Ubuntu App Launch Documentation

Release 16.04.0

Ted Gould

Mar 29, 2017

Contents

Ubuntu App Launch is the abstraction that creates a consistent interface for managing apps on Ubuntu Touch. It is used by Unity8 and other programs to start and stop applications, as well as query which ones are currently open. It doesn't have its own service or processes though, it relies on the system init daemon to manage the processes (currently systemd) but configures them in a way that they're discoverable and usable by higher level applications.

CHAPTER 1

Environment Variables

There are a few environment variables that can effect the behavior of UAL while it is running.

UBUNTU_APP_LAUNCH_DEMANGLER Path to the UAL demangler tool that will get the Mir FD for trusted prompt session.

UBUNTU_APP_LAUNCH_DISABLE_SNAPD_TIMEOUT Wait as long as Snapd wants to return data instead of erroring after 100ms.

UBUNTU_APP_LAUNCH_LEGACY_ROOT Set the path that represents the root for legacy applications.

UBUNTU_APP_LAUNCH_LIBERTINE_LAUNCH Path to the libertine launch utility for setting up libertine containers and XMir based legacy apps.

UBUNTU_APP_LAUNCH_OOM_HELPER Path to the setuid helper that configures OOM values on application processes that we otherwise couldn't, mostly this is for Oxide.

UBUNTU_APP_LAUNCH_OOM_PROC_PATH Path to look for the files to set OOM values, defaults to */proc*.

UBUNTU_APP_LAUNCH_SNAP_BASEDIR The place where snaps are installed in the system, */snap* is the default.

UBUNTU_APP_LAUNCH_SNAP_LEGACY_EXEC A snappy command that is used to launch legacy applications in the current snap, to ensure the environment gets configured correctly, defaults to */snap/bin/unity8-session.legacy-exec*

UBUNTU_APP_LAUNCH_SNAPD_SOCKET Path to the snapd socket.

UBUNTU_APP_LAUNCH_SYSTEMD_CGROUP_ROOT Path to the root of the cgroups that we should look in for PIDs. Defaults to */sys/fs/cgroup/systemd/*.

UBUNTU_APP_LAUNCH_SYSTEMD_PATH Path to the dbus bus that is used to talk to systemd. This allows us to talk to the user bus while Upstart is still setting up a session bus. Defaults to */run/user/$uid/bus*.

UBUNTU_APP_LAUNCH_SYSTEMD_NO_RESET Don't reset the job after it fails. This makes it so it can't be run again, but leaves debugging information around for investigation.

UBUNTU_APP_LAUNCH_XMIR_HELPER Tool that helps to start XMir and sets the DISPLAY variable for applications

UBUNTU_APP_LAUNCH_XMIR_PATH Specifies the location of the XMir binary to use

CHAPTER 2

API Documentation

AppID

struct `ubuntu::app_launch::`**`AppID`**

The set of information that is used to uniquely identify an application in Ubuntu.

Application ID's are derived from the packaging system and the applications that are defined to work in it. It resolves down to a specific version of the package to resolve problems with upgrades and reduce race conditions that come from installing and removing them while trying to launch them. While it always resolves down to a specific version, there are functions avilable here that search in various ways for the current version so higher level apps can save just the package and application strings and discover the version when it is required.

Public Types

enum `ApplicationWildcard`

Control how the application list of a package is searched in the *discover()* functions.

Values:

`FIRST_LISTED`
First application listed in the manifest

`LAST_LISTED`
Last application listed in the manifest

`ONLY_LISTED`
Only application listed in the manifest

enum `VersionWildcard`

Control how the versions are searched in the *discover()* set of functions

Values:

`CURRENT_USER_VERSION`
The current installed version

Public Functions

operator std::string() const
Turn the structure into a string. This is required for many older C based interfaces that work with *AppID*'s, but is generally not recommended for anyting other than debug messages.

AppID()
Empty constructor for an *AppID*. Makes coding with them easier, but generally there is nothing useful about an empty *AppID*.

bool **empty() const**
Checks to see if an *AppID* is empty.

AppID(Package *pkg*, AppName *app*, Version *ver*)
Constructor for an *AppID* if all the information is known about the package. Provides a precise and fast way to create an *AppID* if all the information is already known.

Parameters

- `package`: Name of the package
- `appname`: Name of the application
- `version`: Version of the package

Public Members

Package **package**
The package name of the application. Typically this is in the form of $app.$developer so it could be my-app.my-name, though other formats do exist and are used in the wild.

In the case of legacy applications this will be the empty string.

AppName **appname**
The string that uniquely identifies the application. This comes from the package manifest. In a Click package this is the string that exists under the "hooks" key in the JSON manifest.

Version **version**
Version of the package that is installed. This is always resolved when creating the struct.

Note For snaps this is actually the 'revision' instead of the version since that is unique where 'version' is not.

Public Static Functions

AppID **parse**(**const** std::string &*appid*)
Parse a string and turn it into an *AppID*. This assumes that the string is in the form: __ and will return an empty *AppID* if not.

Parameters

- `appid`: String with the concatenated *AppID*

AppID **find**(**const** std::string &*sappid*)

Find is a more tollerant version of *parse()*, it handles legacy applications, short AppIDs ($package_$app) and other forms of that are in common usage. It can be used, but is slower than *parse()* if you've got well formed data already.

Note This will use the default registry instance, it is generally recommended to have your own instead of using the default.

Parameters

- `sappid`: String with the concatenated *AppID*

AppID **find**(**const** std::shared_ptr<*Registry*> &*registry*, **const** std::string &*sappid*)

Find is a more tollerant version of *parse()*, it handles legacy applications, short AppIDs ($package_$app) and other forms of that are in common usage. It can be used, but is slower than *parse()* if you've got well formed data already.

Parameters

- `registry`: *Registry* instance to use for persistant connections
- `sappid`: String with the concatenated *AppID*

bool **valid**(**const** std::string &*sappid*)

Check to see whether a string is a valid *AppID* string

Parameters

- `sappid`: String with the concatenated *AppID*

AppID **discover**(**const** std::string &*package*, *ApplicationWildcard* *appwildcard* = ApplicationWildcard::FIRST_LISTED, *VersionWildcard* *versionwildcard* = VersionWildcard::CURRENT_USER_VERSION)

Find the *AppID* for an application where you only know the package name.

Note This will use the default registry instance, it is generally recommended to have your own instead of using the default.

Parameters

- `package`: Name of the package
- `appwildcard`: Specification of how to search the manifest for apps
- `versionwildcard`: Specification of how to search for the version

AppID **discover**(**const** std::string &*package*, **const** std::string &*appname*, *VersionWildcard* *versionwildcard* = VersionWildcard::CURRENT_USER_VERSION)

Find the *AppID* for an application where you know the package name and application name.

Note This will use the default registry instance, it is generally recommended to have your own instead of using the default.

Parameters

- `package`: Name of the package
- `appname`: Name of the application
- `versionwildcard`: Specification of how to search for the version

AppID **discover**(**const** std::string &*package*, **const** std::string &*appname*, **const** std::string &*version*)
Create an *AppID* providing known strings of packages and names

Note This will use the default registry instance, it is generally recommended to have your own instead of using the default.

Parameters

- `package`: Name of the package
- `appname`: Name of the application
- `version`: Version of the package

AppID **discover**(**const** std::shared_ptr<*Registry*> &*registry*, **const** std::string &*package*, *ApplicationWildcard* *appwildcard* = ApplicationWildcard::FIRST_LISTED, *VersionWildcard* *versionwildcard* = VersionWildcard::CURRENT_USER_VERSION)
Find the *AppID* for an application where you only know the package name.

Parameters

- `registry`: *Registry* instance to use for persistant connections
- `package`: Name of the package
- `appwildcard`: Specification of how to search the manifest for apps
- `versionwildcard`: Specification of how to search for the version

AppID **discover**(**const** std::shared_ptr<*Registry*> &*registry*, **const** std::string &*package*, **const** std::string &*appname*, *VersionWildcard* *versionwildcard* = VersionWildcard::CURRENT_USER_VERSION)
Find the *AppID* for an application where you know the package name and application name.

Parameters

- `registry`: *Registry* instance to use for persistant connections
- `package`: Name of the package
- `appname`: Name of the application
- `versionwildcard`: Specification of how to search for the version

AppID **discover**(**const** std::shared_ptr<*Registry*> &*registry*, **const** std::string &*package*, **const** std::string &*appname*, **const** std::string &*version*)
Create an *AppID* providing known strings of packages and names

Parameters

- `registry`: *Registry* instance to use for persistant connections
- `package`: Name of the package
- `appname`: Name of the application
- `version`: Version of the package

Application

class ubuntu::app_launch::**Application**

Class to represent an application, whether running or not, and query more information about it.

Generally the *Application* object represents an *Application* in the system. It hooks up all of it's signals, finds out information about it and controls whether it is running or not. This class is what most users of Ubuntu App Launch will do the majority of their work.

Subclassed by *ubuntu::app_launch::app_impls::Base*

Public Functions

virtual ~Application()

virtual *AppID* **appId**() = 0

Get the *Application* ID of this *Application*

virtual std::shared_ptr<*Info*> **info**() = 0

Get a *Application::Info* object to describe the metadata for this application

virtual bool **hasInstances**() = 0

A quick check to see if this application has any running instances

virtual std::vector<std::shared_ptr<*Instance*>> **instances**() = 0

Get a vector of the running instances of this application

virtual std::shared_ptr<*Instance*> **launch**(**const** std::vector<URL> &*urls* = {}) = 0

Start an application, optionally with URLs to pass to it.

Parameters

- urls: A list of URLs to pass to the application command line

virtual std::shared_ptr<*Instance*> **launchTest**(**const** std::vector<URL> &*urls* = {}) = 0

Start an application with text flags, optionally with URLs to pass to it.

Parameters

- urls: A list of URLs to pass to the application command line

virtual std::shared_ptr<*Instance*> **findInstance**(**const** pid_t &*pid*) = 0

Get a a pointer to the running instances of this application based on the pid

Parameters

- pid: The pid to find the instance of

Public Static Functions

std::shared_ptr<*Application*> **create**(**const** *AppID* &*appid*, **const** std::shared_ptr<*Registry*> &*registry*)

Function to create an *Application* object. It determines the type of application and returns a pointer to that application object. It uses the registry for shared connections and is given an *AppID*. To find the *AppID* for a given application use the *AppID::discover()* functions.

Parameters

- `appid`: *Application* ID for the application
- `registry`: Shared registry to use

class `Info`

Information and metadata about the application for programs that are displaying the application to users.

The *Info* class has all the metadata including user visible strings and other nicities that users expect to see about applications. For most formats this is gotten from the Desktop file, but those may be in different locations depending on the packaging format.

Subclassed by *ubuntu::app_launch::app_info::Desktop*

Public Functions

virtual `~Info`()

virtual const Name &**`name`**() = 0
Name of the application

virtual const Description &**`description`**() = 0
Textual description of the application

virtual const IconPath &**`iconPath`**() = 0
Path to the icon that represents the application

virtual const DefaultDepartment &**`defaultDepartment`**() = 0
Default department of the application

virtual const IconPath &**`screenshotPath`**() = 0
Path to the screenshot of the application

virtual const Keywords &**`keywords`**() = 0
List of keywords for the application

virtual const Popularity &**`popularity`**() = 0
Get the relative popularity of the application, 0 is not popular

virtual *Splash* **`splash`**() = 0
Get information for the splash screen

virtual *Orientations* **`supportedOrientations`**() = 0
Return which orientations are supported

virtual RotatesWindow **`rotatesWindowContents`**() = 0
Return whether the window contents can be rotated or not

virtual UbuntuLifecycle **`supportsUbuntuLifecycle`**() = 0

struct `Orientations`
Orientation and placement

Public Functions

bool **`operator==`**(**const** *Orientations* &*b*) **const**
Check to see if two *Orientations* are the same

Public Members

bool **portrait**
Can support portrait

bool **landscape**
Can support landscape

bool **invertedPortrait**
Can support inverted portrait

bool **invertedLandscape**
Can support inverted landscape

struct Splash
Information to be shown on the app splash screen

Public Members

Title **title**
Title text on the screen

Image **image**
Image to put on the screen

Color **backgroundColor**
Color of the background

Color **headerColor**
Color of the header (if shown)

Color **footerColor**
Color of the footer

ShowHeader **showHeader**
Whether the standard UI Toolkit header should be shown

class Instance
Interface representing the information about a specific application running instance. This includes information on the PIDs that make up the *Application::Instance*.

Subclassed by *ubuntu::app_launch::jobs::instance::Base*

Public Functions

virtual ~Instance()

virtual bool **isRunning**() = 0
Check to see if the instance is currently running. The object can exist even after the instance has stopped running.

virtual pid_t **primaryPid**() = 0
Get the primary PID for this *Application::Instance*, this will return zero when it is not running. The primary PID is the PID keeping the instance alive, when it exists the others get reaped.

virtual bool **hasPid**(pid_t *pid*) = 0
Check to see if a PID is in the cgroup for this application instance. Each application instance tracks all the PIDs that are currently being used

virtual std::vector<pid_t> **pids**() = 0
Check to see if a specific PID is part of this *Application::Instance*

virtual void **setOomAdjustment**(**const** oom::Score *score*) = 0
Sets the value of the OOM Adjust kernel property for the all of the processes this instance.

virtual const oom::Score **getOomAdjustment**() = 0
Gets the value of the OOM Adjust kernel property for the primary process of this instance.

Note This function does not check all the processes and ensure they are consistent, it just checks the primary and assumes that.

virtual void **pause**() = 0
Pause, or send SIGSTOP, to the PIDs in this *Application::Instance*

virtual void **resume**() = 0
Resume, or send SIGCONT, to the PIDs in this *Application::Instance*

virtual void **stop**() = 0
Stop, or send SIGTERM, to the PIDs in this *Application::Instance*, if the PIDs do not respond to the SIGTERM they will be SIGKILL'd

virtual void **focus**() = 0
Signal the shell to focus the *Application::Instance*

Helper

class `ubuntu::app_launch::Helper`
Class representing an untrusted helper in the system. Untrusted helpers are used by trusted helpers to get some amount of functionality from a package in the system. Typically this is via a Click hook in a Click package.

In order to setup a untrusted helper the trusted helper needs to install a small executable that gives the equivallent of a Desktop Exec string to the system. This is done by installing the executable in the `/usr/lib/ /ubuntu-app-launch/$(helper type)/exec-tool`. A simple example can be seen in URL Dispatcher's URL Overlay helper. It is important to note that the helper will be confined with the apparmor profile associated with the *AppID* that is being used. For Click based applications this means that an untrusted helper should be its own stanza in the Click manifest with its own `apparmor` hook. This will configure the confinement for the helper.

Many times an untrusted helper runs in a non-user-facing mode, it is important that UAL **DOES NOT** implement a lifecycle for the helper. It is the responsibility of the trusted helper to do that. Many times this is a timeout or other similar functionality. These are the tools to implement those in a reasonable fashion (services don't have to worry about AppArmor, cgroups, or jobs) but it doesn't not implement them by itself.

Subclassed by *ubuntu::app_launch::helper_impls::Base*

Public Functions

virtual *AppID* **appId**() = 0
Get the *AppID* for this helper

virtual bool **hasInstances**() = 0
Check to see if there are any instances of this untrusted helper

virtual std::vector<std::shared_ptr<*Instance*>> **instances**() = 0
Get the list of instances of this helper

virtual std::shared_ptr<*Instance*> **launch**(std::vector<URL> *urls* = {}) = 0
Launch an instance of a helper with an optional set of URLs that get passed to the helper.

Parameters

- `urls`: List of URLs to passed to the untrusted helper

virtual std::shared_ptr<*Instance*> **launch**(MirPromptSession **session*, std::vector<URL> *urls* = {}) = 0
Launch an instance of a helper that is run in a Mir Trusted Prompt session. The session should be created by the trusted helper using the Mir function `mir_connection_create_prompt_session_sync()`.

Parameters

- `session`: Mir trusted prompt session
- `urls`: List of URLs to passed to the untrusted helper

Public Static Functions

std::shared_ptr<*Helper*> **create**(Type *type*, *AppID* *appid*, std::shared_ptr<*Registry*> *registry*)
Create a new helper object from an *AppID*

Parameters

- `type`: Type of untrusted helper
- `appid`: *AppID* of the helper
- `registry`: Shared registry instance

void **setExec**(std::vector<std::string> *exec*)
Set the exec from a helper utility. This function should only be used inside a helper exec util.

Parameters

- `exec`: The exec line to use for the helper with the *AppID* given

class Instance
Running instance of a a *Helper*

Subclassed by ubuntu::app_launch::helper_impls::BaseInstance

Public Functions

virtual bool **isRunning**() = 0
Check to see if this instance is running

virtual void **stop**() = 0
Stop a running helper

Registry

class `ubuntu::app_launch::`**Registry**

The application registry provides a central source for finding information about the applications in the system. This includes installed applications and running applications.

This class also holds onto shared resources for Ubuntu App Launch objects and functions. Generally speaking, there should only be one of them in the process. There are singleton functions, *getDefault()* and *clearDefault()*, which can be used to port applications from the old C API to the new C++ one but their use is discouraged.

Public Types

enum **FailureType**

Sometimes apps fail, this gives us information on why they failed.

Values:

CRASH

The application was running, but failed while running.

START_FAILURE

Something in the configuration of the application made it impossible to start the application

Public Functions

Registry()

~Registry()

void **clearManager**()

Remove the current manager on the registry

Public Static Functions

std::list<std::shared_ptr<*Application*>> **runningApps**(std::shared_ptr<*Registry*> *registry* = getDefault())

List the applications that are currently running, each will have a valid *Application::Instance* at call time, but that could change as soon as the call occurs.

Parameters

- `registry`: Shared registry for the tracking

std::list<std::shared_ptr<*Application*>> **installedApps**(std::shared_ptr<*Registry*> *registry* = getDefault())

List all of the applications that are currently installed on the system. Queries the various packaging schemes that are supported to get thier list of applications.

Parameters

- `registry`: Shared registry for the tracking

core::Signal<**const** std::shared_ptr<*Application*>&, **const** std::shared_ptr<*Application*::*Instance*>&> &**appStarted**(**const** std::share
&reg = get-De-fault ())

Get the signal object that is signaled when an application has been started.

Note This signal handler is activated on the UAL thread

Parameters

- `reg`: *Registry* to get the handler from

core::Signal<**const** std::shared_ptr<*Application*>&, **const** std::shared_ptr<*Application*::*Instance*>&> &**appStopped**(**const** std::share
&reg = get-De-fault ())

Get the signal object that is signaled when an application has stopped.

Note This signal handler is activated on the UAL thread

Parameters

- `reg`: *Registry* to get the handler from

core::Signal<**const** std::shared_ptr<*Application*>&, **const** std::shared_ptr<*Application*::*Instance*>&, *Registry*::*FailureType*> &**ap**

Get the signal object that is signaled when an application has failed.

Note This signal handler is activated on the UAL thread

Parameters

- `reg`: *Registry* to get the handler from

core::Signal<**const** std::shared_ptr<*Application*>&, **const** std::shared_ptr<*Application*::*Instance*>&, **const** std::vector<pid_t>&>

Get the signal object that is signaled when an application has been paused.

Note This signal handler is activated on the UAL thread

Parameters

- `reg`: *Registry* to get the handler from

core::Signal<**const** std::shared_ptr<*Application*>&, **const** std::shared_ptr<*Application*::*Instance*>&, **const** std::vector<pid_t>&>

Get the signal object that is signaled when an application has been resumed.

Note This signal handler is activated on the UAL thread

Parameters

- `reg`: *Registry* to get the handler from

core::Signal<**const** std::shared_ptr<*Application*>&> &**appInfoUpdated**(**const** std::shared_ptr<*Registry*> &*reg* = getDefault ())

Get the signal object that is signaled when an application's info has been updated.

Note This signal handler is activated on the UAL thread

Parameters

- `reg`: *Registry* to get the handler from

void **setManager**(**const** std::shared_ptr<*Manager*> &*manager*, **const** std::shared_ptr<*Registry*> &*registry*)

Set the manager of applications, which gives permissions for them to start and gain focus. In almost all cases this should be Unity8 as it will be controlling applications.

This function will failure if there is already a manager set.

Parameters

- `manager`: A reference to the *Manager* object to call
- `registry`: *Registry* to register the manager on

std::list<std::shared_ptr<*Helper*>> **runningHelpers**(*Helper*::Type *type*, std::shared_ptr<*Registry*> *registry* = getDefault ())

Get a list of all the helpers for a given helper type

Parameters

- `type`: *Helper* type string
- `registry`: Shared registry for the tracking

core::Signal<**const** std::shared_ptr<*Helper*>&, **const** std::shared_ptr<*Helper*::*Instance*>&> &**helperStarted**(*Helper*::Type *type*, **const** std::shared_pt &*reg* = getDefault())

Get the signal object that is signaled when helper has been started.

Note This signal handler is activated on the UAL thread

Parameters

- `type`: *Helper* type string
- `reg`: *Registry* to get the handler from

core::Signal<**const** std::shared_ptr<*Helper*>&, **const** std::shared_ptr<*Helper*::*Instance*>&> &**helperStopped**(*Helper*::Type *type*, **const** std::shared_pt &*reg* = getDefault())

Get the signal object that is signaled when a helper has stopped.

Note This signal handler is activated on the UAL thread

Parameters

- `type`: *Helper* type string
- `reg`: *Registry* to get the handler from

core::Signal<**const** std::shared_ptr<*Helper*>&, **const** std::shared_ptr<*Helper*::*Instance*>&, *Registry*::*FailureType*> &**helperFa**

Get the signal object that is signaled when a helper has failed.

Note This signal handler is activated on the UAL thread

Parameters

- `type`: *Helper* type string
- `reg`: *Registry* to get the handler from

std::shared_ptr<*Registry*> **getDefault** ()
Use the *Registry* as a global singleton, this function will create a *Registry* object if one doesn't exist. Use of this function is discouraged.

void **clearDefault** ()
Clear the default. If you're using the singleton interface in the *Registry::getDefault()* function you should call this as your service and/or tests exit to ensure you don't get Valgrind errors.

class Manager
The *Application Manager*, almost always if you're not Unity8, don't use this API. Testing is a special case. Subclass this interface and implement these functions.

Each function here is being passed a function object that takes a boolean to reply. This will accept or reject the request. The function object can be copied to another thread and executed if needed.

The reply is required for the application to start. It will block (not currently implemented) until approval is given. If there are multiple requests sent they may be replied out of order if desired.

Public Functions

virtual void **startingRequest** (**const** std::shared_ptr<*Application*> &*app*, **const** std::shared_ptr<*Application*::*Instance*> &*instance*, std::function<void) bool
> *reply* = 0*Application* wishes to startup

Note This signal handler is activated on the UAL thread
Parameters
- `app`: *Application* requesting startup
- `instance`: Instance of the app, always valid but not useful unless mulit-instance app.
- `reply`: Function object to reply if it is allowed to start

virtual void **focusRequest** (**const** std::shared_ptr<*Application*> &*app*, **const** std::shared_ptr<*Application*::*Instance*> &*instance*, std::function<void) bool
> *reply* = 0*Application* wishes to have focus. Usually this occurs when a URL for the application is activated and the running app is requested.

Note This signal handler is activated on the UAL thread
Parameters
- `app`: *Application* requesting focus
- `instance`: Instance of the app, always valid but not useful unless mulit-instance app.
- `reply`: Function object to reply if it is allowed to focus

virtual void **resumeRequest** (**const** std::shared_ptr<*Application*> &*app*, **const** std::shared_ptr<*Application*::*Instance*> &*instance*, std::function<void) bool
> *reply* = 0*Application* wishes to resume. Usually this occurs when a URL for the application is activated and the running app is requested.

Note This signal handler is activated on the UAL thread
Parameters
- `app`: *Application* requesting resume
- `instance`: Instance of the app, always valid but not useful unless mulit-instance app.
- `reply`: Function object to reply if it is allowed to resume

CHAPTER 3

Implementation Details

Application Implementation Base

class `ubuntu::app_launch::app_impls::`**`Base`**

Provides some helper functions that can be used by all implementations of application. Stores the registry pointer which everyone wants anyway.

Inherits from *ubuntu::app_launch::Application*

Subclassed by *ubuntu::app_launch::app_impls::Legacy*, *ubuntu::app_launch::app_impls::Libertine*, *ubuntu::app_launch::app_impls::Snap*

Public Functions

`Base`(**const** std::shared_ptr<*Registry*> &*registry*)

`~Base`()

bool **`hasInstances`**()

A quick check to see if this application has any running instances

std::string **`getInstance`**(**const** std::shared_ptr<app_info::*Desktop*> &*desktop*) **const**

Generates an instance string based on the clock if we're a multi-instance application.

virtual std::shared_ptr<*Application*::*Instance*> **`findInstance`**(**const** std::string &*instanceid*) = 0

std::shared_ptr<*Application*::*Instance*> **`findInstance`**(**const** pid_t &*pid*)

Get a a pointer to the running instances of this application based on the pid

Parameters

- `pid`: The pid to find the instance of

Protected Attributes

std::shared_ptr<*Registry*> **_registry**
Pointer to the registry so we can ask it for things

Protected Static Functions

std::list<std::pair<std::string, std::string>> **confinedEnv**(**const** std::string &*package*, **const** std::string &*pkgdir*)
Function to create all the standard environment variables that we're building for everyone. Mostly stuff involving paths.

Parameters

- `package`: Name of the package
- `pkgdir`: Directory that the package lives in

Application Implementation Legacy

class `ubuntu::app_launch::app_impls::Legacy`
Application Implementation for *Legacy* applications. These are applications that are typically installed as Debian packages on the base system. The standard place for them to put their desktop files is in /usr/share/applications though other directories may be used by setting the appropriate XDG environment variables. This implementation makes use of the GIO Desktop Appinfo functions which do caching of those files to make access faster.

AppIDs for legacy applications only include the Appname variable. Both the package and the version entries are empty strings. The appname variable is the filename of the desktop file describing the application with the ".desktop" suffix.

More info: https://specifications.freedesktop.org/desktop-entry-spec/latest/

Inherits from *ubuntu::app_launch::app_impls::Base*

Public Functions

Legacy(**const** *AppID*::AppName &*appname*, **const** std::shared_ptr<*Registry*> &*registry*)

AppID **appId**()
Get the *Application* ID of this *Application*

std::shared_ptr<*Application*::*Info*> **info**()
Get a *Application::Info* object to describe the metadata for this application

std::vector<std::shared_ptr<*Application*::*Instance*>> **instances**()
Get a vector of the running instances of this application

std::shared_ptr<*Application*::*Instance*> **launch**(**const** std::vector<*Application*::URL> &*urls* = {})
Create an UpstartInstance for this *AppID* using the UpstartInstance launch function.

Parameters

- `urls`: URLs to pass to the application

std::shared_ptr<*Application*::*Instance*> **launchTest** (**const** std::vector<*Application*::URL> *&urls* = {})

Create an UpstartInstance for this *AppID* using the UpstartInstance launch function with a testing environment.

Parameters

- `urls`: URLs to pass to the application

std::shared_ptr<*Application*::*Instance*> **findInstance** (**const** std::string *&instanceid*)

Private Functions

std::list<std::pair<std::string, std::string>> **launchEnv** (**const** std::string *&instance*)

Grabs all the environment for a legacy app. Mostly this consists of the exec line and whether it needs XMir. Also we set the path if that is specified in the desktop file. We can also set an AppArmor profile if requested.

Private Members

AppID::AppName **_appname**

std::string **_basedir**

std::shared_ptr<GKeyFile> **_keyfile**

std::shared_ptr<app_info::*Desktop*> **appinfo_**

std::string **desktopPath_**

std::regex **instanceRegex_**

Application Implementation Libertine

class `ubuntu::app_launch::app_impls::Libertine`

Application Implmentation for the *Libertine* container system. *Libertine* sets up containers that are read/write on a read only system, to all for more dynamic packaging systems (like deb) to work. This provides some compatibility for older applications or those who are only distributed in packaging systems requiring full system access.

Application IDs for *Libertine* applications have the package field as the name of the container. The appname is similar to that of the Legacy() implementation as the filename of the desktop file defining the application without the ".desktop" suffix. UAL has no way to know the version, so it is always hard coded to "0.0".

Libertine applications always are setup with XMir and started using the libertine-launch utility which configures the environment for the container.

More info: https://wiki.ubuntu.com/Touch/Libertine

Inherits from *ubuntu::app_launch::app_impls::Base*

Public Functions

Libertine(**const** *AppID*::Package &*container*, **const** *AppID*::AppName &*appname*, **const** std::shared_ptr<*Registry*> &*registry*)

AppID **appId**()
Get the *Application* ID of this *Application*

std::shared_ptr<*Application*::*Info*> **info**()
Get a *Application::Info* object to describe the metadata for this application

std::vector<std::shared_ptr<*Application*::*Instance*>> **instances**()
Get a vector of the running instances of this application

std::shared_ptr<*Application*::*Instance*> **launch**(**const** std::vector<*Application*::URL> &*urls* = {})

std::shared_ptr<*Application*::*Instance*> **launchTest**(**const** std::vector<*Application*::URL> &*urls* = {})

std::shared_ptr<*Application*::*Instance*> **findInstance**(**const** std::string &*instanceid*)

Private Functions

std::list<std::pair<std::string, std::string>> **launchEnv**()
Grabs all the environment variables for the application to launch in. It sets up the confinement ones and then adds in the APP_EXEC line and whether to use XMir.

This function adds 'libertine-launch' at the beginning of the Exec line with the container name as a parameter. The command can be overridden with the UBUNTU_APP_LAUNCH_LIBERTINE_LAUNCH environment variable.

Private Members

AppID::Package **_container**

AppID::AppName **_appname**

std::string **_container_path**

std::shared_ptr<GKeyFile> **_keyfile**

std::string **_basedir**

std::shared_ptr<app_info::*Desktop*> **appinfo_**

Private Static Functions

std::shared_ptr<GKeyFile> **keyfileFromPath**(**const** std::string &*pathname*)

std::shared_ptr<GKeyFile> **findDesktopFile**(**const** std::string &*basepath*, **const** std::string &*subpath*, **const** std::string &*filename*)

Application Implementation Snappy

class `ubuntu::app_launch::app_impls::`**Snap**

Class implementing a Applications that are installed in the system as Snaps. This class connects to snapd to get information on the interfaces of the installed snaps and sees if any of them are applicable to the user session. Currently that means if the command has the mir, unity8, unity7 or x11 interfaces.

For *Application* IDs snaps use a very similar scheme to Click packages. The package field is the name of the snap package, typically this is the overall application name. The appname is the command in the snap package, which needs to be associated with one of our supported interfaces and have a desktop file. Lastly the version field is actually the snap revision, this value changes even on updates between channels of the same version so it provides a greater amount of uniqueness.

Inherits from *ubuntu::app_launch::app_impls::Base*

Public Types

typedef std::tuple<app_info::*Desktop*::*XMirEnable*, *Application*::*Info*::UbuntuLifecycle> **InterfaceInfo**

Public Functions

Snap(**const** *AppID* &*appid*, **const** std::shared_ptr<*Registry*> &*registry*)

Uses the *findInterfaceInfo()* function to find the interface if we don't have one.

Parameters

- `appid`: *Application* ID of the snap
- `registry`: *Registry* to use for persistent connections

Snap(**const** *AppID* &*appid*, **const** std::shared_ptr<*Registry*> &*registry*, **const** *InterfaceInfo* &*interfaceInfo*)

Creates a *Snap* application object. Will throw exceptions if the *AppID* doesn't resolve into a valid package or that package doesn't have a desktop file that matches the app name.

Parameters

- `appid`: *Application* ID of the snap
- `registry`: *Registry* to use for persistent connections
- `interfaceInfo`: Metadata gleaned from the snap's interfaces

AppID **appId**()

Returns the stored *AppID*

std::shared_ptr<*Application*::*Info*> **info**()

Returns a reference to the info for the snap

std::vector<std::shared_ptr<*Application*::*Instance*>> **instances**()

Get all of the instances of this snap package that are running

std::shared_ptr<*Application*::*Instance*> **launch**(**const** std::vector<*Application*::URL> &*urls* = {})

Create a new instance of this *Snap*

Parameters

- `urls`: URLs to pass to the command

std::shared_ptr<*Application*::*Instance*> **launchTest**(**const** std::vector<*Application*::URL> &*urls* = {})

Create a new instance of this *Snap* with a testing environment setup for it.

Parameters

- `urls`: URLs to pass to the command

std::shared_ptr<*Application*::*Instance*> **findInstance**(**const** std::string &*instanceid*)

Public Static Functions

static std::list<std::shared_ptr<*Application*>> **list**(**const** std::shared_ptr<*Registry*> &*registry*)

Snap::*InterfaceInfo* **findInterfaceInfo**(**const** *AppID* &*appid*, **const** std::shared_ptr<*Registry*> &*registry*)

Asks Snapd for the interfaces to determine which ones the application can support.

Parameters

- `appid`: *Application* ID of the snap
- `registry`: *Registry* to use for persistent connections

bool **checkPkgInfo**(**const** std::shared_ptr<snapd::*Info*::*PkgInfo*> &*pkginfo*, **const** *AppID* &*appid*)

Checks a PkgInfo structure to ensure that it matches the *AppID*

Private Functions

std::list<std::pair<std::string, std::string>> **launchEnv**()

Return the launch environment for this snap. That includes whether or not it needs help from XMir (including *Libertine* helpers)

Private Members

AppID **appid_**

AppID of the *Snap*. Should be the name of the snap package. The name of the command. And then the revision.

std::shared_ptr<app_info::*Desktop*> **info_**

The app's displayed information. Should be from a desktop file that is put in ${SNAP_DIR}/meta/gui/${command}.desktop

std::shared_ptr<snapd::*Info*::*PkgInfo*> **pkgInfo_**

Information that we get from Snapd on the package

Application Info Desktop

class `ubuntu::app_launch::app_info::Desktop`
Inherits from *ubuntu::app_launch::Application::Info*

Subclassed by *ubuntu::app_launch::app_impls::SnapInfo*

Public Types

typedef *TypeTagger*<XMirEnableTag, bool> **XMirEnable**

typedef *TypeTagger*<ExecTag, std::string> **Exec**

typedef *TypeTagger*<SingleInstanceTag, bool> **SingleInstance**

Public Functions

Desktop (**const** *AppID* &*appid*, **const** std::shared_ptr<GKeyFile> &*keyfile*, **const** std::string &*basePath*, **const** std::string &*rootDir*, std::bitset<2> *flags*, std::shared_ptr<*Registry*> *registry*)

const *Application*::*Info*::Name &**name** ()
Name of the application

const *Application*::*Info*::Description &**description** ()
Textual description of the application

const *Application*::*Info*::IconPath &**iconPath** ()
Path to the icon that represents the application

const *Application*::*Info*::DefaultDepartment &**defaultDepartment** ()
Default department of the application

const *Application*::*Info*::IconPath &**screenshotPath** ()
Path to the screenshot of the application

const *Application*::*Info*::Keywords &**keywords** ()
List of keywords for the application

const *Application*::*Info*::Popularity &**popularity** ()
Get the relative popularity of the application, 0 is not popular

Application::*Info*::*Splash* **splash** ()
Get information for the splash screen

Application::*Info*::*Orientations* **supportedOrientations** ()
Return which orientations are supported

Application::*Info*::RotatesWindow **rotatesWindowContents** ()
Return whether the window contents can be rotated or not

Application::*Info*::UbuntuLifecycle **supportsUbuntuLifecycle** ()

virtual *XMirEnable* **xMirEnable** ()

virtual *Exec* **execLine** ()

virtual *SingleInstance* **singleInstance** ()

Protected Attributes

std::shared_ptr<GKeyFile> **_keyfile**

std::string **_basePath**

std::string **_rootDir**

Application::*Info*::Name **_name**

Application::*Info*::Description **_description**

Application::*Info*::IconPath **_iconPath**

Application::*Info*::DefaultDepartment **_defaultDepartment**

Application::*Info*::IconPath **_screenshotPath**

Application::*Info*::Keywords **_keywords**

Application::*Info*::Popularity **_popularity**

Application::*Info*::*Splash* **_splashInfo**

Application::*Info*::*Orientations* **_supportedOrientations**

Application::*Info*::RotatesWindow **_rotatesWindow**

Application::*Info*::UbuntuLifecycle **_ubuntuLifecycle**

XMirEnable **_xMirEnable**

Exec **_exec**

SingleInstance **_singleInstance**

Application Info Snap

class `ubuntu::app_launch::app_impls::`**SnapInfo**

Subclassing the desktop info object so that we can override a couple of properties with interface definitions. This may grow as we add more fields to the desktop spec that come from Snappy interfaces.

Inherits from *ubuntu::app_launch::app_info::Desktop*

Public Functions

SnapInfo(**const** *AppID* &*appid*, **const** std::shared_ptr<*Registry*> &*registry*, **const** *Snap*::*InterfaceInfo* &*interfaceInfo*, **const** std::string &*snapDir*)

Exec **execLine**()

Figures out the exec line for a snappy command. We're not using the Exec in the desktop file exactly, but assuming that it is kinda what we want to be run. So we're replacing that with the script, which we have to use as we can't get the command that is in the snap metadata as Snapd won't give it to us. So we're parsing the Exec line and replacing the first entry. Then putting it back together again.

Private Members

AppID **appId_**

AppID of snap

Application Icon Finder

class `ubuntu::app_launch::`**IconFinder**

Class to search for available application icons and select the best option.

This object attempts to find the highest resolution icon based on the freedesktop icon theme specification found at: https://standards.freedesktop.org/icon-theme-spec/icon-theme-spec-latest.html It parses the theme file for the hicolor theme and identifies all possible directories in the global scope and the local scope.

Public Functions

IconFinder(std::string *basePath*)

Create an *IconFinder*

Parameters

- `basePath`: the root directory to begin searching for themes

virtual ~IconFinder()

Application::*Info*::IconPath **find**(**const** std::string &*iconName*)

Find the optimal icon for the given icon name.

Finds an icon in the search paths that we have for this path

Parameters

- `iconName`: name of or path to application icon

Private Members

std::list<*ThemeSubdirectory*> **_searchPaths**

std::string **_basePath**

Private Static Functions

bool **hasImageExtension**(**const** char **filename*)

Check to see if this is an icon name or an icon filename

std::string **findExistingIcon**(**const** std::string &*path*, **const** std::string &*iconName*)

Check in a given path if there is an existing file in it that satifies our name

std::list<*IconFinder*::*ThemeSubdirectory*> **validDirectories**(**const** std::string &*themePath*, gchar **directory*, int *size*)

Create a directory item if the directory exists

std::list<*IconFinder*::*ThemeSubdirectory*> **addSubdirectoryByType**(std::shared_ptr<GKeyFile> *themefile*, gchar **directory*, **const** std::string &*themePath*)

Take the data in a directory stanza and turn it into an actual directory

std::list<*IconFinder*::*ThemeSubdirectory*> **searchIconPaths**(std::shared_ptr<GKeyFile> *themefile*, gchar **directories, **const** std::string &*themePath*)
Parse a theme file's various stanzas for each directory

std::list<*IconFinder*::*ThemeSubdirectory*> **themeFileSearchPaths**(**const** std::string &*themePath*)
Try to get theme subdirectories using .theme file in the given theme path if it exists

std::list<*IconFinder*::*ThemeSubdirectory*> **themeDirSearchPaths**(**const** std::string &*basePath*)
Look into a theme directory and see if we can use the subdirectories as icon folders. This is a fallback, and is sadly inefficient.

std::list<*IconFinder*::*ThemeSubdirectory*> **iconsFromThemePath**(**const** gchar **themeDir*)
Gets all search paths from a given theme directory via theme file or manually scanning the directory.

std::list<*IconFinder*::*ThemeSubdirectory*> **getSearchPaths**(**const** std::string &*basePath*)
Gets search paths based on common icon directories including themes and pixmaps.

struct ThemeSubdirectory

Public Members

std::string **path**

int **size**

Application Storage Base

class ubuntu::app_launch::app_store::**Base**
Inherits from ubuntu::app_launch::info_watcher::Base

Subclassed by *ubuntu::app_launch::app_store::Legacy*, *ubuntu::app_launch::app_store::Libertine*, *ubuntu::app_launch::app_store::Snap*

Public Functions

Base()

~Base()

virtual bool **verifyPackage**(**const** *AppID*::Package &*package*, **const** std::shared_ptr<*Registry*> &*registry*) = 0

virtual bool **verifyAppname**(**const** *AppID*::Package &*package*, **const** *AppID*::AppName &*appname*, **const** std::shared_ptr<*Registry*> &*registry*) = 0

virtual *AppID*::AppName **findAppname**(**const** *AppID*::Package &*package*, *AppID*::*ApplicationWildcard* *card*, **const** std::shared_ptr<*Registry*> &*registry*) = 0

virtual *AppID*::Version **findVersion**(**const** *AppID*::Package &*package*, **const** *AppID*::AppName &*appname*, **const** std::shared_ptr<*Registry*> &*registry*) = 0

virtual bool **hasAppId**(**const** *AppID* &*appid*, **const** std::shared_ptr<*Registry*> &*registry*) = 0

virtual std::list<std::shared_ptr<*Application*>> **list** (**const** std::shared_ptr<*Registry*> &*registry*) = 0

virtual std::shared_ptr<app_impls::*Base*> **create** (**const** *AppID* &*appid*, **const** std::shared_ptr<*Registry*> &*registry*) = 0

Public Static Functions

std::list<std::shared_ptr<*Base*>> **allAppStores** ()

Application Storage Legacy

class `ubuntu::app_launch::app_store::Legacy`
Inherits from *ubuntu::app_launch::app_store::Base*

Public Functions

Legacy ()

~Legacy ()

bool **verifyPackage** (**const** *AppID*::Package &*package*, **const** std::shared_ptr<*Registry*> &*registry*)
Ensure the package is empty

Parameters

- `package`: Container name
- `registry`: persistent connections to use

bool **verifyAppname** (**const** *AppID*::Package &*package*, **const** *AppID*::AppName &*appname*, **const** std::shared_ptr<*Registry*> &*registry*)
Looks for an application by looking through the system and user application directories to find the desktop file.

Parameters

- `package`: Container name
- `appname`: *Application* name to look for
- `registry`: persistent connections to use

AppID::AppName **findAppname** (**const** *AppID*::Package &*package*, *AppID*::*ApplicationWildcard* *card*, **const** std::shared_ptr<*Registry*> &*registry*)
We don't really have a way to implement this for *Legacy*, any search wouldn't really make sense. We just throw an error.

Parameters

- `package`: Container name
- `card`: *Application* search paths
- `registry`: persistent connections to use

AppID::Version **findVersion** (**const** *AppID*::Package &*package*, **const** *AppID*::AppName &*appname*, **const** std::shared_ptr<*Registry*> &*registry*)
Function to return an empty string

Parameters

- `package`: Container name (unused)
- `appname`: *Application* name (unused)
- `registry`: persistent connections to use (unused)

bool **hasAppId** (**const** *AppID* &*appid*, **const** std::shared_ptr<*Registry*> &*registry*)
Checks the *AppID* by ensuring the version and package are empty then looks for the application.

Parameters

- `appid`: *AppID* to check
- `registry`: persistent connections to use

std::list<std::shared_ptr<*Application*>> **list** (**const** std::shared_ptr<*Registry*> &*registry*)

std::shared_ptr<app_impls::*Base*> **create** (**const** *AppID* &*appid*, **const** std::shared_ptr<*Registry*> &*registry*)

Application Storage Libertine

class `ubuntu::app_launch::app_store::Libertine`
Inherits from *ubuntu::app_launch::app_store::Base*

Public Functions

Libertine ()

~Libertine ()

bool **verifyPackage** (**const** *AppID*::Package &*package*, **const** std::shared_ptr<*Registry*> &*registry*)
Verify a package name by getting the list of containers from liblibertine and ensuring it is in that list.

Parameters

- `package`: Container name
- `registry`: persistent connections to use

bool **verifyAppname** (**const** *AppID*::Package &*package*, **const** *AppID*::AppName &*appname*, **const** std::shared_ptr<*Registry*> &*registry*)
Gets the list of applications from the container using liblibertine and see if is in that list.

Parameters

- `package`: Container name
- `appname`: *Application* name to look for
- `registry`: persistent connections to use

AppID::AppName **findAppname**(**const** *AppID*::Package &*package*, *AppID*::*ApplicationWildcard* *card*, **const** std::shared_ptr<*Registry*> &*registry*)

We don't really have a way to implement this for *Libertine*, any search wouldn't really make sense. We just throw an error.

Parameters

- `package`: Container name
- `card`: *Application* search paths
- `registry`: persistent connections to use

AppID::Version **findVersion**(**const** *AppID*::Package &*package*, **const** *AppID*::AppName &*appname*, **const** std::shared_ptr<*Registry*> &*registry*)

Function to return "0.0"

Parameters

- `package`: Container name (unused)
- `appname`: *Application* name (unused)
- `registry`: persistent connections to use (unused)

bool **hasAppId**(**const** *AppID* &*appid*, **const** std::shared_ptr<*Registry*> &*registry*)

Checks the *AppID* by making sure the version is "0.0" and then calling *verifyAppname()* to check the rest.

Parameters

- `appid`: *AppID* to check
- `registry`: persistent connections to use

std::list<std::shared_ptr<*Application*>> **list**(**const** std::shared_ptr<*Registry*> &*registry*)

std::shared_ptr<app_impls::*Base*> **create**(**const** *AppID* &*appid*, **const** std::shared_ptr<*Registry*> &*registry*)

Application Storage Snap

class `ubuntu::app_launch::app_store::Snap`

Inherits from *ubuntu::app_launch::app_store::Base*

Public Functions

Snap()

~Snap()

bool **verifyPackage**(**const** *AppID*::Package &*package*, **const** std::shared_ptr<*Registry*> &*registry*)

Look to see if a package is a valid *Snap* package name

Parameters

- `package`: Package name

- `registry`: *Registry* to use for persistent connections

bool **verifyAppname**(**const** *AppID*::Package &*package*, **const** *AppID*::AppName &*appname*, **const** std::shared_ptr<*Registry*> &*registry*)

Look to see if an appname is a valid for a *Snap* package

Parameters

- `package`: Package name
- `appname`: Command name
- `registry`: *Registry* to use for persistent connections

AppID::AppName **findAppname**(**const** *AppID*::Package &*package*, *AppID*::*ApplicationWildcard* *card*, **const** std::shared_ptr<*Registry*> &*registry*)

Look for an application name on a *Snap* package based on a wildcard type.

Parameters

- `package`: Package name
- `card`: Wildcard to use for finding the appname
- `registry`: *Registry* to use for persistent connections

AppID::Version **findVersion**(**const** *AppID*::Package &*package*, **const** *AppID*::AppName &*appname*, **const** std::shared_ptr<*Registry*> &*registry*)

Look for a version of a *Snap* package

Parameters

- `package`: Package name
- `appname`: Not used for snaps
- `registry`: *Registry* to use for persistent connections

bool **hasAppId**(**const** *AppID* &*appid*, **const** std::shared_ptr<*Registry*> &*registry*)

Checks if an *AppID* could be a snap. Note it doesn't look for a desktop file just the package, app and version. This is done to make the lookup quickly, as this function can be used to select which backend to use and we want to reject quickly.

Parameters

- `appid`: *Application* ID of the snap
- `registry`: *Registry* to use for persistent connections

std::list<std::shared_ptr<*Application*>> **list**(**const** std::shared_ptr<*Registry*> &*registry*)

Lists all the Snappy apps that are using one of our supported interfaces. Also makes sure they're valid.

Parameters

- `registry`: *Registry* to use for persistent connections

std::shared_ptr<app_impls::*Base*> **create**(**const** *AppID* &*appid*, **const** std::shared_ptr<*Registry*> &*registry*)

Helper Implementation Base

class ubuntu::app_launch::helper_impls::**Base**
Inherits from *ubuntu::app_launch::Helper*

Public Functions

Base(**const** *Helper*::Type &*type*, **const** *AppID* &*appid*, **const** std::shared_ptr<*Registry*> &*registry*)

AppID **appId**()
Get the *AppID* for this helper

bool **hasInstances**()
Check to see if there are any instances of this untrusted helper

std::vector<std::shared_ptr<*Helper*::*Instance*>> **instances**()
Get the list of instances of this helper

std::shared_ptr<*Helper*::*Instance*> **launch**(std::vector<*Helper*::URL> *urls* = {})

std::shared_ptr<*Helper*::*Instance*> **launch**(MirPromptSession **session*, std::vector<*Helper*::URL> *urls* = {})

std::shared_ptr<*Helper*::*Instance*> **existingInstance**(**const** std::string &*instanceid*)
Find an instance that we already know the ID of

Private Functions

std::list<std::pair<std::string, std::string>> **defaultEnv**()
Sets up the executable environment variable based on the appid and the type of helper. We look for the exec-tool, but if we can't find it we're cool with that and we just execute the helper. If we do find an exec-tool we'll use that to fill in the parameters. For legacy appid's we'll allow the exec-tool to set everything.

Private Members

Helper::Type **_type**

AppID **_appid**

std::shared_ptr<*Registry*> **_registry**

Jobs Manager Base

class ubuntu::app_launch::jobs::manager::**Base**
Subclassed by ubuntu::app_launch::jobs::manager::SystemD

Public Functions

Base(**const** std::shared_ptr<*Registry*> &*registry*)

~Base()

virtual std::shared_ptr<*Application*::*Instance*> **launch**(**const** *AppID* &*appId*, **const** std::string &*job*, **const** std::string &*instance*, **const** std::vector<*Application*::URL> &*urls*, launchMode *mode*, std::function<std::list<std::pair<std::string, std::string>>) void > &*getenv* = 0

virtual std::shared_ptr<*Application*::*Instance*> **existing**(**const** *AppID* &*appId*, **const** std::string &*job*, **const** std::string &*instance*, **const** std::vector<*Application*::URL> &*urls*) = 0

std::list<std::shared_ptr<*Application*>> **runningApps**()
Get application objects for all of the applications based on the appids associated with the application jobs

std::list<std::shared_ptr<*Helper*>> **runningHelpers**(**const** *Helper*::Type &*type*)
Get application objects for all of the applications based on the appids associated with the application jobs

virtual std::list<std::string> **runningAppIds**(**const** std::list<std::string> &*jobs*) = 0

virtual std::vector<std::shared_ptr<instance::*Base*>> **instances**(**const** *AppID* &*appID*, **const** std::string &*job*) = 0

const std::list<std::string> &**getAllApplicationJobs**() **const**

core::Signal<**const** std::shared_ptr<*Application*>&, **const** std::shared_ptr<*Application*::*Instance*>&> &**appStarted**()

core::Signal<**const** std::shared_ptr<*Application*>&, **const** std::shared_ptr<*Application*::*Instance*>&> &**appStopped**()

core::Signal<**const** std::shared_ptr<*Application*>&, **const** std::shared_ptr<*Application*::*Instance*>&, *Registry*::*FailureType*> &**ap**

core::Signal<**const** std::shared_ptr<*Application*>&, **const** std::shared_ptr<*Application*::*Instance*>&, **const** std::vector<pid_t>&>
Grab the signal object for application paused. If we're not already listing for those signals this sets up a listener for them.

core::Signal<**const** std::shared_ptr<*Application*>&, **const** std::shared_ptr<*Application*::*Instance*>&, **const** std::vector<pid_t>&>
Grab the signal object for application resumed. If we're not already listing for those signals this sets up a listener for them.

core::Signal<**const** std::shared_ptr<*Helper*>&, **const** std::shared_ptr<*Helper*::*Instance*>&> &**helperStarted**(*Helper*::Type *type*)

core::Signal<**const** std::shared_ptr<*Helper*>&, **const** std::shared_ptr<*Helper*::*Instance*>&> &**helperStopped**(*Helper*::Type *type*)

core::Signal<**const** std::shared_ptr<*Helper*>&, **const** std::shared_ptr<*Helper*::*Instance*>&, *Registry*::*FailureType*> &**helperFa**

virtual core::Signal<**const** std::string&, **const** std::string&, **const** std::string&> &**jobStarted**() = 0

virtual core::Signal<**const** std::string&, **const** std::string&, **const** std::string&> &**jobStopped**() = 0

virtual core::Signal<**const** std::string&, **const** std::string&, **const** std::string&, *Registry*::*FailureType*> &**jobFailed**() = 0

void **setManager**(std::shared_ptr<*Registry*::*Manager*> *manager*)
Set the manager for the registry. This includes tracking the pointer as well as setting up the signals to call back into the manager. The signals are only setup once per registry even if the manager is cleared and changed again. They will just be no-op's in those cases.

void **clearManager**()
Clear the manager pointer

Public Static Functions

std::shared_ptr<*Base*> **determineFactory**(std::shared_ptr<*Registry*> *registry*)
Should determine which jobs backend to use, but we only have one right now.

Protected Attributes

std::weak_ptr<*Registry*> **registry_**
A link to the registry

std::list<std::string> **allApplicationJobs_**
A set of all the job names used by applications

std::shared_ptr<GDBusConnection> **dbus_**
The DBus connection we're connecting to

std::shared_ptr<*Registry*::*Manager*> **manager_**
Application manager instance

Private Functions

void **pauseEventEmitted**(core::Signal<**const** std::shared_ptr<*Application*>&, **const** std::shared_ptr<*Application*::*Instance*>&, **const** std::vector<pid_t>&> &*signal*, **const** std::shared_ptr<GVariant> &*params*, **const** std::shared_ptr<*Registry*> &*reg*)
Core handler for pause and resume events. Includes turning the GVariant pid list into a std::vector and getting the application object.

Private Members

core::Signal<**const** std::shared_ptr<*Application*>&, **const** std::shared_ptr<*Application*::*Instance*>&> **sig_appStarted**
Signal object for applications started

core::Signal<**const** std::shared_ptr<*Application*>&, **const** std::shared_ptr<*Application*::*Instance*>&> **sig_appStopped**
Signal object for applications stopped

core::Signal<**const** std::shared_ptr<*Application*>&, **const** std::shared_ptr<*Application*::*Instance*>&, *Registry*::*FailureType*> **sig_**
Signal object for applications failed

core::Signal<**const** std::shared_ptr<*Application*>&, **const** std::shared_ptr<*Application*::*Instance*>&, **const** std::vector<pid_t>&>
Signal object for applications paused

core::Signal<**const** std::shared_ptr<*Application*>&, **const** std::shared_ptr<*Application*::*Instance*>&, **const** std::vector<pid_t>&>
Signal object for applications resumed

std::map<std::string, std::shared_ptr<core::Signal<**const** std::shared_ptr<*Helper*>&, **const** std::shared_ptr<*Helper*::*Instance*>&>

std::map<std::string, std::shared_ptr<core::Signal<**const** std::shared_ptr<*Helper*>&, **const** std::shared_ptr<*Helper*::*Instance*>&>

std::map<std::string, std::shared_ptr<core::Signal<**const** std::shared_ptr<*Helper*>&, **const** std::shared_ptr<*Helper*::*Instance*>&,

ManagedDBusSignalConnection **handle_managerSignalFocus** = { DBusSignalUnsubscriber{}}
GDBus signal watcher handle for app focused signal

ManagedDBusSignalConnection **handle_managerSignalResume** = { DBusSignalUnsubscriber{}}
GDBus signal watcher handle for app resumed signal

ManagedDBusSignalConnection **handle_managerSignalStarting** = { DBusSignalUnsubscriber{}}
GDBus signal watcher handle for app starting signal

ManagedDBusSignalConnection **handle_appPaused** = { DBusSignalUnsubscriber{}}
GDBus signal watcher handle for app paused signal

ManagedDBusSignalConnection **handle_appResumed** = { DBusSignalUnsubscriber{}}
GDBus signal watcher handle for app resumed signal

std::once_flag **flag_managerSignals**
Variable to track to see if signal handlers are installed for the manager signals of focused, resumed and starting

std::once_flag **flag_appStarted**
Variable to track to see if signal handlers are installed for application started

std::once_flag **flag_appStopped**
Variable to track to see if signal handlers are installed for application stopped

std::once_flag **flag_appFailed**
Variable to track to see if signal handlers are installed for application failed

std::once_flag **flag_appPaused**
Variable to track to see if signal handlers are installed for application paused

std::once_flag **flag_appResumed**
Variable to track to see if signal handlers are installed for application resumed

Private Static Functions

std::tuple<std::shared_ptr<*Application*>, std::shared_ptr<*Application*::*Instance*>> **managerParams**(**const** std::shared_ptr<GVariant>
&*params*,
const
std::shared_ptr<*Registry*>
&*reg*)
Take the GVariant of parameters and turn them into an application and and instance. Easier to read in the smaller function

guint **managerSignalHelper**(**const** std::shared_ptr<*Registry*> &*reg*, **const** std::string &*signalname*, std::function<void) **const** std::shared_ptr<*Registry*> &reg, **const** std::shared_ptr<*Application*> &app, **const** std::shared_ptr<*Application*::*Instance*> &instance, **const** std::shared_ptr<GDBusConnection>&, **const** std::string&, **const** std::shared_ptr<GVariant>&

> *responsefunc*Register for a signal for the manager. All of the signals needed this same code so it got pulled out into a function. Takes the same of the signal, the registry that we're using and a function to call after we've messaged all the parameters into being something C++-ish.

Jobs Instance Base

class `ubuntu::app_launch::jobs::instance::Base`
Inherits from *ubuntu::app_launch::Application::Instance*

Subclassed by ubuntu::app_launch::jobs::instance::SystemD

Public Functions

Base(**const** *AppID* &*appId*, **const** std::string &*job*, **const** std::string &*instance*, **const** std::vector<*Application*::URL> &*urls*, **const** std::shared_ptr<*Registry*> &*registry*)

virtual ~Base()

bool **isRunning**()
Checks to see if we have a primary PID for the instance

bool **hasPid**(pid_t *pid*)
Looks at the PIDs in the instance cgroup and checks to see if is in the set.

Parameters

- `pid`: PID to look for

void **pause**()
Pauses this application by sending SIGSTOP to all the PIDs in the cgroup and tells Zeitgeist that we've left the application.

void **resume**()
Resumes this application by sending SIGCONT to all the PIDs in the cgroup and tells Zeitgeist that we're accessing the application.

void **focus**()
Focuses this application by sending SIGCONT to all the PIDs in the cgroup and tells the Shell to focus the application.

const std::string &**getInstanceId**()

void **setOomAdjustment**(**const** oom::Score *score*)
Sets the OOM adjustment by getting the list of PIDs and writing the value to each of their files in proc

Parameters

- `score`: OOM Score to set

const oom::Score **getOomAdjustment**()
Figures out the path to the primary PID of the application and then reads its OOM adjustment file.

Protected Functions

std::vector<pid_t> **forAllPids** (std::function<void) pid_t
> *eachPid*Go through the list of PIDs calling a function and handling the issue with getting PIDs being a racey condition.

Parameters

- `eachPid`: Function to run on each PID

Protected Attributes

const *AppID* **appId_**
Application ID

const std::string **job_**
Upstart job name

const std::string **instance_**
Instance ID environment value, empty if none

std::vector<*Application*::URL> **urls_**
The URLs that this was launched for. Only valid on launched jobs, we should look at perhaps changing that.

std::shared_ptr<*Registry*> **registry_**
A link to the registry we're using for connections

Protected Static Functions

void **pidListToDbus** (**const** std::shared_ptr<*Registry*> &*reg*, **const** *AppID* &*appid*, **const** std::string &*instanceid*, **const** std::vector<pid_t> &*pids*, **const** std::string &*signal*)
Send a signal that we've change the application. Do this on the registry thread in an idle so that we don't block anyone.

Parameters

- `pids`: List of PIDs to turn into variants to send
- `signal`: Name of the DBus signal to send

void **signalToPid** (pid_t *pid*, int *signal*)
Sends a signal to a PID with a warning if we can't send it. We could throw an exception, but we can't handle it usefully anyway

Parameters

- `pid`: PID to send the signal to
- `signal`: signal to send

void **oomValueToPid** (pid_t *pid*, **const** oom::Score *oomvalue*)
Writes an OOM value to proc, assuming we have a string in the outer loop

Parameters

- `pid`: PID to change the OOM value of
- `oomvalue`: OOM value to set

void **oomValueToPidHelper**(pid_t *pid*, **const** oom::Score *oomvalue*)
Use a setuid root helper for setting the oom value of Chromium instances

Parameters

- `pid`: PID to change the OOM value of
- `oomvalue`: OOM value to set

std::string **pidToOomPath**(pid_t *pid*)
Get the path to the PID's OOM adjust path, with allowing for an override for testing using the environment variable UBUNTU_APP_LAUNCH_OOM_PROC_PATH

Parameters

- `pid`: PID to build path for

GCharVUPtr **urlsToStrv**(**const** std::vector<*Application*::URL> &*urls*)
Reformat a C++ vector of URLs into a C GStrv of strings

Parameters

- `urls`: Vector of URLs to make into C strings

Registry Implementation

class `ubuntu::app_launch::Registry::Impl`
Private implementation of the *Registry* object.

Public Functions

Impl(*Registry* &*registry*)

Impl(*Registry* &*registry*, std::list<std::shared_ptr<app_store::*Base*>> *appStores*)

virtual ~Impl()

void **clearManager**()

std::shared_ptr<*IconFinder*> **getIconFinder**(std::string *basePath*)

void **zgSendEvent**(*AppID* *appid*, **const** std::string &*eventtype*)
Send an event to Zietgeist using the registry thread so that the callback comes back in the right place.

const std::string &**oomHelper**() **const**

core::Signal<**const** std::shared_ptr<*Application*>&> &**appInfoUpdated**(**const** std::shared_ptr<*Registry*> &*reg*)

std::list<std::shared_ptr<app_store::*Base*>> **appStores**()

void **setAppStores** (std::list<std::shared_ptr<app_store::*Base*>> &*newlist*)

Public Members

GLib::ContextThread **thread**
Shared context thread for events and background tasks that UAL subtasks are doing

std::shared_ptr<GDBusConnection> **_dbus**
DBus shared connection for the session bus

snapd::*Info* **snapdInfo**
Snapd information object

std::shared_ptr<*jobs*::manager::Base> **jobs**

Public Static Functions

static void **setManager** (**const** std::shared_ptr<*Registry*::*Manager*> &*manager*, **const** std::shared_ptr<*Registry*> &*registry*)

std::string **printJson** (std::shared_ptr<JsonObject> *jsonobj*)
Helper function for printing JSON objects to debug output

std::string **printJson** (std::shared_ptr<JsonNode> *jsonnode*)
Helper function for printing JSON nodes to debug output

void **watchingAppStarting** (bool *rWatching*)
Variable to track if this program is watching app startup so that we can know to not wait on the response to that.

bool **isWatchingAppStarting** ()
Accessor for the internal variable to know whether an app is watching for app startup

static std::shared_ptr<info_watcher::Zeitgeist> **getZgWatcher** (**const** std::shared_ptr<*Registry*> &*reg*)

Protected Attributes

std::shared_ptr<info_watcher::Zeitgeist> **zgWatcher_**
ZG Info Watcher

std::once_flag **zgWatcherOnce_**
Init checker for ZG Watcher

Private Members

Registry &**_registry**
The *Registry* that we're spawned from

std::shared_ptr<ZeitgeistLog> **zgLog_**
Shared instance of the Zeitgeist Log

std::unordered_map<std::string, std::shared_ptr<*IconFinder*>> **_iconFinders**
All of our icon finders based on the path that they're looking into

std::string **oomHelper_**
Path to the OOM *Helper*

std::list<std::shared_ptr<app_store::*Base*>> **_appStores**
Application stores

core::Signal<**const** std::shared_ptr<*Application*>&> **sig_appInfoUpdated**
Signal for application info changing

std::once_flag **flag_appInfoUpdated**
Flag to see if we've initialized the info watcher list

std::list<std::pair<std::shared_ptr<info_watcher::Base>, core::ScopedConnection>> **infoWatchers_**
List of info watchers along with a signal handle to our connection to their update signal

Snapd Info

class `ubuntu::app_launch::snapd::Info`
Class that implements the connection to Snapd allowing us to get info from it in a C++ friendly way.

Public Functions

Info()
Initializes the info object which mostly means checking what is overridden by environment variables (mostly for testing) and making sure there is a snapd socket available to us.

virtual ~Info()

std::shared_ptr<*Info*::*PkgInfo*> **pkgInfo**(**const** *AppID*::Package &*package*) **const**
Gets package information out of snapd by using the REST interface and turning the JSON object into a C++ Struct

Parameters

- `package`: Name of the package to look for

std::set<*AppID*> **appsForInterface**(**const** std::string &*interface*) **const**
Gets all the apps that are available for a given interface. It asks snapd for the list of interfaces and then finds this one, turning it into a set of AppIDs

Parameters

- `in_interface`: Which interface to get the set of apps for

std::set<std::string> **interfacesForAppId**(**const** *AppID* &*appid*) **const**
Finds all the interfaces for a specific appid

Parameters

- `appid`: *AppID* to search for

Private Functions

std::shared_ptr<JsonNode> **snapdJson**(**const** std::string *&endpoint*) **const**
Asks the snapd process for some JSON. This function parses the basic response JSON that snapd returns and will error if a return code error is in the JSON. It then passes on the "result" part of the response to the caller.

Parameters

- `endpoint`: End of the URL to pass to snapd

void **forAllPlugs**(std::function<void) JsonObject *plugobj
> *plugfunc* **const**Looks through all the plugs in the interfaces and runs a function based on them. Avoids pulling objects out of the parsed JSON structure from Snappy and making sure they have the same lifecycle as the parser object which seems to destroy them when it dies.

Parameters

- `plugfunc`: Function to execute on each plug

Private Members

std::string **snapdSocket**
Path to the socket of snapd

std::string **snapBasedir**
Directory to use as the base for all snap packages when making paths. This can be overridden with UBUNTU_APP_LAUNCH_SNAP_BASEDIR

bool **snapdExists** = false
Result of a check at init to see if the socket is available. If not all functions will return null results.

struct PkgInfo
Information that we can get from snapd about a package

Public Members

std::string **name**
Name of the package

std::string **version**
Version string provided by the package

std::string **revision**
Numerical always incrementing revision of the package

std::string **directory**
Directory that the snap is uncompressed into

std::set<std::string> **appnames**
List of appnames in the snap

Type Tagger

template <typename *Tag*, typename *T*>
class `ubuntu::app_launch::`**TypeTagger**

A small template to make it clearer when special types are being used.

The *TypeTagger* a small piece of C++ so that we can have custom types for things in the Ubuntu App Launch API that should be handled in special ways, but really have basic types at their core. In this way there is explicit code to convert these items into their fundamental type so that is obvious and can be easily searched for.

Public Functions

const T &**value**() **const**

Getter to get the fundamental type out of the *TypeTagger* wrapper

operator T() **const**

Getter to get the fundamental type out of the *TypeTagger* wrapper

bool **operator==**(**const** *TypeTagger*<Tag, T> &*b*) **const**

bool **operator==**(**const** T &*b*) **const**

~TypeTagger()

Public Static Functions

static *TypeTagger*<Tag, T> **from_raw**(**const** T &*value*)

Function to build a *TypeTagger* object from a fundamental type

Private Functions

TypeTagger(**const** T &*value*)

Private constructor used by *from_raw()*

Private Members

T **_value**

The memory allocation for the fundamental type

CHAPTER 4

Quality

Merge Requirements

This documents the expections that the project has on what both submitters and reviewers should ensure that they've done for a merge into the project.

Submitter Responsibilities

- Ensure the project compiles and the test suite executes without error
- Ensure that non-obvious code has comments explaining it

Reviewer Responsibilities

- Did the Jenkins build compile? Pass? Run unit tests successfully?
- Are there appropriate tests to cover any new functionality?
- **If this MR effects application startup:**
  - Run test case: ubuntu-app-launch/click-app
  - Run test case: ubuntu-app-launch/legacy-app
  - Run test case: ubuntu-app-launch/secondary-activation
- **If this MR effect untrusted-helpers:**
  - Run test case: ubuntu-app-launch/helper-run

Index

U