scientiamobile

"The first step in a great mobile experience"

INFUZE C++ WRAPPER USER GUIDE

Support

The ScientiaMobile Enterprise Support Portal is open to all WURFL users, both commercial license holders and evaluation users. It represents the combined knowledge base for the WURFL community. Commercial licensees are invited to post questions in the forum using the account to which their licenses are associated. This may mean faster handling of those posts by ScientiaMobile's personnel.

For commercial license holders, there are tiered support levels to address a variety of business support needs. After logging into your account, commercial licensees with support options can access the Enterprise Support portal to post tickets. These tickets will receive expedited attention.

To inquire about support plans, use our License Inquiry or our General Inquiry form.

Update Notifications

If you would like to be notified of our API updates, major data updates, and other technical changes, please subscribe to our ScientiaMobile Announcements list

scientiamobile

www.scientiamobile.com
Tel +1.703.310.6650
E-mail: sales@scientiamobile.com

WURFL InFuze C++ API Wrapper: User Guide

Introduction

The WURFL InFuze C++ Wrapper is a single HPP header file encapsulating the WURFL InFuze C API in C++ classes. This provides a handy and intuitive OOP interface for WURFL InFuze.

Supported Platforms

The C++ Wrapper is supported on the same platforms as the WURFL InFuze C API, providing a (C++98 or above) C++ compiler. Among these, there are multiple Linux distros such as Ubuntu, CentOS, RedHat, Fedora, and FreeBSD. Other supported operating systems include Windows and Mac OS X.

Installation

Being a header-only package, no explicit installation is needed. You just need to add:

```
#include <wurfl/wurfl.hpp>
```

to begin using the InFuze C++ Wrapper in your code. The wrapper relies on a proper installation of WURFL InFuze.

Basic Usage

Here is a quick code sample you can run to get started with the C++ Wrapper:

```
#include <wurfl/wurfl.hpp>
#include <iostream>

int main(int argc, char **argv){
  // declare a Builder and set a data file as root
  wurfl::Builder builder;
  builder.setRoot("/usr/share/wurfl/wurfl.zip");

  // let the Builder build a Manager
  wurfl::Manager manager(builder.build());

  // use the Manager to obtain a Device by looking up a User Agent string
  wurfl::Device device = manager.lookup("Mozilla/5.0 (Linux; Android 5.0; SAMSUNG SM-G925 Build/LRX21V) Appl
eWebKit/537.36 (KHTML, like Gecko) SamsungBrowser/4.0 Chrome/44.0.2403.133 Mobile Safari/537.36");

  // retrieve Device ID
  std::cout << "device: " << device.id() << std::endl;

  // retrieve a static and virtual capability
  std::cout << "is_console: " << device.capability("is_console") << std::endl;
  std::cout << "form_factor: " << device.virtualCapability("form_factor") << std::endl; // NOTE: virtual capabilities
are calculated at runtime!
}
```

There is no need to call the XXX_destroy() method. Destruction of the Builder, Manager, and Device objects are handled automatically by the C++ destructor mechanism when the instances go out of scope.

As you will see, the Manager object wraps wurfl_XXX() InFuze C API calls, while the Device wraps the wurfl_device_XXX() InFuze C API calls. The Builder is a utility class introduced in the C++ layer which wraps an engine configuration and creation. All required calls are conveniently issued in order in the Build::build() method, after you have configured your engine with all the desired Builder::setXXX() calls.

More on Lookup

Looking up a single user agent string is a basic use case. More realistic scenarios call for look-ups using a user-defined HTTP headers retrieval callback function - similar to the WURFL C API wurfl_lookup() call:

```
const char *lookup_callback(const char *header_name, const void *headers_data)
{
   ...
   your code that receives header data in the "headers_data" parameter,
   receives the requested header name in the "header_name" parameter
   and returns the value of the requested header
   ...
}

...

// declare a Builder and set a data file as root
wurfl::Builder builder;
builder.setRoot("/tmp/wurfl.zip");

// let the Builder build a Manager
wurfl::Manager manager(builder.build());

// use the Manager to obtain a Device by looking up header values passed with a callback function
wurfl::Device device = manager.lookup(&lookup_callback, your_headers_data)
```

If you choose to roll out your own user-defined HTTP header retrieval callback, you should perform a case-insensitive comparison on header names, and/or verify if your scenario supplies header names in a different case, rather than the standard one expected by WURFL.

Return values

While the WURFL C API call typically returns integer error codes (i.e.,wurfl_error), the C++ WURFL API Wrapper uses C++ exception mechanisms to report usage failures:

Exception Thrown	Source of the Exception	Reason for the Exception
std::logic_error	private Manager::setXXX() methods called from Builder::build()	Configuration parameters values supplied to Builder setXXX() methods are invalid. On Builder::build() call, Manager creation fails.
std::runtime_error	Manager and Device query methods	the query cannot be satisfied (wrong device id, unexistent cap/vcap, etc)

The underlying WURFL C API error message can be retrieved by calling thewhat() method of the std::exception being thrown. It is up to the client code to correctly wrap the WURFL creation/query code in adequate try/catch constructs.

The WURFL Updater

Since InFuze 1.8.3.0, a native internal Updater Module is available.

The WURFL Updater will automatically keep your data file up-to-date with ScientiaMobile's data release schedule. It handles all download and reload operations - even in multithreaded, multiprocess scenarios, along with optional logging of operations, network errors, etc.

While the WURFL InFuze engine construction calls are completely hidden in theBuilder class, we decided to expose the updater functionalities both in the Builder and in the Manager interfaces. This is because one could want a completely configured and working updater since the Manager creation, or he/she could prefer to configure and/or start the updater later. To create a Manager instance with a pre-configured background updater, you can do something like this:

```
wurfl::Builder builder;
builder.setRoot("/tmp/wurfl.zip");
```

```
builder.setUpdaterDataURL("your_personal_WURFL_Snapshot_URL");
builder.setUpdaterDataFrequency(WURFL_UPDATER_FREQ_DAILY);
builder.setUpdaterLogPath("wurfl_cpp_updater.log");
builder.updaterStart();

wurfl::Manager manager(builder.build());
```

Or, if you prefer to build a Manager instance without an updater, and eventually configure and start it later:

```
wurfl::Builder builder;
builder.setRoot(wurflFullDataFile);
wurfl::Manager manager(builder.build());

// ...later:
manager.setUpdaterDataURL("your_personal_WURFL_Snapshot_URL");
manager.setUpdaterDataFrequency(WURFL_UPDATER_FREQ_DAILY);
manager.setUpdaterLogPath("wurfl_cpp_updater.log");
manager.updaterStart();
```

You can check for detailed updater operations in the log file set with the setUpdaterLogPath() call. Logging is not mandatory but highly recommended, and the best way to troubleshoot network problems and the like.

Please note that the only mandatory call for the updater module to work is setUpdaterDataURL(), where you set your personal WURFL Snapshot URL (located in your license account page). This, in turn, is dependent on a successful setRoot() call:

- The WURFL data file, and the path specified in the setRoot() call, **MUST** have write/rename access. The old data file will be replaced (i.e, a rename operation will be performed on it) with the updated version upon successful update completion, and the directory where it resides will be used for remote file download, etc.
- ScientiaMobile does not distribute uncompressed XML data files via the updater. This means that if you plan to use this feature, you **MUST** use a compressed (i.e, a ZIP or a XML.GZ) file as data root in the setRoot() call.

Please note that setUpdaterDataFrequency() sets the frequency of Updater checks for the data file, not how often the engine data file is actually updated.

The WURFL InFuze Updater functionality relies on the presence and features of the curl command-line utility. A check for correct curl installation on the system being used is done in the setUpdaterDataURL() call.

WURFL InFuze C++ API Wrapper Reference

class wurfl::Builder

The Builder class abstracts the configuration and the construction of a WURFL engine.

Method	Description
Builder()	Default constructor. Creates a Builder instance set to InFuze defaults.
Builder& setRoot(path_to_root_xml)	Sets the root WURFL data file to be used by WURFL InFuze to a specific path in your file system. This call is mandatory in order to obtain a working Manager from a build() call.

Method	Description
Builder& addPatch(path_to_patch_xml)	Adds a patch to WURFL by taking the path to the patch xml file.
Builder& addPatches(begin_iterator, end_iterator)	Adds several patches paths to WURFL by specifying the begin and end iterators of a paths string collection.
Builder& addPatches(collection)	Adds a whole collection of patches' paths to WURFL.
void clearPatches()	Clears all the patches' paths stored in the Builder instance. Useful if you want to build several engines reusing the same Builder instance.
Builder& addCapability(capability_name)	Adds a capability to the capabilities filter. If unused, all capabilities are loaded.
Builder& addCapabilities(begin_iterator, end_iterator)	Adds a collection of capabilities to be filtered by WURFL by specifying the begin and end iterators of a capabilities names collection.
Builder& addCapabilities(collection)	Adds a whole collection of filtered capabilities names to WURFL.
void clearCapabilities()	Clears all the filtered capabilities names stored in the Builder instance. Useful if you want to build several engines reusing the same Builder instance.
Builder& setCacheProvider(wurfl_cache_provider, const char*)	Sets the WURFL Cache provider to be used: WURFL_CACHE_PROVIDER_NONE or WURFL_CACHE_PROVIDER_LRU (default). Depending on the cache provider, a cache configuration string can also be specified (see also WURFL C API documentation).
DEPRECATED std::list<std::string> mandatoryCapabilities()	Returns a temporary list of mandatory capabilities names. Now useless and deprecated, mandatory capabilities are always silently added in case of capability filtering.
DEPRECATED std::list<std::string> virtualCapabilities()	Returns a temporary list of virtual capabilities names. Has been deprecated in favor of getVirtualCapabilities()
const std::vector<std::string>& getVirtualCapabilities()	Returns a cached const vector of virtual capabilities names. Please use this instead of the deprecated virtualCapabilities() method.
Builder& setUpdaterLogPath(log_file_path)	Instruct the internal WURFL InFuze Updater to log any operation/error to the named file. If not used, the updater will not log anything.

Method	Description
Builder& setUpdaterDataURL(url)	Set remote data file URL to be downloaded via internal WURFL Infuze Updater. This is the only mandatory call if you want to use the InFuze Updater.
Builder& setUpdaterDataFrequency(wurfl_updater_frequency)	Sets how often the updater checks for any new/updated WURFL data file to be downloaded and used by the engine: WURFL_UPDATER_FREQ_DAILY (default) or WURFL_UPDATER_FREQ_WEEKLY.
Builder& updaterStart()	Tells the Builder to start the background updater thread on the Manager instance as soon as is built and returned.
Manager build()	Builds and returns a Manager instance configured with the previously selected options(engine target, cache, root data file, updater options etc).

class wurfl::Manager

The Manager class abstract the runtime usage of a WURFL engine. Amongst its responsibilities, the most important one is to perform lookups on user agent strings, or more generally on request headers, in order to identify the associated devices. Each Manager instance also has a built-in updater instance, by which both synchronous or asynchronous updates of the WURFL data file can be performed.

Method	Description
Manager(const Manager&)	Copy Constructor. The creation of a new Manager instance is to be done by copy construction from the return of a Builder instance build() call. No Manager default constructor is provided. Please see code examples above.
wurfl_handle getHandle()	Returns the underlying wrapped WURFL handle.
const char* info()	Returns a string describing the loaded WURFL data file and optional patch files.
const char* loadTime()	Returns the timestamp of the latest successful WURFL load/update.
DEPRECATED std::list<std::string> virtualCapabilities()	Returns a temporary list of virtual capabilities names. Has been deprecated in favor of getVirtualCapabilities()
const std::vector<std::string>& getVirtualCapabilities()	Returns a cached const vector of virtual capabilities names. Please use this instead of the deprecated virtualCapabilities() method.

Method	Description
DEPRECATED std::list<std::string> capabilities()	Returns a temporary list of supported capabilities names. Has been deprecated in favor of getCapabilities()
const std::vector<std::string>& getCapabilities()	Returns a cached const vector of supported capabilities names. Please use this instead of the deprecated capabilities() method.
DEPRECATED std::list<std::string> deviceIds()	Returns a temporary list of supported device identifiers. Has been deprecated in favor of getDeviceIds()
const std::vector<std::string>& getDeviceIds()	Returns a cached const vector of supported device identifiers names. Please use this instead of the deprecated deviceIds() method.
Device lookup(const char* useragent)	Query WURFL for a device matching the given user agent string. It returns a Device instance.
Device lookup(wurfl_header_retrieve_callback, const void*)	Query WURFL passing a header-retrieval user defined callback. It returns a Device instance.
DEPRECATED Device lookup(const std::map<std::string, std::string> &headers)	Query WURFL passing a std::map of headers key-value pairs. It returns a Device instance. It has been deprecated in favor of the usage of the lookup(const HeadersMap&) call because std::map has case-sensitive value retrieval (you should stay case-insensitive when working with HTTP headers)
Device lookup(wurfl::HeadersMap &headers)	Query WURFL passing a helper key-case-insensitive std::map wrapper of headers key-value pairs. It returns a Device instance
Device device(const char*)	Returns a Device instance given the WURFL device ID.
bool hasCapability(const char* capabilityName)	Checks if the given name identifies a capability.
bool hasVirtualCapability(const char* virtualCapabilityName)	Checks if the given name identifies a virtual capability.
Manager& setUpdaterLogPath(const std::string& log_file_path)	Instruct the internal WURFL Infuze updater to log to file any operation/error. If unused, the updater will not log anything.
Manager& setUpdaterDataURL(const std::string& url)	Set remote data file URL to be downloaded via internal WURFL InFuze Updater. This is the only MANDATORY call if you want to use the InFuze Updater.

Method	Description
Manager& setUpdaterDataFrequency(wurfl_updater_frequency)	Sets how often the updater checks for any new/updated WURFL data file to be downloaded and used by the engine: WURFL_UPDATER_FREQ_DAILY or WURFL_UPDATER_FREQ_WEEKLY.
void updaterStart()	Starts the background, non-blocking updater thread.
void updaterStop()	Stops the background, non-blocking updater thread.
void updaterRunOnce()	Starts a foreground, blocking update synchronous operation.

class wurfl::Device

The Device class abstracts a device retrieved by WURFL via lookup or direct device ID search. It encapsulates the wurfl_device_XXX() C API calls, exposing several methods to query a Device instance for its capabilities and state.

Method	Description
wurfl_device_handle getHandle()	Returns the underlying wrapped WURFL handle.
const char *capability(const char *cap_name)	Queries the Device instance for the value of the requested capability.
const char *virtualCapability(const char *vcap_name)	Queries the Device instance for the value of the requested virtual capability.
const char *userAgent()	Queries the Device instance for the value of the original user agent string used in the lookup.
const char *normalizedUserAgent()	Queries the Device instance for the value of the user agent string used in the lookup after the normalization applied by WURFL engine.
const char *id()	Queries the Device instance for his WURFL device ID string.
const char *rootId()	Queries the Device instance for the device ID string of his root device.
bool isActualDeviceRoot()	Tests if the Device instance represents a root device.