# Features / Feature packages

## 1. Motivation

### A. OpenCV + CUDA

[OpenCV](http://opencv.org/) is a computer vision library that can optionally be built with CUDA support to massively accelerate certain tasks when using computers with NVidia GPUs. For users without NVidia GPUs, building with CUDA support provides no benefit. [CUDA](http://www.nvidia.com/object/cuda_home_new.html) is provided only via a 1.3 GB installer (at the time of this authoring), which requires administrator access to install and modifies the global system state.

Therefore, there is significant value in enabling users to choose whether they find CUDA support valuable for their particular scenario.

### B. OpenCV + OpenCV\_contrib

The community around [OpenCV](http://opencv.org/) has built up a library of extensions called [OpenCV\_contrib](https://github.com/opencv/opencv_contrib). However, these extensions are a source-level patch onto the main OpenCV codebase and therefore must be applied *during* the core OpenCV build. Further confounding the problem, it is the author’s understanding that these community extensions have only been developed with [CUDA](http://www.nvidia.com/object/cuda_home_new.html) enabled and cannot be built without that dependency.

Therefore, if CUDA is disabled, OpenCV\_contrib must also be disabled. Likewise, when a user requests OpenCV\_contrib, CUDA must be enabled. It would be convienent, but not a requirement, to enable CUDA without enabling the community extensions.

Finally, these extensions add additional exports and headers which could be depended upon by other libraries. For maintainers, there must be a way to specify this requirement such that vcpkg install mylib-depends-ocv-contrib will verify/build/rebuild OpenCV with the community extensions enabled.

### C. C++ REST SDK + SignalR

The [C++ REST SDK](https://github.com/Microsoft/cpprestsdk) is a networking library that provides (among other features) HTTP and Websockets clients. To implement the HTTP client functionality on Windows Desktop, only the core Win32 platform APIs are needed (zlib is optional). However, the websockets client is based on [Websockets++](https://www.zaphoyd.com/websocketpp/), which adds mandatory dependencies on boost, openssl, and zlib. Many users of the C++ REST SDK do not use the websockets component, so to minimize their overall dependency footprint it can be disabled at build time. Ideally, this option would be easily accessible to users who are concerned about the final size or licensing of their deployment.

However, [SignalR-Client-Cpp](https://github.com/aspnet/SignalR-Client-Cpp) depends on the websockets functionality provided by the C++ REST SDK. Therefore, the maintainers of the signalrclient port would ideally like to express this dependency such that cpprestsdk will be correctly built for their needs. Note that signalrclient does not *inherently* care about boost, websocketspp or openssl, rather it depends only on the public websocket client APIs provided by cpprestsdk. It would be much more maintainable to declare dependencies based on the public APIs rather than the dependencies themselves.

## 2. Other design concerns

* General-purpose Open Source projects must be able to easily and succinctly describe their build dependencies inside Vcpkg. This should be no more verbose than a single vcpkg install line and, when that command succeeds, there is a guarantee that all required functionality/headers/imports are available.
* The internal state of the Vcpkg enlistment must be either extremely transparent OR managed by version control (git). This enables larger projects to efficiently transfer the entire state of their customized Vcpkg system between machines (and onto build servers) by having the destination clone and then run a single vcpkg install line for the subset of dependencies required. The results of this operation should be as repeatable as reasonably achievable given the current limits of the underlying toolchain.

## 3. Proposed solution

A key summary of the above motivations is that they are all scenarios surrounding APIs that are not independently buildable from each other. We have an existing solution for APIs that are independently buildable: separate packages. Therefore, we seek to extend the user-facing notion of “packages” to include capabilities and contracts that may not be build-time independent.

This document proposes “features” (also called feature packages). These features are intended to model semi-independently toggleable API sets/contracts such that they can be sanely depended upon by other packages. It is not a goal to model exclusive alternatives (such as implementation choices that are not user-observable) through this mechanism.

* Individual libraries within boost may be reasonably represented as features.
* Whether a graphics library is built on DirectX xor OpenGL (where one but not both must be chosen) is not representable as a feature.

From a user experience perspective (i.e. from vcpkg install) feature packages act as much as possible like completely independent packages. However, internally, any change to a package’s features will result in a rebuild of the associated “parent” package. This will invoke a package rebuild experience similar to upgrading.

When using vcpkg install <package>, some features will be enabled by default. These default features can be avoided by referring to the packages as <package>/core and features can be added by supplying them on the same installation line.

### A. Proposed User experience

#### i. User with no preference about options

Install of a library with default features:

> vcpkg install cpprestsdk  
// -- omitted build information -- //  
Package cpprestsdk/core:x86-windows is installed.  
Package cpprestsdk/compression:x86-windows is installed.  
Package cpprestsdk/ws-client:x86-windows is installed.

Removal of that library:

> vcpkg remove --purge cpprestsdk  
The following packages will be removed:  
 cpprestsdk:x86-windows  
Removing package cpprestsdk:x86-windows...  
Removing package cpprestsdk:x86-windows... done  
Purging package cpprestsdk:x86-windows...  
Cleaned up D:\src\vcpkg\packages\cpprestsdk\_x64-windows  
Purging package cpprestsdk:x86-windows... done

Installation of a library with optional features:

> vcpkg install opencv  
// -- omitted build information -- //  
Package opencv/core:x86-windows is installed.

#### ii. User desires CUDA support for OpenCV directly, and is unfamiliar with feature packages

Developer Bob knows he wants OpenCV, so he guesses what the package is called

> vcpkg install opencv  
// -- omitted build information -- //  
Package opencv/core:x86-windows is installed.

Bob attempts to build his application against OpenCV (assuming CUDA), which fails at runtime or compile time indicating that OpenCV wasn’t built with CUDA. Bob comes back to vcpkg, not knowing about the “feature packages” feature. The primary inquiry tools for Vcpkg are search and list, so he runs vcpkg search:

> vcpkg search opencv  
opencv 3.2.0 computer vision library  
opencv/cuda support for NVidia CUDA  
opencv/contrib community supported extensions for OpenCV  
  
If your library is not listed, please open an issue at:  
 https://github.com/Microsoft/vcpkg/issues

He isn’t immediately sure what the lack of a version number means, but anything in vcpkg search can be applied to vcpkg install, so he runs:

> vcpkg install opencv/cuda  
The following packages will be rebuilt:  
 opencv:x86-windows  
  
To rebuild with this feature, use:  
 vcpkg remove --purge opencv:x86-windows  
 vcpkg install opencv/core,cuda:x86-windows

Bob follows the instructions…

> vcpkg remove --purge opencv:x86-windows  
// -- omitted results as above -- //  
> vcpkg install opencv/core,cuda:x86-windows  
// -- omitted build information -- //  
Package opencv/core:x86-windows is installed.  
Package opencv/cuda:x86-windows is installed.

and he can now use OpenCV’s CUDA support in his application.

#### iii. User is familiar with feature packages, and wants to opt-out of a feature

Developer Alice has used cpprestsdk, built it from source, and she knows about the option to disable websockets. She uses search to find the complete list of features:

> vcpkg search cpprestsdk  
cpprestsdk 2.9.0-2 C++11 JSON, REST, and OAuth library The C++ RES...  
cpprestsdk/compression Gzip compression support in the HTTP client.  
cpprestsdk/ws-client Websocket client support based on websocketspp.  
  
If your library is not listed, please open an issue at:  
 https://github.com/Microsoft/vcpkg/issues

She decided she only wants cpprestsdk/compression, so she installs only that feature:

> vcpkg install cpprestsdk/compression  
// -- omitted build information -- //  
Package cpprestsdk/core:x86-windows is installed.  
Package cpprestsdk/compression:x86-windows is installed.

She receives a quick recursive build that only depends on zlib.

She’s now interested in some additional libraries built on top of cpprestsdk: azure-storage-cpp and signalrclient.

> vcpkg install azure-storage-cpp  
// -- omitted build information -- //  
Package azure-storage-cpp/core:x86-windows is installed.  
  
> vcpkg install signalrclient  
Package signalrclient:x86-windows depends on cpprestsdk/ws-client:x86-windows.  
  
The following packages will be rebuilt:  
 \* azure-storage-cpp:x86-windows  
 \* cpprestsdk:x86-windows  
  
To rebuild the current package graph with this feature, use:  
 vcpkg remove --purge cpprestsdk:x86-windows azure-storage-cpp:x86-windows  
 vcpkg install cpprestsdk/core,compression,ws-client:x86-windows  
 vcpkg install azure-storage-cpp/core:x86-windows  
 vcpkg install signalrclient/core:x86-windows

She follows the above script and can use both azure-storage-cpp and signalrclient in her code.

Some time has passed, she decided not to use signalrclient, and she’s interested in shipping her application. She wants to minimize her final install size, so she’d like to remove all unneeded packages like boost and openssl.

> vcpkg remove boost openssl  
The following packages and features will be removed:  
 \* signalrclient/core:x86-windows  
 \* cpprestsdk/ws-client:x86-windows  
 boost/core:x86-windows  
 openssl/core:x86-windows  
  
The following packages will be rebuilt:  
 \* azure-storage-cpp:x86-windows  
 \* cpprestsdk:x86-windows  
  
Removing features requires rebuilding packages.  
To rebuild the current package graph without these features, use:  
 vcpkg remove --purge cpprestsdk:x86-windows azure-storage-cpp:x86-windows signalrclient:x86-windows openssl:x86-windows boost:x86-windows  
 vcpkg install cpprestsdk/core,compression:x86-windows  
 vcpkg install azure-storage-cpp/core:x86-windows

In the end, her final vcpkg list outputs:

> vcpkg list  
zlib/core:x86-windows 1.2.11 A compression library  
azure-storage-cpp/core:x86-windows 2.6.0 Microsoft Azure Storage Client SDK for ...  
cpprestsdk/core:x86-windows 2.9.0-2 C++11 JSON, REST, and OAuth library  
cpprestsdk/compression:x86-windows Gzip compression support in the HTTP client.

### B. Technical model

* Each package can have any number “features”.
* Features follow the same naming conventions as packages, but when referenced are always “namespaced” by the parent package.
  + cpprestsdk/ws-client is a completely orthogonal feature from poco/ws-client.
* Features are valid dependencies.
  + signalrclient depends on cpprestsdk/ws-client
* Features can have dependencies (including other features).
  + cpprestsdk/ws-client depends on boost, openssl, and websocketspp
  + opencv/cuda depends on cuda
  + opencv/contrib depends on opencv/cuda
  + boost/python depends on libpython
* Every package has an implicit feature called core, which covers the core library without any features enabled. All features implicitly depend on the core feature of their parent package
  + azure-storage-cpp depends on cpprestsdk/core
* Each package declares a list of default features that are enabled when the package is referred to by its raw name, and core is implicitly a default feature.
  + cpprestsdk declares ws-client to be a default feature. Any reference cpprestsdk implicitly means cpprestsdk/core *and* cpprestsdk/ws-client.
  + opencv does not declare cuda nor contrib to be default features.

As a conclusion of the above, it is expected that all packages will be buildable with all features disabled (just the core feature) OR with all features enabled.

## 4. Related Work

### Cargo’s Features (from Rust): <http://doc.crates.io/manifest.html#the-features-section>

The proposed feature packages are exceedingly similar to Cargo’s Features, with the following changes:

* We avoid any collision problems because features are always namespaced by the owning package
* We do not have a concept of “feature groups”, instead we allow dependencies from one feature to another within the same package (Note: This may be how “feature groups” are implemented internally to Cargo – it was not clear from the documentation).
* Because of the nature of C and C++, it is extremely commonplace that large software packages can have features disabled to remove their dependencies upon other libraries. Changing this configuration requires a rebuild of the package and potentially rippling ABI changes to any downstream dependencies. Therefore, we expect significantly more use of this feature to manage optional API contracts instead of the intended use in Cargo (curation).
* We do not intend feature packages to be used to express the curation relationship.

### Gentoo’s USE flags: <https://wiki.gentoo.org/wiki/Handbook:X86/Working/USE>

Gentoo’s USE flags can be shortly summarized as a global set of keywords that is used to make cross-cutting changes to the entire package graph’s build configuration. This system standardizes many common settings such that they can be simultaneously toggled for the entire graph. The most common example of this would be using KDE vs Gnome. A user who knows that, given the choice, they would prefer the KDE/Qt interface can manage the massive space of package configuration efficiently without learning the particular term that each package has decided to call “build using Qt instead of GTK”.

USE flags can be customized hierarchically when needed, including at the per-package level. They can be depended upon by other packages, both positively and negatively. USE flags themselves can be used in any boolean expression to determine the complete set of package dependencies, including removing dependencies when flags are enabled.

Problems with USE flags:

* They require significant coordination from package maintainers. This increases the burden of adding a package – to author a good package, I need to be aware of every uncommon USE flag and evaluate how those could map onto my local configuration space.
* Based on research online, it seems extremely common that users need to tweak flags at a per-package level. This calls into question how valuable the cross-cutting power above is.
* The vast majority of common USE flags are essentially a list of all the common packages and focus on giving the user a view of dependencies (which a package manager is designed to hide when possible) instead of features (which is what users normally care about).
* Dependency analysis with USE flags becomes a SAT problem with an enormous state space – P\*F bits – which compounds with any versioning relations. This works acceptably in practice via heuristics, but it implies that a) there is a looming performance wall which could suddenly create a poor user experience and b) the heuristics may not correctly model the user’s needs, which can cause a disconnect in desire vs practice, which again causes a poor user experience.