

NSO IN DOCKER

- Docker has transformed the software development world
- Let's explore how we can transform the software development experience around NSO

Speaker notes

Note to readers of exported presentations: I have filled in the speaker notes as complete sentences so they can be read as a narrative. They were originally just some keywords to help me present. Full sentences makes it easier to understand as a reader but is not entirely true to how I originally presented the topic at Cisco NSO developers days in New York December 2019.

Docker has transformed the software development world. I want you to join me in exploring how we can transform the software development experience around NSO.

WHY TRUST ME?



- Kristian Larsson
- Deutsche Telekom / TeraStream
 - Network Automation System
 - Architect / Design Lead
- Using Cisco NSO for >6 years
- NSO in Docker for >4 years
- IP & Opto Network Architect @ Tele2 SWIPNET [AS1257]
- Cisco hat

Speaker notes

Who am I and why should you trust me?

I'm Kristian Larsson. For the last five odd years, I've been the system architect or design lead for the network automation system within a project called TeraStream at Deutsche Telekom. We've done all sorts of cool stuff around NSO that we're not going to talk about here today, but more importantly we have been using NSO with Docker for 4 maybe even five years I think.

I've personally been using Cisco NSO for a bit more than that, think it's about six years.

Prior to Deutsche Telekom, I was an IP and Opto network architect at Tele2. My background is in networking. Over the years I've sorted of learnt a bit of programming so I can get by.

Rather recently I've also taken on a roll at Cisco. I maintain my position at DT but have started working with the NSO team. The very first project I picked up in my new role at Cisco, was...

NSO + DOCKER = ❤️

- 4 years of experience
- public and open source
- maintained by Cisco

Speaker notes

...yupp, you guessed it, NSO in Docker.

I built the CI testing environment at DT about 4-5 years ago and we used Docker quite extensively. Since then it's expanded even further. What I've done here recently is a complete revamp where I've tried to incorporate the best aspects of that previous work.

Let me say this. I am pretty damn happy about how it turned out and I'm excited to be here and present it to you. I hope you will join me in my enthusiasm for NSO in Docker and what it means for development and testing around NSO.

What I show you here today is open source and publically available. It will be maintained by Cisco.

I should however add that it is in somewhat of a development stage right now. Consider this a beta or soft launch.

Also, if you have questions I encourage you to ask them as we go through the presentation. It's usually easier to talk about with the context fresh in mind. I might ask you to defer if it's being answered in the presentation though.

WHAT IS DOCKER?

- a little LXC and COW union FS
- but mostly **ergonomics**
 - it's about improving life of developers

Speaker notes

Okay, so back to basics sort of. What is Docker? Some people will talk about LXC, or Linux Containers, there's some clever use of copy-on-write union file systems.

To me, I think Docker is all about ergonomics for developers. You can talk about the underlying technology but what it really provides is improved ergonomics. Developers want to focus on getting their stuff done, which is mostly about writing code. Everything else, like starting test environments for reproducing problems or testing with all the moving parts that comprise a complete system, is a time sink - it takes away from the time that can be spent on writing code. Docker helps in this area. There are alternative ways of doing the same thing but it is simple and that's exactly what we want here. We want to focus our brain on writing code, not on system administration like activities around it.

Docker helps streamline these kinds of activities and that is why it is loved.

NSO COMPONENTS

NSO system consists of many components

- **NSO**
- **NEDs**
- **Java VM**
- **Java libs**
- **Python VM**
- **Python libraries**
- **service packages**

Speaker notes

A production NSO system consists of multiple components, like the NSO application itself. We have NEDs and our service packages. There's a Java and Python VM to run our service code. There might be libraries related to these two run times.

All of this comes together to form the system.

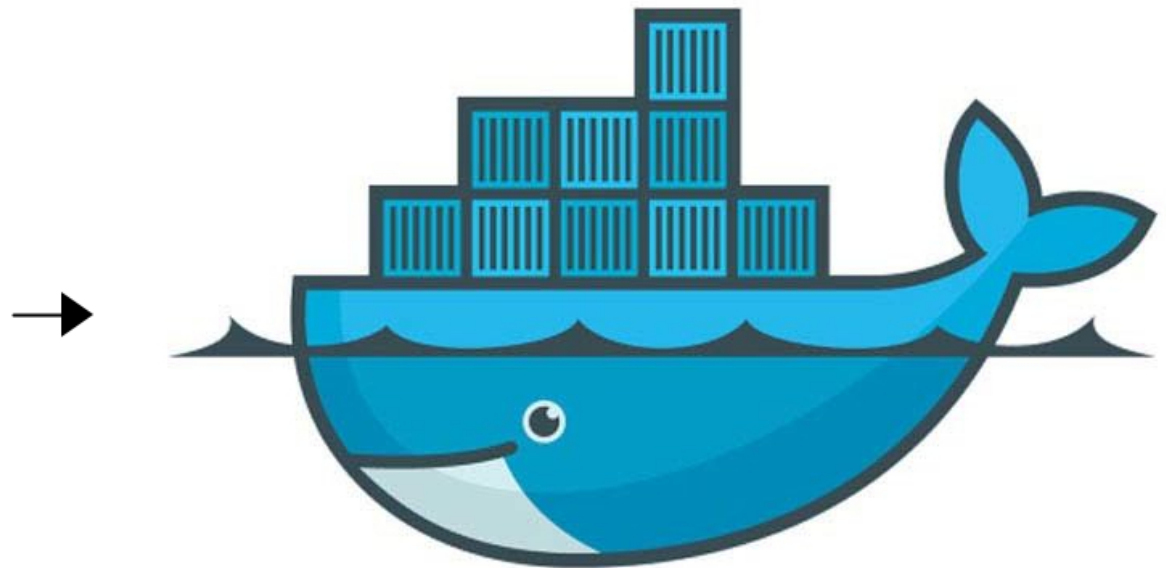
All of these components can be versioned independently, however, they must remain compatible with each other for the system as a whole to work.

Changes to a NED package can invalidate configuration templates in a service package and a service package will depend on some version of the Python VM or some set of Python libraries.

NSO IN DOCKER

NSO system consists of many components

- NSO
- NEDs
- Java VM
- Java libs
- Python VM
- Python libraries
- service packages

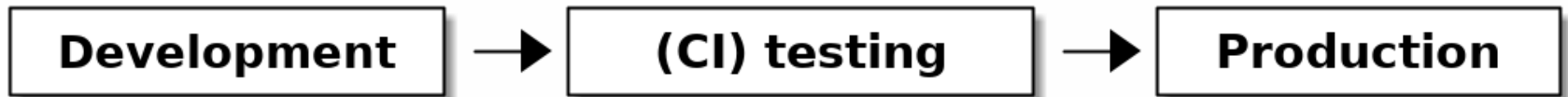


Speaker notes

With NSO in Docker, we take all these components and build a Docker image out of it.

It becomes a single atomic unit that contains all the components we need. This means we can be reasonably certain that when we run CI tests with this set of component versions and we see it working, we can also be reasonably certain it will work in production. This is an important aspect of NSO in Docker.

ENVIRONMENTS



Speaker notes

Our NSO system will usually run in a number of different environments.

Just looking at the typical software development flow, we start out by writing some code on our laptop, we commit that to git and push to some code hosting platform which then runs CI tests on our code. Finally if all goes well we can deploy the result of our code to a production environment. That's three different environments.

The basic premise of CI testing is that the environment in which you test is a reasonable replica of your production environment and similarly for your development environment. We saw how many components that goes into making a production NSO system. We don't track all of those in our git repository, I mean, your Java VM isn't actually checked into your git repository - we just track our NSO service code. The rest, like the Java VM version and libraries are pulled in from elsewhere.

We have to manage these dependencies in a consistent manner so we can leave guarantees that the tests we run in CI testing are actually applicable in production. Docker helps us do this.

I really can't stress this enough. It's about getting rid of the "It works on my laptop" kind of response to issues. To make things worse, you could have some staging or acceptance environment bringing up the total number of environments even further. And remember, that the development environment isn't actually a single environment. Every developer typically has his own environment, like his laptop, so you can multiply that by the number of developers you have on your team.

GUARANTEE CONSISTENCY

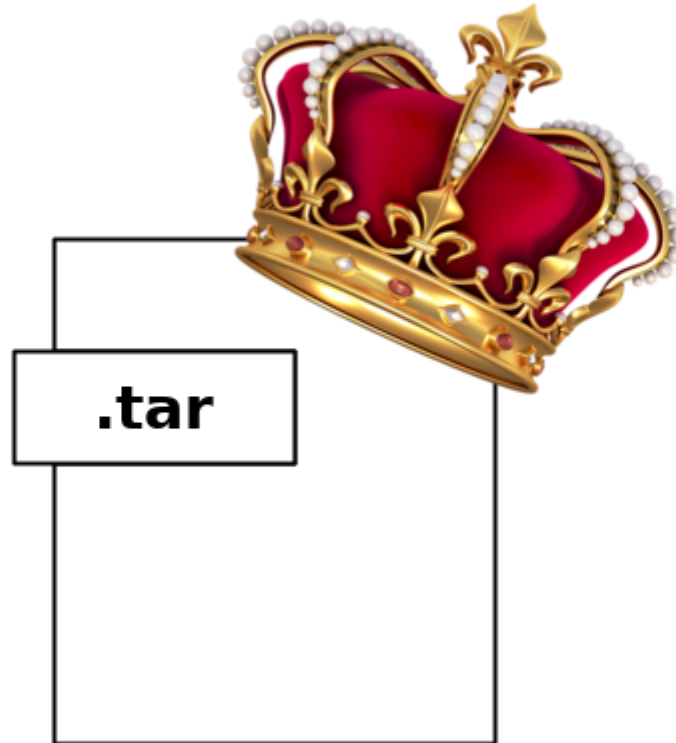
- build docker image **once**
 - test in **CI** -> deploy to **production**
 - also reuse for **dev** and **lab**

Speaker notes

The basic Docker flow is that you build an image once and then deploy that in many different environments. So we build an image in CI, we run all the tests with that image and then deploy the same image to production.

We also reuse the image for development and lab purposes.

GLORIFIED TAR FILE?



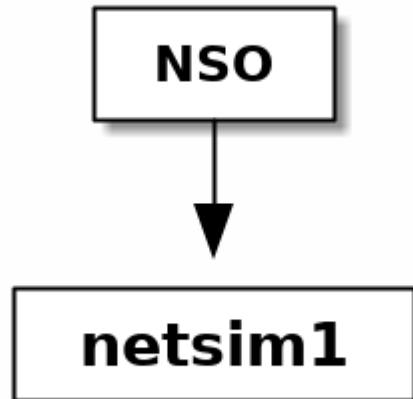
Speaker notes

Okay, so Docker helps us ship stuff around as an atomic unit.

Is that just a glorified tar file? From one perspective yes. Looking solely at the packaging aspect of Docker, it is essentially a glorified tar file.

However, there is more. Docker helps us run things in a convenient manner.

TEST IN A LOCAL WORLD

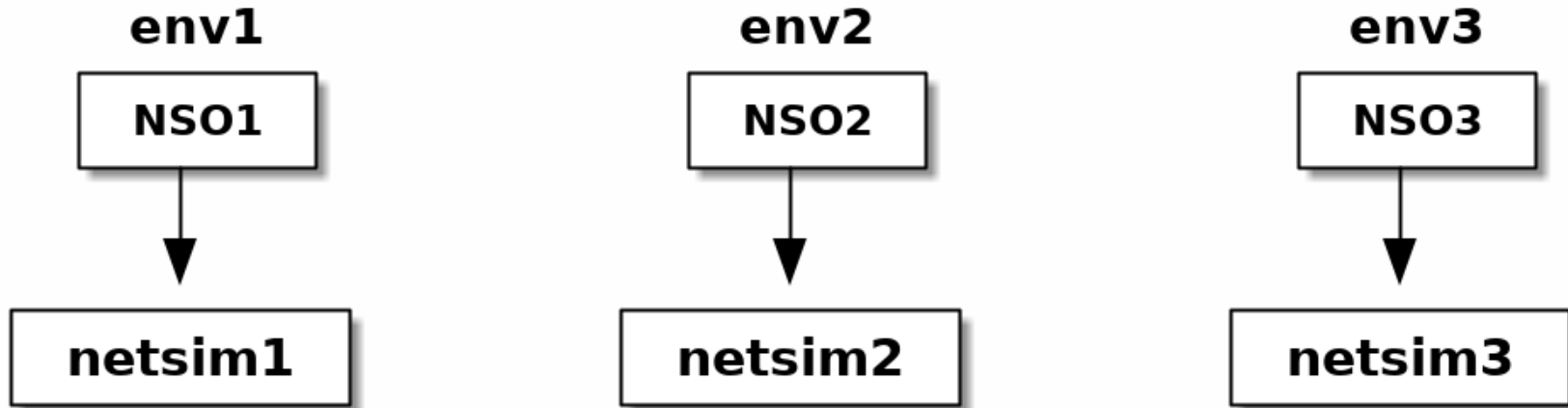


```
<devices xmlns="http://tail-f.com/ns/ncs">
  <device>
    <name>my-test-device</name>
    <address>netsim1</address>
    ...
  </device>
</devices>
```

Speaker notes

For testing, it is very convenient to be able to refer to things in a static manner. Like let's say we have an NSO instance and a netsim in our test environment. If we can always refer to the netsim instance in a static fashion, by its name and a static port, then writing the test scaffolding becomes trivial.

TEST IN A GLOBAL WORLD

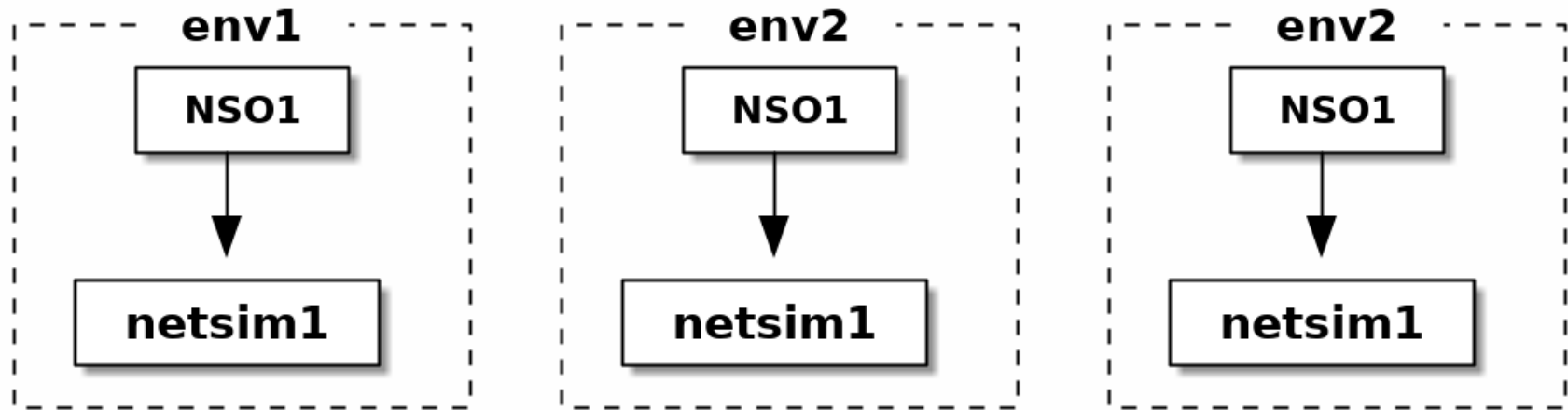


Speaker notes

On the contrary, consider an environment, like CI, where we want to be able to run multiple tests at the same. This requires that ports or names don't collide and implies in a classic environment that dynamic allocation of these resources is used. This means that the test harness becomes considerably more complex. By no means an insurmountable challenge for a programmer but it unnecessarily shifts our attention away from writing valuable code. Figuring out the address of a netsim is just the top of the ice berg, once you get to building virtual router network topologies this will become much more complex.

I would argue that having a local world view where things are static, is a key attribute of an easy to use test bed.

TEST LOCAL IN GLOBAL WORLD



Speaker notes

What we want is to be able to start many instances but pretend that the world in which they live is much smaller. They don't need to see everyone else in the big wide world, they need only see their neighbor. This is in essence namespaces, which is something Docker is really quite good at. Thus, this is another key aspect to why NSO in Docker makes sense. We'll get back to the details of how this works in practice.

WHAT DOES IT LOOK LIKE?

- production image contains the **customers** service packages
 - implies **you**, the **customer** must build the image
- what can Cisco do?

Speaker notes

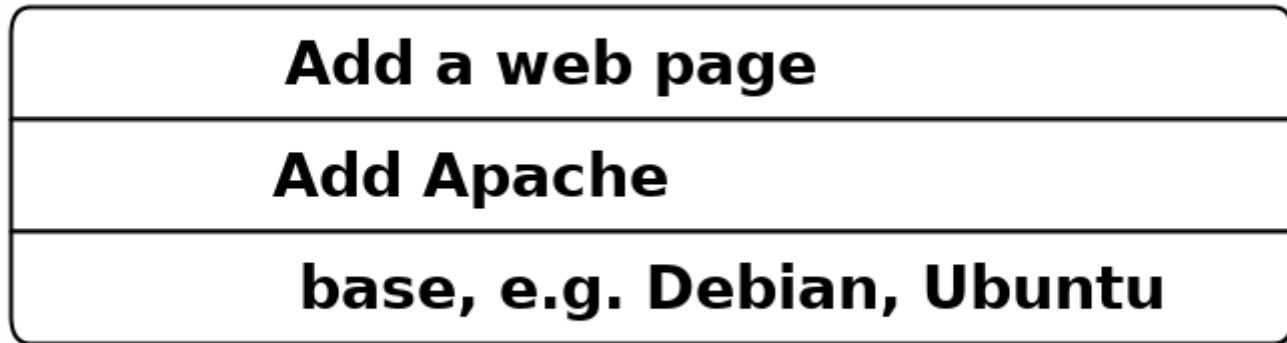
As we looked at before, a production Docker image of NSO contains services packages of the NSO user or customer. This means Cisco can't build a complete production NSO image for you. You, the users in this room, have to add in **your** service packages into **your** image for it to become suitable for **your** environment.

But what can be done to simplify this? What can Cisco do?

To answer that, we first have to understand a little more about Docker...

UNDERSTAND DOCKER LAYERS

- A Docker image is built out of layers
- naturally lends itself to extending a *base* image by adding things on top



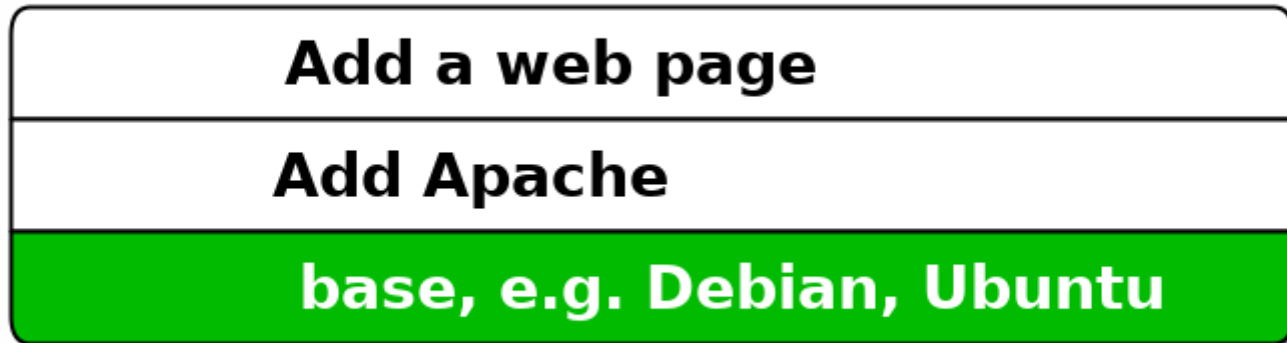
Speaker notes

Specifically about Docker image layers.

A Docker image is built out of layers. You start from a base layer,

UNDERSTAND DOCKER LAYERS

- A Docker image is built out of layers
- naturally lends itself to extending a *base* image by adding things on top

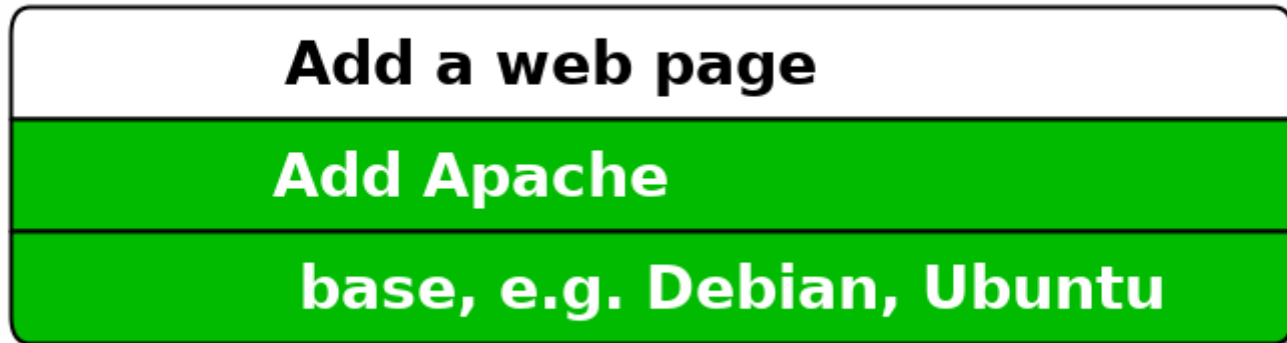


Speaker notes

...typically an image someone else has built, like a Debian or Ubuntu image. Then you add your stuff on top. In this case, we install the Apache web server

UNDERSTAND DOCKER LAYERS

- A Docker image is built out of layers
- naturally lends itself to extending a *base* image by adding things on top



Speaker notes

... and add in our static web pages.

UNDERSTAND DOCKER LAYERS

- A Docker image is built out of layers
- naturally lends itself to extending a *base* image by adding things on top

Add a web page

Add Apache

base, e.g. Debian, Ubuntu

Speaker notes

Out of this we get a Docker image that we can run as a container which will then be able to serve our web pages.

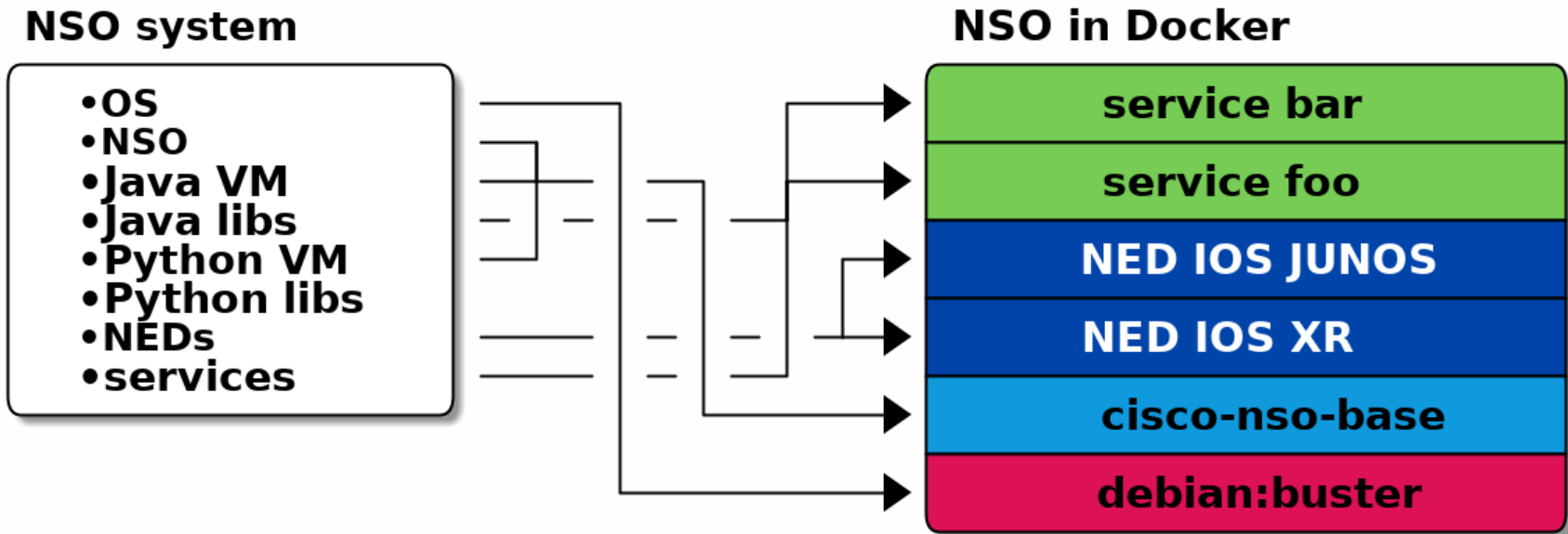
DOCKER IMAGE VS CONTAINER

- you build a docker **image**
- start one or more **container** from that image
- a container is an *instance* of an image
- image does not contain instance specific data, like IP addresses
 - IP etc is allocated at run time

Speaker notes

I also want to clarify the difference between an image and a container. We talked here about building an image. When you do `docker run`, you start a container based on that image. It becomes an instance of an image, and any changes it does to its local data is just for the container instance - not the image. The image doesn't change until you build it again. It actually doesn't change then either unless you give it new input. Docker will see if you are building an image based on the same input and just use the cached result from a previous build.

NSO COMPONENTS



Speaker notes

Building a Docker image out of the components of an NSO system will similarly result in a number of layers. We have the operating system, which is the very base layer in an Docker image, then on top of that we get cisco-nso-base, that includes NSO itself as well as the Java and Python VMs. Then on top of that you would typically add the NEDs that you need and finally your own service packages.

Cisco will publish the NEDs but we can't put them in this Docker image for you. There are like a hundred different NEDs, maybe more. We don't know which ones you want so you have to pick yourself and add them in. Nor can we add in your service packages.

Looking at this stack, we see the common parts are the base NSO layer and whatever lies beneath it. Let's zoom in on that.

NSO BASE DOCKER IMAGE

cisco-nso-base

debian:buster

- we are building a docker image for use as the base image
 - multiple images for different NSO versions
- also building a development image
 - contains build tools
 - make, ant, ncsc etc

Speaker notes

We're building a docker image for use as the base in your images. It's called cisco-nso-base.

There's actually multiple of these images - one for every version of NSO, so you pick the version of NSO you want by picking different docker images as base for yours.

In addition, there is a development image that contains, in addition to what is in this cisco-nso-base, various build tools like make, ant etc that you need to build a NSO package.

NOW WHAT?

- Let's use it!
- NED or package author
- Prod NSO developer
 - Composing multiple packages -> one system
 - Testing of the whole

Speaker notes

So how can we use it?

We are going to look at two examples for different use cases or user stories.

In the first one, we assume the role of a NED developer and we will look at how to build and test a NED using NSO in Docker.

In the second example, we will look at how to build a production NSO image. Which is naturally what most users of NSO will actually be interested in. We start with the NED case however, simply because it's fewer moving parts.

NSO-IN-DOCKER FOR NEDS

- compile & package NED into Docker image
 - ned image - contains just the NED (not runnable)
 - netsim image - acts as a netsim based on the NED
 - NSO image - NSO system with this specific NED loaded

Speaker notes

NSO in Docker for NED. We will start with compiling and packaging up the NED into a few Docker images. I say a few because we will build multiple images. The first one, contains just the NED itself and nothing else. We're using docker as a tar file. You can't run that image because there is nothing executable inside.

Then we build a netsim image which you can run. Starting a container from it will simply act as a netsim instance based on the NED. This is useful for testing, both in this repo and in other repos.

Finally we will build an NSO image, which is just a basic NSO system and our specific NED. Again, this is useful for testing as we will see.

NED DOCKERFILE

```
ARG BASE_IMAGE # for example cisco-nso-base:5.2.1
ARG BUILD_IMAGE # for example cisco-nso-dev:5.2.1 (has to match base image version)
```

```
FROM $BUILD_IMAGE AS build
ARG PACKAGE_NAME # this is the name of the NED package
COPY ${PACKAGE_NAME} /var/opt/ncs/packages/${PACKAGE_NAME}
RUN make -C /var/opt/ncs/packages/${PACKAGE_NAME}/netsim
RUN make -C /var/opt/ncs/packages/${PACKAGE_NAME}/src
```

```
FROM $BUILD_IMAGE as netsim
ARG PACKAGE_NAME
ENV PNAME=${PACKAGE_NAME}
COPY --from=build /var/opt/ncs/packages /var/opt/ncs/packages
COPY /run-netsim.sh /run-netsim.sh
CMD /run-netsim.sh ${PNAME} ${PNAME}
```

```
FROM $BASE_IMAGE AS nso
COPY --from=build /var/opt/ncs/packages /var/opt/ncs/packages
```

```
FROM scratch AS ned
COPY --from=build /var/opt/ncs/packages /var/opt/ncs/packages
```

build:

```
docker build --target netsim -t $(IMAGE_PATH)$(PROJECT_NAME)-netsim:${DOCKER_TAG} --build-arg BUILD_IMAGE=$(BUILD_IMAGE) --build-arg
docker build --target nso -t $(IMAGE_PATH)$(PROJECT_NAME)-nso:${DOCKER_TAG} --build-arg BUILD_IMAGE=$(BUILD_IMAGE) --build-arg BASE_
docker build --target ned -t $(IMAGE_PATH)$(PROJECT_NAME):${DOCKER_TAG} --build-arg BUILD_IMAGE=$(BUILD_IMAGE) --build-arg BASE_IMAG
```

Speaker notes

How many of you have seen a Dockerfile before? How many have seen a multi-stage Dockerfile?

Okay, this is a multi-stage Dockerfile. It's quite simple.

At the bottom we can see the makefile which runs the docker build commands which in turn reads the dockerfile to produce a docker image. These are some long lines and there are lots of variables in this making it a little hard to comprehend but we can see how we run docker build three times, each for one respective target. netsim, nso and ned. The -t argument is what specific the tag or sort of the name of the docker image and again, we see how -netsim, -nso and just nothing, is reflected in the name.

Looking at the Dockerfile we actually have four targets. The first is build, which is used to compile the NED. It starts with reading in an argument for the package name. We then use COPY to copy the files from outside of the docker image into the docker image, then we run make for the NED and its netsim directory. That's the end of that image.

Going to the next, we have the netsim image. We read in the PACKAGE_NAME again, because these arguments are scoped per target. This time we copy in the package, but using --from=build we copy the package not from outside of our docker environment but from the previously built stage, the one called build. We then copy in a run-netsim script and set it as the default CMD. This means that when this docker image is run as a container, it will per default execute this run-netsim script, which will run the NED as a netsim.

Then there's the nso image which is even simpler. Using our cisco-nso-base image as a base, we just add the NED we built. Again, it's from the build stage, so it's already compiled and ready to run.

Finally, the ned stage which also just copies in the package. Unlike the previous nso stage though, this starts from an image called scratch. It's a special image that simply contains nothing at all. Thus, you cannot run this image. We just use it as a vessel for shipping our compiled package elsewhere.

NED TESTING

test:

```
docker network create $(CNT_PREFIX)-$(PROJECT_NAME)
docker run -td --network $(CNT_PREFIX)-$(PROJECT_NAME) --name $(CNT_PREFIX)-$(PROJECT_NAME)-nso $(IMAGE_PATH)$$(PROJECT_NAME)-nso:$(D
docker run -td --network $(CNT_PREFIX)-$(PROJECT_NAME) --name $(CNT_PREFIX)-$(PROJECT_NAME)-netsim --network-alias dev1 $(IMAGE_PATH
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'nsc --wait-started 600'
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'echo "show packages" | ncs_cli -u admin -g ncsadmin' | grep "oper-status
@echo "Add device to NSO"
docker cp add-device.xml $(CNT_PREFIX)-$(PROJECT_NAME)-nso:/add-device.xml
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'echo -e "configure\nload merge /add-device.xml\ncommit\nexit" | ncs_cli -
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'echo "show devices brief" | ncs_cli -u admin -g ncsadmin'
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'echo "request devices device dev1 ssh fetch-host-keys" | ncs_cli -u admin
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'echo "request devices device dev1 sync-from" | ncs_cli -u admin -g ncsadm
@echo "Configure hostname on device through NSO"
docker cp test/device-config-hostname.xml $(CNT_PREFIX)-$(PROJECT_NAME)-nso:/device-config-hostname.xml
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'echo -e "configure\nload merge /device-config-hostname.xml\ncommit\nexit"
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'echo -e "show configuration devices device dev1 config" | ncs_cli -u admin
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'echo -e "request devices device dev1 sync-from" | ncs_cli -u admin -g ncs
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'echo -e "show configuration devices device dev1 config" | ncs_cli -u admin
```

Speaker notes

This is extracted from the Makefile of the example NED we have that follows the NSO in Docker concept. Again a little long but we'll go through it together.

You remember how I talked about having a local world in which everything could look the same, yet exist in a global world which could contain multiple isolated test environments? In order to have such an isolated environment, we create a new docker network. The name consists of two variables, first `CNT_PREFIX` which is our container prefix and then the project name. Together they form a globally unique identifier. Then we start two containers from the `nso` and `netnsim` image respectively. Notice on the `netnsim` image how we provide a `--network-alias dev1`. This means that inside of this docker network, we can always access the `netnsim` container through the hostname `dev1` and this is what makes it possible to have a local world with fixed hostname references.

The following commands are to wait for NSO to start up. We check that packages have been loaded correctly. Then proceed to load a small configuration file into NSO. This file will add the `netnsim` as a device in NSO. After which we can fetch ssh keys and do a `sync-from`. We then load a second configuration file, this one will set the hostname on the `netnsim` device. Finally we read the configuration of the device and `grep` for the string `foobarhostname`. This is the magic string that we set the hostname too, so by seeing that it is set, we have confirmed that we were able to successfully write the hostname configuration to the `netnsim` device. We follow up with a `sync-from` and check the same thing again, just to make sure `sync-from` works too.

CI RUN

```
←[32;1m$ echo Building for NSO version ${NSO_VERSION}←[0;m
Building for NSO version 5.2.1
←[32;1m$ make build←[0;m
docker build --target netsim -t registry.gitlab.com/nso-developer/docker-example-ned-ietf/docker-example-ned-ietf-netsim:356737718 --b
Sending build context to Docker daemon 142.3kB

Step 1/15 : ARG BASE_IMAGE
Step 2/15 : ARG BUILD_IMAGE
Step 3/15 : FROM $BUILD_IMAGE AS build
---> c261847ee7cf
Step 4/15 : ARG PACKAGE_NAME
---> Using cache
---> d15aa875b33d
Step 5/15 : COPY ${PACKAGE_NAME} /var/opt/ncs/packages/${PACKAGE_NAME}
---> 3a1c5236c152
Step 6/15 : SHELL ["/bin/sh", "-lc"]
---> Running in 1eb74e397255
Removing intermediate container 1eb74e397255
---> 92eb5b7a004b
Step 7/15 : RUN make -C /var/opt/ncs/packages/${PACKAGE_NAME}/netsim
---> Running in a7b665a9efed
make: Entering directory '/var/opt/ncs/packages/ietf-ned/netsim'
/opt/ncs/current/netsim/confd/bin/confdc --yangpath ../src/yang \
    `ls ../src/yang/ietf-system-ann.yang > /dev/null 2>&1 && \
    echo "-a ../src/yang/ietf-system-ann.yang" ` \
    -c -o ietf-system.fxs ../src/yang/ietf-system.yang
make: Leaving directory '/var/opt/ncs/packages/ietf-ned/netsim'
Removing intermediate container a7b665a9efed
---> 73fe14a2c8f7
Step 8/15 : RUN make -C /var/opt/ncs/packages/${PACKAGE_NAME}/src
---> Running in d107e3a5f18f
make: Entering directory '/var/opt/ncs/packages/ietf-ned/src'
mkdir -p ncsc-out
mkdir -p ../load-dir
mkdir -p ../shared-jar
mkdir -p ../private-jar
mkdir -p java/src/com/example/ietfncd/namespaces
mkdir -p ../python/ietf_ned
rm -rf ../package-meta-data.xml
if [ -x /opt/ncs/current/support/ned-make-package-meta-data ]; then \
    /opt/ncs/current/support/ned-make-package-meta-data package-meta-data.xml.in; \
else
    cp package-meta-data.xml.in ../package-meta-data.xml; \
fi
--ncs-ned-id ietf-ned-nc-1.0:ietf-ned-nc-1.0
chmod -w ../package-meta-data.xml
/opt/ncs/current/bin/ncsc --ncs-compile-bundle yang \
--ncs-device-dir ncsc-out
```

```
        --ncs-device-type netconf \
        --ncs-ned-id ietf-ned-nc-1.0:ietf-ned-nc-1.0 \
        && \
        cp ncsc-out/modules/fixs/*.fixs ../load-dir;
for f in `echo ../load-dir/*.fixs`; do \
    n=`basename $f | sed 's/\.fixs//'; \
    /opt/ncs/current/bin/ncsc --java-disable-prefix --exclude-enums --fail-on-warnings --java-package com.example.ietfnc.namespaces -- \
    /opt/ncs/current/bin/ncsc --emit-python ../python/ietf_ned $f || exit 1; \
done; \
```

done

```
←[91mWarning: the following symbols have been suppressed due to a
conflict with an enum or bit with the same mapped name but a different value:
'server'.
```

Use tailf:code-name on the conflicting enums, bits, or nodes to avoid the conflict.

```
←[0mtouch ncsc-out/.done
```

```
cd java && ant -q all
```

```
[javac] warning: [options] bootstrap class path not set in conjunction with -source 6
[javac] warning: [options] source value 6 is obsolete and will be removed in a future release
[javac] warning: [options] target value 1.6 is obsolete and will be removed in a future release
[javac] warning: [options] To suppress warnings about obsolete options, use -Xlint:-options.
[javac] 4 warnings
```

BUILD SUCCESSFUL

Total time: 0 seconds

```
make -C ../netsim all
```

```
make[1]: Entering directory '/var/opt/ncs/packages/ietf-ned/netsim'
```

```
make[1]: Nothing to be done for 'all'.
```

```
make[1]: Leaving directory '/var/opt/ncs/packages/ietf-ned/netsim'
```

```
make: Leaving directory '/var/opt/ncs/packages/ietf-ned/src'
```

Removing intermediate container d107e3a5f18f

```
---> 47504a7f5881
```

```
Step 9/15 : FROM $BUILD_IMAGE as netsim
```

```
---> c261847ee7cf
```

```
Step 10/15 : ARG PACKAGE_NAME
```

```
---> Using cache
```

```
---> d15aa875b33d
```

```
Step 11/15 : ENV PNAME=${PACKAGE_NAME}
```

```
---> Running in f3edf53efcdc
```

Removing intermediate container f3edf53efcdc

```
---> c4486ee3648c
```

```
Step 12/15 : COPY --from=build /var/opt/ncs/packages /var/opt/ncs/packages
```

```
---> 56c03f81cdd8
```

```
Step 13/15 : SHELL ["/bin/sh", "-lc"]
```

```
---> Running in e76d50923150
```

Removing intermediate container e76d50923150

```
---> 0854dc6fd442
```

```
Step 14/15 : COPY /run-netsim.sh /run-netsim.sh
```

```
---> 07d3be3e661d
```

```
Step 15/15 : CMD /run-netsim.sh ${PNAME} ${PNAME}
```

```
---> Running in 0eb655bc4237
```

Removing intermediate container 0eb655bc4237

```
---> 467caead72aa
```

```
Successfully built 467caead72aa
Successfully tagged registry.gitlab.com/nso-developer/docker-example-ned-ietf/docker-example-ned-ietf-netsim:356737718
docker build --target nso -t registry.gitlab.com/nso-developer/docker-example-ned-ietf/docker-example-ned-ietf-nso:356737718 --build-arg B
Sending build context to Docker daemon 142.3kB

Step 1/17 : ARG BASE_IMAGE
Step 2/17 : ARG BUILD_IMAGE
Step 3/17 : FROM $BUILD_IMAGE AS build
---> c261847ee7cf
Step 4/17 : ARG PACKAGE_NAME
---> Using cache
---> d15aa875b33d
Step 5/17 : COPY ${PACKAGE_NAME} /var/opt/ncs/packages/${PACKAGE_NAME}
---> Using cache
---> 3a1c5236c152
Step 6/17 : SHELL ["/bin/sh", "-lc"]
---> Using cache
---> 92eb5b7a004b
Step 7/17 : RUN make -C /var/opt/ncs/packages/${PACKAGE_NAME}/netsim
---> Using cache
---> 73fe14a2c8f7
Step 8/17 : RUN make -C /var/opt/ncs/packages/${PACKAGE_NAME}/src
---> Using cache
---> 47504a7f5881
Step 9/17 : FROM $BUILD_IMAGE as netsim
---> c261847ee7cf
Step 10/17 : ARG PACKAGE_NAME
---> Using cache
---> d15aa875b33d
Step 11/17 : ENV PNAME=${PACKAGE_NAME}
---> Using cache
---> c4486ee3648c
Step 12/17 : COPY --from=build /var/opt/ncs/packages /var/opt/ncs/packages
---> Using cache
---> 56c03f81cdd8
Step 13/17 : SHELL ["/bin/sh", "-lc"]
---> Using cache
---> 0854dc6fd442
Step 14/17 : COPY /run-netsim.sh /run-netsim.sh
---> Using cache
---> 07d3be3e661d
Step 15/17 : CMD /run-netsim.sh ${PNAME} ${PNAME}
---> Using cache
---> 467caead72aa
Step 16/17 : FROM $BASE_IMAGE AS nso
---> 202452d9e9d4
Step 17/17 : COPY --from=build /var/opt/ncs/packages /var/opt/ncs/packages
---> 487f46cab49b
Successfully built 487f46cab49b
Successfully tagged registry.gitlab.com/nso-developer/docker-example-ned-ietf/docker-example-ned-ietf-nso:356737718
docker build --target ned -t registry.gitlab.com/nso-developer/docker-example-ned-ietf/docker-example-ned-ietf:356737718 --build-arg B
Sending build context to Docker daemon 142.3kB

Step 1/19 : ARG BASE_IMAGE
Step 2/19 : ARG BUILD_IMAGE
```

```
Step 3/19 : FROM $BUILD_IMAGE AS build
---> c261847ee7cf
Step 4/19 : ARG PACKAGE_NAME
---> Using cache
---> d15aa875b33d
Step 5/19 : COPY ${PACKAGE_NAME} /var/opt/ncs/packages/${PACKAGE_NAME}
---> Using cache
---> 3a1c5236c152
Step 6/19 : SHELL ["/bin/sh", "-lc"]
---> Using cache
---> 92eb5b7a004b
Step 7/19 : RUN make -C /var/opt/ncs/packages/${PACKAGE_NAME}/netsim
---> Using cache
---> 73fe14a2c8f7
Step 8/19 : RUN make -C /var/opt/ncs/packages/${PACKAGE_NAME}/src
---> Using cache
---> 47504a7f5881
Step 9/19 : FROM $BUILD_IMAGE as netsim
---> c261847ee7cf
Step 10/19 : ARG PACKAGE_NAME
---> Using cache
---> d15aa875b33d
Step 11/19 : ENV PNAME=${PACKAGE_NAME}
---> Using cache
---> c4486ee3648c
Step 12/19 : COPY --from=build /var/opt/ncs/packages /var/opt/ncs/packages
---> Using cache
---> 56c03f81cdd8
Step 13/19 : SHELL ["/bin/sh", "-lc"]
---> Using cache
---> 0854dc6fd442
Step 14/19 : COPY /run-netsim.sh /run-netsim.sh
---> Using cache
---> 07d3be3e661d
Step 15/19 : CMD /run-netsim.sh ${PNAME} ${PNAME}
---> Using cache
---> 467caead72aa
Step 16/19 : FROM $BASE_IMAGE AS nso
---> 202452d9e9d4
Step 17/19 : COPY --from=build /var/opt/ncs/packages /var/opt/ncs/packages
---> Using cache
---> 487f46cab49b
Step 18/19 : FROM scratch AS ned
--->
Step 19/19 : COPY --from=build /var/opt/ncs/packages /var/opt/ncs/packages
---> a98f5f4130c0
Successfully built a98f5f4130c0
Successfully tagged registry.gitlab.com/nso-developer/docker-example-ned-ietf/docker-example-ned-ietf:356737718
←[32;1m$ make test-[];m
make stop
make[1]: Entering directory '/builds/nso-developer/docker-example-ned-ietf'
docker stop 356737718-netsim 356737718-nso
Error response from daemon: No such container: 356737718-netsim
Error response from daemon: No such container: 356737718-nso
make[1]: Leaving directory '/builds/nso-developer/docker-example-ned-ietf'
```

```
make clean
make[1]: [Makefile:81: stop] Error 1 (ignored)
make[1]: Entering directory '/builds/nso-developer/docker-example-ned-ietf'
docker rm -f 356737718-netsim 356737718-nso
Error: No such container: 356737718-netsim
Error: No such container: 356737718-nso
make[1]: [Makefile:84: clean] Error 1 (ignored)
docker network rm ci-356737718
Error: No such network: ci-356737718
make[1]: [Makefile:85: clean] Error 1 (ignored)
make[1]: Leaving directory '/builds/nso-developer/docker-example-ned-ietf'
docker network create ci-356737718
482c560e1abb70f4007af9e9bfa046de35930809e08e6a52e0857bc8dfe98acc
docker run -td --network ci-356737718 --name 356737718-nso registry.gitlab.com/nso-developer/docker-example-ned-ietf/docker-example-ned-ietf:1413bdc8bd164a76835b5080d65d9af2aea4b709e2b2a2f4329a7f98df7e345
docker run -td --network ci-356737718 --name 356737718-netsim --network-alias dev1 registry.gitlab.com/nso-developer/docker-example-ned-ietf:7e077a1cdc5f63bbb5ca8be9859f70490e6eb99421a293804adf4099fe90282f
docker exec -t 356737718-nso bash -lc 'ncs --wait-started 600'
docker exec -t 356737718-nso bash -lc 'echo "show packages" | ncs_cli -u admin -g ncsadmin'
packages package ietf-ned-nc-1.0
package-version 1.0
description "Generated netconf package"
ncs-min-version [ 4.5 ]
directory /nso/run/state/packages-in-use/1/ietf-ned
component ietf-ned
ned netconf ned-id ietf-ned-nc-1.0
ned device vendor Acme
oper-status up
docker exec -t 356737718-nso bash -lc 'echo "show packages" | ncs_cli -u admin -g ncsadmin' | grep "oper-status up"
oper-status up
Add device to NSO
Copy config straight, for NSO 5
cp test/add-device.xml add-device.xml
On NSO 4 we remove the ned-id from the config since that's a NSO 5 thing
echo 5.2.1 | grep "^4" && sed -e '/<ned-id/d' test/add-device.xml > add-device.xml
make: [Makefile:67: test] Error 1 (ignored)
docker cp add-device.xml 356737718-nso:/add-device.xml
docker exec -t 356737718-nso bash -lc 'echo -e "configure\nload merge /add-device.xml\ncommit\nexit" | ncs_cli -u admin -g ncsadmin'
Commit complete.
docker exec -t 356737718-nso bash -lc 'echo "show devices brief" | ncs_cli -u admin -g ncsadmin'
NAME ADDRESS DESCRIPTION NED ID
-----
dev1 dev1 - ietf-ned-nc-1.0
docker exec -t 356737718-nso bash -lc 'echo "request devices device dev1 ssh fetch-host-keys" | ncs_cli -u admin -g ncsadmin'
result updated
fingerprint {
  algorithm ssh-rsa
  value b6:d8:9b:3d:e0:bc:a6:71:cf:c8:02:e1:5b:fd:12:f6
}
docker exec -t 356737718-nso bash -lc 'echo "request devices device dev1 sync-from" | ncs_cli -u admin -g ncsadmin'
result true
Configure hostname on device through NSO
docker cp test/device-config-hostname.xml 356737718-nso:/device-config-hostname.xml
docker exec -t 356737718-nso bash -lc 'echo -e "configure\nload merge /device-config-hostname.xml\ncommit\nexit" | ncs_cli -u admin -g ncsadmin'
Commit complete.
```

```

docker exec -t 356737718-nso bash -lc 'echo -e "show configuration devices device dev1 config sys:system hostname" | ncs_cli -u admin
hostname foobar;
docker exec -t 356737718-nso bash -lc 'echo -e "request devices device dev1 sync-from" | ncs_cli -u admin -g ncsadmin'
result true
docker exec -t 356737718-nso bash -lc 'echo -e "show configuration devices device dev1 config sys:system hostname" | ncs_cli -u admin
hostname foobar;
+ [32;1m$ if [ "${DOCKER_PUSH}" != "false" ]; then make push; fi-[0;m
docker push registry.gitlab.com/nso-developer/docker-example-ned-ietf/docker-example-ned-ietf:356737718
The push refers to repository [registry.gitlab.com/nso-developer/docker-example-ned-ietf/docker-example-ned-ietf]
a893c3a9447f: Preparing
a893c3a9447f: Pushed
356737718: digest: sha256:a96e3eec17c6b89209599b75ba5f6c14ddc3c15937a1d26f46385e574e653574 size: 526
+ [32;1m$ if [ "${CI_COMMIT_REF_NAME}" = "master" ]; then make tag-release; fi-[0;m
Setting docker tag for release
docker tag registry.gitlab.com/nso-developer/docker-example-ned-ietf/docker-example-ned-ietf:356737718 registry.gitlab.com/nso-developer/docker-example-ned-ietf-netsim:356737718
docker tag registry.gitlab.com/nso-developer/docker-example-ned-ietf/docker-example-ned-ietf-netsim:356737718 registry.gitlab.com/nso-developer/docker-example-ned-ietf-netsim:5.2.1
+ [32;1m$ if [ "${CI_COMMIT_REF_NAME}" = "master" ] && [ "${DOCKER_PUSH}" != "false" ]; then make push-release; fi-[0;m
docker push registry.gitlab.com/nso-developer/docker-example-ned-ietf/docker-example-ned-ietf:5.2.1
The push refers to repository [registry.gitlab.com/nso-developer/docker-example-ned-ietf/docker-example-ned-ietf]
a893c3a9447f: Preparing
a893c3a9447f: Layer already exists
5.2.1: digest: sha256:a96e3eec17c6b89209599b75ba5f6c14ddc3c15937a1d26f46385e574e653574 size: 526
section_end:1574286492:build_script
+ [0Ksection_start:1574286492:after_script
+ [0K-[32;1mRunning after script...-[0;m
+ [32;1m$ make stop clean-[0;m
docker stop 356737718-netsim 356737718-nso
356737718-netsim
356737718-nso
docker rm -f 356737718-netsim 356737718-nso
356737718-netsim
356737718-nso
docker network rm ci-356737718
ci-356737718
section_end:1574286504:after_script
+ [0Ksection_start:1574286504:archive_cache
+ [0Ksection_end:1574286506:archive_cache
+ [0Ksection_start:1574286506:upload_artifacts_on_success
+ [0Ksection_end:1574286508:upload_artifacts_on_success
+ [0K-[32;1mJob succeeded
+ [0;m

```

WHAT DID WE TEST?

- compilation as NED and netsim works
- loads correctly in NSO
- loads correctly in netsim
 - fails on XR models with global constraints

Speaker notes

So this proves that a NED compiles, both in NED form as well as netsim form.

Note how compile time verification is different from load time. It's when we load a NED or start a netsim that we get to see it come together. For example, on certain versions of XR, the YANG models have constraints that require some top level nodes to be set. This means you can't start netsim with an empty configuration. You have to load an XML file to seed CDB.

We also showed how we can do simple testing of the interaction between NSO and netsim. We could naturally add a lot more tests to actually test our NED. This is just to show how the pieces fit together to form a test harness.

NED TEMPLATE

- the build and test scaffolding for NEDs is generic
- using NiD NED concept is simple
 - copy NED template repo
 - ncs-make-package
 - modify NED specific config file
 - ...
 - profit!

Speaker notes

One of the coolest aspects of this NED build is that it is almost completely generic. As we've seen in both the Dockerfile and Makefiles, there are a lot of variables. It makes it a little hard to read at first but it's worth it. The three main files that you need to build and test a NED like this are 100% generic. That's the CI configuration itself, the Dockerfile and the Makefile. The one place where we have NED specific configuration is the test file that sets the hostname. Naturally, you have to have a configuration path and value that is specific to the NED you are testing. Everything else is generic.

This enables you to build and test a NED in the same manner through a few simple steps. You grab the gitlab-ned-repo-template by cloning its public repository. You copy those files into a new directory, you add the NED package itself, perhaps on you built through ncs-make-package, modify the NED specific config file for the test, commit and off you go. It's that simple!

PROD NSO

```
ARG NSO_IMAGE_PATH
ARG NSO_VERSION
ARG PKG_PATH

# pull in remote images of NEDs and make them locally available through alias
FROM ${PKG_PATH}docker-example-ned-ietf/docker-example-ned-ietf:${NSO_VERSION} AS ietf-ned
FROM ${PKG_PATH}ned-iosxr-621/ned-iosxr-621:${NSO_VERSION} AS ned-iosxr-621

# compile local packages
FROM ${NSO_IMAGE_PATH}cisco-nso-dev:${NSO_VERSION} AS build
COPY hostname /var/opt/ncs/packages/hostname
RUN make -C /var/opt/ncs/packages/hostname/src

# build the final production NSO image
FROM ${NSO_IMAGE_PATH}cisco-nso-base:${NSO_VERSION} AS nso

# copy in already built NEDs from external repos
COPY --from=ietf-ned /var/opt/ncs/packages/ietf-ned /var/opt/ncs/packages/ietf-ned
COPY --from=ned-iosxr-621 /var/opt/ncs/packages/ned-iosxr-621 /var/opt/ncs/packages/ned-iosxr-621

# copy in local package
COPY --from=build /var/opt/ncs/packages/hostname /var/opt/ncs/packages/hostname
```

Speaker notes

Next up, we will build an example production NSO image. I kept the comments in this one because the file is short enough as it is.

Similar to the previous image we have some arguments. We then have these two dummy FROM statements which we use to alias the fully qualified path of the NEDs into a shorter name that we can then use in the COPY operation further down.

Next stage is a build stage where we compile a service package called hostname. It's quite a silly service, it just sets the hostname of a device, but it'll serve as an example package. We copy in the package and compile it.

We then move on to the final stage which is about building the NSO image. Now we copy in the ietf-ned and a XR ned. Notice how we use the `-from` argument to COPY to reference the previous build stage. Finally we copy in the compiled hostname service package.

PROD NSO TESTING

testenv-start:

```
-docker network create $(CNT_PREFIX)-$(PROJECT_NAME)
docker run -td --network $(CNT_PREFIX)-$(PROJECT_NAME) --name $(CNT_PREFIX)-$(PROJECT_NAME)-nso --label testenv-$(CNT_PREFIX)-$(PROJECT_NAME)
docker run -td --network $(CNT_PREFIX)-$(PROJECT_NAME) --name $(CNT_PREFIX)-$(PROJECT_NAME)-netsim-ietf1 --network-alias ietf1 --label testenv-$(CNT_PREFIX)-$(PROJECT_NAME)-ietf1
docker run -td --network $(CNT_PREFIX)-$(PROJECT_NAME) --name $(CNT_PREFIX)-$(PROJECT_NAME)-netsim-xr1 --network-alias xr1 --label testenv-$(CNT_PREFIX)-$(PROJECT_NAME)-xr1
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'ncs --wait-started 600'
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'echo "show packages" | ncs_cli -u admin -g ncsadmin'
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'echo "show packages" | ncs_cli -u admin -g ncsadmin' | grep "oper-status"
```

testenv-configure:

```
docker cp test/add-device.xml $(CNT_PREFIX)-$(PROJECT_NAME)-nso:/add-device.xml
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'echo -e "configure\nload merge /add-device.xml\ncommit\nexit" | ncs_cli -u admin -g ncsadmin'
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'echo "show devices brief" | ncs_cli -u admin -g ncsadmin'
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'echo "request devices device * ssh fetch-host-keys" | ncs_cli -u admin -g ncsadmin'
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'echo "request devices device * sync-from" | ncs_cli -u admin -g ncsadmin'
```

testenv-test:

```
docker cp test/configure-hostname-service.xml $(CNT_PREFIX)-$(PROJECT_NAME)-nso:/configure-hostname-service.xml
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'echo -e "configure\nload merge /configure-hostname-service.xml\ncommit\nexit" | ncs_cli -u admin -g ncsadmin'
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'echo "show configuration hostname" | ncs_cli -u admin -g ncsadmin'
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'echo -e "request hostname * re-deploy" | ncs_cli -u admin -g ncsadmin'
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'echo -e "show configuration devices device ietf1 config" | ncs_cli -u admin -g ncsadmin'
docker exec -t $(CNT_PREFIX)-$(PROJECT_NAME)-nso bash -lc 'echo -e "show configuration devices device xr1 config" | ncs_cli -u admin -g ncsadmin'
```

Speaker notes

This is quite similar to what the NED tests did. It's structured a little differently. Instead of a single make target we now have three targets; `testenv-start`, `testenv-configure` and `testenv-test`. There's a `stop` and `clean` target too but they're not included here for brevity's sake. Having different target makes it possible to start the test environment once and rerun the test part multiple times. Good for doing development with in an iterative fashion.

Just like with the NED, we start by creating a docker network with a unique name. We start three containers. The NSO container, one IETF NED netsim and one IOS XR netsim. We proceed to check that the packages loaded correctly, just like before.

We proceed to load some configuration into NSO that contains the devices. That XML file includes two devices, one for the IETF NED netsim and one for the XR netsim. `ssh fetch host keys` and `sync-from` on both devices.

Now on to the test make target where we again load configuration. This time the configuration is for the hostname service. We verify that the service did what it was supposed to do, which is to set the hostname of these devices, by reading out the configuration of each device and `grep` for the magic string `foobarhostname`.

That's it. A simple demonstration of how you can compose your production NSO image out of both locally sourced and compiled packages as well as externally built NEDs plus testing it in a fairly simple manner.

CI RUN

```
Building for NSO version 5.2.1
+[[32;1m$ make build-[[0;m
docker build --target nso -t registry.gitlab.com/nso-developer/docker-example-prod-nso-compose/docker-example-prod-nso-compose:356752313
Sending build context to Docker daemon 217.1kB

Step 1/12 : ARG NSO_IMAGE_PATH
Step 2/12 : ARG NSO_VERSION
Step 3/12 : ARG PKG_PATH
Step 4/12 : FROM ${PKG_PATH}docker-example-ned-ietf/docker-example-ned-ietf:${NSO_VERSION} AS ietf-ned
---> a98f5f4130c0
Step 5/12 : FROM ${PKG_PATH}ned-iosxr-621/ned-iosxr-621:${NSO_VERSION} AS ned-iosxr-621
---> a4dbdc1d8f3a
Step 6/12 : FROM ${NSO_IMAGE_PATH}cisco-nso-dev:${NSO_VERSION} AS build
---> c261847ee7cf
Step 7/12 : COPY hostname /var/opt/ncs/packages/hostname
---> Using cache
---> ccaa61c6ab6c
Step 8/12 : RUN make -C /var/opt/ncs/packages/hostname/src
---> Using cache
---> fd645e81be14
Step 9/12 : FROM ${NSO_IMAGE_PATH}cisco-nso-base:${NSO_VERSION} AS nso
---> 202452d9e9d4
Step 10/12 : COPY --from=ietf-ned /var/opt/ncs/packages/ietf-ned /var/opt/ncs/packages/ietf-ned
---> Using cache
---> 09c374ffae07
Step 11/12 : COPY --from=ned-iosxr-621 /var/opt/ncs/packages/ned-iosxr-621 /var/opt/ncs/packages/ned-iosxr-621
---> Using cache
---> b27ce8425fea
Step 12/12 : COPY --from=build /var/opt/ncs/packages/hostname /var/opt/ncs/packages/hostname
---> Using cache
---> b4138a879a64
Successfully built b4138a879a64
Successfully tagged registry.gitlab.com/nso-developer/docker-example-prod-nso-compose/docker-example-prod-nso-compose:356752313
+[[32;1m$ make test-[[0;m
make testenv-start
make[1]: Entering directory '/builds/nso-developer/docker-example-prod-nso-compose'
docker network create 356752313-docker-example-prod-nso-compose
3fcddb28c5331ad0a396c6615f99dabead115b056ea6fc97ed394391c7266c92
docker run -td --network 356752313-docker-example-prod-nso-compose --name 356752313-docker-example-prod-nso-compose --label testenv
e5af2a2f3177d7578012a7d872c479c8bcbad8cb624b0ccf3ae0ff4c6525bd96
docker run -td --network 356752313-docker-example-prod-nso-compose --name 356752313-docker-example-prod-nso-compose-netsim-ietf1 --net
16eb31461e204336acbb7976eb33ea3ac5a77ce690f3f9fb7ebf368551d2b4d2
docker run -td --network 356752313-docker-example-prod-nso-compose --name 356752313-docker-example-prod-nso-compose-netsim-xr1 --netwo
c44c35b57e341814ec1665dd2fe527acelc3964b5f505b8f2e703d690f967e2
docker exec -t 356752313-docker-example-prod-nso-compose-nso bash -lc 'ncs --wait-started 600'
docker exec -t 356752313-docker-example-prod-nso-compose-nso bash -lc 'echo "show packages" | ncs_cli -u admin -g ncsadmin'
packages package hostname
package-version 1.0
```

```

description "Generated Python package"
ncs-min-version [ 5.2 ]
python-package vm-name hostname
directory /nso/run/state/packages-in-use/1/hostname
templates [ hostname-template ]
component main
  application python-class-name hostname.main.Main
  application start-phase phase2
oper-status up
packages package ietf-ned-nc-1.0
package-version 1.0
description "Generated netconf package"
ncs-min-version [ 4.5 ]
directory /nso/run/state/packages-in-use/1/ietf-ned
component ietf-ned
  ned netconf ned-id ietf-ned-nc-1.0
  ned device vendor Acme
oper-status up
packages package ned-iosxr-621-nc-1.0
package-version 1.0
description "Generated netconf package"
ncs-min-version [ 4.5 ]
directory /nso/run/state/packages-in-use/1/ned-iosxr-621
component ned-iosxr-621
  ned netconf ned-id ned-iosxr-621-nc-1.0
  ned device vendor Acme
oper-status up
docker exec -t 356752313-docker-example-prod-nso-compose-nso bash -lc 'echo "show packages" | ncs_cli -u admin -g ncsadmin' | grep "op
oper-status up
oper-status up
oper-status up
make[1]: Leaving directory '/builds/nso-developer/docker-example-prod-nso-compose'
make testenv-configure
make[1]: Entering directory '/builds/nso-developer/docker-example-prod-nso-compose'
docker cp test/add-device.xml 356752313-docker-example-prod-nso-compose-nso:/add-device.xml
docker exec -t 356752313-docker-example-prod-nso-compose-nso bash -lc 'echo -e "configure\nload merge /add-device.xml\ncommit\nexit" |
Commit complete.
docker exec -t 356752313-docker-example-prod-nso-compose-nso bash -lc 'echo "show devices brief" | ncs_cli -u admin -g ncsadmin'
NAME ADDRESS DESCRIPTION NED ID
-----
ietf1 ietf1 - ietf-ned-nc-1.0
xr1 xr1 - ned-iosxr-621-nc-1.0
docker exec -t 356752313-docker-example-prod-nso-compose-nso bash -lc 'echo "request devices device * ssh fetch-host-keys" | ncs_cli -
devices device ietf1 ssh fetch-host-keys
  result updated
  fingerprint {
    algorithm ssh-rsa
    value b6:d8:9b:3d:e0:bc:a6:71:cf:c8:02:e1:5b:fd:12:f6
  }
devices device xr1 ssh fetch-host-keys
  result updated
  fingerprint {
    algorithm ssh-rsa
    value b6:d8:9b:3d:e0:bc:a6:71:cf:c8:02:e1:5b:fd:12:f6
  }

```



```
docker exec -t 356752313-docker-example-prod-nso bash -lc 'echo "request devices device * sync-from" | ncs_cli -u admin -g ncsadm'
devices device ietfl sync-from
    result true
devices device xr1 sync-from
    result true
make[1]: Leaving directory '/builds/nso-developer/docker-example-prod-nso-compose'
make testenv-test
make[1]: Entering directory '/builds/nso-developer/docker-example-prod-nso-compose'
docker cp test/configure-hostname-service.xml 356752313-docker-example-prod-nso-compose-nso:/configure-hostname-service.xml
docker exec -t 356752313-docker-example-prod-nso-compose-nso bash -lc 'echo -e "configure\nload merge /configure-hostname-service.xml\'
Commit complete.
docker exec -t 356752313-docker-example-prod-nso-compose-nso bash -lc 'echo "show configuration hostname" | ncs_cli -u admin -g ncsadm'
hostname ietfl {
    hostname foobarhostname;
}
hostname xr1 {
    hostname foobarhostname;
}
docker exec -t 356752313-docker-example-prod-nso-compose-nso bash -lc 'echo -e "request hostname * re-deploy" | ncs_cli -u admin -g ncsadm'
docker exec -t 356752313-docker-example-prod-nso-compose-nso bash -lc 'echo -e "show configuration devices device ietfl config" | ncs_cli'
    hostname foobarhostname;
docker exec -t 356752313-docker-example-prod-nso-compose-nso bash -lc 'echo -e "show configuration devices device xr1 config" | ncs_cli'
    host-name foobarhostname;
make[1]: Leaving directory '/builds/nso-developer/docker-example-prod-nso-compose'
←[32;1m$ if [ "${DOCKER_PUSH}" != "false" ]; then make push; fi←[0;m
←[32;1m$ if [ "${CI_COMMIT_REF_NAME}" = "master" ]; then make tag-release; fi←[0;m
docker tag registry.gitlab.com/nso-developer/docker-example-prod-nso-compose/docker-example-prod-nso-compose:356752313 registry.gitlab.com/nso-developer/docker-example-prod-nso-compose:356752313
←[32;1m$ if [ "${CI_COMMIT_REF_NAME}" = "master" ] && [ "${DOCKER_PUSH}" != "false" ]; then make push-release; fi←[0;m
section_end:1574287616:build_script
←[0Ksection_start:1574287616:after_script
←[0K←[32;1mRunning after script...←[0;m
←[32;1m$ make stop clean←[0;m
make testenv-stop
make[1]: Entering directory '/builds/nso-developer/docker-example-prod-nso-compose'
docker ps -q --filter label=testenv-356752313-docker-example-prod-nso-compose | xargs --no-run-if-empty docker stop
c44c35b57e34
16eb31461e20
e5af2a2f3177
make[1]: Leaving directory '/builds/nso-developer/docker-example-prod-nso-compose'
make testenv-clean
make[1]: Entering directory '/builds/nso-developer/docker-example-prod-nso-compose'
docker ps -aq --filter label=testenv-356752313-docker-example-prod-nso-compose | xargs --no-run-if-empty docker rm -f
c44c35b57e34
16eb31461e20
e5af2a2f3177
docker network rm 356752313-docker-example-prod-nso-compose
356752313-docker-example-prod-nso-compose
docker volume rm 356752313-docker-example-prod-nso-compose-packages
Error: No such volume: 356752313-docker-example-prod-nso-compose-packages
make[1]: [Makefile:85: testenv-clean] Error 1 (ignored)
make[1]: Leaving directory '/builds/nso-developer/docker-example-prod-nso-compose'
make: *** No rule to make target 'clean'. Stop.
section_end:1574287628:after_script
←[0Ksection_start:1574287628:archive_cache
←[0Ksection_end:1574287630:archive_cache
```

```
←[0Ksection_start:1574287630:upload_artifacts_on_success
←[0Ksection_end:1574287631:upload_artifacts_on_success
←[0K←[32;1m]Job succeeded
←[0;m
```

CISCO DELIVERABLE

- Cisco can't actually deliver that `cisco-nso-base` image
 - due to legal requirements
- so we do the next best thing
- provide a complete *recipe* for building a base NSO docker image
 - just add the NSO install file!

DOCKER PERSISTENCE

- containers don't have persistent disk
- restarting a container means its starts clean from image

Speaker notes

One of the most frequent questions I get is how persistence is done. Containers aren't supposed to have state they say. Containers can't persist any data, they say.

This is sort of true. Containers per default don't have a way to persist data - once a running container is removed the data in it is gone too.

However, no application that provides any real value can really be stateless. You really do need to save data. In cloud native stuff it's just that the data is separated into a separate database and the database gets special treatment.

NSO has CDB built-in, which really is a database. Thus it needs special treatment.

NSO PERSISTENCE

- docker offers persistent disk through volumes
- design time choice of placement in image vs volume

image	volume
NSO core	CDB
Java & Python VM	notification streams
NEDs	rollbacks
service packages	logs

Speaker notes

... and here's a reason to use these images rather than bake your own. Maybe you will get all of this right. I'm pretty happy with the way I've setup the nso-docker images but in a earlier incarnation I had the paths for image local vs persist data setup differently. This was in the Deutsche Telekom environment. The CDB directory was a symbolic link from the container local directory to the persistent shared volume and while this worked well for NSO itself, it didn't work well for the ncs-backup script. It didn't follow symbolic links per default and I think it took us a year to realize that our backups didn't contain the CDB content we wanted, but just a single symbolic link. We fixed that, but the backup restore still wouldn't work because it couldn't work with that symbolic link.

NSO+DOCKER ECOSYSTEM

SUMMARY

- NSO in Docker ecosystem
 - base image
 - NED concept
 - prod NSO concept
- fits together
- common look and feel
- well tested (19 tests on nso-docker)
- parallel environments (CNT_PREFIX)
- test with multiple NSO version!
- netsim -> vrnetlab

Speaker notes

To sum up the NSO in Docker ecosystem and what it provides... we have the base images, the NED concept and what we call a production NSO concept for how to do composition. I showed you how the latter two are built.

We're not going to look in-depth at how the base NSO docker images themselves are built. I would however like to put emphasis on one aspect. These images are quite well tested. There are currently 19 tests which tests the behaviour of the images, which is 19 more than the majority of Docker images out there that I've seen.

Parallel environments are supported throughout. This means you can run CI tests in parallel or run multiple parallel environments in your local development environment.

The NSO version is parameterized everywhere! This means it is trivial to test your entire environment with different versions of NSO. I can't really emphasize enough how important this is. You should always be testing your code with the current version of NSO you are running as well as with your next target release such that you are ready to make the upgrade or if there are problems you can deal with them over time.

The netsim containers I propose here when building NEDs are useful for certain types of testing. You are inevitably going to advance and require a proper virtual router for which I can recommend vrnetlab. As both these netsims and vrnetlab use Docker containers it becomes an incredibly simple operation to switch. It fits well into the NSO in Docker ecosystem.

IT'S PUBLIC!

- cisco-nso-base & cisco-nso-dev
 - <https://gitlab.com/nso-developer/nso-docker/>
- NED example
 - <https://gitlab.com/nso-developer/docker-example-ned-ietf/>
- NED repo template
 - <https://gitlab.com/nso-developer/gitlab-ned-repo-template/>
- prod NSO example
 - <https://gitlab.com/nso-developer/docker-example-prod-nso-compose/>

