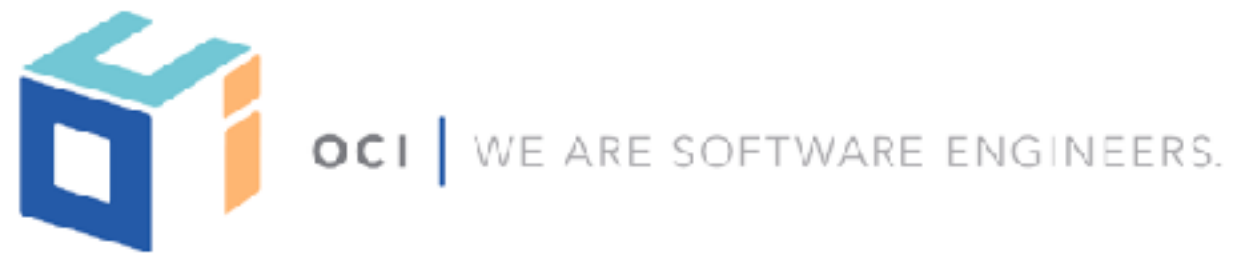


Saint Louis C++ Users Group



August 8th, 2018

Hi, welcome to the Saint Louis C++ Users Group meeting for August 2018



ociweb.com

Tonight is sponsored by Object Computing, Inc.

OCI - We Are Software Engineers

Visit ociweb.com for information.



My name is Austin Gilbert.

I'm a professional C++ developer and entrepreneur with over 15 years experience working on distributed systems and ultra-low latency real-time applications for finance and embedded systems.



Tonight we will be talking about setting up and using conan.io in your software construction process.

While conan is pretty easy to learn and use, there is a lot of material to cover tonight. As such, I ask if you have questions to please take notes and save them until the end of the presentation and then I can answer them in better detail.

What is conan?

- ❑ **C / C++ Package Manager for Developers**
- ❑ **Distributed Client/Server Dependency Manager**

First, let's talk a bit about conan and what it can do for you.

Conan is a C/C++ package manager for developers, primarily intended for use during software construction. Saying conan is “for developers” is somewhat important here as there are many other dependency & package managers out there. Many of which focus on deploying and managing dependencies for completed software. Conan in contrast is there to assist you as you build software.

Conan is also a dependency manager. As a dependency manager, conan locates and imports your project's dependencies in a useable form. Sometimes this means pulling down pre-compiled binary libraries, and sometimes this means pulling down recipes for building from source with your specific compiler profile.

Conan Features

- ☐ **Cross-platform**
- ☐ **Build-system agnostic**
- ☐ **Describes how to build a library or executable**
- ☐ **Describes dependencies on other libraries or headers**
- ☐ **Describes build-time dependencies, e.g. build tools needed**
- ☐ **Transitive Dependencies**

Conan is cross-platform. It will run anywhere python can run.

Conan is build-system agnostic, it will work with CMake, msbuild, makefiles, etc. A little bit later in this presentation, I'll be showing you how conan integrates with cmake.

Conan allows you to describe how to build your library, to declare build-tool dependencies and source code dependencies, and allows you to export

- ☐ **Packages use semantic versioning**
- ☐ **Provides channel semantics for choosing different variants of a package, e.g. release vs. release candidate vs. feature branch**
- ☐ **Create binary, source, and/or header-only packages as deliverable artifacts**
- ☐ **Test linking of libraries before deploying**

- ☐ **Improves developer productivity, especially when on-boarding**
- ☐ **Flexibility - packages are just Python**
- ☐ **Private and public repositories are supported**
- ☐ **FOSS**

Conan uses a user/channel mechanism to allow package variants to co-exist in the same environment. This is useful for maintaining release packages alongside release candidates, or perhaps publishing a conan package of your feature branch when collaborating with others.

Conan is Free and Open Source Software hosted on GitHub.

<< **Installing Conan** >>

Step by step...

- ☐ Install python
- ☐ Install pip
- ☐ pip install virtualenv
- ☐ virtualenv conan
- ☐ Activate your virtual environment: `. ~/conan/bin/activate`
- ☐ pip install conan

You will need to install python, pip, and virtualenv if you don't already have them. I'm going to assume you already have these installed.

If you prefer to use **pipenv** you can, but I'm going to stick with **virtualenv** in this example. You don't need a virtual environment at all, really, but it is useful for testing new conan versions side-by-side existing versions before committing to upgrading.

install_conan.mov

<< **Conan Directory Structure** >>

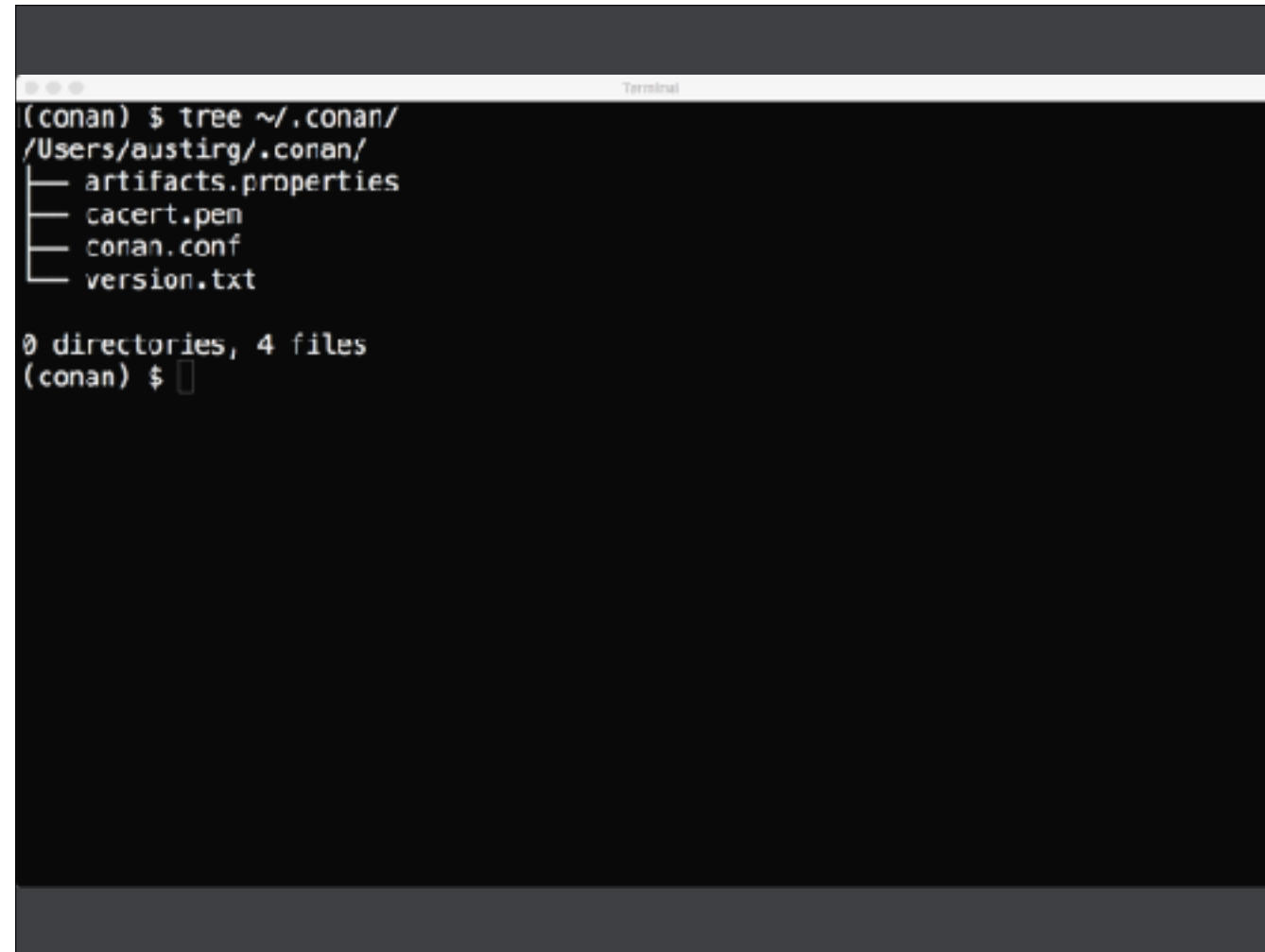
The first time you run conan, it will create a ~/.conan directory structure under your home directory.



The first time you run conan, it will create a **.conan** directory in your home directory. As you can see there isn't much there at the moment.

The **conan.conf** file is the most significant file here at the moment; it contains two settings of interest, where your conan cache directory will live and what proxy if any you need to go through when using conan.

There is also a **version.txt** file here which helps you keep track of which conan version you're running.

A terminal window titled "Terminal" showing the output of the command `tree ~/.conan/`. The output lists four files: `artifacts.properties`, `cacert.pem`, `conan.conf`, and `version.txt`. Below the file list, it states `0 directories, 4 files`. The prompt `(conan) $` is followed by a cursor.

```
(conan) $ tree ~/.conan/  
/Users/austing/.conan/  
├── artifacts.properties  
├── cacert.pem  
├── conan.conf  
└── version.txt  
  
0 directories, 4 files  
(conan) $
```

Here is the `~/.conan` directory structure as a still.

<< **Simple Example** >>

Okay, so we've talked a little bit out conan, now let's walk through a simple example to demonstrate some basics.



proj1_build_fails.mov

So I've setup a trivial project which uses CMake to build a static library. It has one function called "hello" which will be in the public interface.

We're going to make this trivial library dependent on Boost.Format just so we can demonstrate how conan deals with dependencies.

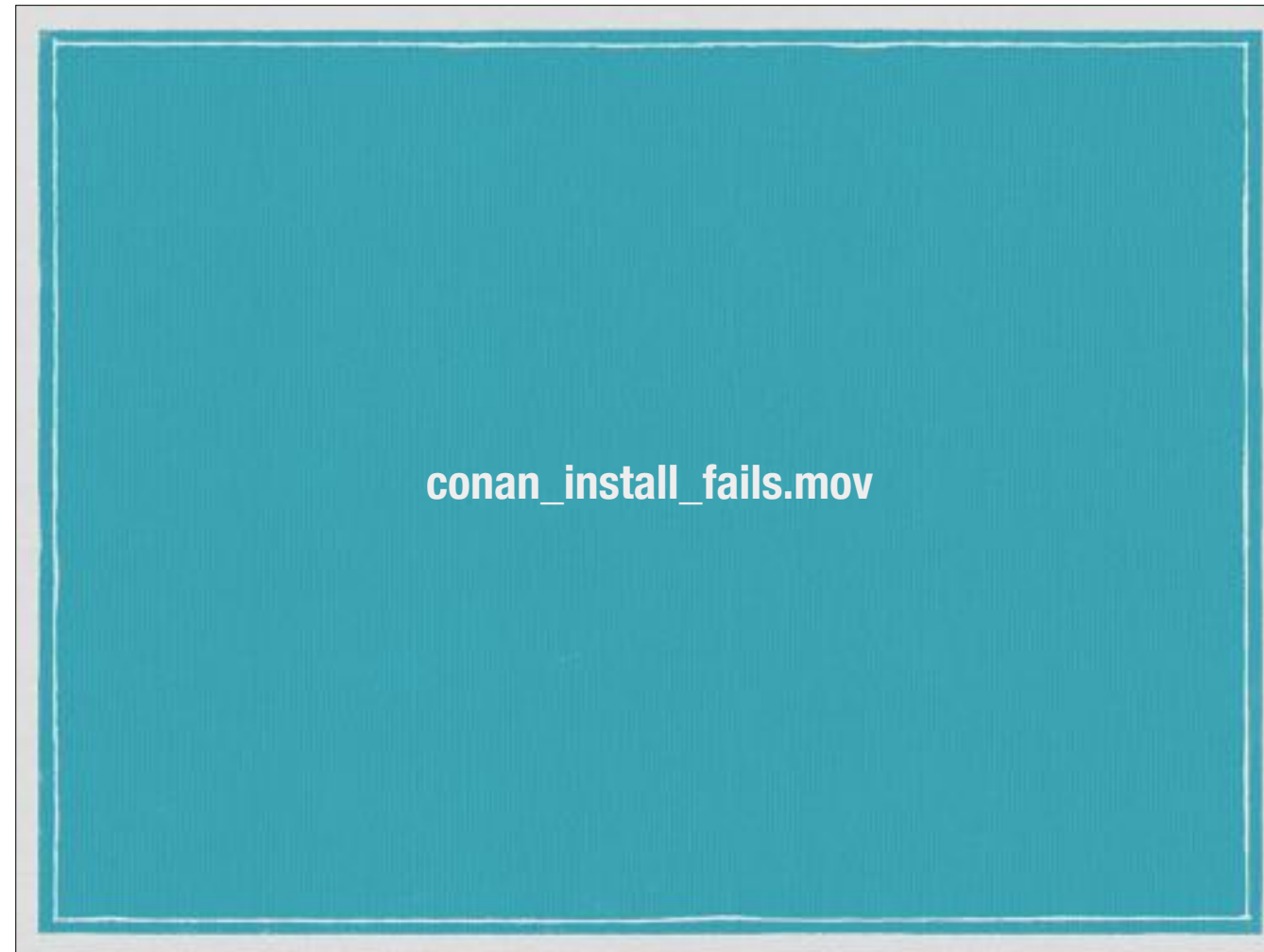
As you can see, the CMake is currently configured to find boost using a CMake Finder. However, because I'm using conan, I do not have boost installed locally - so when we attempt to build this project, it will fail because it cannot find boost or UnitTest++. We're not actually using UnitTest++, so I'll just remove that dependency.



<<< conanfile.txt >>>

So we have a trivial static library which depends on boost. This is the simplest use case for conan, we are only consuming packages (declaring dependencies).

To handle our scenario, all we need to do is define a **conanfile.txt** with the packages we require.



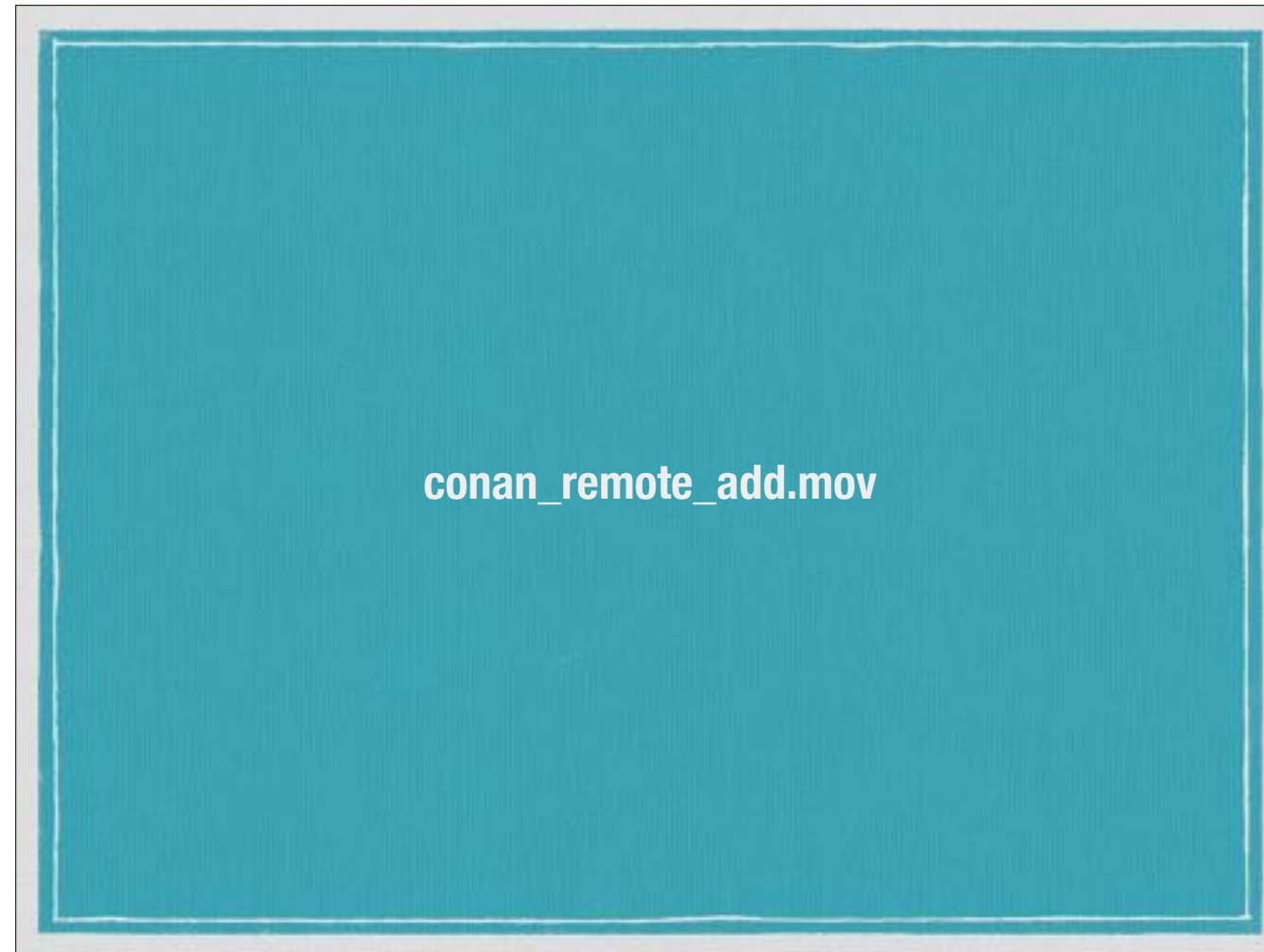
We're creating our **conanfile.txt** and adding a **requires** section, under which we will declare we want a version of boost between 1.63.0 and 1.68.0.

And now when we run “**conan install**” it fails. This is the first time we've run “**conan install**” so it is going to create some files we don't have yet, like our **default profile** and **registry.txt**.

<<< **Setup BinTray** >>>

Conan failed to locate our dependency because it did not exist in our local cache, or in any remote repository we had defined. We can remedy this by adding a remote repo where the package can be found.

We're going to use the public conan repository hosted on BinTray and populated by Bincrafters.



To do this, we use the “**conan remote add**” command, passing it the name of the remote and the URL as parameters. **Success.**

Now, we’re going to query the remote for our dependency to make sure its there. We use the “**conan search**” command for this. We’re using the “-r” option to search a remote, if this option is left off, the search is through our own local cache only.

Our query is successful, the repo has several versions to choose from.

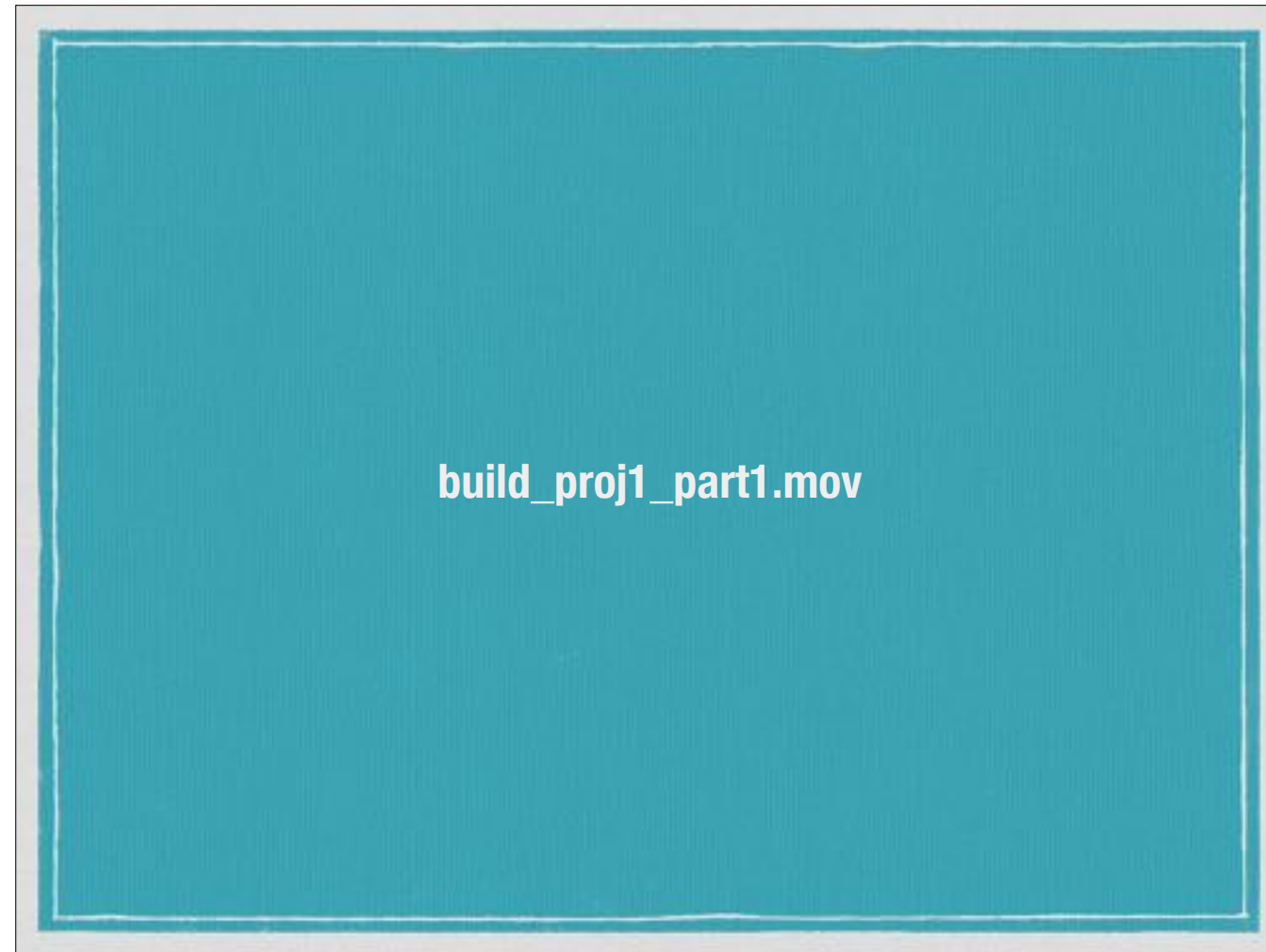


<https://bincrafters.github.io/>

To keep up with the Bincrafters project visit their website at bincrafters.github.io

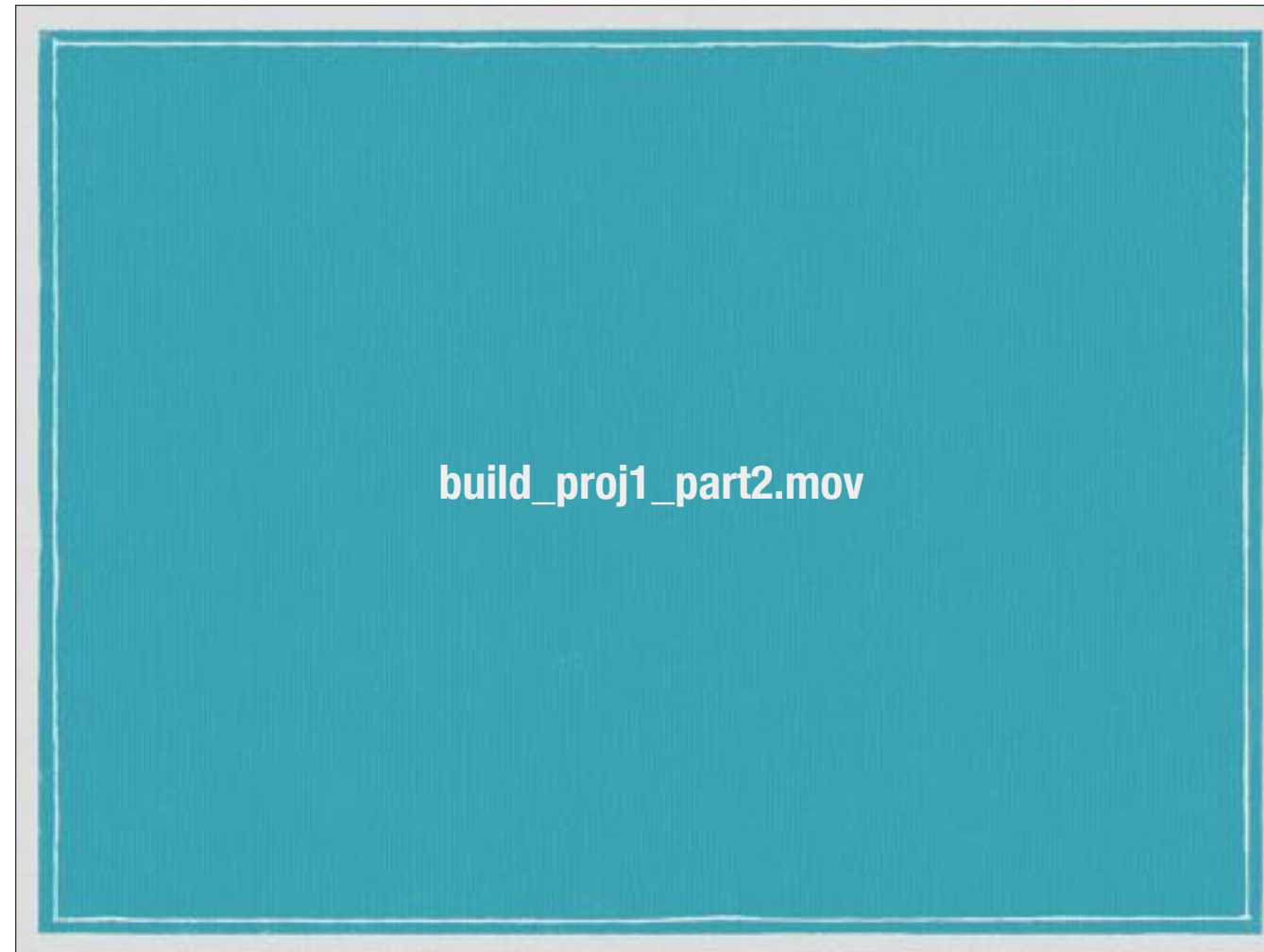
<<< **Building It** >>>

Let's build our project again.



Okay, good, conan has contacted the bincrafters repo and has found boost and is building everything we need to.

Ah, the build still fails. Okay, we forgot a few things...



We need to remove our normal CMake dependency declarations to prevent CMake from using its find scripts.

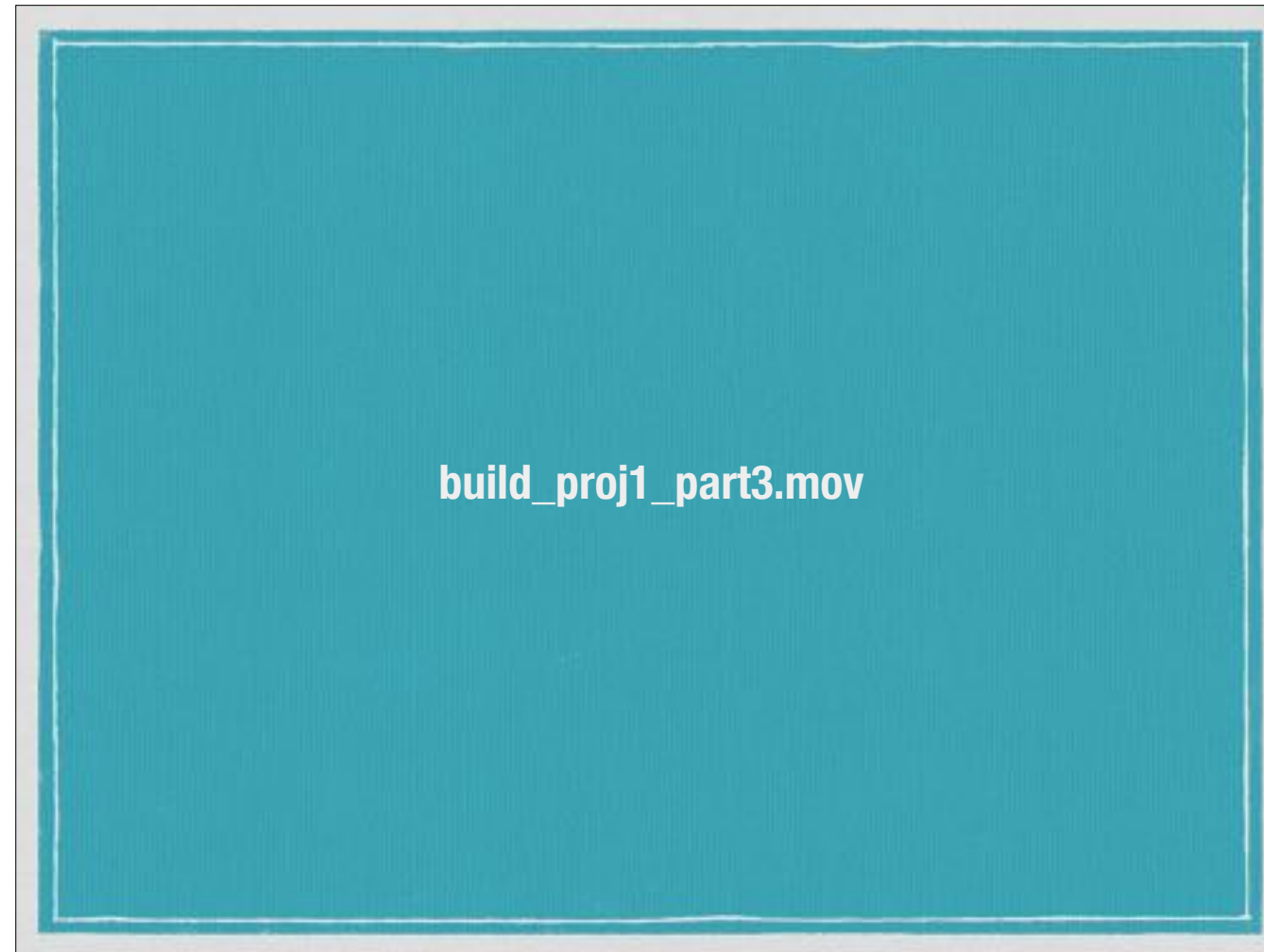
And we need to change how we reference our dependencies for linking - we'll change these to **CONAN_PKG::boost_format**. CMake will complain about using this type of dependency, it can be suppressed using `cmake_policy`, but we'll just ignore it for expediency.

And now, great, conan runs without issues... and we generate CMake without issues.

Let's look briefly at what conan generated for CMake to use. Basically, conan just creates a list of include directories and library directories for CMake to use when building and linking.

Now let's look at our `~/conan` directory again, there's more stuff there now. Specifically, you can see there is a **data** directory. This directory is the local cache folder where conan stores packages we need to use. So in our **data** dir we can see we've got some boost stuff as expected. And if we keep drilling down, we can get a peak at the conan package structure. These details aren't normally important, it can be convenient to know where to look if you're in deep trouble debugging a conan issue; if you're interested in knowing more, I suggest using conan and diving in.

Okay, let's build our project by running **ninja**. And great, it built and we have a static lib available. To recap, the sequence is “**conan install**”, “**cmake**”, and then invoke your build tool.

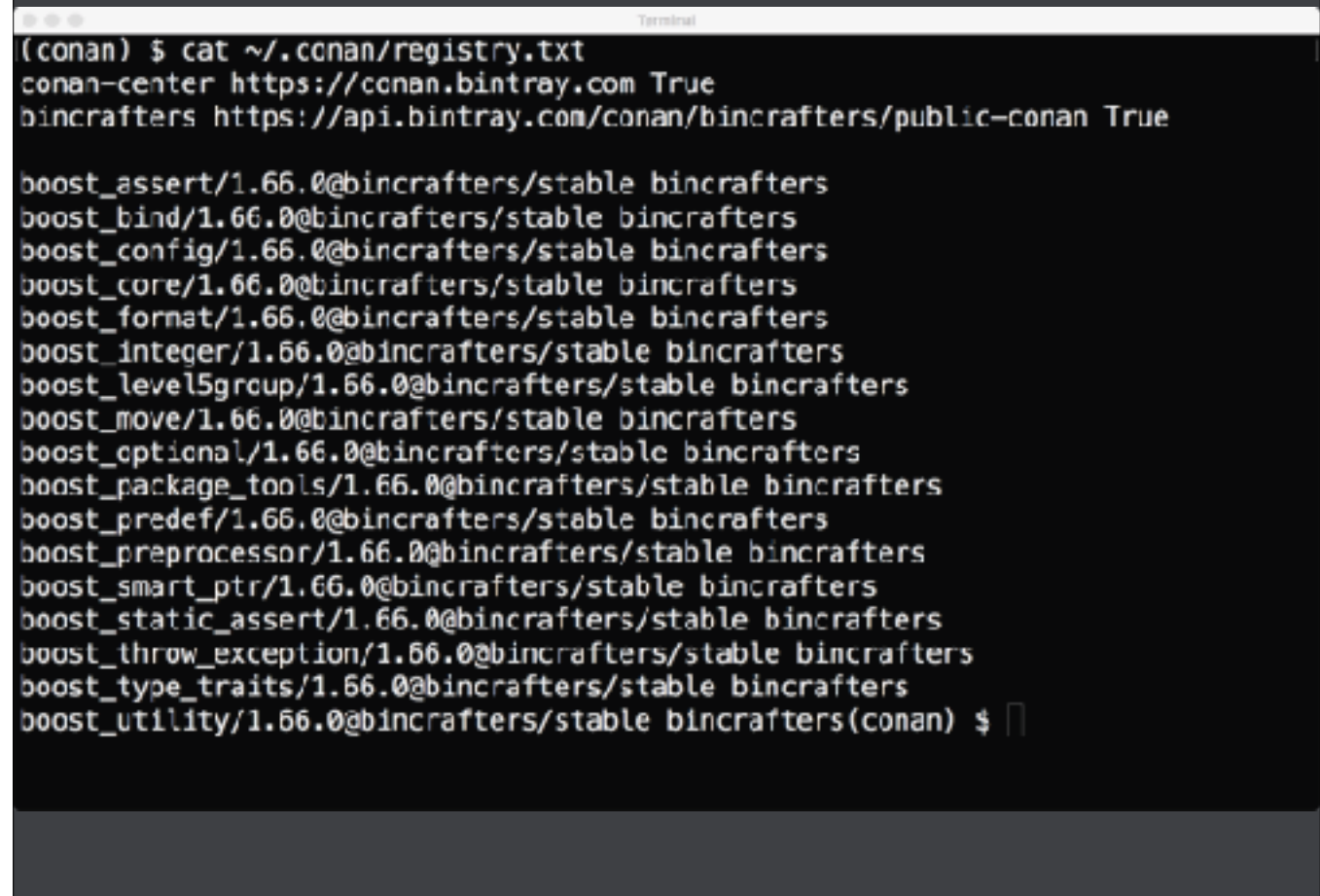


Let's take another look at what conan added after we successfully ran “**conan install**”.

It added a default profile with some conan setting information. Here we can see, I'm on macOS, I'm using clang, the architecture is amd64, the compiler version is 9.1, the libcxx is **libc++** (note there's no version information for libcxx, this is important later). And we have our build_type. These settings are meta data that are encoded in conan packages we create.

We have a **registry.txt** file. This holds all the conan repositories and packages we know about.

The **settings.yml** is an interesting file. It controls what values are considered valid for some settings. Here you can see we've some **os** options, some compilers etc. If you find something in your environment is missing, you can edit this file to add it. Just be aware everyone you work with will need the edits. We'll cover a convenient way to distribute the same settings to everyone later.

A terminal window titled "Terminal" with a dark background and light text. It shows the command `cat ~/.conan/registry.txt` and its output. The output lists two repositories: `conan-center` and `bincrafters`, both pointing to `https://bintray.com` with `True` for the `public-conan` flag. Below this, a list of 17 Boost packages is shown, each with its version `1.66.0`, the repository `bincrafters`, and the channel `stable`.

```
(conan) $ cat ~/.conan/registry.txt
conan-center https://conan.bintray.com True
bincrafters https://api.bintray.com/conan/bincrafters/public-conan True

boost_assert/1.66.0@bincrafters/stable bincrafters
boost_bind/1.66.0@bincrafters/stable bincrafters
boost_config/1.66.0@bincrafters/stable bincrafters
boost_core/1.66.0@bincrafters/stable bincrafters
boost_format/1.66.0@bincrafters/stable bincrafters
boost_integer/1.66.0@bincrafters/stable bincrafters
boost_level5group/1.66.0@bincrafters/stable bincrafters
boost_move/1.66.0@bincrafters/stable bincrafters
boost_optional/1.66.0@bincrafters/stable bincrafters
boost_package_tools/1.66.0@bincrafters/stable bincrafters
boost_predef/1.66.0@bincrafters/stable bincrafters
boost_preprocessor/1.66.0@bincrafters/stable bincrafters
boost_smart_ptr/1.66.0@bincrafters/stable bincrafters
boost_static_assert/1.66.0@bincrafters/stable bincrafters
boost_throw_exception/1.66.0@bincrafters/stable bincrafters
boost_type_traits/1.66.0@bincrafters/stable bincrafters
boost_utility/1.66.0@bincrafters/stable bincrafters
(conan) $
```

Let's take another look at our `~/.conan/registry.txt` file. After successfully running "**conan install**" and pulling in our boost dependencies, we can see that our **registry.txt** file contains some additional information at the bottom.

It now shows us not only the repositories available, but what packages we have in our local cache, and which repository they came from.



https://docs.conan.io/en/latest/getting_started.html

If you need a recap of what we've done thus far. Visit this URL.

You'll notice they included the conan libraries differently in CMake on account they're using a less advanced CMake setup, their way is probably preferred if it will work in your setup.



```
<<< conanfile.py >>>
```

So we built our library. But what we'd really like to do is add this library to our conan cache so we can use it in our other projects. However, we can't create and export packages using a **conanfile.txt**, to do this we need to use a **conanfile.py** file format.

Let's remove our **conanfile.txt** file and create a **conanfile.py** now.



The quickest way to get a **conanfile.py** is to use the “**conan new**” command.

Here, I’m passing the “—test” option to add a **test_package/** directory which will be used to test linking our library, this is useful to prevent bad builds from being shared.



<<Edit video>> keep: 0-0:42, splice in conan_create_splice.mov, keep 10:30-end

Once we have everything working, if we owned a conan repository, we could then upload our package using “**conan upload**”, a command we’ll cover a bit later.

<< Package Reference >>

Now let's go back a moment and take a closer look at how we added boost as a dependency to our project.

In previous versions, this was called a package reference or a package specifier, but it appears docs.conan.io is referring to it more simply as a “package” now. For this presentation, we'll use these as interchangeable terms.

To declare a dependency, we added a package specifier or package reference to our **conanfile.txt** and **conanfile.py** files. Let's take a look at how package specifiers are built.



package/1.0.1@user/channel

Each conan package is described with a triplet.



package/1.0.1@user/channel

The first part is the package name. As we saw when using a portion of the boost libraries, our package name was boost_format.



`package/1.0.1@user/channel`

The second part is the package version, which is also set in a conanfile.py file.

Conan uses **semantic versioning** for package versions. Here we are requiring a specific version of ‘package’

A large orange square with a thin black border, containing the URL <https://semver.org/> in white text.

<https://semver.org/>

If you aren't familiar with semantic versioning, it's documented at semver.org. It's a double edged sword, conan works awesome until you need to use a project which doesn't use semantic versioning exactly as defined by semver.org.... In those cases, you have some extra munging work to do to make the project useable by conan.

A large orange square with a thin black border, containing the text 'package/~=1.0@user/channel' in white.

`package/~=1.0@user/channel`

Semantic versioning allows us to say things like “I want a version around 1.0”. This would match 1.0.1, 1.0.2, and so on, but not match 1.1.0

`package/[>1.0.0,<1.4.0]@user/channel`

Allow for a range of packages.



`package/[>1.2 || 1.9.9]@user/channel`

Version conditions can be logically OR'd



`package/1.0.1@user/channel`

The final part of a package reference are the package **owner (or user)** and **channel information**. Canonically, the owner is typically named after the server or group hosting the package. For the name of the channel, **Stable** is used for stable releases, **testing** is used for early access packages, and **develop** might be used for in development packages.



<< proj2 & proj3 >>

Now we're going to quickly add two more projects to our example to demonstrate how our local conan cache can be used for resolving dependencies.



Here, I've create a project basically like proj1, except our function says “goodbye” instead of “hello”. This trivial project is also dependent on boost, and I've already created it, so it exists in our local cache.



Proj 3 will use proj1 and proj2 to say hello and goodbye.

Proj3 will be an executable that uses proj1 and proj2.

In the conanfile.py we can see we're declaring proj1 and proj2 as dependencies. It is important to note, since we're using pre-release tags we have to use a specially defined range for semver to match correctly. Greater than 0.0.0-0 and less than 0.0.1 in this case.

Let's build our project.

Okay, we got an error. This error is telling us we have conflicting dependencies. The proj2 available in our cache used boost 1.65.1, and the proj1 available in our cache used boost 1.66.0. *To me, this is one of the coolest features of conan; when there is a conflict in dependencies, it stops.*

Looking at our conanfile.py files for each project, we can confirm the error is correct. We need to resolve the conflict before the project will build. We'll change the range proj2 is using so it can also match against boost 1.66.0.

Now we'll just rebuild it with "conan create" so the latest version is in our conan cache.

And once we go back to proj3, we should be able to build successfully now.

And there it is, great.



<< conan remove >>

Okay, we've seen "**conan create**" and "**conan install**" which can add conan packages to our local cache, now we'll quickly demonstrate how to remove a package from our local cache using "**conan remove**"

You might use "**conan remove**" to clean up old versions packages you aren't using anymore or to clean out an errant package.



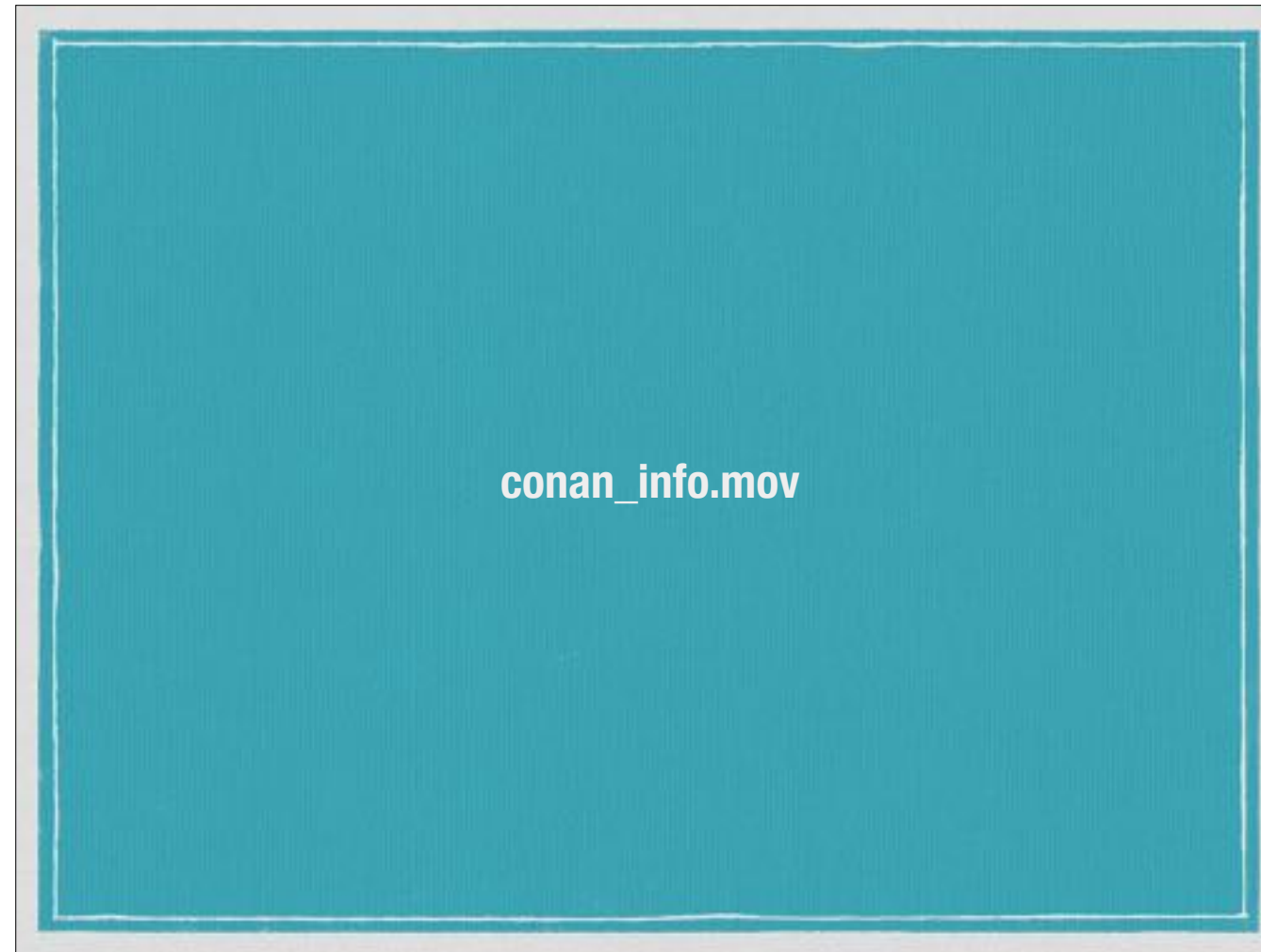
conan_remove.mov

Let's say we need to remove some package from our conan cache. We could locate it in the file system and delete it, but the recommend approach is to use “**conan remove**”.

If you do get into a jam you can't figure out how to resolve, you can **rm -rf ~/.conan/data** and rebuild everything.

conan remove is particularly useful when you have many development packages and you want to cleanup space on the filesystem.

<< conan info >>



“**conan info**” is a command that’s useful for looking at your projects dependencies. It will give some information about each package, like it’s URL and license.

For each printed entry, it will tell you what packages require it and what packages it requires.

I use this most often to look at the dependency graph. “**conan info —graph dependencies.dot**”. This particular graph is very busy, and if my project were larger, I would likely edit the DOT file before generating an image, because I don’t care about the boost dependencies so much and I’d mostly be interested in seeing the dependencies that existed between modules I’d created myself.

<< TRUTH TIME >>

**Conan is not terribly
useful ...**



... by itself

We've covered two useful conan scenarios so far.

- (1) Consuming packages created by others and publicly available.
- (2) Creating our own packages which exist only in our local conan cache.

While you can get a fair amount of mileage with just this functionality, it isn't until you begin to share packages between developers that you truly harness Conan's potential.

To demonstrate more fully the potential impact conan can have, we need to see it working in concert other development infrastructure. We're going to quickly setup and use a more enterprise environment with conan.

An Enterprise Environment

- ☐ GitLab to host C/C++ source code
- ☐ GitLab repository to host Conan Settings
- ☐ GitLab Runner configured to build conan packages on commit
- ☐ Built packages are exported to JFrog Artifactory artifact repository
- ☐ Conan.io to define and manage dependencies between library sources

I will now walk through setting up and using conan in an enterprise environment.

We will host our source code in GitLab. I'm using GitLab because it has **docker images** available which makes setup easy. You can substitute most any enterprise quality source control system instead.

Conan has the ability to import settings from a URL, so we'll be setting up a repo to hold conan settings which we can use to distribute a common conan configuration to all our corporate conan users. This is a big deal, doing this will save you much pain and suffering on-boarding developers to using conan.

We'll be using GitLab's CI pipelines to build our C++ projects inside a docker. Conan will be used to initiate the builds, and once a successful build is completed, conan will publish the package to our artifact repository, JFrog's artifactory.

I'll be using JFrog's Artifactory to host conan packages because docker images are available making setup easy, and because it has excellent support for conan.

And of course, we'll be using conan to define a package we want to build and to manage dependencies between some library sources.

<< **JBrog Artifactory** >>

We'll setup artifactory first

Pull Docker Image

```
$ docker pull docker.bintray.io/jfrog/artifactory-cpp-ce
```

Run Image

```
$ sudo docker run -d --restart=always -p 8081:8081 docker.bintray.io/jfrog/artifactory-cpp-ce
```

Here we use “docker pull” to grab the artifactory image from [bintray.io](https://bintray.com/jfrog/artifactory-cpp-ce). Then we start the image in a container while mapping some ports to make them available on localhost.

Note we’re using the community edition that includes support for C++ here. Artifactory is not a free product, please review the terms of use with your attorneys before using the community edition in production.

<<< Configure Artifactory >>>

config_artifactory.mov



Let's add our artifactory repo to our conan registry. We use “**conan remote add**” to do this.

We can see conan added our remote to our registry.txt file.

Now let's see if we can communicate with our repo, yes, our package is not there when we search for it, that's what we expected. Now let's check our local cache. Good, it's in our local cache as expected.

Now let's upload our package and search again. When you want binary artifacts to be stored in artifactory, DO NOT forget the **—all** here or conan will only upload the recipe for your package and you'll waste hours trying to figure out why binary builds are not working.

Great, conan listed all the files it uploaded. Everything looks to be in order. And let's search our repo again. And there it is. So our artifactory setup appears to be in working order.

(Cut at 1:10)



I'd like to quickly show you why JFrog's artifactory is a good choice for hosting your conan repository.

Let's lookup our package in artifactory and browse some of the details.

Make sure search type is set to "package" and package type is set to "conan".

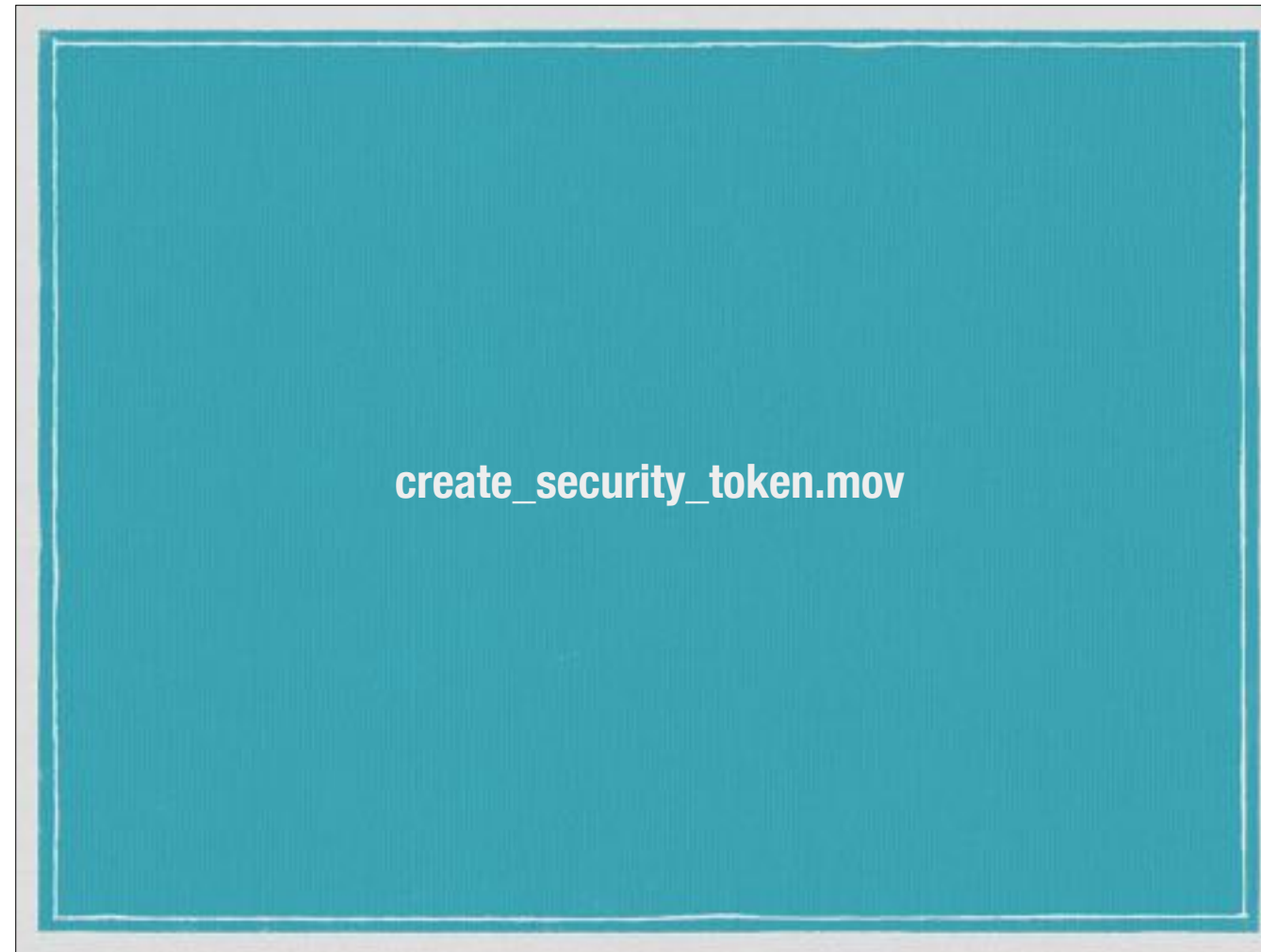
And there's our package.

Artifactory shows us a wealth of useful information. The version, the user, the channel, the repository its hosted in, and the date it was last modified. We can sort by each of these if need be.

And if we drill into the package, artifactory will show us the package details. There will be one package entry here for each binary built with different settings.

On the right we can see the repository path, the time the package was created, and the user who deployed the package.

And if we drill into the package we can see the files it contains. The conan settings the package was built with, the options the package was built with, and which other packages this package requires.



So far, I've uploaded packages using an artifactory username and password, but once we setup the CI pipeline, we'll want to use a security token.

You create a security token by using artifactory's REST API. Note, the username I'm supplying here does not exist and SHOULD NEVER exist. The group I'm adding this fake user to DOES exist and has permission to add packages to our conan repo.



<< GitLab >>

Okay, now for GitLab

Pull Docker Image



```
$ docker pull gitlab/gitlab-ce
```

Run Image



```
$ sudo docker run -d --restart=always -p 8080:80 gitlab/gitlab-ce
```

Basically, the same setup as when we ran artifactory, except the ports and image are different.

Again, we're running a community edition of GitLab, please review the terms of use with your attorneys before using in production.



config_gitlab.mov
gitlab_register_and_create_accounts.mov

Real quickly, we're going to get GitLab going and setup an administrative user. Then register a normal user to upload code with.

Now we'll create a group for our code, and we'll create a repo for each of our libraries and our executable. (Play in fast forward?)



¡Magic!

I went ahead and pushed our 3 projects into GitLab, I'm going to skip walking through how to do that as git and gitlab is a bit out of scope.



This is the part of the video where I was going to setup GitLab Runners in docker that would build our project and upload the conan packages to conan automatically.

However, due to some quirks with docker and network name resolution and how GitLab Runner works, I wasn't able to do this. I was going to fallback and simply use Jenkins, but I ran out of time, sorry.

An Enterprise Environment

- ☐ GitLab to host C/C++ source code
- ☐ GitLab repository to host Conan Settings
- ☐ ~~GitLab Runner configured to build conan packages on commit~~
- ☐ Built packages are exported to JFrog Artifactory artifact repository
- ☐ Conan.io to define and manage dependencies between library sources

So, we're scratching GitLab Runner and CI integration. I'm very sorry about this. You'll just have to take my word for it, you want CI pipeline integration uploading packages to artifactory.

I highly recommend using GitLab Runner in your production environment, the integration with GitLab works really really well....

Okay, moving on.

<< **Settings & Profiles** >>

The last step we need to complete before we can “deploy” conan in our enterprise environment is to configure a common set of settings and profiles which will be distributed to all our developers and build servers.

This is done to ensure everyone is using the same compiler settings, etc. But, as I will demonstrate, it greatly reduces the time needed to on-board developers into the conan workflow.



<<< **conan config** >>>

The command to import settings into conan is “**conan config install**” with a URL hosting the settings.

The URL can point to a ZIP file or a git repository. I prefer using a git repo for obvious reasons, it can be used in the event settings have changed but you need to reproduce an earlier build with different settings.

Settings

- ☐ **settings.yml**
- ☐ **remotes.txt**
- ☐ **conan/conan.conf**
- ☐ **profiles/**
- ☐ **profiles/MyProfile**

Here are the files you can distribute via ZIP file or Git repository:

- **settings.yml** is the same file as we've already seen.
- **remotes.txt** will contain remote repositories in the order we want them, and will put this into our **~/.conan/registry.txt** file.
- **conan/conan.conf** will allow us to override only a few settings, like the proxy server or cache location.
- we can put as many different profiles as we wish in the **profiles/** folder.
- Putting a **MyProfile** file in the **profiles/** folder will cause conan to override the contents of the default profile. This is useful if you have a standard development image all developers should be using.

Platforms

- ☐ **macOS High Sierra**
- ☐ **Ubuntu 18.04**
- ☐ **openSUSE Leap 15.0**
- ☐ **Centos 6**
- ☐ **Centos 7**
- ☐ **OpenBSD 6.3**

We will be setting up conan settings & profiles, then demonstrating how they can be used to ensure our project works across many platforms.

Conan has a small problem supporting Linux. Each distribution of Linux, and potentially each version of each distribution comes with a potentially different compiler version, different runtime libraries, and different versions of default system libraries, libraries such as OpenSSL.

Conan accounts for compiler versions but not minor compiler version numbers. This can be problematic because the behavior of the compiler when dealing with undefined behaviors or compiler specific behaviors are allowed to change between minor update versions (depending on the compiler - I've seen this with Visual Studio in particular). We on our own dealing with this.

More importantly, Conan does NOT account for different versions of the standard runtime on different systems. While conan allows us to differentiate between libstdc++ and libc++ we cannot differentiate between different versions of libc++ using conan. We are on our own. Many bugs were filed about this issue, but were put down.

Since conan doesn't account for these differences, they are not encoded in the meta data for conan packages, meaning you could get a binary built against a different libstdc++ for example. Your first clue something is wrong could be missing symbols when linking; it is not fun to track down the first time it happens.

Tonight, I will walk you through what I believe is the most elegant way to deal with this in conan, by using settings and profiles. It adds a bit of noise to the command lines we've been using, but if you work in an environment where your home directory is hosted on a network share and mounted when you log into various machines, it is a necessary step to avoid time wasting pitfalls.

We will setup settings for dealing with Linux distributions, and we will setup profiles to support the following platforms.



Here is the basic layout of the conan settings folder.

The **settings.yml** file will overwrite the settings.yml in ~/.conan/.

The **remotes.txt** file will overwrite the remotes section of the registry.txt file in ~/.conan

The **conan/conan.conf** allows overwriting of some settings, like a proxy server or cache directory.

The biggest win is the ability to distribute pre-configured profiles. The profiles are in the **profiles/**. As you can see, I've defined a profile and a debug profile for each platform we have in our environment.



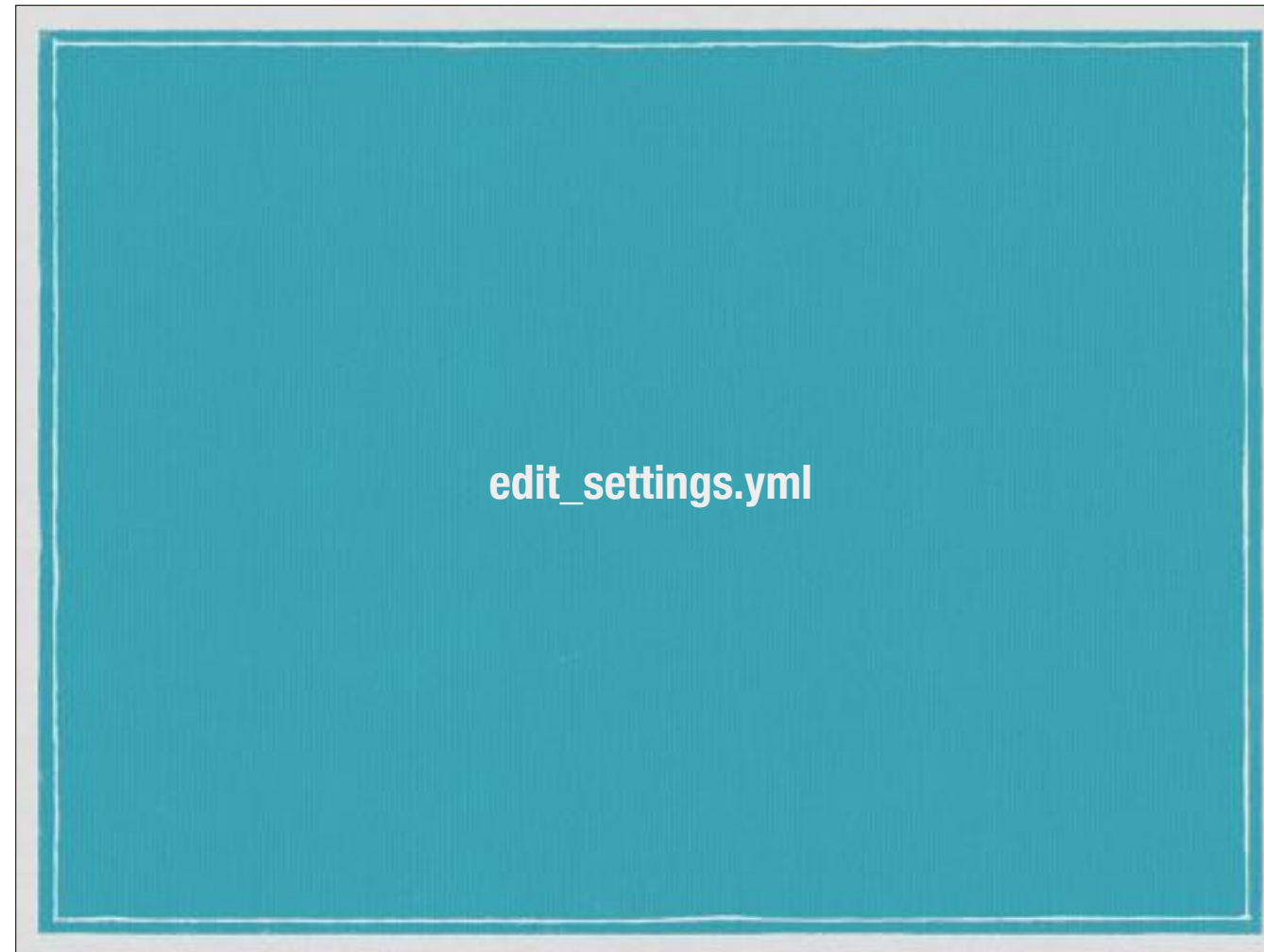
add_configure.mov

Now let's demonstrate how to solve the Linux Distribution issue we talked about. First, in our `conanfile.py` we will add a `configure()` function to check some settings coming in from our conan profile file.

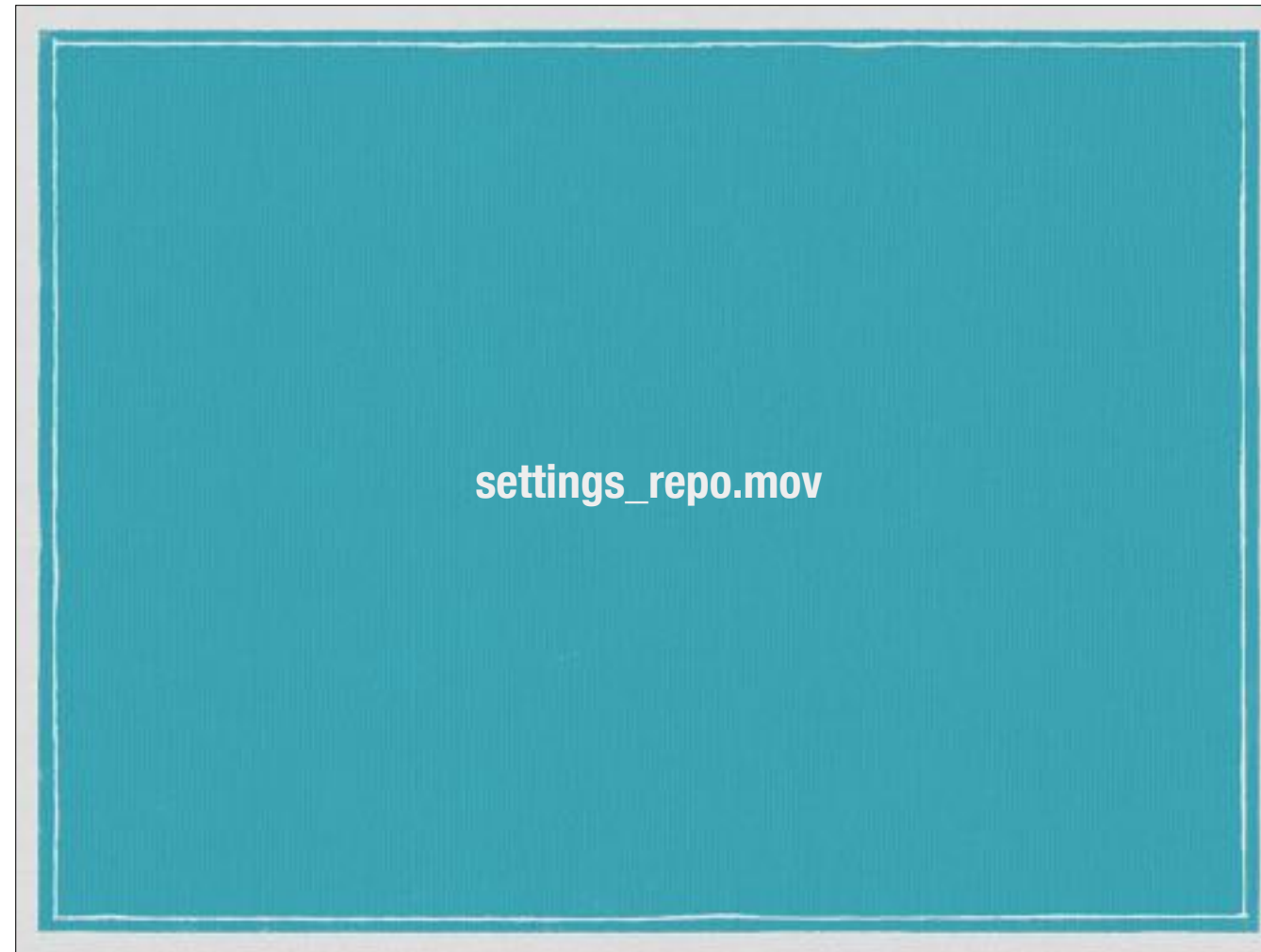
We're going to make this one a little more realistic, on Linux, if the distribution is not defined, we will raise an exception.

On macOS, we need to know the version as the compiler versions change with the macOS version and that means the libc++ version could have changed as well.

Likewise on Windows, we might want to know which subsystem we're targeting.



Now we're going to edit our settings.yml file to add Linux distributions we care about - we have to do this because these are not there by default.



Of course, we have our conan settings and profiles in a git repo.



profiles_in_action.mov

So I setup VMs with a few different distros we might find in an enterprise environment. I've got them loaded with our development tools. CMake 3.10 and git 2.x, plus their native compilers.

Now we want to see what the on-boarding experience would be like for a new developer coming onto a project who needs to do some work on proj3. In the process, we'll demonstrate how profiles work.



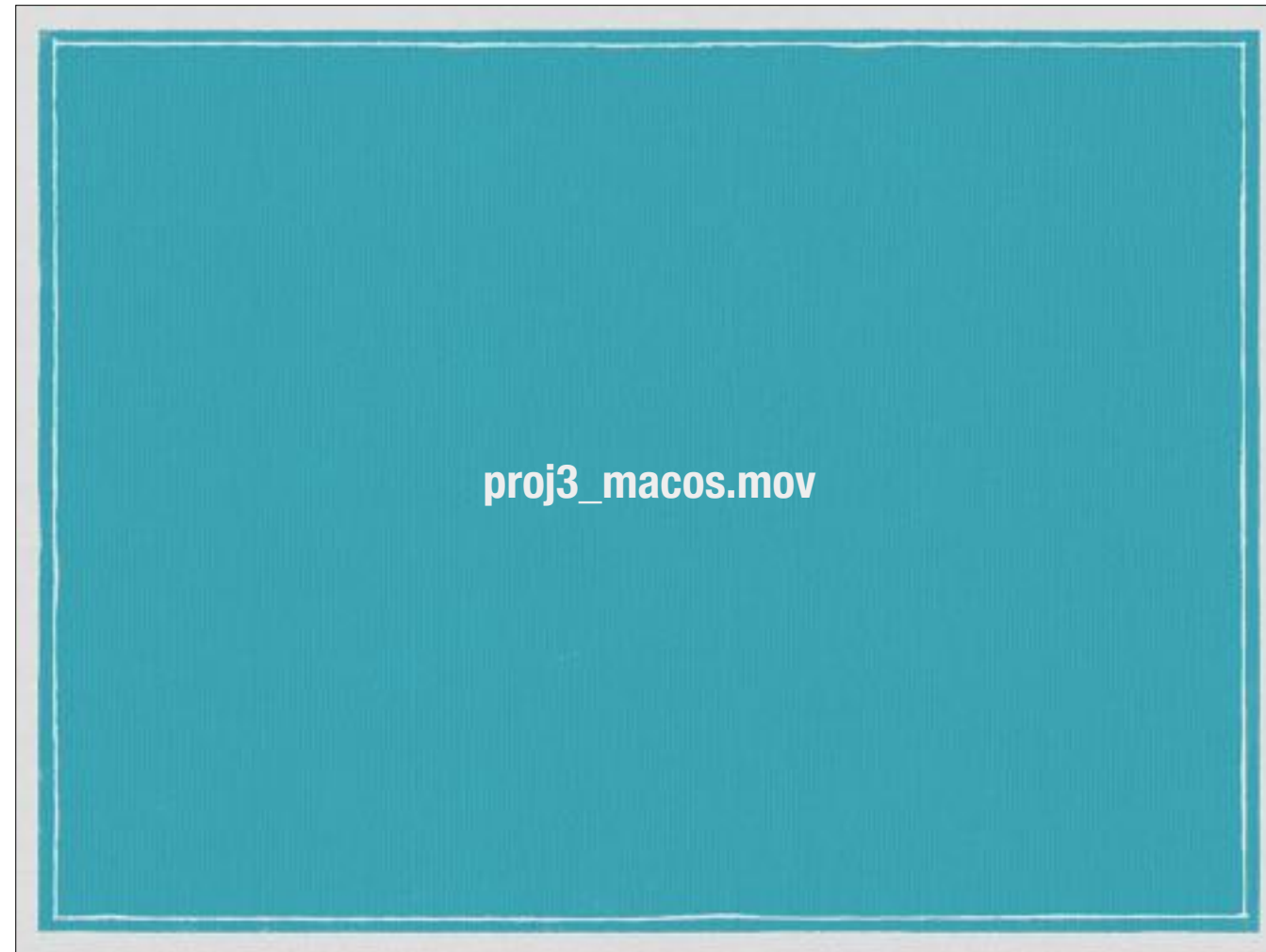
proj3_openbsd.mov

I'm going to delete my entire `~/conan` directory to simulate a new developer starting working on a project from nearly scratch, all the dev tools are already installed.

Then I'm going to install all our conan settings & profiles, and copy the profile I want to be my default, in case I ever forget to use `—profile` when using conan.

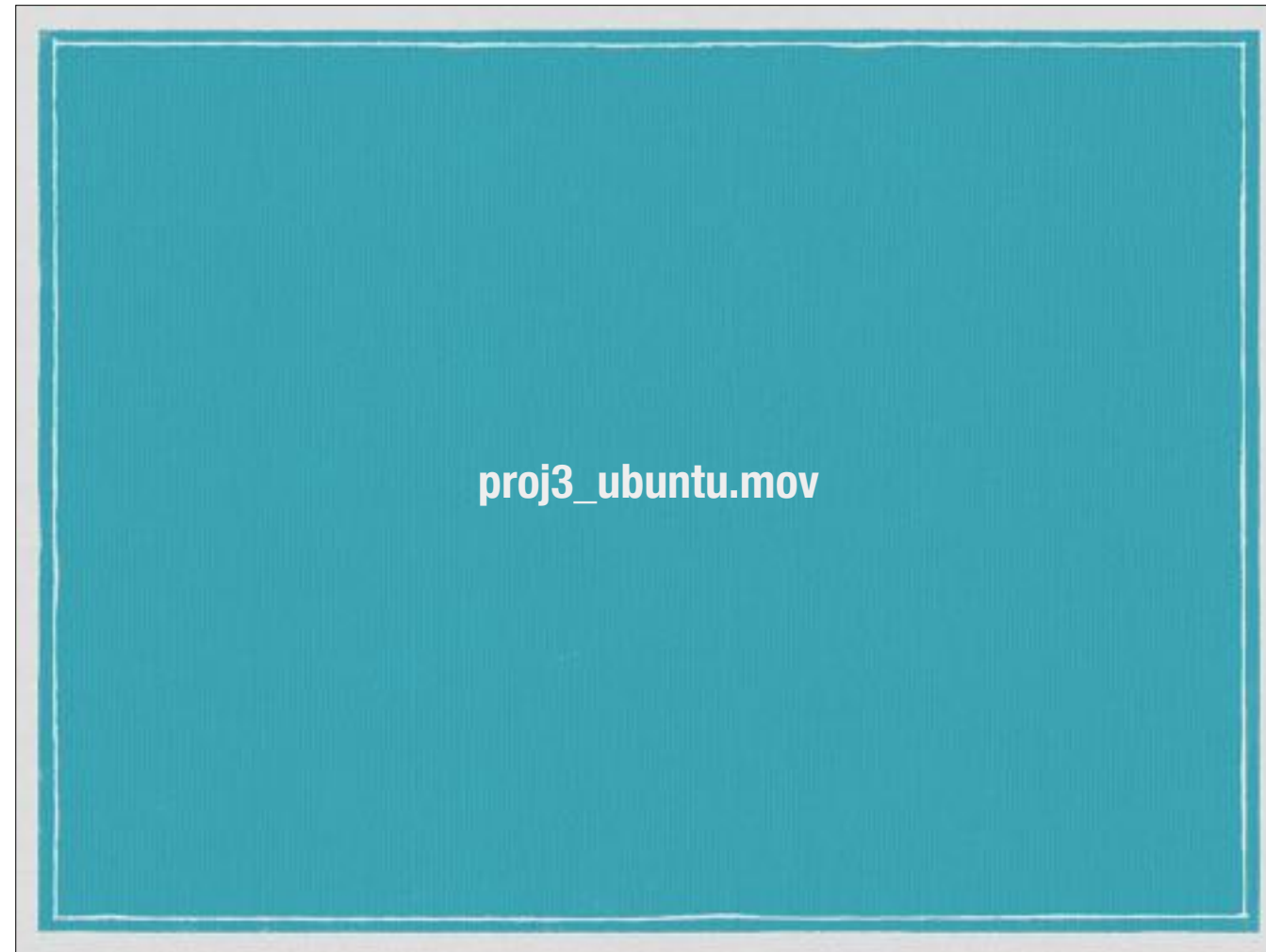
Now I'll pull the source code for the executable I'll be working on. Make a build directory, run “conan install” using `—profile` and `—build=missing`. Run `cmake`. And finally run `ninja` to build.

Obviously, if I wanted to use code blocks or make files, I would use a different CMake generator.



Now we'll do the same thing on macOS, and everything just works.

((play in fast forward))



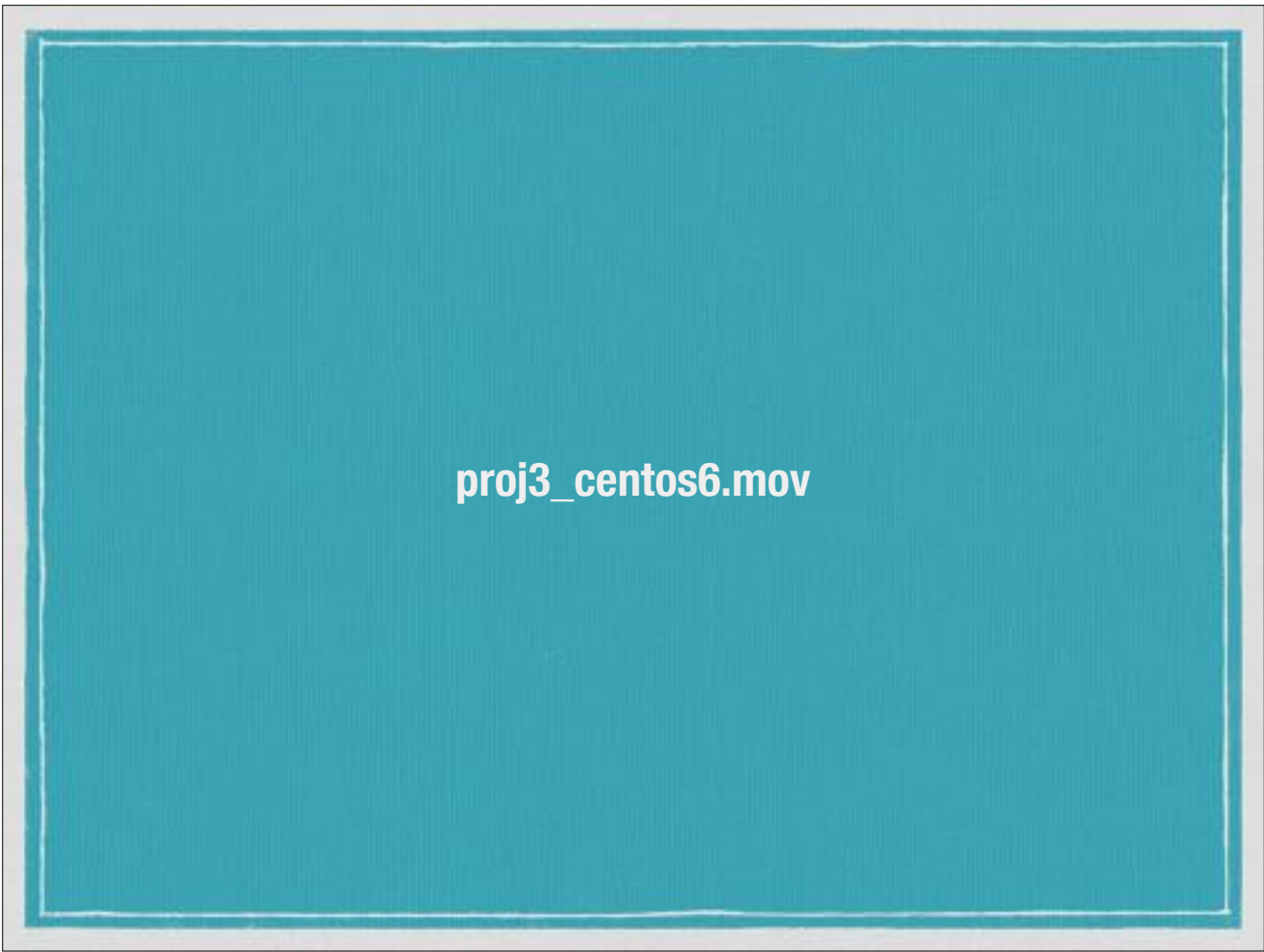
Ubuntu 18.04...

((play in fast forward))



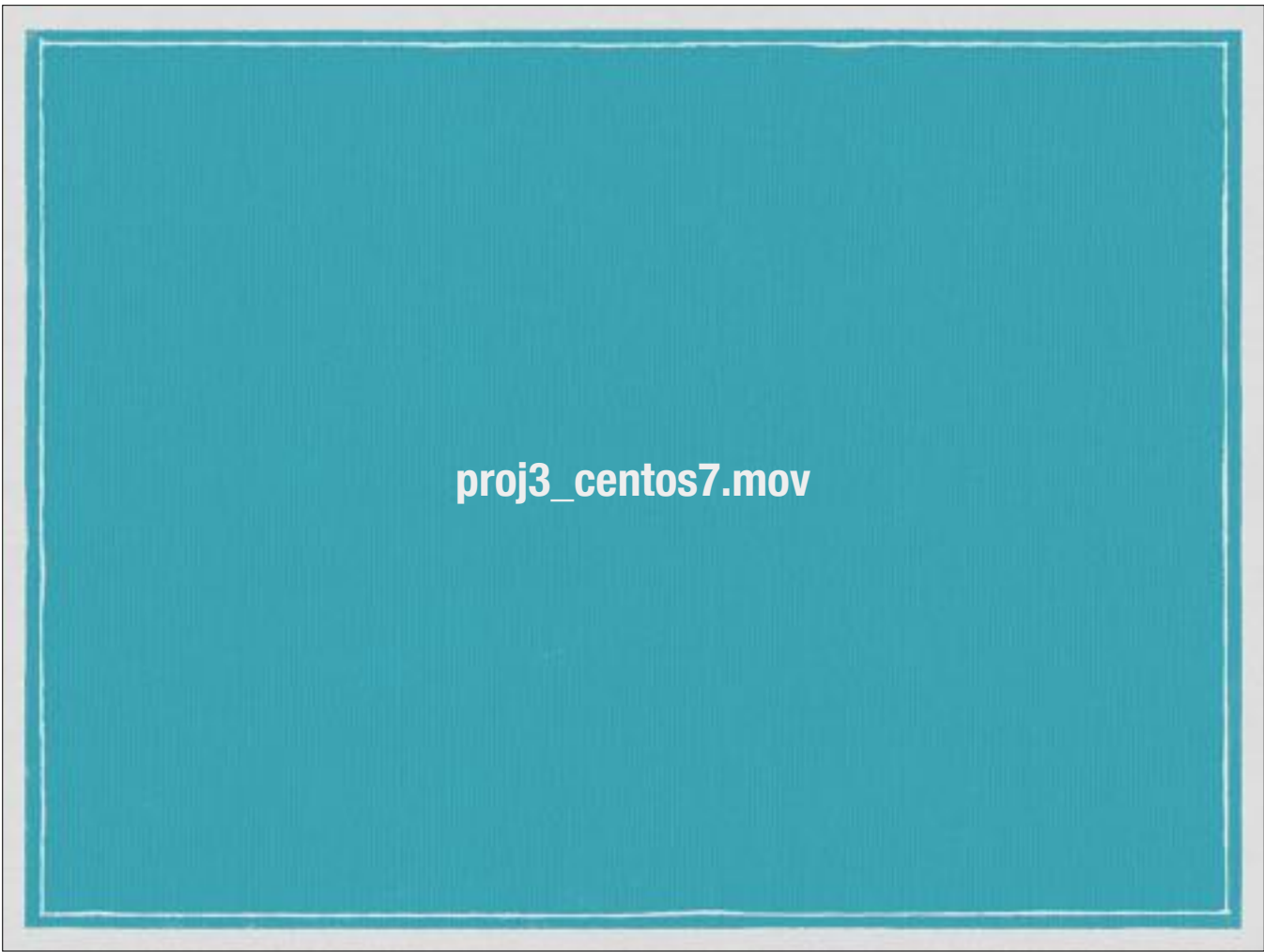
OpenSUSE Leap 15.0...

((play in fast forward))



Centos 6...

((play in fast forward))



Centos 7...

((play in fast forward))



<< Gotchas >>

There's quite a few gotchas when using conan.

Some of the most frustrating I think are mismatches in configuration between conan and your build tool - especially things that cause a mismatch in the c++ runtime libs or compiler settings.

At the root of the issue is that conan runs before CMake (or your project generator). This opens the door for mismatches in the C++ standard being declared/used between conan and cmake, between debug and release etc. E.g. declaring a build debug in conan doesn't make it so, at least it didn't in early version of conan, you still needed to pass `-DCMAKE_BUILD_TYPE=Debug` to cmake to get the desired result.

Gotchas

- ❑ **Making changes to conanfile.py but not publishing to artifactory**
- ❑ **Time-lag between pushing to git and CI pipeline uploading new package to artifactory**
- ❑ **Mismatch between what is declared in conan and what is used in CMake (or other build system)**
- ❑ **~/.conan hosted on network share, logging into different build machines with different compiler**

This first gotcha is the main reason I recommend setting up a CI build pipeline.

However, once you setup a CI pipeline that automatically publishes packages to your conan repository, you need to be aware there will be a time lag between pushing to git and having the changes available in conan.

The 3rd gotcha is because conan is not tightly coupled with our build tools. It runs before our build tools creating config files ... config files we have to take advantage of to be useful. If we have everything declared nicely in conan, but not in CMake it is wasted.

The 4th gotcha won't effect everyone. It is a huge time waster if your ~/.conan directory is mounted on a network share and you need to log into and build on multiple different machines with different OSes or distros. The solution is to always use the **—profile** parameter with all conan commands. If you forget to do this, conan will attempt to build and link using settings and binaries from your default profile which may not work on the machine you're logged into. The typical symptom is undefined symbols when linking.

Gotchas (cont)

- ❑ When your build is broken, there is another layer where mistakes could be hiding
- ❑ When using older version of GCC on Linux, setting your `stdcpp` in your recipe isn't enough, you need to change your `libcxx=libstdc++11`

The first bullet point here, when conan/cmake is working its like magic, but when it breaks, you have an additional layer non-trivial layer of gears that have to be debugged.

Holy cow, I wasted a lot of time setting up this demo on this second bullet point. I wasted a full day on this at least. I knew there was something I'd done at Exegy to get things working on linux... but I couldn't remember and couldn't go back and reference it. Setting the c++ standard isn't enough to get things compiling. Conan isn't smart enough in some situations to use the correct c++ runtime - on older versions of GCC on Linux, you'll need to change `libcxx` to **`libstdc++11`** to get everything to compile.

<< **Getting Help** >>

Overall conan is easy to learn and use, but there are a few sharp edges. Now we'll cover a few places you can go to get help while learning.

A large orange square with a thin black border, containing the URL [**https://docs.conan.io**](https://docs.conan.io) in white text.

<https://docs.conan.io>

Conan has online documentation at docs.conan.io. This is the first place to check if you get stuck using conan.

While it has improved since the beta versions of conan, it can still be difficult to find answers for specific nuanced problems. In the event you cannot find help on the documentation website, you can open a help request issue on GitHub or jump on the conan channel on the C++ Slack.

<https://cpplang.slack.com>

#conan

There is a #conan channel under cpplang.slack.com, many of the conan core developers hang out here and I've found them to be extremely helpful in the past.

This is probably the quickest way for you to get around a road block.



<https://github.com/conan-io/conan/issues>

If you can't find it in the docs, and nobody on the #conan channel knows, then you probably need to open a **help** question on the conan issues page.

The core developers and community are very good about helping out new users here. Just be sure to search the issues history before opening a new issue.

<< **In Closing** >>

Before we wrap up, I'd like to quickly call out some similar projects.

Related Work

- | | |
|--|--|
| <input type="checkbox"/> vcpkg
https://github.com/Microsoft/vcpkg | <input type="checkbox"/> CMake
https://cmake.org/ |
| <input type="checkbox"/> NuGet
https://docs.microsoft.com/en-us/nuget/what-is-nuget | <input type="checkbox"/> Nix
https://nixos.org/ |
| <input type="checkbox"/> Build2
https://build2.org/ | <input type="checkbox"/> Homebrew
https://brew.sh |
| <input type="checkbox"/> biicode
https://github.com/biicode | <input type="checkbox"/> Maven |
| | <input type="checkbox"/> Ant |

This list is not exhaustive. I tried to list what people mostly use or have used for C and C++ in the not so distant past.

CMake and Build2 are mostly for building C/C++ code, but both have packaging systems. I'm not familiar with build2, so I won't comment on it, but the external package functionality in CMake can be made to work, but is more primitive than what's available in conan.

Nix and Homebrew are related to a degree but in my mind are really for distributing packages of completed software, not for use during construction. We're mentioning them for completeness.

I threw in a couple of unconventional options, Maven and Ant, which are not designed for C/C++ but can be made to work and which were used frequently in the past given a lack of anything else.

To me vcpkg looks like a really interesting option and I'd definitely be interested in learning more about it in the near future.



<https://github.com/conan-io/cmake-conan>

Also related work...

The conan team has released a CMake wrapper for conan so you don't have to manually run "conan install".

I haven't tried this yet, but it may prove useful if you're using conan and cmake, especially if you have a large developer group who understands make but doesn't know anything about conan.



<< EOT >>

That's it for the presentation.

**Questions or
Comments ?**

Are there any questions or comments?

Thank You!