Software Management

Egor Pugin

June 3, 2021

Note: this is an early draft. It's known to be outdated, incomplet and incorrekt, and it has lots of bad formatting.

Please, report any issues to author. Contributions are welcome.

Homepage	https://software-network.org/
Git Repository (client tool)	https://github.com/SoftwareNetwork/sw
The latest version of this document	https://software-network.org/client/sw.pdf

Contents

Introduction

This document

This document describes a modern approach to software management. It consists of several chapters which include the new approach itself, documentation for client-side tools and utilities and server-side (website) functionality description.

License for this documentation is unspecified yet.

Overview

Software Network is a project dedicated to better software management.

Originally started as a package manager (CPPAN - https://github.com/cppan/cppan) to C/C++ languages based on CMake build system, it is evolved to independent set of tools and libraries. CPPAN or v1 was a playground where different ideas were studied and checked.

Main user tool is called ‘sw‘. Whole project also may be called as SW. Pronounce it as you like: ‘[software]‘ or ‘[sw]‘ or ‘[sv]‘.

History

CPPAN or SW v1

Idea was formed during years of work with CMake, different projects, porting things to different OSs. Development started in the early 2016.

YAML was choosen as configuration file format. It was mixed declarative approach with imperative CMake insertions.

SW or CPPAN v2

Development started in the late 2017.

Decarative approach showed weaknesses, so it was decided to switch to the most flexible thing - complete programming language. C++ was choosen over Python (Conan) and, possibly, other languages such as Lua (Premake).

More on CPPAN

CPPAN – C++ Archive Network. This section is not related to current status of SW and given for historical reasons.

The idea comes from:

1. Comprehensive Perl Archive Network (CPAN), CRAN (R-language), CTAN (TeX).
2. Java packages and build systems (Maven).
3. C++ Modules proposals and presentations by Gabriel Dos Reis.

In the beginning project aimed on C++ project with Modules only. So, the project should evolve by their release. But during development, CPPAN shows great capabilities of handling current C++98/C++11/C++14 projects and even some C libraries.

General principles of CPPAN are listed below.

1. Source code only! You do not include your other stuff like tests, benchmarks, utilities etc. Only headers and sources (if any). On exception here: project's license file. Include it if you have one in the project tree.

2. Semantic versioning http://semver.org.

3. Zero-configure (zero-conf.) projects. Projects should contain their configurations in headers (relying on toolchain macros) or rely on CPPAN utilities (macros, different checkers in configuration file) or have no config steps at all (header only projects). Still CPPAN provides inlining of user configuration steps, compiler flags etc.

4. All or Nothing rule for dependencies. Many projects have optional dependencies. In CPPAN they should list them and they'll be always included to the build or not included at all. So, no optional dependencies.

Projects' naming

Project names are like Modules from this C++ proposal http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/p01
Words are delimeted by points '.'.

Root names are:

1. pvt - for users' projects. E.g. 'pvt.cppan.some_lib'. 2. org - for organization projects with open source licenses. E.g. 'org.boost.algorithm'. 3. com - for organization projects with commercial licenses. E.g. 'com.some_org.some_proprietary_lib'.

Project versions

Each project will have its versions. Version can be a semver number '1.2.8' or a branch name 'master', 'dev'. Branches always considered as unstable unless it is stated by maintainer. Branches can be updated to the latest commit. And fixed versions can not.

Useful links

1. Website - https://software-network.org/

2. Download client - https://software-network.org/client/

Acknowledgements

Chapter 1

Software Management

1.1 Features

Goals:

1. Reliability.
2. Correctness.
3. Repeatability.
4. Robustness.
5. Scalability.
6. Performance. This one is mostly provided by language of implementation (C++).

Features:

1. Build System is integrated with Package Manager in one program. This allows us to have more information on both ends. User does not need any external tools except sw itself.
2. Describe your builds directly in C++(17).
3. Different frontends. Do not like C++? Write you build configuration in declarative YAML, JSON, Python, LUA and even on good old CMake etc. Currently only basic YAML frontend is implemented. Contributions are welcome. SW itself is written in modern C++ (C++17).
4. Easy crosscompilation.
5. Generators for different IDEs (WIP, contributions are welcome).
6. Integrations with other tools (CMake etc.).

1.2 Package Id

1.2.1 Path

1.2.2 Version

1.3 Drivers

1.4 Frontends

Main frontend - C++.
Second frontend - CPPAN (v1) YAML-based frontend.

1.5 Security

SW security is based on GPG digital signatures.
TBD

Chapter 2

Client tools (SW and others)

Main client tool is SW.

A gui frontend is also available. It is Qt-based and widgets-based (no QML).

2.1 Quick Start

2.1.1 Package Management

SW uses packages paths to uniquely identify projects on the web. Project path is similar to Java packages. Version is appended to it.

'org.username.path.to.project.target-version'

It contains:

1. Namespace: 'org'
2. User name: 'username'
3. Path to target: 'path.to.project.target'

Allowed namespaces are:

1. 'org' - organization's public projects with open (free) software license.
2. 'com' - organization's projects with commercial license.
3. 'pub' - user's public projects.
4. 'pvt' - user's private projects.

'version' - is a **semver-like** version.

2.1.2 Build process

SW uses configuration files to describe build process (Build).

Build contains several solutions (Solution). Each Solution is responsible for exactly one configuration. For example, x86+dll+Release or arm64+static+Debug (plus sign is used only for this document).

Solution consists of targets, projects and directories. Each object might include any other object.

SW uses multipass approach if it needs to build more than one solution. (Unlike CMake that is telled which options go to what configuration during single pass.)

It is possible to use different frontends to describe build. Main frontend is called 'sw' as well as the program and uses C++ code directly. Current secondary frontend is 'cppan' YAML frontend used in first version of the program.

Build process

2.1.3 Frontends

2.1.3.1 C++ frontend (sw)

HINT: run 'sw create project' to force sw to create first config and generate an IDE project for you._

To start your first build configuration, create a file called 'sw.cpp'.

sw.cpp

```
// this is our 'main' build function signature
// function accepts current Solution
void build(Solution &s)
{
    // add non copyable target to Solution of type Executable with name 'mytarget'
    auto &t = s.addTarget<Executable>("mytarget");
    t += "src/main.cpp"; // add our first source file
    // t += "dependency.path"_dep; // add needed dependency
}
```

SW uses the latest current standard available - C++17.

To control your build (add definitions, dependencies, link libraries) sw uses user-defined literals. For more help on this, see https://github.com/SoftwareNetwork/sw/blob/master/src/sw/driver/suffix.h Examples:

sw.cpp

```
t += "MY_DEFINITION"_def;
t += "ws2_32.lib"_slib;
t += "src"_idir;
```

Now create ‘src/main.cpp‘ file with contents you want.

Run ‘sw build‘ to perform build or ‘sw generate‘ to generate IDE project.

Visibility

To provide control on properies visibility, one might use following target members:

sw.cpp

```
t.Private += ...; // equals to t += ...;
t.Protected += ...; // available only for this target and current project members
t.Public += ...; // available for this target and all downstream users
t.Inheritance += ...; // available only for all downstream users
```

Underlying implementation uses three bits to describe inheritance.

1. This target T
2. This project P (excluding T)
3. All other projects A (excluding P and T)

This gives 7 different inheritance modes (excluding 000) and you could use them all!

More than one configuration

To create additional configs you might add them on command line or write into configuration. Add following code to your ‘sw.cpp‘:

sw.cpp

```
// configure signature
// Note that it accepts whole build, not a single solution!
void configure(Build &b)
{
    auto &s1 = b.addSolution(); // add first solution
    s1.Settings.Native.LibrariesType = LibraryType::Static;
    s1.Settings.Native.ConfigurationType = ConfigurationType::ReleaseWithDebugInformation;

    auto &s2 = b.addSolution(); // add first solution
    s2.Settings.Native.LibrariesType = LibraryType::Shared;
    s2.Settings.Native.ConfigurationType = ConfigurationType::Debug;

    return;

    // following code can be used for advanced toolchain configuration
    if (b.isConfigSelected("cygwin2macos"))
        b.loadModule("utils/cc/cygwin2macos.cpp").call<void(Solution&)>("configure", b);
    else if (b.isConfigSelected("win2macos"))
        b.loadModule("utils/cc/win2macos.cpp").call<void(Solution&)>("configure", b);
}
```

In this function you could provide full custom toolchain to use (compilers, linkers, librarians etc.). Extended configs can be found here: https://github.com/SoftwareNetwork/sw/tree/master/utils/cc

2.1.3.2 YAML declarative frontend (CPPAN)

CPPAN is a declarative frontend using YAML syntax.

Example: https://github.com/cppan/cppan/blob/v1/doc/cppan.yml

Quick Start

We'll describe initial setup and usage based on this demo project https://github.com/cppan/demo_project.

To start work with cppan install its client into the system. On all systems and work can be done without root privileges.

1. Create your project's initial structure, 'CMakeLists.txt' file etc.
2. Create 'cppan.yml' file in the project's root directory.
3. Open it and write dependencies on which your project depends.

'cppan.yml' files use YAML syntax which is very simple. Example of this file can be:

cppan.yml

```
dependencies:
        pvt.cppan.demo.gsl: master
        pvt.cppan.demo.yaml_cpp: "*"

        pvt.cppan.demo.boost.asio: 1
        pvt.cppan.demo.boost.filesystem: 1.60
        pvt.cppan.demo.boost.log: 1.60.0
```

'dependencies' directive tells the CPPAN which projects are used by your project. '*' means any latest fixed (not a branch) version. Textual name is a branch name ('master'). Versions can be '1.2.8' - exact version, '1.2' means '1.2.*' any version in 1.2 series, '1' means '1.*.*' and version in 1 series. When the new version is available it will be downloaded and replace your current one only in case if you use series versions. If you use strict fixed version, it won't be updated, so no surprises.

Now you should run 'cppan' client in project's root directory where 'cppan.yml' is located. It will download necessary packages and do initial build system setup.

After this you should include '.cppan' subdirectory in your 'CMakeLists.txt' file.

CMake

```
add_subdirectory(.cppan)
```

You can do this after 'project()' directive. In this case all dependencies won't be re-built after any compile flags. If you need your special compile flags to have influence on deps, write 'add_subdirectory(.cppan)' after compiler setup.

For your target(s) add 'cppan' to 'target_link_libraries()':

CMake

```
target_link_libraries(your_target cppan)
```

CMake will try to link all deps into this target. To be more specific you can provide only necessary deps tothis target:

CMake

```
target_link_libraries(your_target org.boost.algorithm-1.60.0) # or
    target_link_libraries(your_target org.boost.algorithm-1.60) # or
    target_link_libraries(your_target org.boost.algorithm-1) # or
    target_link_libraries(your_target org.boost.algorithm) # or
```

All these names are aliases to full name. So, when you have more that 1 version of library (what is really really bad!), you can specify correct version. For custom build steps you may use executables by their shortest name.

Internally cppan generate a 'CMakeLists.txt' file for dependency. It will use all files it found in the dependency dir.

CMake

```
file(GLOB_RECURSE src "*")
```

Syntax

files

In ‘files‘ directive you specify what files to include into the distributable package. It can be a regex expression or a relative file name.

cppan.yml

```
files:
        - include/.*
        - src/.*
```

or

cppan.yml

```
files:
        - sqlite3.h
        - sqlite3ext.h
        - sqlite3.c
```

or just

cppan.yml

```
files: include/.*
```

or

cppan.yml

```
files: # from google.protobuf
      - src/.*\.h
      - src/google/protobuf/arena.cc
      - src/google/protobuf/arenastring.cc
      - src/google/protobuf/extension_set.cc
```

dependencies

‘dependencies‘ contains a list of all dependencies required by your project. Can be ‘private‘ and ‘public‘. ‘public‘ are exported when you add you project to C++ Archive Network. ‘private‘ stays private and can be used by your project’s tools and other stuff.

For example when you develop an application, you want to add unit tests, regression tests, benchmarks etc. You can specify test frameworks as dependencies too. For example, ‘pvt.cppan.demo.google.googletest.gtest‘ or ‘pvt.cppan.demo.google.googletest.gmock‘.

But when you develop a library and want export it to CPPAN, you won’t those libraries in the public dependency list. You can write:

cppan.yml

```
dependencies:
      public:
        org.boost.filesystem: 1
      private:
        pvt.cppan.demo.google.googletest.gtest: master
```

By default all deps are public.

cppan.yml

```
dependencies:
        org.boost.filesystem: 1.60 # public
        pvt.cppan.demo.google.googletest.gtest: master # public now
```

include_directories

Include directories are needed by your project and project users. You must always write ‘private‘ or ‘public‘ keywords. ‘private‘ include dirs available for your lib only. ‘public‘ available for your lib and its users.

cppan.yml

```
include_directories: # boost.math example
      public:
        - include
      private:
        - src/tr1
```

license

Include license file if you have it in the project tree.

license: LICENSE.txt

root_directory

When adding version from remote file (archive) often there is a root dir inside the archive. You can use this directive to specify a path to be added to all relative files and dirs.

cppan.yml

```
root_directory: sqlite-amalgamation-3110000 # sqlite3 example
```

exclude_from_build

Sometimes you want to ship a source file, but do not want to include it into build. Maybe it will be conditionally included from config header or whatever.

cppan.yml

```
exclude_from_build: # from boost.thread
      - src/pthread/once_atomic.cpp
```

options

In ‘options‘ you can provide predefined macros for specific configurations. ‘any‘, ‘static‘ and ‘shared‘ are available. Inside them you can use ‘definitions‘ to provide compile defines. You must write ‘public‘ or ‘private‘ near each define. ‘private‘ is only for current library. ‘public‘ define will see all users and the current lib.

cppan.yml

```
options: # from boost.log
        any:
          definitions:
            public: BOOST_LOG_WITHOUT_EVENT_LOG
            private: BOOST_LOG_BUILDING_THE_LIB=1
        shared:
          definitions:
            public: BOOST_LOG_DYN_LINK
            private: BOOST_LOG_DLL
```

But try to minimize use of such options.

pre_sources, post_sources, post_target, post_alias

You can provide your custom build system insertions with these directives.

cppan.yml

```
post_sources: | # custom step from boost.config
      if (WIN32)
        file(WRITE ${CMAKE_CURRENT_SOURCE_DIR}/include/boost/config/auto_link.hpp "")
      endif()
```

Can be used in options too near with ‘definitions‘.

root_project, projects

Can be used when you’re exporting more than one project from one repository. Inside ‘projects‘ you should describe all exported projects. Relative name of each should be specified after this. Inside each project, write usual command.

cppan.yml

```
root_project: pvt.cppan.demo.google.googletest
      projects:
        gtest: # use relative name here - absolute will be pvt.cppan.demo.google.googletest.gtest
          license: googletest/LICENSE
          files:
            - googletest/include/.*\.h
            - googletest/src/.*
        gmock: # use relative name here
          # ...
```

package_dir

Can be used to choose storage of downloaded packages. Could be 'system', 'user', 'local'. Default is 'user'. 'system' requires root right when running cppan, deps will go to '/usr/share/cppan/packages'. 'user' stores deps in '$HOME/.cppan/packages'. 'local' will store deps in cppan/ in local dir.

package_dir: local

You can selectively choose what deps should go to one of those locations.

cppan.yml

```
dependencies:
        org.boost.filesystem:
            version: "*"
            package_dir: local
```

It is convenient when you want to apply your patches to some dependency.

check_function_exists, check_include_exists, check_type_size, check_symbol_exists, check_library_exists

These use cmake facilities to provide a way of checking headers, symbols, functions etc. Cannot be under 'projects' directive. Can be only under root. They will be gathered during 'cppan' run, duplicates will be removed, so not duplicate work.

cppan.yml

```
check_function_exists: # from libressl
      - asprintf
      - inet_pton
      - reallocarray

    check_include_exists:
      - err.h
```

2.1.3.3 CMake frontend

If you have your projects in CMake, SW is able to load and run them to extract build rules.

CMake frontend is in initial state right now. Try it and report any issues.

2.2 Command Line Reference

2.3 Build System Supported Languages

SW aims to build a wide specter of programming languages.

Mature support:

1. ASM
2. C
3. C++

Hello-world support:

1. C#
2. D

3. Fortran
4. Go
5. Java
6. Kotlin
7. Rust

Ported Intepreters
- perl (partial)
- python
- tcl
Work started
- java

2.4 Integrations

It is possible to use SW as package manager for your favourite build system.
Here you can find list of completed integrations with examples.

2.4.1 CMake

Full example:
https://github.com/SoftwareNetwork/sw/blob/master/test/integrations/CMakeLists.txt
Steps

1. Download SW, unpack and add to PATH.
2. Run ‘sw setup‘ to perform initial integration into system.
3. Add necessary parts to your CMakeLists.txt (see below).

First, make sure CMake is able to found SW package:

CMake
```
find_package(SW REQUIRED)
```

Then add necessary dependencies (packages) with

CMake
```
sw_add_package()
```

You can omit version. In this case the latest will be used.
Next step is to execute SW to prepare imported targets script with

CMake
```
sw_execute()
```

Complete example:

CMake
```
find_package(SW REQUIRED)
sw_add_package(
    org.sw.demo.sqlite3
    org.sw.demo.glennrp.png
)
sw_execute()
```

Last thing is to add dependency to your package:

CMake

```
add_executable(mytarget ${MY_SOURCES})
target_link_libraries(mytarget
    org.sw.demo.glennrp.png
)
```

Configure SW deps
Set SW_BUILD_SHARED_LIBS to 1 to build shared libs instead of static (default)

CMake

```
set(SW_BUILD_SHARED_LIBS 1)
```

2.4.2 Waf

Full example
https://github.com/SoftwareNetwork/sw/blob/master/test/integrations/wscript

1. Copy-paste these lines to your file https://github.com/SoftwareNetwork/sw/blob/master/test/integrations/wscript#L4-L25
2. Replace these lines with your dependencies https://github.com/SoftwareNetwork/sw/blob/master/test/integrations/wscript#L9-L10
3. Use deps like on those lines https://github.com/SoftwareNetwork/sw/blob/master/test/integrations/wscript#L35-L36

To build things call

Shell

```
waf sw_configure sw_build configure build
```

sw_configure step setups SW for waf.
sw_build step builds necessary deps.

2.5 Generators

SW allows you to generate projects to many different existing systems.

Generator	Support
Visual Studio (C++ projects)	Full
Visual Studio (NMake projects)	Full
Compilation Database	Full
Make	Full
NMake	4/5
Ninja	Full
Batch (win)	Full
Shell (*nix)	Full
SW build description	Full
SW execution plan	Full
Raw bootstrap	Full
Missing generators:	
Xcode	
CMake	
qmake	

Raw bootstrap is used to bootstrap programs without sw.

2.6 Reproducible builds

SW supports reproducible builds. Invoke your commands with ‘–reproducible-build‘ parameter. Binaries MUST BE the same on all systems assuming you compare identical build configs.

Shell

```
sw --reproducible-build ... build
```

2.7 Command Line Reference

2.7.1 sw alias

Add command alias.

Shell

```
sw alias aliasname command line args
```

To invoke it, run:

Shell

```
sw aliasname
```

You cannot add alias to existing commands.

2.7.2 sw build

Build current project.

To get build and dependencies graphs, run:

Shell

```
sw -print-graph build
```

2.7.3 sw create

Create templated project.

2.7.4 sw generate

Generate IDE project.

2.7.5 sw upload

Upload current project to Software Network. Project will be fetched from your version control system using specified tag/version/branch.

2.7.6 sw setup

Integrate or remove sw from the system.

Integrate sw into system.

Shell

```
sw setup
```

Remove sw into system.

Shell

```
sw setup --uninstall
```

To control what is cleaned up, use ‘–level‘ option with comma separated list of flags.

Shell

```
sw setup --uninstall --level 1,4
```

Available levels:

- 1 - Remove storage.
- 2 - Remove sw system integration settings.
- 4 - Remove sw settings.
- 8 - Remove sw executable (if possible).
- 16 - All of the above.

2.7.7 sw verify

Verify uploaded file.

Sw provides byte to byte verification for files uploaded to Software Network. If you want to check that someone is not changed project files, you can run this command. It will download original source from the internet, create the archive and compare its hash with file on the Software Network.

2.7.8 sw -help

CLI help.

```
OVERVIEW: SW: Software Network Client

  SW is a Universal Package Manager and Build System

USAGE: sw.client.sw-0.3.1.exe [subcommand] [options]

SUBCOMMANDS:

  abi       - List package ABI, check for ABI breakages.
  build     - Build files, dirs or packages.
  configure - Create build script.
  create    - Create different projects.
  fetch     - Fetch sources.
  generate  - Generate IDE projects.
  install   - Add package to lock.
  integrate - Integrate sw into different tools.
  list      - List packages in database.
  open      - Open package directory.
  override  - Override packages locally.
  remote    - Manage remotes.
  remove    - Remove package.
  run       - Run target (if applicable).
  setup     - Used to do some system setup which may require administrator access.
  test      - Run tests.
  update    - Update lock file.
  upload    - Upload packages.
  uri       - Used to invoke sw application from the website.

  Type "sw.client.sw-0.3.1.exe <subcommand> -help" to get more help on a specific subcommand
```

```
OPTIONS:

General options:

  -B                              - Build always
  -D=<string>                     - Input variables
  -activate=<string>              - Activate specific packages
  -build-name=<string>            - Set meaningful build name instead of hash
  -cc-checks-command=<string>     - Automatically execute cc checks command
  -checks-st                      - Perform checks in one thread (for cc)
  -compiler=<string>              - Set compiler
  -config-name=<string>           - Set meaningful config names instead of hashes
  -configuration=<string>         - Set build configuration.
                                    Allowed values:
                                        - debug, d
                                        - release, r
                                        - releasewithdebuginformation, releasewithdebinfo, rwdi
                                        - minimalsizerelease, minsizerel, msr
                                    Default is release.
                                    Specify multiple using a comma: "d,r".
  -curl-verbose                   -
  -d=<path>                       - Working directory
  -debug-configs                  - Build configs in debug mode
  -do-not-mangle-object-names     -
  -do-not-remove-bad-module       -
  -exclude-target=<string>        - Targets to ignore
  -explain-outdated               - Explain outdated commands
  -explain-outdated-full          - Explain outdated commands with more info
  -host-cygwin                    - When on cygwin, allow it as host
  -host-settings-file=<path>      - Read host settings from file
  -ignore-source-files-errors     - Useful for debugging
  -ignore-ssl-checks              -
  -j=<int>                        - Number of jobs
  -k=<int>                        - Skip errors
  -l                              - Use lock file
  -libc=<string>                  - Set build libc
  -libcpp=<string>                - Set build libcpp
  -list-predefined-targets        - List predefined targets
  -list-programs                  - List available programs on the system
  -log-to-file                    -
  -os=<string>                    - Set build target os
  -platform=<string>              - Set build platform.
                                    Examples: x86, x64, arm, arm64
  -print-checks                   - Save extended checks info to file
  -r=<string>                     - Select default remote
  -s                              - Force server resolving
  -save-all-commands              -
  -save-command-format=<string>   - Explicitly set saved command format (bat or sh)
  -save-executed-commands         -
  -save-failed-commands           -
  -sd                             - Force server db check
  -self-upgrade                   - Upgrade client
  -settings=<string>              - Set settings directly
  -settings-file=<path>           - Read settings from file
  -settings-file-config=<string>  - Select settings from file
  -settings-json=<string>         - Read settings from json string
  -shared-build                   - Set shared build (default)
  -show-output                    -
```

```
  -standalone                   - Build standalone binaries
  -static-build                 - Set static build
  -static-dependencies          - Build static dependencies of inputs
  -storage-dir=<path>           -
  -target=<string>              - Targets to build
  -target-os=<string>           -
  -time-trace                   - Record chrome time trace events
  -toolset=<string>             - Set VS generator toolset
  -trace                        - Trace output
  -verbose                      - Verbose output
  -wait-for-cc-checks           - Do not exit on missing cc checks, wait for user input
  -win-md                       - Set /MD build (default)
  -win-mt                       - Set /MT build
  -write-output-to-file         -

Generic Options:

  -help                         - Display available options (-help-hidden for more)
  -help-list                    - Display list of available options (-help-list-hidden for more)
  -version                      - Display the version of this program
```

2.8 Work behind proxy

To use sw client via proxy, add following to your SW YAML config in ~/.sw/sw.yml:

```
proxy:
    host: hostname:port
    user: user:passwd
```

2.9 Bootstrapping

SW client is distributed in a form of precompiled binary. This binary is built with SW tool, so we have a circular dependency.

It is important to be able to build it without existing SW client. Here comes bootstrapping procedure. To make bootstrap package, use ‘rawbootstrap‘ generator:

Shell

```
sw *needed config* generate -g rawbootstrap
```

2.10 Internals

2.10.1 Components and directory structure

Components

1. Support - small helper library.
2. Manager - does package managing: package paths, package ids, local databases, remote repository selector, package downloader etc.
3. Builder - controls executed commands, file timestamps, timestamp databases.
4. Driver.cpp - main program driver that implements high level build, solutions, targets representation, programs (compilers), source files, languages and generators.
5. Client - main executable.

Directory structure

1. doc - documentation

2. include - some public headers; not well set up
3. src - source code
4. test - tests
5. utils - all other misc. files: crosscompilation configs etc.

Chapter 3

Server tools

Software Network – server frontend – https://software-network.org/
Download clients – https://software-network.org/client

3.1 Publish a project

Publish Instructions

1. Prepare your project.
2. Prepare build script.
3. Set up source code repository.
4. Upload project on some hosting.
5. Create an account on Software Network and your user token at https://software-network.org/u/tokens.
6. Create your first publisher https://software-network.org/u/publishers.
7. Add user token to SW settings. Run: ‘ sw remote alter origin add token PUBLISHERNAME SECRETTOKEN ‘
8. Run locally near build script to upload the project: ‘ sw upload PREFIX ‘

SW will download your sources from the public hosting using Source provided in build script.
—
PREFIX is your namespace (com/org/pub/pvt) + your publisher name + custom subdirectory inside it. “‘ namespace.publisher[.subdirectory] “‘
Examples:

1. org.sw.demo
2. org.boost
3. com.someorgname
4. pub.egorpugin.my.fancy.subdir
5. pvt.egorpugin.my.secret.project

Old cppan info

It uses ssl for any interactions, api calls etc. It uses http to send packages (data) to users. But hashsum checks are performed on client side and hashes are trasferred securely.

On that site you can find projects, organizations, project versions.

To add your project to CPPAN you have to:

1. Register (e.g. ‘cppan‘ account name). 1. Login 1. Create project. It will be created under your account in ‘private‘ - ‘pvt‘ root namespace. E.g. ‘pvt.cppan.my_very_useful_library‘. 1. Add project version.

Project version sources

1. git repository 2. remote file (on some server: http, ftp) 3. local file (from your computer)

You can add either fixed version (‘1.2.8‘) or a branch (‘master‘, ‘develop‘). Branch is updatable. Version is not.

When adding version from git, it tries to find a tag in form ‘prefixX.Y.Z‘ where ‘X.Y.Z‘ is version provided and ‘prefix‘ is custom prefix on your tags. For example, boost uses ‘boost-‘ prefix, some projects use ‘v‘ and some use empty prefix. You cannot change ‘X.Y.Z‘ delimiters. It’s always ‘.‘. So, if you want to add your project, consider changing your tag naming schema for future.

When adding branch from git, it tries to find a branch with same name in the git repo.

Types of projects:

1. library 2. executable 3. root project - an umbrella project which can be downloaded as dependency with all its children. For example, if you write in ‘dependencies‘ ‘pvt.cppan.demo.boost‘ it will download and compile whole boost. 4. directory - an umbrella project which can not be downloaded as dependency.

You can specify custom content of ‘cppan.yml‘ on AddProjectVersion page. It helps if you’re experimenting.

You can create a permanent ‘cppan.yml‘ file in your repository or an archive, so it will be used as input.

Organizations

You can add an organization if you are representing one. Then you can add admins (can create, remove projects) and users (can add versions of their projects).

Organizations will receive two root namespaces: ‘org‘ for projects with open source licensed and ‘com‘ for proprietary code. Private repositories both for users and orgs probably will be introduced later.

F.A.Q.

Q: What is ‘org.sw.demo‘ namespace?

‘org.sw‘ is a utility SW account and ‘org.sw.demo‘ contains many projects, but they’re not official. You could use them as a starting point in uploading your packages. And official packages will be in ‘pub/org.your_name.*‘ namespace. Also for orgs: ‘org.boost.*‘, ‘org.google.*‘, ‘org.facebook.*‘, ‘com.ibm.*‘.

Q: Where does sw download/generate stuff?

By default, in your user home directory. Like ‘$HOME/.sw/storage‘ on *nix or ‘%USERPROFILE%/.sw/storage‘ on Windows.

Q: How to change that directory?

Open ‘$HOME/.sw/sw.yml‘, add ‘storage_dir: your/favourite/dir/for/sw‘.

Q: Will be server side open sourced? When?

Probably yes, later.

TODO

Help is appreciated.

frontends: lua, python, makefile, c, cmake, angelscript