first release is finally ready! Bugs have been fixed, a download page created and signed binaries uploaded. The next step is to request some testing from people following the project, before announcing it in a few Java related forums after the New Year, when people are returning to work.

Beyond eating mince pies the bulk of the work in the last 20 days has been bug fixes to the Linux and Windows online update engines, filing more ideas in the issue tracker and investigating app compatibility issues.

1.2.2 10th December 2018

Rampdown continues in preparation for the first release. Mac and Windows builds have passed the test plans, and testing has now moved on to Linux (which has already revealed a strange visual bug). Many bugs have been fixed during this test phase. Unfortunately, download cancellation had to be disabled, as it needs more work.

A new mailing list has been created in preparation for launch. Please join the graviton-dev mailing list.

I presented Graviton at JavaFX Days Zürich, which was a great conference. Feedback was overwhelmingly positive, and some firms expressed interest in using it as a replacement for Java Web Start. Collaboration with Karakun on their planned reimplementation of Web Start is a possibility in future. I have re-recorded the talk I gave; it can be viewed here:

1.2.3 27th November 2018

I took my last few days of leave for the year to push Graviton Alpha over the finishing line. Significant work has been done and the first release is nearly here! I will be presenting Graviton at JavaFX Days in Zürich next week.

- The big new feature this week is *Graviton apps*, which allows an app to establish communication with the browser despite running in an isolated classloader. It has JavaDocs and is extensible to support many future features. Currently it allows apps to opt back in to running in the main browser window. You can also opt-in to reusing the Graviton JVM and browser window using a manifest entry, for easier setup.
- The Proxy Vole library has been integrated. It enables automatic usage of proxy settings from the user's operating system settings (cross platform), browser settings (Firefox/IE), and can execute the JavaScript in proxy auto-config files. This should work for both GUI and command line apps.
- The UI has received some more visual polish.
- You can now use GitHub URLs as coordinates. It will use JitPack as normal.
- New installs get a few 'showcase' apps in their recent app pickers, so users have somewhere to start.
- JARs without a Main-Class manifest entry will now be scanned to locate a main method.
- A new *Roadmap* page has been added with dozens of feature ideas for where Graviton can go in future.
- Signing keys have been obtained for Windows and macOS.
- Upgraded to Kotlin 1.3 and TinyLog 2

1.2.4 4th November 2018

Many exciting things have happened since the last update. Most importantly a new contributor has arrived! Welcome to Bernhard Luztmann who has contributed several improvements.

- Startup of the invoked program has been totally reworked. By default the program gets its own JVM process now, which increases app compatibility. However in some cases JavaFX apps are still invoked inline.
- The `Cmd-Q` key, which normally quits an app, now goes back to the shell window for inlined apps.
- Reverse DNS cooordinates work now, again thanks to Bernhard. Try `plan99.net:tictactoe`
- Together we've added scrolling to the history list, which was one of the last items blocking release.
- The look has been refreshed to be cleaner.
- You can now right click on individual app history tiles to force a refresh.
- A new tutorial has been started, showing you how to start with the tiniest GUI app possible and incrementally adapt it for Graviton.

1.2.5 29th September 2018

This week Windows command line tool support was finished. Java apps can now be invoked from the CLI or GUI, with all the necessary workarounds for Windows' various console handling issues. ANSI escape support is activated via the Win32 API for users on Windows 10.

1.2.6 23rd September 2018

Not much to report this week:

- Windows runtime updates work now, and the Windows installer has been improved. This brings us closer to first release.
- Some initial work done on upgrading to Kotlin 1.3, but it's not quite ready to go yet.
- The code was simplified to not use coroutines any longer. It wasn't buying me much.
- New art: evening forest.
- Some rudimentary support for cancelling in-flight downloads.

1.2.7 17th September 2018

Graviton had its first day out last week! I got up on stage and demoed a customised version to the audience at CordaCon, a conference dedicated to the Corda decentralised, peer to peer database system (otherwise known as a a distributed ledger or blockchain).

Corda is fully based on Java/Kotlin, and is thus a great fit for Graviton. Reception was warm, with one of the questions at the end of the talk being "can we have Graviton yesterday?". Equally interesting to me was how many people were struck by my demo of Scene Builder: quite a few came up and asked me what that GUI designer was and where they could get it.

To celebrate being introduced to the world, Graviton got a facelift:

Changes since last time:

- App history is now used to create a basic history list. Still lots of work to do here.
- Some basic support for customising the brand logo and name.
- Better support for jitpack.io
- Refactorings and bug fixes to keep the code clean.
- Added a (currently disabled) login screen, it's a work in progress and I might delete it.
- Possible to use videos now for the backdrop as well as pictures.

1.2.8 3rd September 2018

The summer may be drawing to a close, but it's not all bad: Graviton development has returned! Changes this week:

- History infrastructure upgraded to store name and description from the POM files, ready for rendering.
- Support for building and running on both Java 8 and 10 at once.
- Progress bar tracking for downloads, better command line progress bar.
- Select the highest version of a module in a dependency graph instead of Maven's "nearest wins" heuristic, which was breaking some complex apps.
- Misc refactorings and improvements.

1.2.9 29th May 2018

Progress report

- Apps are updated in the background every 6 hours if the user started them without a version specifier in the coordinate.
- Backported to Java 8. Too many things still break with Java 10, but there'll be another attempt in future with some extra logic added to increase app compatibility.
- Conscrypt is now used by default, it eliminates the overhead of using SSL entirely.
- Silent runtime updates are now fully working and tested on macOS. Free disk space is checked and updates are applied atomically, with signature checking to detect maliciously crafted updates.
- A logo has been selected. It may change in future but it'll do for now.
- A new download animation has been created.
- On macOS the app menu now has an about box and a clear cache option.

1.2.10 20th May 2018

Progress report

After a short break spent on other tasks and video games, Graviton development returns! This week work focused on the Chrome-style runtime auto update mechanism. Many of the pieces of this critical component have been laid previously, and now the final piece is landing: download and activation.

- A new domain name has been acquired: graviton.app. For now it just redirects to the docsite.
- A simple update protocol has been defined and implemented. It is described in browser-update. It still needs to be adapted for Windows, but the bulk of the code is platform independent.
- New background art has been added to the shell, a vector art of Paris.
- Some more future feature ideas have been filed in github.

More work remains on the update framework: free disk space testing, Windows support, making updates fully atomic, checking for download corruption and so on. These small things will come in the next batch of work.

1.2.11 16th April 2018

Progress report

This week continued to fill out the current features:

- JavaFX apps are now invoked directly via instantiating their `Application` class, which lets them take over the main stage. Try `net.plan99:tictactoe` for an example.
- A logging framework has been integrated. Logs rotate when they get too large, they print nicely coloured output to terminals that support it and there are various helpers in the code. Try the `--verbose` flag to see it in action.
- The start of a history manager has been added.
- The app now caches resolved coordinates and classpaths for 24 hours. This means Maven Resolver isn't invoked at all when you use an app regularly, if you start an app without specifying a version number.
- Windows:
  - Background tasks work properly now.
  - JNA has been integrated. It's used to display a message box if an exception is thrown during startup, because Windows won't let you print to the console if you're a GUI app. But JNA will come in useful later for other things too.
  - Some investigation of how to handle the GUI/console app dichotomy that Windows has. Tasks were filed.
- Refactored the code to use co-routines, this enabled more sharing of code between the CLI and GUI frontends and cleaned up the logic quite significantly. A new `AppLauncher` class centralises handling of all app launch tasks.

Next steps

The next big performance win will be to use the background task support to refresh apps in the history list in the background, even when Graviton isn't in use. Most of the infrastructure is there now, it just has to be wired up. Once that's done app startup will be near-instant after first use.

After that it's back to investigating why SSL halves performance.

1.2.12 8th April 2018

Progress report

This was a productive first week!

- An especially big welcome to Anindya Chatterjee who has contributed improved Linux support:
  - Native bootstrap
  - Scheduling using cron
  - And packaging, which we improved to create DEBs. There is still some work to on the Linux package before it's ready however.
- We enabled parallel POM resolution, which doubled the speed of downloading applications.
- Performance investigation showed that SSL is a major performance hit at the moment, disabling it gives another 2x speed increase.
- The background task scheduler is now activated on first run for all three platforms, and removed on uninstallation for Windows.
- The design site was refreshed with a video of the shell, and an update for the altered product vision (see below).

The product vision received some tweaks this week - whereas previously it was imagined that apps would be written specifically for Graviton, we have now introduced the concept of "incremental adaptation" in which existing apps that exist in Maven repositories and on GitHub can be used out of the box, with no Graviton specific changes. Adding code to interact with the platform will improve the user experience but is not a technical requirement. This is the result of seeing that it's feasible to run apps direct from Maven repositories interactively.

Next steps

Try to discover why SSL slows things down so much. Experimenting with an OkHttp backend to Maven Resolver might be a good next step here, as Java SSL is known to be slow and OkHttp supports the Conscrypt security provider that uses BoringSSL under the covers.

Improve the Linux package to install files into the numbered directory (or make it irrelevant for the Linux bootstrap program).

Implement a module that downloads and signature checks new platform-specific native images.

1.3 Roadmap

Here are various feature ideas and plans broken down by area, to give an idea of where the project might go. GitHub Issues is the canonical issue tracker for the project.

1.3.1 Shell UI

Beyond coordinates. In future, other forms of app identifier beyond Maven coordinates may be considered, like git/github URLs.

Agile. The shell will become separated from the main browser and updatable independently, so we can rapidly push new versions and iterate the content.

URL handler. Graviton will register a URL handler so it can be invoked from the web. Clicking a URL will open the shell window with the app coordinate pre-filled but the user will still be expected to press enter to run it. This is so the

user gets a chance to opt in or out of sandboxing, and also to train the user to go straight to Graviton to run apps (it's faster for them and reduces the risk of browser makers trying to kill the platform by blocking the URL scheme).

App streaming. The average web page is 2mb in size. Experimentation shows that many apps can easily be made to fit within this size budget using pack200 compression and by not re-downloading commonly used dependencies. Please see *Code fetching* for more information on how code is fetched and kept fresh. Various optimisations will be implemented to move beyond the rudimentary Maven repository protocol and enable "instant on" apps.

Custom app tile artwork. Allow apps to customise their tile with background art, rich text, quick launch buttons.

Tabbed UI. Allow multiple apps to run simultaneously with a Chrome-style tabbed UI.

Loading splash. Apps can provide a logo image that'll be displayed whilst an app is being initialized.

JediTerm for Windows users. The Windows shell is notoriously not very good. Allow CLI apps to be run from the Graviton GUI for Windows users, using JetBrains JediTerm.

View source. When an artifact has an attached source JAR, allow the user to 'view source' on the app with a simple embedded source code viewer for whatever screen the user happens to be on.

History/navigation. If an app wants it, expose a form of browser-style page oriented navigation with Graviton controlled back button, history list and ability to wrap up current app state in a bookmark/textual form, similar to URLs. This would be entirely opt in but may be useful for apps that want to allow users to rapidly share locations and context via a URL-like copy/pasteable string.

1.3.2 App discovery

Showcase. The shell may create tiles for apps the user has never used before, to advertise great or interesting desktop apps that Graviton can launch.

mDNS/Bonjour discovery. Be able to locate domain names and apps that are advertising themselves on the local network. Whilst this would be ineffective for very large enterprise networks that are multi-segment and do not support broadcast, it is sufficient for IoT devices to advertise themselves (e.g. printers, wifi hotspots, etc). It is also sufficient for factory floor applications, smaller offices, and so on. mDNS/Bonjour names look like this "foobar.local" where the name is chosen by the app itself.

Enterprise internal repositories. For larger networks, Graviton will support a variety of "well known" (i.e. hard coded) domain names that may be pointed at internal Maven repositories. By linking continuous integration systems to an internal Artifactory or Nexus deployment, code will be automatically pushed direct from source repositories through to the desktops in a smooth and silent manner.

IoT secure repositories. A server may present a self signed certificate if it was reached via mDNS. This certificate is then remembered and may not change in future without generating a giant red scary warning page. This is different to how web browsers use SSL and is intended to make life easier for internal app developers and embedded devices that struggle to obtain a web PKI certificate today.

P2P networks. In future the peer to peer DAT protocol may be interesting as an additional protocol, if consumer use cases turn out to be more popular than hoped for.

Active Directory and other SSO integration. Internet Explorer and some other browsers allow for automatic remote sign-in based on local credentials, when the network is properly configured. It'd be nice to enable this automatically for apps that can benefit from it, for arbitrary remote servers and app repositories.

1.3.3 Versions and updates

Version control. Allow the user to pin apps to a particular version (partially implemented already). Allow network admins to do the same for both apps and the JVM itself, and also to trigger rollbacks in case of regressions.

Don't use metered connections. Try to detect roaming connections and don't do updates when using them.

Target versioning. Allow apps to declare what version of Graviton/Java they were tested against. Enable backwards compatibility goo for older apps to keep them working, allowing developers to opt-in to potentially breaking changes when they're ready.

Just-in-time module downloads. Expose Maven resolution via the Graviton API, so that apps can request plugins and additional features be downloaded and added to their classpath on the fly.

Better JitPack integration. If an app request is being satisfied by jitpack.io then monitor build progress and feed it back to the user.

Support reproducible builds inside SGX enclaves. Investigate using Oblivium to do what JitPack does, but in a remotely auditable manner, so anyone can run build servers.

Better app distribution format than JARs. The JAR format is old and works well enough for now but pack200 showed it's possible to improve on it significantly.

Canary channel. Distribute regular automated daily OpenJDK and OpenJFX builds through a canary channel to allow for rapid-fire testing of new code.

Delta updates. Download only what changed between Graviton releases, to reduce bandwidth consumption and improve reliability.

1.3.4 Security

Sandboxing. Resurrect the applet sandbox and make it fit for the 21st century. Use the `SecurityManager` infrastructure to implement the PowerBox pattern, in which sandboxing rules are tweaked on the fly via natural UI interactions like using the operating system file open dialogs, drag and drop and so on. Allow an app's activity to be watched, and allow the sandbox to have multiple levels of aggression to reflect the varying nature of trust. No code signing required.

Dependency pinning. Avoid problems with hacked CDNs or overlapping artifacts, by allowing JARs to specify the hashes of their entire dependency graphs. Useful in combination with:

Signature consistency enforcement. If an app is signed with a public key (or sub-key), require it to continue being signed with that key. This ensures only the original app publisher can publish updates and is similar to the Android scheme, in which app identity is linked to the signing key. *This is not the same as CA legal identity code signing*. Sandbox permissions would become linked to the signing keys rather than coordinates.

JVM security upgrades with Arabica. Arabica is a way to run some of the native libraries that come with a JVM (like e.g. GUI libraries) inside a Google NaCL sandbox for native code. It could be integrated to provide a more robust virtual machine.

Execution timeouts. Terminate an app forcibly if it goes into an infinite loop - requires Graal, see below.

1.3.5 Java 11+ features

JavaScript/LLVM. Graviton is not Kotlin or JavaFX specific. It should come with the Graal compiler and Truffle backends, as GraalVM itself does. In this way apps should be authorable in JavaScript, Python, Ruby, C++, Rust, Haskell and so on, if they depend on the right runtime modules that Graal can recognise.

Graal is on the verge of offering several features that are of particular interest:

- Support for NodeJS modules.
- Ability to impose execution time limits and interrupt execution asynchronously, to break infinite loops. This is effectively a compiler-supported version of the deprecated Thread.stop() and is useful for browser style code sandboxing. In early versions it is acceptable for Graviton to hang in the face of a DoS attack by a malicious app - it is unlikely to matter for the initial use cases.

- Support for Python, Ruby and LLVM. Thus Graviton programs could conceivably utilise sandboxed modules written in C/C++, offering an alternative to WebAssembly.

Modules. Graviton should assemble the module and classpaths automatically, placing modular JARs onto the module path by default and the rest onto the classpath. Graviton may additionally split JARs into multiple layers in order to automatically resolve version conflicts, when an app has to use two different and conflicting versions of the same module (common with Guava). When not run in a sandbox all modules should be opened to the app, or the app should be able to request a list of –add-opens flags, to avoid crashes due to module encapsulation.

Auto AOT. Graviton should pre-optimise the JVM image using ahead of time compilation for the `java.base` module, and consider AOT compiling modules of apps that have opted in via a background task, for faster startup. Tools that indicate they can support it could be automatically fed through SubstrateVM to generate small native binaries.

JavaFX. Bundle all JavaFX modules out of the box so app developers don't have to worry about this.

1.3.6 Command line tools

Graviton already provides some nice features to CLI programs, such as activating ANSI escapes on Windows and auto-update. But there is much more we can do here.

Aliases. When an app is used for the first time, make a small startup script in a special directory named (by default) after the artifact name that acts as an alias for *graviton coordinate*. This directory can be added to the end of the PATH, allowing you to start the tool in a natural way after the first time.

CLI store. In the background update task, pre-populate a different special directory with launchers for popular tools you *haven't* used yet, selected by the Graviton team. If this is at the end of the PATH, then you may find there is no need to ever install CLI tools again: on first run the tool will be downloaded and then kept fresh for you.

Update callbacks. If a CLI app is Graviton-aware, run a callback if a new version is downloaded. If the app is a long running server, it can then choose to restart itself to pick up the new version. This allows for self-updating servers.

Start-on-boot registration. Abstract the OS start-on-boot mechanisms e.g. systemd and Windows Services, to make server installation and maintenance absolutely painless.

Sandboxing. CLI specific sandboxing, so untrusted CLI tools can be invoked. Never run `curl ... | bash` ever again! For apps where it fits, command line parameter parsing can be used to auto-configure the sandbox e.g. to only include paths and servers that were passed in.

1.3.7 Gaming features

EGL and advanced graphics. The Java game dev community is surprisingly large and successful - consider that Minecraft came out of it. They would be a great market segment to target and a potentially enthusiastic userbase. For this to work they need access to OpenGL contexts. A simple starting point is to let them run unsandboxed in a separate JVM instance. Later versions of Graviton could offer an API to open a new OpenGL window and expose the handle back such that it could be combined with JMonkeyEngine, LWJGL and other popular game engines. Chrome uses an open source layer to implement EGL on top of Direct3D which improves support on Windows, and it could be integrated into Graviton. Once this work is done an eGL surface for JavaFX apps should be relatively straightforward for experienced systems/graphics programmers.

API for opening ports via UPnP. Expose an API to allow sandboxed apps to request firewall/NAT port forwarding.

Sandboxed full screen mode. Provide a browser-style full screen mode which helps the user exit it, independently of the app itself.

1.3.8 Desktop integration

Integration with native desktop IPC. Graviton apps should be able to expose control surfaces via platform native OO IPC mechanisms, in particular, COM and DBUS. This would allow scripting and interaction with Graviton apps from tools like MS Office macros.

Support cross platform file associations. Enable registration of file extensions to Graviton apps. The user would be prompted the first time such a registration is used (*not* registered) if they want to open the file type with that app, so registration can be seamless and 'optimistic'.

Single-instance mode. Only allow a single instance of an app to be run at once, if the app has requested that.

Connectivity callbacks. Expose a JavaFX observable bean that indicates the current status of network connectivity, along with a higher level API so downloads can be transparently paused and resumed when the network is back.

1.4 Graviton apps

Graviton can run any JVM app with a main method or subclass of `javafx.application.Application`. However, apps can opt-in to new features that Graviton makes available to you if you agree to run inside its JVM.

You can reuse the Graviton window to avoid annoying flicker and disappearing/reappearing shell windows. This may also make startup faster because you will also be running in the Graviton JVM. This imposes some requirements on you, but they are not onerous. Here are some things to be aware of:

1. The primary requirement is that you test your app thoroughly, and in particular, you test your app after starting it from the same shell twice. At this time Graviton does not provide total isolation between app runs - if you reconfigure the JVM by e.g. registering classes with various APIs, those registered classes may still be there when you restart.
2. You should avoid making big changes to the primary stage, and accept it how you find it. You can change the title.
3. Don't use System.exit or explicitly shut down JavaFX. That will quit the browser. You may use `Window.setOnCloseRequest` to register a callback for when the user closes the window or presses Alt-F4/Cmd-Q.
4. You should use `setOnCloseRequest` to tidy up any threads that you started which may still be running. Otherwise in the current version of Graviton they may hang around and continue executing after your app appears to have quit.

Note: Future versions of Graviton will use varying levels of sandboxing to help your app clean up.

There are two ways to opt in to Graviton features: by adding an attribute to your app's MANIFEST.MF file, or by implementing the Graviton API.

1.4.1 Using the API

To use the API add a dependency on the `app.graviton:graviton-api` library. This provides various types that can be used to opt-in to features and communicate with the browser. Although this will become a runtime dependency, it's very small (contains only interfaces and annotation) and adds no overhead when Graviton is not in use.

1.4.2 Testing your app locally

To verify your app runs well in Graviton, you can publish to a special local Maven repository called `dev-local` that is examined by Graviton as if it were a remote repository. It exists in your `$HOME/.m2/dev-local` directory, or

Windows equivalent.

Here's how you would configure Gradle to publish to this repository:

```
publishing {
    publications {
        app(MavenPublication) {
            from components.java
        }
    }
    repositories {
        maven {
            name = 'gravitonLocal'
            def homePath = System.properties['user.home']
            url = "file://${homePath}/.m2/dev-local"
        }
    }
}
```

If you add this snippet to the top level of your Gradle file, you can now run `gradle publishAppPublicationToGravitonLocalRepository` (you can also use the IntelliJ GUI to do this, as it's a bit of a mouthful) and the app will become visible to Graviton.

With Maven you don't need to change your POM, just change the command line:

```
mvn deploy -DaltDeploymentRepository=dev-local::default::file://$HOME/.m2/dev-local
```

1.4.3 Set the main class in your app

You need to start by making sure your JAR is executable. This involves specifying the main class in your manifest.

Here's how you do it with Maven:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>2.6</version>
    <configuration>
        <archive>
            <manifest>
                <mainClass>tictactoe.TicTacToe</mainClass>
            </manifest>
        </archive>
    </configuration>
</plugin>
```

Or for Gradle:

```
jar {
    manifest {
        attributes("Main-Class": "tictactoe.TicTacToe")
    }
}
```

This class should contain your static main method.

1.4.4 Reusing the shell window: Manifest

Add `Graviton-Features: inline` alongside the `Main-Class` entry in your JAR manifest. With Maven you can do it like this:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <configuration>
        <archive>
            <manifestEntries>
                <Graviton-Features>inline</Graviton-Features>
                <Main-Class>com.example.Main</Main-Class>
            </manifestEntries>
        </archive>
    </configuration>
</plugin>
```

Or with Gradle, like this:

```
jar {
    manifest {
        attributes(
            'Graviton-Features': 'inline',
            'Main-Class', 'com.example.Main'
        )
    }
}
```

The JavaFX `start` method of your app will be called as normal with a hidden stage, so you can change the top level scene of the stage then show it.

1.4.5 Reusing the top level window: API

If you use the Graviton API to reuse the window, you get access to a `Graviton` interface that lets you interface with the browser. This feature is not available if you go the manifest route. To do this:

1. Implement a JavaFX app by subclassing `javafx.application.Application` as normal. Set this to be your main class in your application manifest as above.
2. Add a dependency on the `app.graviton:graviton-api` library in your build file.
3. Implement the `GravitonRunInShell` interface on your main class. It requires one method `createScene`, which takes a `Graviton` object and returns a JavaFX `Scene`.
4. Refactor your `start(Stage)` method so the part that configures your `Scene` is moved into the `createScene` method. From `start` just pass null to the parameter.
5. Adjust your `start` method so if the stage is already visible, you don't attempt to set the scene or modify the window in other ways beyond adjusting the title.
6. Adjust your `createScene` method so if the Graviton parameter is non-null, you pass the width and height obtainable via that object into the `Scene` constructor (assuming you want your scene to fill the whole shell area).

Here's an example:

```
public class MyApp extends Application implements GravitonRunInShell {
   @Override
   public Scene createScene(Graviton graviton) {
       Button root = new Button("Hello world!");
       if (graviton != null)
           return new Scene(root, graviton.getWidth(), graviton.getHeight());    //␣
↪Fill the browser area.
       else
           return new Scene(root);
   }

   @Override
   public void start(Stage primaryStage) {
       primaryStage.setTitle("MyApp");
       if (primaryStage.isShowing()) return;    // Running in Graviton so bail out.

       // Running outside of Graviton, set up the stage.
       primaryStage.setScene(createScene(null));
       primaryStage.show();
   }
}
```

1.5 Tutorial: graphical hello world

Graviton can run unmodified Java apps, but you can improve the user experience by making small upgrades to your program.

This tutorial takes you through how to make a simple GUI app and then improve how it runs. This isn't the only way to publish software for Graviton - it's just a gentle getting started guide.

Note: You can find the code for this app with one commit per stage here, on GitHub.

1.5.1 Step 1: publish from GitHub

Make a hello world app that uses JavaFX. It could also use Swing, SWT, OpenGL etc but for now pick JavaFX. Here's what such an app might look like:

```
package hello

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HelloWorld extends Application  {
    @Override
    public void start(Stage primaryStage) throws Exception {
        StackPane helloPane = new StackPane(new Button("Hello world!"));
        primaryStage.setScene(new Scene(helloPane));
        primaryStage.setWidth(300);
        primaryStage.setHeight(200);
```

(continues on next page)

(continued from previous page)

```
        primaryStage.show();
    }
}
```

Of course it could also use Kotlin, Scala, Haskell, Ruby etc as these languages all run on the JVM.

Use Gradle or Maven as the build system. Your IDE can generate such build files if you aren't sure where to start. A simple Gradle `build.gradle` file for the above program might look like this:

```
plugins {
    id 'java'
}

group 'com.github.yourusername'
version '1'
```

The group should be set to a package / reverse DNS name you control. If you don't have one, you can use the GitHub pattern shown above.

Talking of which, now upload your hello world app to GitHub and use the GitHub UI to make a release of version 1.

Note: The description you provide for your GitHub repository will appear in the recent apps list, so set a good one!

Open Graviton and enter `com.github.yourusername:reponame` where obviously the string is derived from the URL of your repository (`https://github.com/yourusername/reponame` for the prior example). Graviton will work in conjunction with JitPack to check out your source code from your repository, build it, package it into JARs, publish it via a Maven repository and then download it to your computer and run it.

Try clearing the cache and starting the app again. It'll download much faster this time as JitPack won't need to download your code or build it.

Now try starting your app once last time from the recent apps list. It should start instantly. That's how it'll be for your users after their first run too, as updates are applied asynchronously in Graviton. Your app will also work offline.

Now change the label on the button and make a new GitHub release. Either go do something else for a few hours, or just right click on the app tile and select 'Refresh'. When you start your app from Graviton again, it'll be on the latest version. Your users should get the new version within about 6 hours or so if their computer is switched on, or shortly after they've logged in when it's been off for a while.

That's it! You just published your first desktop app, direct from your git repository.

Note: JitPack is useful for development purposes but you will probably want to do your own builds in future. For that you should investigate BinTray, which lets you easily create and publish programs to the JCenter Maven repository. In future this tutorial may cover how to do that.

1.5.2 Step 2: Specify the main class name

In the first step, we made a class that subclasses `javafx.application.Application`. That was sufficient for Graviton to figure out where the program should begin. But this way to start apps will get slower as our app gets bigger, because Graviton had to scan all the files in our app looking for `Application` subclasses.

It's more efficient to tell Graviton exactly where the program is meant to start. Add the `application` plugin in your Gradle file, set the `mainClassName` property and then set the version to 2, so it looks like this:

```
plugins {
    id 'java'
    id 'application'
}

group 'com.github.yourusername'
version '2'

mainClassName = 'hello.HelloWorld'
```

Commit it, push to GitHub and make a new release. You probably won't see any observable difference in speed just yet, but it's good practice to always explicitly specify your entry point.

1.6 Online update

One of the most annoying tasks with distributing desktop apps is keeping them up to date. This is something browsers excel at and, along with sandboxing and a smooth developer on-ramp, can largely explain the web's success as an app platform.

1.6.1 Goals

- Chrome-style silent upgrade of the JVM, of Graviton and of the apps.
- Updates can be applied whilst the browser is running, without any user intervention or disruption.
- Updates can be applied even when the browser is not running.
- Most updates do not require platform-specific rebuilds.

1.6.2 Non-goals

- In this first iteration, network administrator control is out of scope. It can be tackled in future.
- Linux support - Linux users have their own platform level update mechanisms that can be reused.

1.6.3 Design

Note: This design is **implemented**.

The browser uses Java 8 and the javapackager tool, which produces a directory layout and native installer for each platform, with a small native startup tool that loads JVM parameters from a config file.

We modify the directory layout before the native installers are produced to replace the startup program with our own, the bootstrap program, which selects a 'sub install' to run. In this way new versions can be unpacked alongside the current running version. Apps are downloaded to a Maven cache stored in the OS specific cache directories, which uses the same design of versioned directories.

The operating system task scheduler on each platform is used to invoke Graviton at 6 hour intervals. It checks for new versions of itself, and uses a history file that records the last 20 app invocations the user made. It then uses Maven Resolver to re-resolve each in turn - if the app had no version specified this will download the latest version (otherwise it's a no-op unless the cache was cleared).

Bootstrap

The directory structure of the browser app looks like this:

```
/graviton
/1/bin/graviton
/1/lib/...
/2/bin/graviton
/2/lib/...
```

In other words, there is a native executable at the root location, and then several sub-installations identified by sub-directories with integer names. The native executable at the root scans the directory list at startup and selects the directory with the highest name before re-executing the binary found under that location with the same set of command line arguments. The native executable is called the *bootstrap* and it does very little, so will need to change rarely. It is be written in Kotlin/Native and there are different versions for macOS and Windows.

This design means that it's possible for a running instance of the browser to download and unpack new versions of itself whilst running. Partially unpacked installations should be put in a directory with a name that starts with a letter, and then the directory can be renamed to the final integer name when unpacking and preparation is finished. The browser can monitor this directory to see if a new version has become available whilst running and notify the user with an unobtrusive hint in the user interface, as Chrome does it.

It is assumed the user is either an admin user (on macOS) or that the app was installed to the user's home directory. On Windows the installer we provide does not offer any customisation options, thus, it will always be in the user's home directory (under the hidden AppData folder).

The `javapackager` tool is used to assemble the final download.

Update tracking and batched updates

The browser stores its current integer version (all versions are integers in the update system) in a file called `last-run-version`. If the browser starts and discovers its hard-coded current version is different to the version stored in the file, it knows it has been updated.

There may be situations where the user has been offline for a long time or uninstalled the browser without removing its data files. The browser may thus end up skipping versions, where the last run version was e.g. 5 but the current version is 10. The browser must be designed to be able to iteratively upgrade its data files from older versions.

We use this design rather than allow the browser to only have to upgrade from the last version of itself because, whilst the update system could step through each update in sequence, this would mean no way to recover from botched releases. More importantly, it would mean if the user uninstalled the app and reinstalled it a long time later, we'd have to download and run all the old versions to step through the upgrade sequence, which would be bad. So instead we insist that the browser be able to read and upgrade data files from any prior version of itself.

Background updates

The browser runtime accepts a command line flag `--background-update` which causes it to poll a remote server to see if new versions are available instead of starting the normal codepaths. If the running version is the latest version, it exits silently. Otherwise it begins a background download of an update, and proceeds to create a new directory with the new version inside it so it will be picked up next time the browser is started.

On each platform the operating system task scheduler is used to trigger this process. A new library, *Graviton Scheduler* has been created to abstract the OS task scheduling functionality. On Windows, the Task Scheduler is configured to run this program once a day by running `c:\Windows\System32\Schtasks.exe` with an XML file like this one. This will take place on first run. On macOS there is a similar process, in which an XML file is dropped into the `~/Library/LaunchAgents` directory, and on Linux the user-local cron is used.

In this way the browser is guaranteed to be up to date (secure) even if the user hasn't used it for a while, which will be typical in the early days when there aren't many apps.

Update security

Updates take two forms.

On macOS and Linux, it is a platform specific JAR file that's unpacked to the target directory by the update process itself. It is not a platform specific installer. Tasks that are needed will have to be done on first-run (identified by an old or missing `last-run-version` file). The JAR does not contain class files. Instead we use it only for its signing capability.

We sign the update pack with the `jarsigner` tool. There may be multiple signatures required from different parties, to provide a more secure multi-signature update scheme (everyone reproduces the build and a quorum of developers signs it).

The unpacking code verifies each signature against a hard coded list of public keys. If any file fails to present the right list of signatures, the update is discarded and will be retried.

Note: This means if a bad update is pushed users will keep trying to re-download it until it's fixed.

On Windows, the update is the same NSIS installer that users download the first time they install the browser. It is run with special command line flags that make it run invisibly in the background, and ignore files that are the same version or newer (i.e. comparing the PE headers). Because of the numbered install directories, this installer can be run whilst Graviton itself is running.

Update protocol

The updater requests the URL `https://update.graviton.app/<osname>/control?c=5` where 5 is the current version of the app and "osname" is either "mac" or "win". The control file is a properties file that must have at least one key, "Latest-Update-URL" which contains a relative URL to the update pack. The value of this key will be interpreted as if it were an HTML link, so, you can use either absolute URLs or a path like "/foo/bar" in it.

The filename must be of the form "5.something.whatever", i.e. a dot separated name where the first component is the integer version number. It will be downloaded and unpacked only if the version number in the filename is higher than the currently executing version. The other components are arbitrary and ignored.

The signed pack will be downloaded, verified and either unpacked into the numbered directory indicated by the file name, or executed. On UNIX systems the execute bit is set on a hard-coded OS specific path to ensure the main executable can be invoked. Once this is done the update is complete.

Updating the updater

Because the update process is performed by the app itself, triggered by a command line flag, the update process also by implication update the updater. In the unlikely event that the bootstrap program needs to be changed, that can also be handled by special case code, assuming the user can write to that program. However given it does so little the hope is it never needs to be updated once created.

This mechanism can be used to change the signing keys that are authorised to push upgrades, as the set of developers evolves over time.

JRE minimisation

Java 9+ introduces a nice feature; the jlink and javapackager tools can now minimise the JRE by stripping out modules that aren't needed. Unfortunately it comes with a huge caveat - this only works for fully modularised apps, and the tooling, Gradle and Kotlin support for this is half baked. Building and jlinking a modular Kotlin app is still far from easy. For now we will punt this to later in the hope that the ecosystem eventually catches up.

1.6.4 Preparing an update

Make sure you have a keystore.p12 file that contains an PKCS#12 key store. You can create a signing key like this:

```
openssl ecparam -out ec_key.pem -name secp256r1 -genkey
openssl req -new -key ec_key.pem -nodes -x509 -days 3650 -out update_cert.pem
# Enter some plausible sounding details here. The cert details don't matter.
openssl pkcs12 -inkey ec_key.pem -in update_cert.pem  -export -out keystore.p12 -
↪alias $USER
```

After running these commands, you will have a p12 file that contains an elliptic curve private key and certificate, under the alias of your current username.

The procedure for pushing an update to the browser and runtime is as follows.

1. Increase the version number in the root build.gradle file (it's represented as a string but must be an integer value)
2. Run the package-osname script in the root directory for each OS in turn. For instance `package-mac.sh`
3. This will run the procedure to generate a native installer, unpack it, and then output a signed JAR of the update in the current directory.
4. Upload this JAR to the server.
5. Update the control file.

> **Warning:** On macOS make sure you don't have any prior disk images mounted when running, as it can interfere with the build process.

1.7 Command line apps

A big part of Graviton is "instant on" streaming apps, which keep themselves fully up to date. This capability is just as useful for command line apps as it is for GUI apps. All major operating systems have command line package managers now. Linux paved the way with tools like apt-get and yum, macOS has brew, Windows has chocolately.

Unfortunately, it sucks to write cross platform CLI apps. Java apps are ugly from the command line as you must set up the classpath manually, and prefix commands with "java -jar programname.jar" which is unnatural. And that doesn't help you keep them up to date, or manage dependencies. On the other hand, few other platforms have robust operating system abstractions (e.g. Go isn't that great on Windows). If you want to distribute a cross platform command line tool you're pretty much restricted to Python and pip: it's fragile and isn't ideal for more complex apps where static typing or high performance is desirable.

Finally, many tools make assumptions that aren't valid on Windows, e.g. by using ANSI terminal escapes.

Graviton can move into this niche if it has great support for command line tools.

1.7.1 Goals

- Natural, easy access to programs from the command line.
- Apps are identified by Maven coordinates, stored in Maven repositories and dependencies work correctly.
- Apps update automatically in the background unless pinned.
- Minimal OS overhead in the common case of no work for Graviton to do.
- One-liner to obtain new apps.
- Optional sandboxing.
- Support ANSI escapes.

1.7.2 Non-Goals

- Exposing new services or APIs for command line apps, like an argument parsing API.

1.7.3 Design

Usage

Either at install time or on first run, Graviton will either add itself to the path (on Windows) or symlink itself to `/usr/local/bin/graviton` (on Mac/Linux). The user can run an app like this:

```
graviton org.jetbrains.kotlin:kotlin-compiler --help
```

If an app will be used frequently, a symlink can be requested with the expected name:

```
graviton --alias org.jetbrains.kotlin:kotlin-compiler=kotlinc
```

If there's a `Command-Line-Name` entry in the MANIFEST.MF and if there is no other command on the path with that name, an alias to that name will be set up the first time the app is run. From that point on the short name can be used.

Graviton remembers which packages were requested via this command line interface versus downloaded as part of dependency resolution.

If a version number was specified in the coordinate when first installed, that version is pinned. If a range was specified, or no version was specified at all, then Graviton will update these apps during its regular background update check. Because Maven repositories are immutable, this will not disturb any running applications.

Every so often, old versions of apps that were installed via the CLI are removed and the local Maven repository garbage collected to free up space.

Windows

Windows requires special support. EXE files have to be marked in the header as either GUI or console apps. GUI apps can't print to the console if launched from the command prompt and automatically go into the background. Console apps pop up a console window if launched from the GUI, regardless of whether or not the app uses it, which is ugly. Finally, although Windows 10 added support for ANSI colour and cursor escape codes, the console ignores them by default. Enabling them requires Win32 API calls.

We solve this with a somewhat complex bootstrap:

1. The Graviton bootstrapper tool (which is a native app) is compiled twice, once in GUI mode and once in console mode.

2. The console mode version enables ANSI escapes support and then starts the main Graviton app and waits for it to finish.
3. The GUI app starts. At this point the Windows kernel has detached it from the console and closed its input/output handles. We re-attach to the parent console and then re-open the input/output handles, repeating the part of the JVM startup sequence that initialises System.{in,out,err}

1.8 Code fetching

A big part of the browser experience is how code is downloaded and kept fresh, whilst being as fast as possible.

1.8.1 Repositories

To help us get started quickly and be useful right away, Graviton currently fetches code only from Maven repositories. It comes pre-configured with the following repos:

1. Maven Central
2. Bintray
3. Jitpack.io - this one conveniently turns github repositories into Maven repositories on the fly, see below for more info.
4. A local repository in ~/.m2/dev-local which you can deploy to directly. See below.

To make an app available it and all its dependencies must be findable via one of these four sources. In future we will add support for auto-discovery of other repositories on the local network using registry keys, mDNS, and so on.

Because of this decision, all code modules in Graviton are identified by Maven "GAV" coordinates (group ID, artifact ID, version). An "artifact" in Maven-speak is just a file. A group ID is a Java-style reverse DNS name. And a version is self explanatory (if you need more info here is a good article on Maven version numbers.

1.8.2 Future support

It's not intended to use Maven repositories forever. Over time we could develop better formats with higher security and performance. For example, apps could commit to the hashes of their dependency tree for more security against compromised repository mirrors, pack200 could be brought back from the grave, and classes from many modules could be multiplexed together into a single stream with a 'watermark' where execution should begin to allow apps to be started before they've fully downloaded.

1.8.3 Jitpack

You can run apps directly from GitHub by using jitpack.io - just use coordinates of the following form:

```
com.github.username:repositoryname:commit-hash
```

For example, `com.github.mikehearn:tictactoe:master-SNAPSHOT`.

1.8.4 Local development repository

Graviton will not use your `~/.m2/repository` directory if you have one, both for internal technical reasons and to keep your development environment separated. However it will

1.8.5 Optimisations

Making an app start as fast as a web page is partly about download optimisation and we have many planned or already implemented:

1. Parallel resolution of the dependency tree (DONE).
2. Local caching of artifacts, so commonly used libraries are not re-downloaded repeatedly (DONE).
3. Only re-check for new versions once a day (DONE).
4. Cache the resolved classpath so it doesn't have to be re-calculated on each use (DONE).
5. Pre-generation of a dependency tree file, so a POM walk isn't necessary.
6. Proxies for common Maven repos that respond to failed download attempts by fetching the requested artifacts and recompressing with pack200, thus automatically optimising distribution of apps that are being frequently requested.
7. Early launch - monitoring a "training run" of the app and observing when classloading activity pauses for a few seconds. Any modules accessed before that time are assumed to be needed and will be resolved before startup, any modules accessed after that will be downloaded whilst the app is running. An attempt to access a class in a module that wasn't loaded yet will hang until loading completes. In this way apps can be adjusted to stream features in the background.
8. Pre-fetch of commonly used libraries so apps don't pay any download cost for them.
9. Identification of artifacts by secure hash rather than just coordinates.

All the app-specific adaptations can be made easy with Maven and Gradle plugins.

1.9 Testing

Here are some initial notes on testing Graviton. In the early days, testing is done through a mix of unit tests and manually executed test plans. As the product definition and UI stabilise, this will shift towards heavy usage of integration tests.

1.9.1 Using WireMock to simulate Maven repositories

WireMock is a Java program that can simulate web servers, with many features. Especially useful for us is the ability to record and replay interactions with real servers, customised in various ways and with fault/slowdown injection. The interactions are stored to JSON files.

WireMock can be used via Graviton itself. To make a recording of the interaction with a Maven repository follow these steps.

Firstly, run `graviton com.github.tomakehurst:wiremock-standalone` in a new empty directory.

Now run:

```
curl -d '{"targetBaseUrl": "http://repo1.maven.org/maven2", "extractBodyCriteria": {
↪"binarySizeThreshold": "1kb"}}' http://localhost:8080/__admin/recordings/start
```

This will set up a recording proxy that will interpose on Maven Central (obviously change the URL as desired but watch out, don't add a trailing slash).

Invoke Graviton like this:

```
graviton --cache-path=/tmp/gravicache --repositories=http://localhost:8080 com.github.
↪ricksbrown:cowsay moo
```

to download via the proxy into a fresh cache directory.

Finally, use `curl -d '{}' http://localhost:8080/__admin/recordings/stop` to stop the recording.

The directory you ran WireMock in will now have two directories, files and mappings. Any binary larger than 1kb will have been put in the files directory, textual responses or small binary responses will be in JSON files in mappings. If you re-run the same Graviton command against the WireMock server, all the answers will come from the recording. You can test this by disabling your wifi or ethernet connection.

1.9.2 Test plans

This document contains some manual test plans. At some point we should automate them, although testing software like Graviton can be tricky. We'll need to use an HTTP server that can replay recorded interactions with real Maven repositories, and use lots of TestFX for the GUI, mock filesystems and so on. Then integration tests for the OS integrations/background updates.

Pre-flight checklist

Build the package for each OS in turn. Ensure there are no Graviton cache directories anywhere and remove them if so. Install the package.

Mac installer

- Make sure your Mac security settings are configured to be the default, i.e. Gatekeeper is active.
- Ensure there are no Graviton installs or data directories already.
- Open the mac DMG and drag it to the Applications folder. Ensure the icon and DMG background look right.
- Start the app from the Applications folder and ensure it starts, that the menu bar shows the name Graviton.

Windows installer

- Click the Windows installer. Ensure it installs automatically and with no user interaction.
- Find the app in the start menu and start it. Shut it down.
- Open task scheduler and check the background update task is registered. Run it manually and check that in the log that the update process ran up to the point that it knows it's up to date.
- Go into the control panel and uninstall Graviton. Ensure the uninstall completes without errors (this is a common failure point!)

CLI

- Run `graviton --help` and ensure a useful help output is created. On Windows, ensure it's in colour.
- Run `graviton --version` and ensure the version number is correct.

- Run `graviton com.github.ricksbrown:cowsay --cowthink moo!` and ensure it downloads and runs correctly.
- Run it again and ensure there's no download or other UI text this time, just the cow.
- Go offline (disable wifi) and run `graviton --clear-cache com.github.ricksbrown:cowsay "cannot work"`. Check the error message you get is sensible and helpful.
- Run `graviton --clear-cache com.github.ricksbrown:cowsay "Downloaded again!"` and ensure it re-downloads and runs like before.
- Run `graviton -r com.github.ricksbrown:cowsay moo` and ensure it checks for an update again.
- Edit the history file to set the time of the last update to more than 24 hours ago. Then run `graviton --offline com.github.ricksbrown:cowsay moo` and ensure it doesn't check for an update even though the history entry is old.

GUI shell

- Start the GUI. Run Tic-Tac-Toe and ensure it downloads properly, the main window disappears, TTT starts. Quit and ensure GUI is restored.
- Ensure the history list populates with an app tile for it.
- Start `com.github.rohitawate:everest` and click cancel during the download. Make sure it stops.
- Start Everest again, this time, quit during the download and ensure Graviton properly quits.
- Start it for a third time and this time let it succeed.
- Right click on one of the tiles and ensure each item works.

Online update

- Clear your cache. Start `net.plan99:tictactoe:1.0.1` - it should download an old version of the app. Now go into your history file and edit it so the version number is missing from the coordinate fragment field. This makes it look like the user just hasn't updated for a while and a new version has been released.
- Start Graviton GUI. Now from the command line run `graviton --background-update` and check the log file to ensure it updated you to the latest version of tictactoe. Start the app in the GUI ensure it's now the latest version.

If the bootstrapper, installers, packaging or runtime update code was changed, for each platform:

- Bump the version number and prepare an update site that has the dummy update
- Install the current (under test) version
- Leave the machine or VM until an update check interval has passed, with Graviton running
- Make sure that the update has been downloaded, applied and restarting Graviton makes it appear

1.9.3 Next steps to improve testing

1. Implement or find a recording HTTP proxy that can simulate errors and slowness. WireMock isn't quite there for us, I could never make its browser proxy mode really work. Make recordings of installing various apps. The manual test cases should now be executable entirely offline.
2. Write an integration test tool that verifies, given a random/clean cache directory: * the CLI tool can be used to install and run basic programs * when run against the proxy recordings

3. Write a basic TestFX test that generates a clean cache directory and installs/runs TicTacToe from the proxy recordings.
4. Extend the test to ensure that the clear cache function works.