KMS Developer's Guide

**[Introduction](#heading=h.3ryfxms6dayy) 2[Development tools](#heading=h.evxbn7317x90) 2[Source code repositories](#heading=h.murx7u1qjfn7) 2**[Repository dependency graph](#heading=h.6h0i3bcykmlq) 4**[Development 101](#heading=h.jqa0hhcyywp7) 5**[Libraries](#heading=h.nlivqna0fsab) 5[Debian packages](#heading=h.fjdhtiab3ilh) 5[Build tools](#heading=h.vsdd0wllpcuc) 5**[Working with KMS sources](#heading=h.b48slpbonsev) 6**[Developing KMS](#heading=h.f2cqt4qsssv1) 7[Install development tools](#heading=h.dd3tsf9e0rco) 7[Install development libraries for KMS](#heading=h.8ihzsa4iwudv) 7[Install KMS fork libraries](#heading=h.609k0arnxeht) 7[Download KMS](#heading=h.6mgvkss6vcta) 8[Build KMS](#heading=h.mcgi08vkzoz5) 8[Launch KMS](#heading=h.z7g2qrjjpvra) 8[Build and run KMS tests](#heading=h.tg1iekgks85v) 9[Using the CLion IDE](#heading=h.au654pso4kp1) 9[Working on a forked library](#heading=h.lucy3kkfe71k) 9[Full cycle](#heading=h.9mrjfisoj5lq) 9[In-place linking](#heading=h.qpckhfxalhu0) 10[Generating Debian packages](#heading=h.yt2yfwfwgpyk) 10[Example: kms-core](#heading=h.8azw348hgf6) 10[Dependency resolution: to repo or not to repo](#heading=h.oaz0uphd5fa2) 11[Package generation script](#heading=h.ytrz089jccs2) 12[Building KMS on Ubuntu 14.04 (Trusty)](#heading=h.btau06yrmy3l) 12**[How-to’s](#heading=h.tpj61rxz4scz) 13**[How to add or update an external library to kurento](#heading=h.i3ieir1ipc31) 13[How to add a new fork library to kurento](#heading=h.d9utjhog5oi4) 13**[Known problems](#heading=h.iialgwayrtbg) 14**

# Introduction

Kurento is a multimedia framework that eases the task of building multimedia applications with the following features:

* **Dynamic WebRTC Media pipelines:** Kurento allows custom media pipelines connected to WebRTC peers like web browsers and mobile apps. These media pipelines can be composed by players, recorders, mixers, etc. and can be changed dynamically when the media is flowing.
* **Client/Server Architecture:** Apps developed with Kurento follow a client/server architecture. Kurento Media Server (KMS) is the server and offers a WebSocket interface implementing the Kurento Protocol, which allows Client Applications to define pipeline topologies.
* **Java and JavaScript Client Applications:** The typical use case of a KMS deployment consists of a three-layer architecture, where the user's browser interacts with the KMS server by means of an intermediate Client Application. There are several official Kurento Client Libraries, supporting the use of Java and JavaScript for the Client Applications. Clients for other languages can be easily implemented following the WebSocket protocol.
* **Third party Modules:** KMS has a pluggable architecture and allows third parties to implement modules that can be combined with other built-in or third party modules in the same pipeline. For example, there are modules for Computer Vision with features such as face detection, barcode reading, etc.

This document is a high level explanation of how to become a KMS developer. Development of Kurento Client Applications is out of the scope for this document, and won't be explained here.

# Development tools

This is an overview of the tools and technologies used by KMS:

* The code is written in C and C++ languages.
* The code style is heavily influenced by that of Gtk and GStreamer projects.
* CMake is the construction tool.
* The source code is versioned in several GitHub repositories.
* The officially supported platforms are Ubuntu LTS distributions: 14.04 (Trusty) and 16.04 (Xenial).
* The heart of KMS is the GStreamer multimedia framework.
* In addition to GStreamer, KMS uses other libraries like boost, jsoncpp, libnice, etc.

# Source code repositories

Kurento source code is stored in several GitHub repositories at <https://github.com/Kurento>. Each one of these repositories has a specific purpose and usually contains the code required to build a shared library of the same name.

There are several types of repositories:

* **Fork Repositories:** KMS depends on several open source libraries, the main one being GStreamer. Sometimes these libraries show specific behaviors that need to be tweaked in order to be useful for KMS; other times there are bugs that have been fixed but the patch is not accepted at the upstream source for whatever reason. In these situations, while the official path of feature requests and/or patch submit is still tried, we have created a fork of the affected libraries. The repositories that contain these forked libraries are called “Fork Repositories”. These are the current Fork Repositories, as of KMS version 6.6.1:
  + [gstreamer](https://github.com/Kurento/gstreamer) (libgstreamer1.5)
  + [gst-plugins-base](https://github.com/Kurento/gst-plugins-base)
  + [gst-plugins-good](https://github.com/Kurento/gst-plugins-good)
  + [gst-plugins-bad](https://github.com/Kurento/gst-plugins-bad)
  + [gst-plugins-ugly](https://github.com/Kurento/gst-plugins-ugly)
  + [gst-libav](https://github.com/Kurento/gst-libav)
  + [jsoncpp](https://github.com/Kurento/jsoncpp)
  + [libsrtp](https://github.com/Kurento/libsrtp)
  + [libnice](https://github.com/Kurento/libnice) (gstreamer0.10-nice, gstreamer1.0-nice, gstreamer1.5-nice)
  + [openwebrtc-gst-plugins](https://github.com/Kurento/openwebrtc-gst-plugins)
  + [openh264](https://github.com/Kurento/openh264)
  + [usrsctp](http://usrsctp)
* **Main Repositories:** The core of KMS is located in Main Repositories. As of version 6.6, these repositories are:
  + [kms-core](https://github.com/Kurento/kms-core)
  + [kms-elements](https://github.com/Kurento/kms-elements)
  + [kms-filters](https://github.com/Kurento/kms-filters)
  + [kms-jsonrpc](https://github.com/Kurento/kms-jsonrpc)
  + [kms-cmake-utils](https://github.com/Kurento/kms-cmake-utils)
  + [kurento-media-server](https://github.com/Kurento/kurento-media-server)
  + [kurento-module-creator](https://github.com/Kurento/kurento-module-creator)
* **Omni-Build Repository:** The [kms-omni-build](https://github.com/Kurento/kms-omni-build) repository is a dummy umbrella for the other KMS Main Repositories. It has no actual code; instead, it only has the required CMake code to allow building the whole KMS project in one go. For this, it gets a copy of the required repositories via Git submodules.
* **Module Repositories:** KMS is distributed with some basic GStreamer pipeline elements, but other elements are available in form of modules. These modules are stored individually in Module Repositories. Currently, we have the following ones:
  + [kms-crowddetector](https://github.com/Kurento/kms-crowddetector)
  + [kms-chroma](https://github.com/Kurento/kms-chroma)
  + [kms-pointerdetector](https://github.com/Kurento/kms-pointerdetector)
  + [kms-platedetector](https://github.com/Kurento/kms-platedetector)
* **Client Repositories:** Client Applications can be developed in Java, JavaScript with Node.js, or JavaScript directly in the browser. Each of these languages have their support tools made available in their respective repositories.
* **Tutorial or demo repositories:** There are several repositories that contain sample code for developers that use Kurento or want to develop a custom Kurento module. Currently these are:
  + [kms-datachannelexample](https://github.com/Kurento/kms-datachannelexample)
  + [kms-plugin-sample](https://github.com/Kurento/kms-plugin-sample)
  + [kms-opencv-plugin-sample](https://github.com/Kurento/kms-opencv-plugin-sample)
  + [kurento-tutorial-java](https://github.com/Kurento/kurento-tutorial-java)
  + [kurento-tutorial-node](https://github.com/Kurento/kurento-tutorial-node)
  + [kurento-tutorial-js](https://github.com/Kurento/kurento-tutorial-js)

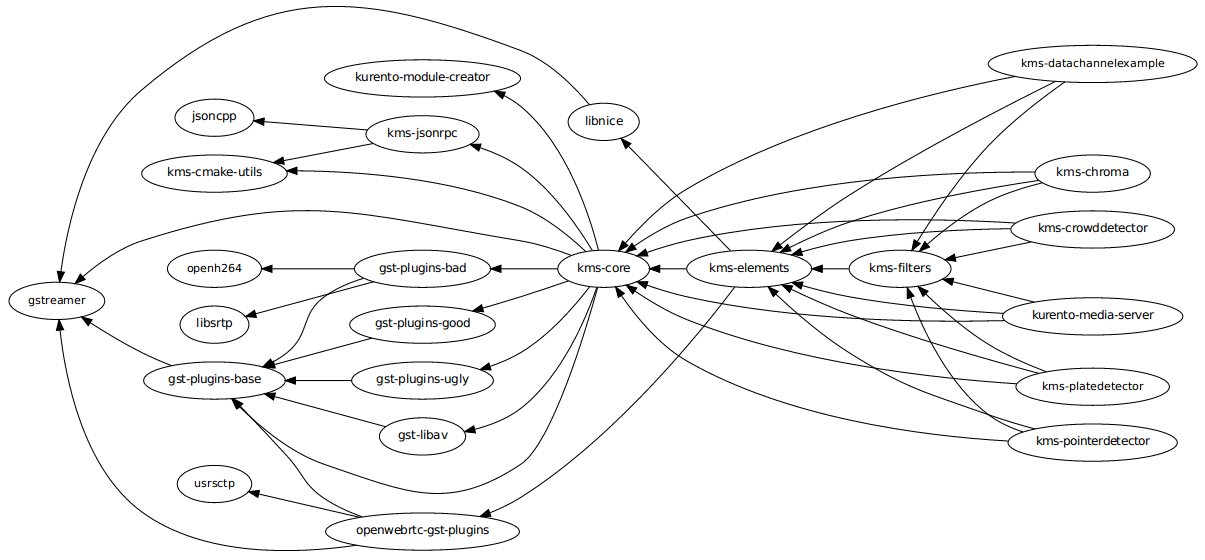
KMS Main Repositories have the following contents:

* **kms-core**: Contains the core GStreamer code. This is the base library that is needed for other libraries. It has 80% C code and a 20% C++ code.
* **kms-elements**: Contains the main elements offering pipeline capabilities like WebRtc, Rtp, Player, Recorder, etc. It has 80% C code and a 20% C++ code.
* **kms-filters**: Contains the basic video filters included in KMS. It has 65% C code and a 35% C++ code.
* **kms-jsonrpc**: Kurento protocol is based on JsonRpc, and makes use of a JsonRpc library contained in this repository. It has C++ code.
* **kms-cmake-utils**: Contains a set of utilities for building KMS with CMake.
* **kurento-media-server**: Contains the main entry point of KMS. That is, the main() function for the server executable code. This application depends on libraries located in the above repositories. It has mainly C++ code.
* **kurento-module-creator**: It is a code generation tool for generating code scaffolding for plugins. This code includes KMS code and Kurento client code. It has mainly Java code.

A KMS developer must know how to work with KMS Fork and Main Repositories and understand that each of these have a different development lifecycle. The majority of development for KMS will occur at the KMS Main Repositories, while it's unusual to make changes in Fork Repositories except for updating their upstream versions.

## Repository dependency graph

This graph shows the dependencies between projects:



# Development 101

KMS is a C/C++ project developed with an Ubuntu system as main target, which means that its dependency management and distribution is based on the Debian package system.

## Libraries

It is not a trivial task to configure the compiler to use a set of libraries because a library can be composed of several .so and .h files. To make this task easier, [pkg-config](https://www.freedesktop.org/wiki/Software/pkg-config/) is a helper tool used when compiling applications and libraries. In short: when a library is installed in a system, it registers itself in the pkg-config database with all its required files, which allows to later query those values in order to compile with the library in question.

For example, if you want to compile a C program which depends on GLib 2.0, you can run:

gcc -o program program.c $(pkg-config --libs --cflags glib-2.0)

## Debian packages

In a Debian/Ubuntu system, development libraries are distributed as Debian packages which are made available in public package repositories. When a C/C++ project is developed in these systems, it is usual to distribute it also in Debian packages. It is then possible to install them with the command apt-get install, which will handle automatically all the package's dependencies.

When a library is packaged, the result usually consists of several packages. These are some pointers on the most common naming conventions for packages, although they are not always strictly enforced by Debian or Ubuntu maintainers:

* **bin package:** Package containing the binary files for the library itself. Applications are linked against them during development, and they are also loaded in production. The package name starts with “lib”, followed by the name of the library.
* **dev package**: Contains files needed to link with the library during development. The package name starts with “lib” and ends with “-dev”. For example, libboost-dev or libglib2.0-dev.
* **dbg package**: Contains debug symbols to ease error debugging during development. The package name starts with “lib” and ends with “-dbg”. For example, libboost-dbg.
* **doc package**: Contains documentation for the library. Used in development. The package name starts with “lib” and ends with “-doc”. For example, libboost-doc.
* **src package**: Package containing the source code for the library. It uses the same package name as the bin version, but it is accessed with the command apt-get source instead of apt-get install.

## Build tools

There are several tools for building C/C++ projects: Autotools, Make, CMake, Gradle, etc. The most prominent tool for building projects is the Makefile, and all the other tools tend to be simply wrappers around this one. KMS uses CMake, which generates native Makefiles to build and package the project. There are some IDEs that recognize CMake projects directly, such as [JetBrains CLion](https://www.jetbrains.com/clion/) or [Qt Creator](https://www.qt.io/ide/).

A CMake projects consists of several CMakeLists.txt files, which define how to compile and package native code into binaries and shared libraries. These files also contain a list of the libraries (dependencies) needed to build the code.

To specify a dependency it is necessary to know how to configure this library in the compiler. The already mentioned pkg-config tool is the standard de-facto for this task, so CMake comes with the ability to use pkg-config under the hood. There are also some libraries built with CMake that use some specific CMake-only utilities.

# Working with KMS sources

KMS uses CMake to build KMS Main Repositories. Fork repositories contain its own build system (typically Autotools or native Make). This depends on the preferences of the original creators of each project.

KMS Main Repositories declare libraries in CMake, assuming they are or can be installed in the system. For example, **kms-elements** depends on the following items:

* **kms-core**, a library located in a Main Repository.
* **libnice**, a library located in a Fork Repository.
* **ffmpeg**, a public library.

For this reason, **kms-core**, **ffmpeg** and **libnice** libraries have to be installed in the system before building the project **kms-elements**.

In KMS, we have developed a custom CMake command to search a library in several places. This command is called **generic\_find** and it is located in the **kms-cmake-utils** repository.

**kms-omni-build** is an special project because it is designed to build all KMS Main Repositories from a single entry point. **kms-omni-build** includes the other KMS Main Repositories as Git submodules: it makes KMS development easier because if you build **kms-omni-build** project, you don’t need to install libraries of KMS Main Repositories manually. However, all development libraries must still be installed manually.

To build KMS from sources you first have to decide on which part you want to work:

* **Main KMS development:** You want to make code changes in Main Repositories and test them in your development machine, to see how the changes affect KMS. Or maybe you want to debug KMS with GDB or analyze it with Valgrind.
* **Change a forked library:** You want to update a Fork Repository and check if all is working as expected. In this case, you have two options:
  + Change code in the current fork.
  + Synchronize the fork with a new release of forked library.
* **Generate Debian packages:** To distribute KMS is necessary to generate Debian packages from KMS Fork and Main Repositories.

As you can see, there are a lot of possibilities. In the next sections we’ll explain the best way to build KMS in these different contexts.

## Developing KMS

To work with KMS Main Repositories the easiest way is using the module **kms-omni-build**.

To work with the KMS codebase you should follow the next steps:

* Install development tools (Git, C Compiler, CMake, etc…).
* Install KMS development libraries.
* Install KMS fork libraries.
* Clone **kms-omni-build** and update submodules.
* Run CMake and Make.
* Run the newly compiled KMS.
* Run KMS tests.
* Optionally, develop using the CLion ID.

### Install development tools

Run as root:

apt-get install --no-install-recommends \

build-essential gdb pkg-config cmake \

clang debhelper valgrind \

git wget maven 'openjdk-[8|7]-jdk'

### Install development libraries for KMS

Run as root:

apt-get install --no-install-recommends \

libboost-dev \

libboost-filesystem-dev \

libboost-log-dev \

libboost-program-options-dev \

libboost-regex-dev \

libboost-system-dev \

libboost-test-dev \

libboost-thread-dev \

libevent-dev \

libglib2.0-dev \

libglibmm-2.4-dev \

libsigc++-2.0-dev \

libopencv-dev

### Install KMS fork libraries

Run as root:

DIST="xenial-dev"

tee /etc/apt/sources.list.d/kurento.list > /dev/null <<EOF

# Kurento Packages repository

deb http://ubuntu.kurento.org ${DIST} kms6

EOF

wget http://ubuntu.kurento.org/kurento.gpg.key -O - | apt-key add -

apt-get update

**Note 1**: For Ubuntu 14.04 (Trusty), use "trusty-dev":

DIST="trusty-dev"

**Note 2**: In order to install the Release version, remove the "-dev" suffix:

DIST="xenial"

Now, the fork packages can be installed from the Kurento repo.

Run as root:

apt-get install --no-install-recommends \

libgstreamer1.5-dev \

libgstreamer-plugins-base1.5-dev \

gstreamer1.5-plugins-base \

gstreamer1.5-plugins-good \

gstreamer1.5-plugins-bad \

gstreamer1.5-plugins-ugly \

gstreamer1.5-libav \

gstreamer1.5-nice \

libnice-dev \

openwebrtc-gst-plugins-dev \

libvpx-dev \

libxml2-utils \

uuid-dev \

libsoup2.4-dev \

libssl-dev \

kmsjsoncpp-dev \

ffmpeg

### Download KMS

Run:

git clone https://github.com/Kurento/kms-omni-build.git

cd kms-omni-build

git submodule init

git submodule update --remote

### Build KMS

Run:

TYPE=Debug

mkdir build-$TYPE && cd build-$TYPE

cmake -DCMAKE\_BUILD\_TYPE=$TYPE ..

make

CMake accepts the following build types, to be used in the first line above: Debug, Release, RelWithDebInfo. So, for a Release build, you would run TYPE=Release instead of TYPE=Debug.

**Important note**: the standard way of compiling a project with CMake is to create a build directory and run the cmake and make commands from there. This allows the developer to have different build folders for different purposes. However do not use this technique if you are trying to compile a subdirectory of **kms-omni-build**. For example, if you do this to build kms-ombi-build/kms-core, no more that one build folder can be present at a time in kms-ombi-build/kms-core/build.

If you want to keep several builds of a single module, it is better to just work on a separate Git clone of that repository.

### Launch KMS

Run:

kurento-media-server/server/kurento-media-server \

--modules-path=. \

--modules-config-path=./modules\_config \

--conf-file=../kurento-media-server/kurento.conf.json \

--gst-plugin-path=. \

--gst-debug-level=3 \

--gst-debug=Kurento\*:4 --gst-debug=kms\*:4

You can set the logging level of specific categories with the option --gst-debug, which can be used multiple times, once for each category. Besides, the global logging level is specified with --gst-debug-level.

### Build and run KMS tests

KMS uses the Check unit testing framework for C (<https://libcheck.github.io/check/>).

To build and run all tests, change the last one of the build commands: make check.

To build and run one specific test, use `make <TestName>.check`.

For example: make test\_agnosticbin.check

If you want to analyze memory usage with Valgrind, use `make <TestName>.valgrind`.

For example: make test\_agnosticbin.valgrind

### Using the CLion IDE

You can use whatever IDE you want to develop KMS. The KMS team has used KDevelop, QtCreator and JetBrains CLion. In this section, CLion configuration is explained.

**Install CLion**

Download from <https://www.jetbrains.com/clion/>, uncompress and run.

**Import KMS in CLion**

Run CLion > Import project from sources > Select **kms-omni-build** folder

**Configure project:**

File > Settings… > Build, Execution, Deployment > CMake >

CMake options: -DGENERATE\_TESTS=TRUE -DFORCE\_GENERIC\_FIND=TRUE -DCMAKE\_C\_COMPILER=clang -DCMAKE\_CXX\_COMPILER=clang++

Wait to “Building symbols...” or “Updating symbols…” process to terminate. And then use the “Compile button”.

CLion hints:

* Compile project: make
* Regenerate CMake: File > Invalidate caches and restart.
* Run CMake from cache variables: CMake > Refresh.

## Working on a forked library

These are the two typical workflows used to work with fork libraries:

### Full cycle

This workflow has the easiest and fastest setup, however it also is the slowest one. To make a change, you would edit the code in the library, then build it, generate Debian packages, and lastly install those packages over the ones already installed in your system. It would then be possible to run KMS and see the effect of the changes in the library.

This is of course an extremely cumbersome process to follow during anything more complex than a couple of edits in the library code.

### In-place linking

The other work method consists on changing the system library path so it points to the working copy where the fork library is being modified. Typically, this involves building the fork with its specific tool (which often is Automake), changing the environment variable LD\_LIBRARY\_PATH, and running KMS with such configuration that any required shared libraries will load the modified version instead of the one installed in the system.

This allows for the fastest development cycle, however the specific instructions to do this are very project-dependent. For example, when working on the GStreamer fork, maybe you want to run GStreamer without using any of the libraries installed in the system; see <https://cgit.freedesktop.org/gstreamer/gstreamer/tree/scripts/gst-uninstalled>.

[TODO: Add concrete instructions for every forked library]

## Generating Debian packages

You can create Debian packages for KMS itself and for forked libraries. We have four public repositories, containing packages generated from KMS Main Repositories and KMS Fork Repositories:

* Repositories for Ubuntu 14.04 (Trusty):
  + Release: http://ubuntu.kurento.org trusty kms6
  + Development: http://ubuntu.kurento.org trusty-dev kms6
* Repositories for Ubuntu 16.04 (Xenial):
  + Release: http://ubuntu.kurento.org xenial kms6
  + Development: http://ubuntu.kurento.org xenial-dev kms6

We also have several Continuous-Integration ("CI") jobs such that every time a patch is accepted in Git's master branch, a new development package of that repository is generated and uploaded to the development repositories. Packages are generated by a Python script called compile\_project.py, which is stored in the [adm-scripts](https://github.com/Kurento/adm-scripts) repository, and you can use it to generate Debian packages locally in your machine.

Versions number of Development packages are timestamped, so a developer is able to know explicitly which version of each package has been installed at any given time. On the other hand, Release packages follow the standard Semantic Versioning system (<http://semver.org/>).

### Example: kms-core

Optional: Make sure the system is in a clean state, removing all KMS packages:

for pkg in \

kurento-module-creator-4.0 kms-cmake-utils \

'^kmsjsoncpp.\*' '^kms-jsonrpc-1.0.\*' '^kms-.\*-6.0.\*' \

'^gir1.2-gst.\*1.5' '^gstreamer1.5.\*' '^libgstreamer.\*1.5.\*' \

'^libnice.\*' '^gstreamer.\*-nice' \

'^libboost.\*-dev' '^libopencv.\*-dev' ffmpeg

do apt-get purge --auto-remove --yes $pkg ; done

Optional: Add Kurento Packages Repository (see section "**Dependency resolution**"):

tee /etc/apt/sources.list.d/kurento-xenial-dev.list > /dev/null <<EOF

# Kurento Packages repository

deb http://ubuntu.kurento.org xenial-dev kms6

EOF

wget http://ubuntu.kurento.org/kurento.gpg.key -O - | apt-key add -

apt-get update

Install system tools and Python modules:

apt-get install --no-install-recommends \

curl wget git build-essential fakeroot debhelper subversion flex realpath \

python python-apt python-debian python-git python-requests python-yaml

Download and setup packaging tools:

git clone https://github.com/Kurento/adm-scripts.git

export PATH="$PWD/adm-scripts:$PATH"

Download and build packages for the desired module:

git clone https://github.com/Kurento/kms-core.git

cd kms-core

../adm-scripts/kms/compile\_project.py --base\_url https://github.com/Kurento compile

Notes:

* subversion (svn) is used by compile\_project.py due to GitHub's lack of support for the git-archive protocol (<https://github.com/isaacs/github/issues/554>).
* flex can be installed by gstreamer, when a bug in package version detection gets fixed.
* realpath is used by adm-scripts/kurento\_check\_version.sh.

### Dependency resolution: to repo or not to repo

The script compile\_project.py is able to resolve all dependencies for any given module. For each dependency, the following process will happen:

* If the dependency is already available to apt-get from the Kurento Packages Repository, it will get downloaded and installed. This means that the dependency will not get built locally.
* If the dependency is not available to apt-get, its corresponding project will be cloned from the Git repo, built, and packaged itself. This triggers a recursive call to compile\_project.py, which in turn will try to satisfy all the dependencies corresponding to that sub-project.

It is very important to keep in mind the dependency resolution mechanism that happens in the Python script, because it can affect which packages get built in the development machine. **If the Kurento Packages Repository has been configured for apt-get, then all dependencies for a given module will be downloaded and installed from the repo, instead of being built**. On the other hand, if the Kurento repo has *not* been configured, then all dependencies will be built from source.

This can have a very big impact on the amount of modules that need to be built to satisfy the dependencies of a given project. The most prominent example is **kurento-media-server**: it basically depends on *everything* else. If the Kurento repo is available to apt-get, then all of KMS libraries will be downloaded and installed. If the repo is not available, then all source code of KMS will get downloaded and built, including the whole GStreamer libraries and other forked libraries.

**Important Note**: This only applies to Ubuntu 16.04 (Xenial), for which the official package repositories already contain all required development libraries to build the whole KMS. However, for Ubuntu 14.04 (Trusty) the official repos are missing some required packages, so the Kurento Packages Repository must be configured in the system in order to build all of KMS. Refer to the following sections.

### Package generation script

This is the full procedure followed by the compile\_project.py script:

1. Check if all development dependencies for the given module are installed in the system. This check is done by parsing the file **debian/control** of the project.
2. If some dependencies are not installed, apt-get tries to install them.
3. For each dependency defined in the file **.build.yaml**, the script checks if it got installed during the previous step. If it wasn't, then the script checks if these dependencies can be found in the source code repository given as argument. The script then proceeds to find this dependency's real name and requirements by checking its online copy of the debian/control file.
4. Every dependency with source repository, as found in the previous step, is cloned and the script is run recursively with that module.
5. When all development dependencies are installed (either from package repositories or compiling from source code), the requested module is built, and its Debian packages are generated and installed.

### Building KMS on Ubuntu 14.04 (Trusty)

KMS cannot be built in Trusty without adding the Kurento Packages Repository, because some of the system development libraries are required in a more recent version than the one available by default in the official Ubuntu Trusty repos. This is a non exhaustive list of those required libraries, compared with the versions available in Xenial and in the Kurento repo:

**kms-core**

* libglib2.0-dev (>= 2.46); 14.04 = (2.40); 16.04 = (2.48); Kurento = (2.46).  
  It actually builds and works fine with 2.40, but the required version of glib was first raised from 2.40 to 2.42 and later to 2.46 in commits b10d318b and 7f703bed, justified as providing huge performance improvement in mutex and g\_object\_ref.

**gst-plugins-base**

* libsoup2.4-dev (>= 2.48); 14.04 = (2.44); 16.04 = (2.52); Kurento = (2.50).

**libsrtp**

* libssl-dev (>= 1.0.2); 14.04 = (1.0.1f); 16.04 = (1.0.2g); Kurento = (1.0.2g).

**gst-plugins-bad**

* libde265-dev (Any); 14.04 = (None); 16.04 = (1.0.2); Kurento = (0.9).
* libx265-dev (Any); 14.04 = (None); 16.04 = (1.9); Kurento = (1.7).
* libass-dev (>= 0.10.2); 14.04 = (0.10.1); 16.04 = (0.13.1); Kurento = (0.10.2).
* libgnutls28-dev and librtmp-dev; the latter depends on libgnutls-dev, which conflicts with the former (only in 14.04). Solution: use librtmp-dev from Kurento repo, which doesn't depend on libgnutls-dev.

**kms-elements**

* libnice-dev (>= 0.1.13); 14.04 = (0.1.4); 16.04 = (0.1.13); Kurento = (0.1.13).

**libnice**

* libgupnp-igd-1.0-dev (>= 0.2.4); 14.04 = (0.2.2); 16.04 = (0.2.4); Kurento = (0.2.4).

This means that it is not possible to build the whole KMS without the Kurento Packages Repository already configured in the system. But as we mentioned in the previous section, the mere presence of this repo will skip building as many packages as possible if the build script is able to find them already available for install with apt-get.

In the case that we want to build the whole KMS libraries and modules, the solution to this problem is to clone each module separately, and build them in the order given by their dependency graph, which is this (see section "**Repository dependency graph**"):

1. gstreamer
2. gst-plugins-base
3. gst-plugins-good
4. gst-plugins-ugly
5. libsrtp
6. openh264
7. gst-plugins-bad
8. gst-libav
9. usrsctp
10. openwebrtc-gst-plugins
11. jsoncpp
12. libnice
13. kms-cmake-utils
14. kurento-module-creator
15. kms-jsonrpc
16. kms-core
17. kms-elements
18. kms-filters
19. kurento-media-server
20. kms-chroma
21. kms-crowddetector
22. kms-datachannelexample
23. kms-platedetector
24. kms-pointerdetector

# How-to’s

## How to add or update an external library to kurento

Add it to/Change it in:

* Add dependency to debian/control in the project that uses it
* Add dependency to CMakeLists.txt in the project that uses it

## How to add a new fork library to kurento

* Fork the repository
* Create a .build.yaml in this repository with build instructions
* Add dependency to debian/control in the project that uses it
* Add dependency to CMakeLists.txt in the project that uses it
* Add dependency to .build.yaml in the project that uses it

# Known problems

* Sometimes gstreamer fork doesn't compile correctly. Try again.
* Some tests are failing sometimes. If tests fail, packages are not generated. To change it, edit debian/rules file to disable testing generation and testing execution.