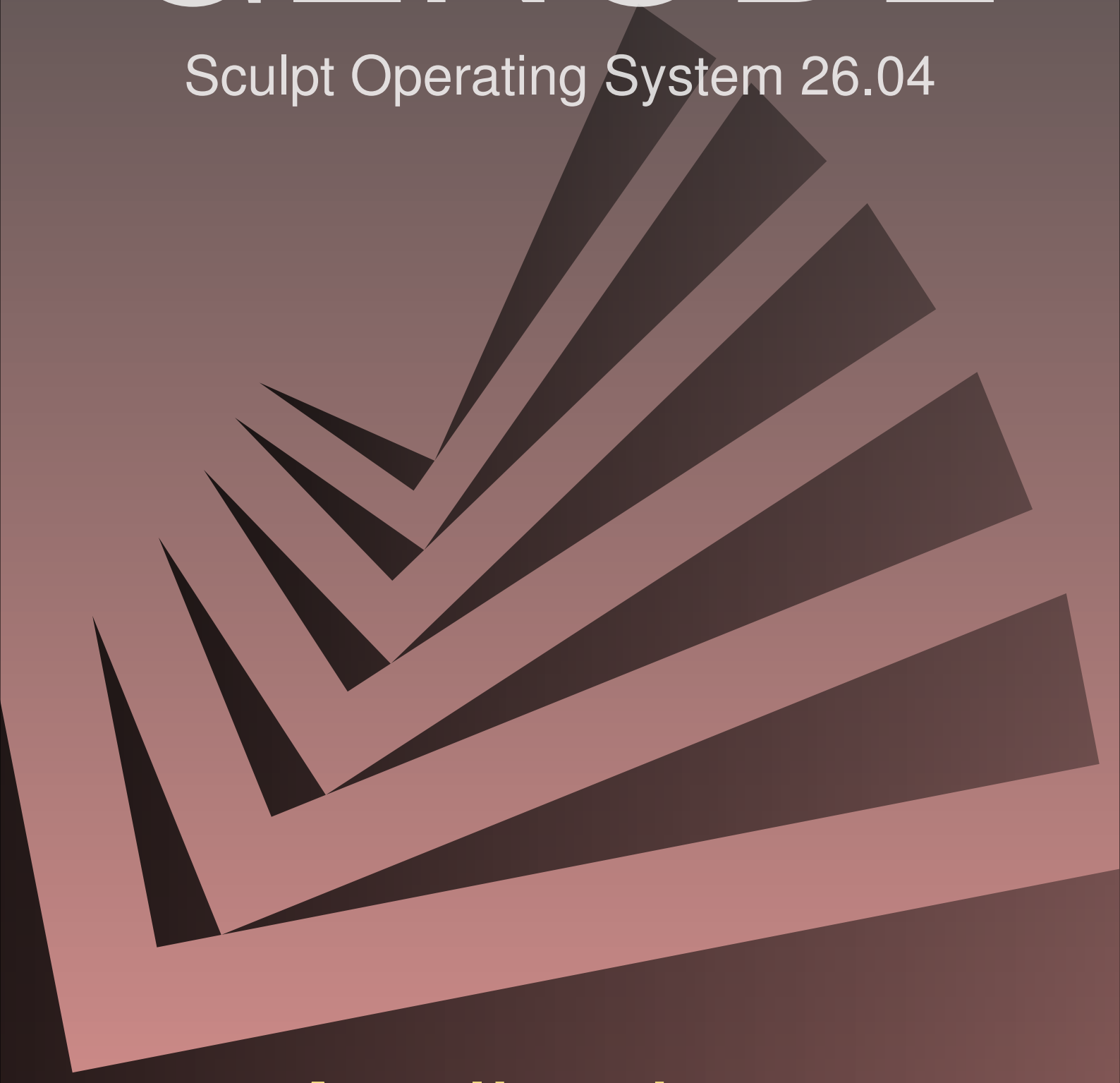
# GENODE

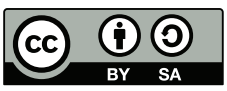
Sculpt Operating System 26.04



# Applications

Johannes Schlatow

## Contents



# 1 Introduction

This document complements the Genode Foundations book with application-level topics. It is primarily intended for application developers. Before studying the Genode Applications material, it is recommended to give the Genode Foundations book a read. The book can be downloaded at https://genode.org.

Another guide worth reading is the Sculpt OS guide[1]. Sculpt OS is one particular incarnation of the Genode OS Framework that puts the user in the position of full control. With Sculpt OS, you are leaving the well-known world of Linux. It is therefore worth familiarizing yourself with these new grounds before building applications for it.

This document features a practical guide for developing and porting applications to Genode. The material leverages the Goa software development kit (SDK), which streamlines the application development for the Genode OS Framework and Sculpt OS in particular.

### Goa SDK tool

https://codeberg.org/genodelabs/goa

Chapter 2 demonstrates the use of the Goa tool in form of a Getting Started guide. Chapter 3 recapitulates the principles of Genode's architecture as well as its most essential libraries, components, and tooling for application developers. Chapter 4 provides assistance when it comes to debugging Genode applications. Finally, Chapter 5 completes the document by a collection of tutorials and stories on application porting/development.

[1] https://genode.org/documentation/articles/sculpt-26-04

## 2 Getting started with Goa

*This section is based on Norman Feske's Goa article[1] at https://genodians.org.*

The development of applications for Genode used to require a lot of learning about Genode's way of organizing source code, the framework's custom build system, and the use of run scripts. The Goa tool aims at largely removing these burdens from application developers.

In contrast to the tools that come with Genode, which were designed for developing complete systems, Goa[2] is focused on the development of individual applications. In a nutshell, it streamlines the following tasks:

1. The porting of 3rd-party software to Genode, which typically involves
   - Downloading 3rd-party source code via Git, Subversion, or in the form of archives,
   - Applying patches to the downloaded source code, and
   - Keeping track of changes locally made to the downloaded source code.
2. Building software using standard build tools like CMake, alleviating the need to deal with Genode's custom build system. Goa takes care of automatically installing the required Genode APIs and supplying the right parameters to the build system so that Genode executables are produced.
3. Rapidly testing the software directly on the developer's Linux host system. Goa automatically downloads Genode components needed for the test scenario.
4. Since Genode executables are binary-compatible between Linux and microkernels, the same binaries as tested on Linux can be deployed on top of the other kernels supported by Genode. Goa takes care of exporting the software in the format expected by Genode's package management.
5. Publishing (archiving and cryptographically signing) the software so that it becomes available to other Genode users, in particular users of Sculpt OS.

[1] https://genodians.org/nfeske/2019-11-25-goa
[2] https://codeberg.org/genodelabs/goa

## 2.1 Installation

1. It is recommended to use the latest long-term support (LTS) version of Ubuntu on a 64-bit x86 PC. Make sure that your installation satisfies the following requirements.
   - *libSDL-dev* needed to run system scenarios directly on your host OS,
   - *tclsh* and *expect* needed by the tools,
   - *bubblewrap* for sandboxed build environments
2. Clone the Goa repository:

   ```
   git clone https://codeberg.org/genodelabs/goa.git
   ```

   The following steps refer to the directory of the clone as `<goa-dir>`.
3. Enable your shell to locate the `goa` tool by either
   - Creating a symbolic link in one of your shell's binary-search locations (e. g., if you use a `bin/` directory in your home directory, issue `ln -s <goa-dir>/bin/goa ~/bin/`), or alternatively
   - Add `<goa-dir>/bin/` to your `PATH` environment variable, e. g., (replace `<goa-dir>` with the absolute path of your clone):

     ```
     export PATH=$PATH:<goa-dir>/bin
     ```
4. Provide the Genode toolchain by either
   - Installing the Genode toolchain following the instructions at https://genode.org/download/tool-chain. Note that Goa uses the same toolchain version as was used for the targeted Sculpt OS version. In consequence, the biennial toolchain update conducted in the framework's May-release will only take effect in Goa with the next Sculpt release.
   - Making sure *squashfs-tools* and *squashfuse* are installed so that Goa is able to download the latest Genode toolchain and make it available in the sandboxed build environment
5. Optionally, enable bash completion by adding the following line to your `~/.bashrc` file:

   ```
   source <goa-dir>/share/bash-completion/goa
   ```

Please feel welcome to explore Goa on your own. A good starting point would be the built-in help command:

```
goa help
```

## 2.2 A first example, using a plain old Makefile

Let's say, you want to build a hello-world application that uses the raw Genode API with no libc whatsoever.

First, create a project directory, let's call it "hello":

```
$ mkdir hello
$ cd hello
hello$
```

By convention, the project name corresponds to the name of the directory. Source codes are stored in a *src/* subdirectory. Let's create a file at *src/hello.cc* with the following content:

```
#include <base/log.h>
#include <base/component.h>

void Component::construct(Genode::Env &)
{
  Genode::log("Hello");
}
```

Besides the *hello.cc* file, let's create a *Makefile* at *src/Makefile* with the following content:

```
hello: hello.cc
```

Now, let's give `goa` a first try:

```
hello$ goa build
```

Goa responds with the following message:

```
[hello] Error: hello has a 'src' directory but lacks an 'artifacts' file.
               You may start with an empty file.
```

The so-called artifacts file tells Goa about the expected end result of the build process. Even though we already know from our *Makefile* that our only build artifact will be the executable binary called "hello", let's follow Goa's advise of starting with an empty *artifacts* file. Note, you may consult `goa help artifacts` for more details on the artifacts file.

```
hello$ touch artifacts
```

As a notable side effect of the `goa build` command, Goa has created a new directory called *var/* within the project directory. The *var/* directory is the designated place for generated files such as the build directory.

Upon the next attempt of issuing the `goa build` command, now with an *artifacts* file in place, Goa attempts to compile our program but with pretty limited success:

```
hello.cc:1:10: fatal error: base/log.h: No such file or directory
 #include <base/log.h>
          ^~~~~~~~~~~~
compilation terminated.
make: *** [hello] Error 1
[hello:make] <builtin>: recipe for target 'hello' failed
Error: build via make failed
```

Our program tries to include a header file that is nowhere to be found. To resolve this problem, one can tell Goa that the project needs to use the Genode base API, by placing a file named *used_apis* with the following content into the project directory.

```
genodelabs/api/base
```

This line tells Goa that the project depends on Genode's base API, which features the *base/log.h* and *base/component.h* headers. When issuing the command `goa build` again, you see the following message:

```
download genodelabs/api/base/2025-04-09.tar.xz
download genodelabs/api/base/2025-04-09.tar.xz.sig
```

Goa automatically downloaded the base API and installed it into a fresh depot at *var/depot/genodelabs/api/base/2025-04-09/*. But not only that, it also re-attempted the build of the program. If you take a look at *var/build/x86_64/*, you will see the `hello` executable. If the output was too unspectacular for your taste, you may append the `--verbose` argument to the `goa build` command to see more details about the steps taken.

To run the program, one needs to tell Goa, which part of the build artifacts are relevant. In our case, it's the `hello` executable binary. You can declare this information in your *artifacts* file by adding the following line. It refers to the respective file relative to the *var/build/x86_64/* directory.

```
hello
```

If you issue the `goa build` command again, you can see that this file appears at *var/bin/x86_64/hello*. The content of the *bin* directory is meant for the integration into a Genode scenario.

Speaking of a Genode scenario, to run the program within a Genode system, you have to define the "contract" between the program and the surrounding system. This contract has the form of a runtime package, which is stored in the *pkg/* subdirectory. A runtime package needs at least two files: a *README* file and a *runtime* file. The *README* file should give brief information about the purpose of the Genode subsystem for human readers. The runtime file contains the contractual information in human-inclined data (HID) syntax. Briefly summarized, its structure is inspired by XML (hierarchy of nodes, nodes can have attributes), yet it omits syntactic decor for calmness and clarity. For more details, please refer to the HID documentation at https://genode.org/documentation/hid.

For this example, create a file *pkg/runtime* with the following content:

```
runtime | ram: 1M | caps: 1000 | binary: hello
+ config
+ content
  + rom hello
-
```

This file declares the binary you want to start, how much RAM and capabilities the subsystem expects, configuration information passed to the subsystem, and the content of the package. In this case, you only have a single ROM module for the binary called "hello".

Note that Goa's built-in help command provides more details on the structure of runtime files.

```
goa help runtime
```

For running the scenario, one can use the `goa run` command:

```
hello$ goa run
download genodelabs/bin/x86_64/base-linux/2026-04-20.tar.xz
download genodelabs/bin/x86_64/base-linux/2026-04-20.tar.xz.sig
download genodelabs/bin/x86_64/init/2026-04-29.tar.xz
download genodelabs/bin/x86_64/init/2026-04-29.tar.xz.sig
download genodelabs/src/base-linux/2026-04-20.tar.xz
download genodelabs/src/base-linux/2026-04-20.tar.xz.sig
download genodelabs/src/init/2026-04-29.tar.xz
download genodelabs/src/init/2026-04-29.tar.xz.sig
download genodelabs/api/os/2026-04-16.tar.xz
download genodelabs/api/os/2026-04-16.tar.xz.sig
download genodelabs/api/report_session/2024-08-28.tar.xz
download genodelabs/api/report_session/2024-08-28.tar.xz.sig
download genodelabs/api/sandbox/2025-12-11.tar.xz
```

```
download genodelabs/api/sandbox/2025-12-11.tar.xz.sig
download genodelabs/api/timer_session/2025-10-12.tar.xz
download genodelabs/api/timer_session/2025-10-12.tar.xz.sig
Genode 26.02-309-g6ae975a32c8
17592186044415 MiB RAM and 19000 caps assigned to init
[init -> hello] Hello
```

You can see that Goa automatically installed the dependencies needed to execute the runtime package, integrates a Genode scenario, and executes it directly on Linux. If you switch to another terminal, you can see the Genode processes:

```
$ ps a | grep Genode

8646 pts/3    Sl+    0:00 [Genode] init
8649 pts/3    Sl+    0:00 [Genode] init -> hello
8650 pts/3    Sl+    0:02 [Genode] init -> timer
```

You can cancel the execution of the Genode scenario via Control-C.

## 2.3 A second example, using CMake

As another step, let us create a new project that executes the 3rd step of the excellent CMake tutorial[1]. Let's call the project "cmake_step3". Instead of copying the code into the *cmake_step3/src/* directory, let us better tell Goa to download the code from the original tutorial. This can be done by creating an *import* file in the project directory. Create the file *cmake_step3/import* with the following content. Please have a look at `goa help import` for a detailed explanation.

```
LICENSE   := BSD
VERSION   := 4.3.2
DOWNLOADS := cmake_step3.sparse-git

URL(cmake_step3) := https://github.com/Kitware/CMake
REV(cmake_step3) := v4.3.2
DIR(cmake_step3) := src
SPARSE_PATH(cmake_step3) := Help/guide/tutorial/Step3
```

This import file describes the download of only the specified subdirectory of the CMake project from GitHub. Let's give it a try:

```
cmake_step3$ goa import
import  download https://github.com/Kitware/CMake
import  git Cloning into '/tmp/tmp.Mp9TWeKK1E'...
...
import  generate import.hash
```

After the command finished, you find the source code sitting nicely in a new *src/* directory. Let's try to build it just after creating an empty *artifacts* file.

```
cmake_step3$ touch artifacts
cmake_step3$ goa build
[cmake_step3:cmake] -- The C compiler identification is GNU 14.2.0
[cmake_step3:cmake] -- The CXX compiler identification is GNU 14.2.0
[cmake_step3:cmake] -- Detecting C compiler ABI info
[cmake_step3:cmake] -- Detecting C compiler ABI info - done
[cmake_step3:cmake] -- Check for working C compiler:
                    /usr/local/genode/tool/25.05/bin/genode-x86-gcc - skipped
[cmake_step3:cmake] -- Detecting C compile features
[cmake_step3:cmake] -- Detecting C compile features - done
[cmake_step3:cmake] -- Detecting CXX compiler ABI info
[cmake_step3:cmake] -- Detecting CXX compiler ABI info - done
[cmake_step3:cmake] -- Check for working CXX compiler:
```

[1] https://cmake.org/cmake-tutorial/

```
                  /usr/local/genode/tool/25.05/bin/genode-x86-g++ - skipped
[cmake_step3:cmake] -- Detecting CXX compile features
[cmake_step3:cmake] -- Detecting CXX compile features - done
[cmake_step3:cmake] -- Configuring done (0.3s)
[cmake_step3:cmake] -- Generating done (0.0s)
[cmake_step3:cmake] -- Build files have been written to: .../var/build/x86_64
[cmake_step3:cmake] [ 25%] Building CXX object
        MathFunctions/CMakeFiles/MathFunctions.dir/MathFunctions.cxx.obj
.../cmake_step3/src/MathFunctions/MathFunctions.cxx:1:10:
        fatal error: iostream: No such file or directory
    1 | #include <iostream>
      |          ^~~~~~~~~~
```

Apparently, the example requires the standard C++ library. You can supply this API to the project by creating a *used_apis* file with the following content:

```
genodelabs/api/posix
genodelabs/api/libc
genodelabs/api/stdcxx
```

The `posix` API is needed because - unlike a raw Genode component - the program starts at a `main` function. The libc is needed as a dependency of the standard C++ library.

When issuing the `goa build` command again, you see that Goa downloads the required APIs and successfully builds the example program:

```
cmake_step3$ goa build
download genodelabs/api/libc/2026-03-11.tar.xz
download genodelabs/api/libc/2026-03-11.tar.xz.sig
download genodelabs/api/posix/2020-05-17.tar.xz
download genodelabs/api/posix/2020-05-17.tar.xz.sig
download genodelabs/api/stdcxx/2026-04-16.tar.xz
download genodelabs/api/stdcxx/2026-04-16.tar.xz.sig
[cmake_step3:cmake] -- Configuring done (0.0s)
[cmake_step3:cmake] -- Generating done (0.0s)
[cmake_step3:cmake] -- Build files have been written to: .../var/build/x86_64
[cmake_step3:cmake] [ 25%] Building CXX object
        MathFunctions/CMakeFiles/MathFunctions.dir/MathFunctions.cxx.obj
[cmake_step3:cmake] [ 50%] Linking CXX static library libMathFunctions.a
[cmake_step3:cmake] [ 50%] Built target MathFunctions
[cmake_step3:cmake] [ 75%] Building CXX object
        Tutorial/CMakeFiles/Tutorial.dir/Tutorial.cxx.obj
[cmake_step3:cmake] [100%] Linking CXX executable Tutorial
[cmake_step3:cmake] [100%] Built target Tutorial
```

The resulting executable binary can be found at *var/build/x86_64/Tutorial/Tutorial*. Let's declare it a build artifact by mentioning it in the *artifacts* by adding the following line.

```
Tutorial/Tutorial
```

To run the program, you need a runtime package that is slightly more advanced than the first hello example. This time, you need to declare that the runtime requires content from other depot archives in addition to your program by creating a file *pkg/archives* with the following content:

```
genodelabs/src/posix
genodelabs/src/libc
genodelabs/src/vfs
genodelabs/src/stdcxx
```

This way, the subsystem incorporates the shared libraries found in those depot archives. A suitable *pkg/runtime* for running the program within a Genode scenario looks like this:

```
runtime | ram: 10M | caps: 1000 | binary: Tutorial

+ config
  + libc | stdout: /dev/log | stderr: /dev/log
  + vfs
  | + dir dev
  |   + log
  + arg | : Tutorial
  + arg | : 24

+ content
  + rom Tutorial
  + rom posix.lib.so
  + rom libc.lib.so
  + rom libm.lib.so
  + rom stdcxx.lib.so
  + rom vfs.lib.so
-
```

Since the tutorial uses the C runtime, you have to supply a configuration that defines how the virtual file system of the component looks like, and where the program's standard output should go. The runtime also specifies the first and second arguments of the POSIX program as "Tutorial" (name of the program) and "24" as its actual argument. The `+ content` lists all ROM modules required.

With this runtime package in place, let's give the Tutorial a run:

```
cmake_step3/$ goa run
[cmake_step3:cmake] -- Configuring done (0.0s)
[cmake_step3:cmake] -- Generating done (0.0s)
[cmake_step3:cmake] -- Build files have been written to: .../var/build/x86_64
[cmake_step3:cmake] [ 50%] Built target MathFunctions
[cmake_step3:cmake] [100%] Built target Tutorial
download genodelabs/bin/x86_64/libc/2026-04-28.tar.xz
download genodelabs/bin/x86_64/libc/2026-04-28.tar.xz.sig
download genodelabs/bin/x86_64/posix/2026-04-16.tar.xz
download genodelabs/bin/x86_64/posix/2026-04-16.tar.xz.sig
download genodelabs/bin/x86_64/stdcxx/2026-04-16.tar.xz
download genodelabs/bin/x86_64/stdcxx/2026-04-16.tar.xz.sig
download genodelabs/bin/x86_64/vfs/2026-04-16.tar.xz
download genodelabs/bin/x86_64/vfs/2026-04-16.tar.xz.sig
...
download genodelabs/api/sandbox/2025-12-11.tar.xz
download genodelabs/api/sandbox/2025-12-11.tar.xz.sig
Genode 26.02-309-g6ae975a32c8
17592186044415 MiB RAM and 19000 caps assigned to init
[init -> cmake_step3] Computing sqrt of 24 to be 12.5
[init -> cmake_step3] Computing sqrt of 24 to be 7.21
[init -> cmake_step3] Computing sqrt of 24 to be 5.26936
[init -> cmake_step3] Computing sqrt of 24 to be 4.912
[init -> cmake_step3] Computing sqrt of 24 to be 4.899
[init -> cmake_step3] Computing sqrt of 24 to be 4.89898
[init -> cmake_step3] Computing sqrt of 24 to be 4.89898
[init -> cmake_step3] Computing sqrt of 24 to be 4.89898
[init -> cmake_step3] Computing sqrt of 24 to be 4.89898
[init -> cmake_step3] Computing sqrt of 24 to be 4.89898
[init -> cmake_step3] The square root of 24 is 4.89898
[init] child "cmake_step3" exited with exit value
```

You see that Goa takes care of downloading all dependencies needed to host the subsystem and subsequently executes the scenario. The program built by the tutorial prints the result “The square root of 24 is 4.89898”.

## 2.4 Running the scenario on Sculpt OS

As icing on the cake, let's give the scenario a spin on a microkernel. Sculpt OS is a Genode-based general-purpose OS compatible with commodity PC hardware. It is used as day-to-day OS by the Genode developers and can be downloaded as ready-to-use system image:

### Sculpt OS download

https://genode.org/download/sculpt

The official Sculpt image is equipped with a specifically tailored preset called "goa testbed" which allows Goa to use the system as a remote test target. Simply hook up the Sculpt system to your wifi or your wired network, and enable the *goa testbed* preset. Note down the IP address and execute the following command on your development system:

```
cmake_step3/$ goa run --target sculpt --target-opt-sculpt-server <sculpt-ip>
[cmake_step3:cmake] -- Configuring done (0.0s)
[cmake_step3:cmake] -- Generating done (0.0s)
[cmake_step3:cmake] -- Build files have been written to: .../var/build/x86_64
[cmake_step3:cmake] [ 50%] Built target MathFunctions
[cmake_step3:cmake] [100%] Built target Tutorial
uploaded Tutorial (local change)
uploaded stdcxx.lib.so (local change)
uploaded vfs.lib.so (local change)
uploaded libc.lib.so (local change)
uploaded libm.lib.so (local change)
uploaded posix.lib.so (local change)
uploaded config (local change)
Trying 192.168.42.103...
Connected to 192.168.42.103.
Escape character is '^]'.
0.0 [monitor] monitor ready
48.2 [monitor -> cmake_step3] Computing sqrt of 24 to be 12.5
48.2 [monitor -> cmake_step3] Computing sqrt of 24 to be 7.21
48.2 [monitor -> cmake_step3] Computing sqrt of 24 to be 5.26936
48.2 [monitor -> cmake_step3] Computing sqrt of 24 to be 4.912
48.2 [monitor -> cmake_step3] Computing sqrt of 24 to be 4.899
48.2 [monitor -> cmake_step3] Computing sqrt of 24 to be 4.89898
48.2 [monitor -> cmake_step3] Computing sqrt of 24 to be 4.89898
48.2 [monitor -> cmake_step3] Computing sqrt of 24 to be 4.89898
48.2 [monitor -> cmake_step3] Computing sqrt of 24 to be 4.89898
48.2 [monitor -> cmake_step3] Computing sqrt of 24 to be 4.89898
48.2 [monitor -> cmake_step3] The square root of 24 is 4.89898
48.2 [monitor] child "cmake_step3" exited with exit value 0
```

For more details, please consult Goa's built-in help command `goa help targets` or refer to Section 4.4.

# 3 Foundations

This chapter summarizes the most essential foundations of the Genode OS Framework. For a more detailed view, please refer to the Genode Foundations book available at https://genode.org.

## 3.1 Genode's init component

Genode's system architecture follows a recursive structure in which a component may invest a part of its resource budget in order to start child components. A detailed account of this is given in Section "Recursive system structure" of the Genode Foundations book.

The standard component used for nesting subsystems in Genode is the *init* component. The configuration of the init component determines what child components to start and how resources are assigned to them. A detailed account of init's configuration is given in Chapter "System configuration" of the Genode Foundations book.

When executing `goa run` or installing a runtime package on Sculpt OS, the binary specified in its *runtime* file is added as a child component to a Goa-managed or Sculpt-managed init component. The runtime package may either consist of a single component binary or make use of the init component itself to start multiple components in its subsystem. See Section 5.1 for an example.

Besides starting components and delegating resources, a parent component such as init also establishes communication channels between its child components. Any component may inform its parent about a service that it provides. Other components are then able to request access to this service. Both sides adhere to a predetermined session interface. A list of common session interfaces is provided in Section "Common session interfaces" of the Genode Foundations book.

One of the most basic session interfaces is the ROM session. It provides read-only access to binary or textual data. For instance, executable binaries and shared libraries are provided as ROM modules. Moreover, Genode components typically access a "config" ROM module, which contains the component's configuration as HID. The configuration of Genode's init component, e. g., contains a `start` node for each child component to be started. For illustration, let's have a look at a simple example:

```
+ start fs_rom | ram: 10M
  + provides | + service ROM
  + config
  + route
    + service File_system | + child vfs
    + any-service         | + parent
```

- The `start` node defines a child component named "fs_rom". By default, the child name is identical to the binary name. A different binary name can be specified by adding a `binary` sub node.
- The `ram` attribute specifies the amount of RAM delegated to the component.
- The `provides` node contains the list of session interfaces provided by the component.

- The `config` node specifies the content of the component's config ROM.
- The `route` node contains routing information for the requested services. In this example, the *File_system* session is routed to the child component named "vfs". All other services are routed to the parent component.

Note that session requests are accompanied by a session label. In order to make session requests distinguishable by the providing component, init adds the name of the requesting component as a prefix to the session label and separates the parts by `" -> "`. One may use session labels to apply more fine-grained routing rules.

**Further reading** For more details, please consult the following sections of the Genode Foundations book available on https://genode.org.

- Section "Recursive system structure"
- Section "The init component"
- Section "Common session interfaces"

## 3.2 Component API

Genode components can be classified into the following categories depending on the used API: native, libc and POSIX.

### 3.2.1 Native Genode components

```
#include <base/component.h>
#include <base/log.h>

void Component::construct(Genode::Env &)
{
  Genode::log("Hello world");
}
```

The *base/component.h* header contains the interface each component must implement. The `construct` function is called by the components execution environment to initialize the component. The interface to the execution environment is passed as argument. This interface allows the application code to interact with the outside world. The simple example above merely produces a log message. The log function is defined in the *base/log.h* header. The component does not exit after the `construct` function returns. Instead, it becomes ready to respond to requests or signals originating from other components. The example above does not interact with other components though. Hence, it will just keep waiting indefinitely.

### 3.2.2 Libc components

A libc-based component is not different from a regular Genode component and reacts on events from the surrounding system. The crucial difference lies in the semantics of the POSIX file operations, which may block on read or select. Therefore, the `Component::construct` function is not implemented in the component code but in the libc. On startup, this function prepares the C runtime, including the virtual file system, before executing the application (or libc-using component) code. The actual application is then entered via `Libc::Component::construct` on its own application context (stack and register set). Consequently, Genode components that use the libc have to implement the `Libc::Component::construct` function. The application context enables the libc to suspend and resume the execution of the application at any appropriate time, e. g., when waiting in select for a file descriptor to become readable.

```
#include <libc/component.h>
#include <stdio.h>

void Libc::Component::construct(Libc::Env &)
```

```
{
  Libc::with_libc([] () {
    printf("Hello world\n");
  });
}
```

When using libc functions in the component, the code must indicate this intention by wrapping code into `Libc::with_libc` defined as a function taking a lambda-function argument in *libc/component.h*. This ensures that code from the libc is executed exclusively by the application context and, therefore, is suspendable.

Section 3.3 provides more details on Genode's C runtime and virtual file system.

### 3.2.3 POSIX components

By using Genode's *posix* library, it is possible to build applications that use the well-known `main()` function.

```
#include <stdio.h>

int main(int argc, char **argv, char **envp)
{
  printf("Hello POSIX\n");
  return 0;
}
```

Internally, the *posix* library uses *libc/component.h* and therefore requires configuration of the C runtime and virtual file system as explained in the following section. In addition, the *posix* libary looks for `arg` and `env` nodes in the component's config ROM in order to fill the `argv` and `envp` arrays. Section 2.3 has already shown an example for this.

## 3.3 C runtime and virtual file system

Genode's C runtime bases on FreeBSD's libc and allows running Unix/POSIX-like applications. However, as a consequence of Genode's architecture, there is no global file system in Genode. Instead, every component has its own virtual file system, i. e. its own sandboxed view. Moreover, since files in Genode are no first-level citizens, special files such as sockets must be emulated.

The individual virtual file system (VFS) of a component is provided by Genode's *vfs* library. This library evaluates the `vfs` node of the component's configuration and instantiates the file-system structures accordingly. A plugin mechanism allows on-demand loading of VFS plugins, which are used to emulate special files or file systems.

If you have followed the tutorial in Section 2.3, you will have already seen the `libc` and `vfs` configuration nodes in action. This section explains their use in greater detail. For more information on writing VFS plugins, please refer to the tutorial in Section 5.3.

### 3.3.1 Libc configuration

Genode's libc library evaluates the `libc` node of the component's configuration. The `libc` node supports the following (optional) attributes:

**`stdout`**

The `stdout` attribute defines the file path in the component's VFS that is used for standard output. It is typically directed to a `log`, `null` or `terminal` file.

**`stderr`**

The `stderr` attribute defines the file path in the component's VFS that is used for error messages from libc code.

**`stdin`**

The `stdin` attribute defines the file path in the component's VFS that is used for standard input. It is typically directed to a `log`, `null` or `terminal` file.

**`rtc`**

The `rtc` attribute defines a file path in the component's VFS that provides real-time-clock data. It is typically directed to an `rtc` file or an `inline` file.

**`pipe`**

The `pipe` attribute defines a path to a `pipe` plugin in the component's VFS and thereby enables the use of POSIX pipes for inter-component communication.

**`socket`**

The `socket` attribute defines a path to a socket file system in the component's VFS. Genode's C runtime maps the BSD socket API to VFS operations in the socket file system as provided by the IP-stack VFS plugins `lwip` and `lxip`.

In addition to these attributes, the `libc` node supports the following (optional) sub nodes.

**`pthread`**

The `pthread` sub node defines the placement strategy of pthreads to CPUs. By default, the libc uses round-robin assignment of pthreads to CPUs. This is equal to `pthread | placement: all-cpus`. By using the "manual" placement strategy, one can manually tune the placement, e. g.:

```
+ libc
  + pthread | placement: manual
    + thread | id: 0 | cpu: 0 | . pthread.0 placed on CPU 0
    + thread | id: 1 | cpu: 2 | . pthread.1 placed on CPU 2
```

### 3.3.2 VFS configuration

Genode's VFS library evaluates the `vfs` node within the component's configuration. Inside the `vfs` node, one can specify an arbitrary directory structure by using nested `dir` nodes. On each level, files and subordinate file systems can be instantiated. The most basic types of these are `inline`, `rom` and `ram`. Let's have a look at an example:

```
+ config
  + vfs
    + dir tmp
      + inline foobar | : Hello!
      + rom    config | binary: false
      + ram
```

The above config specifies a */tmp/* directory with a file *foobar* that has the statically defined content "Hello!". Moreover, the directory also contains the read-only *config* file, which gets its content from the config ROM module. The `ram` node instructs the VFS library to also set up a RAM file system inside */tmp/*, much like the well-known tmpfs from Unix-like systems.

The above example illustrates how the VFS is able to provide access to Genode session interfaces (here: ROM session) via well-known file operations. As another example, one can also integrate a file-system session into a VFS by using the `fs` node:

```
+ config
  + vfs
    + fs
```

Vice versa, the VFS component provides its VFS in form of a file-system session to other components. This enables sharing of a particular VFS between several components and even allows cascading VFS components.

Complete usage examples are available in the *examples/vfs* directory of the Goa repository.

### 3.3.3 VFS plugins

The VFS library comes with various built-in file-system plugins and, moreover, is extensible via a plugin-loading mechanism.

**Built-in VFS plugins** The VFS library has the following built-in single-file systems. Every single-file system has an optional <*name*>, which defines the file's name. If the name is omitted, the node type will be used as file name.

**`+ inline <name>`**

Adds a read-only text file. The content of the `inline` node specifies the file content.

**`+ rom <name> | label: <name> | binary: yes`**

Includes a ROM module as a read-only file. The `label` attribute specifies the ROM session label. If omitted, the name will be used as ROM label. The optional `binary` attribute can be set to "no" in order to treat the ROM content as null-terminated text.

**`+ log <name> | label: <...>`**

Makes a LOG session available as a file. A `label` attribute specifies the session label. Note, read operations on the log file will block indefinitely.

**`+ null <name>`**

Instantiates a file that mimics the behaviour of */dev/null* known from Unix-like systems.

**`+ zero <name> | size: 0`**

Instantiates a file that mimics the behaviour of */dev/zero* known from Unix-like systems. The optional `size` attribute limits the number of bytes that can be read from the file. A value of 0 indicates there is no limit.

**`+ rtc <name>`**

Makes an RTC session available as a read-only file. Read operations to this file will return the current date and time in the format `%Y-%m-%d %H:%M:%S\n`.

**`+ terminal <name> | label: <...> | raw: no`**

Makes a Terminal session available as a file. The `label` attribute specifies the optional session label. The `raw` attribute can be set to "yes" in order to ignore control characters.

**+ symlink <name> | target: <...>**

Adds a symbolic link to the file specified by the `target` attribute.

**+ block <name> | label: <...> | io_buffer: 4M**

Makes a Block session available as a file. The `label` attribute specifies the optional session label. The `io_buffer` attribute sets the size of the internal I/O buffer.

Furthermore, the VFS library has the following built-in subordinate file systems:

**+ ram**

Instantiates a temporary file system that stores all data in RAM much like a *tmpfs* known from Unix-like systems.

**+ fs | label: <...> | root: / | writeable: yes | buffer_size: 128K**

Makes a file-system session available. The `label` attributes specifies the optional session label. The `root` attribute specifies the root directory of the session. Furthermore, the file system can be set to read only via the `writeable` attribute. The `buffer_size` attribute sets the size of the session's TX buffer.

**+ tar <name>**

Makes the content of a tar archive available as a read-only file system. The <*name*> attribute specifies the label of the ROM module providing the archive data.

**External VFS plugins** In addition to the aforementioned built-in plugins, the VFS library tries to load additional plugins from shared libraries. For any unknown HID node found in its configuration, the VFS library looks for a shared library file named *vfs_<node_name>.lib.so*. The VFS plugin libraries are typically found in similarly named depot archives *src/vfs_<node_name>*. A tutorial for writing VFS plugins is available in Section 5.3.

There are the following single-file system plugins. As above, the optional <*name*> can be used to change the file name.

**+ ram_log <name> | limit: 16K**

Provides an append-only file with a maximum size specified by the `limit` attribute. Should the content written to the file exceed the configured limit, the oldest content is replaced by zeros, making room for the most recent logged information.

**+ jitterentropy <name>**

Provides a random number generator or entropy source based on CPU jitter. It is typically used for emulation of */dev/random*.

**+ xoroshiro <name> | seed_path: <...>**

Provides a pseudo-random number generator (PRNG) based on the Xoroshiro128+ algorithm. It reseeds itself after a specific amount of state was consumed. The `seed_path` attribute specifies the file in the VFS that is read to reseed the PRNG. It is best suited for emulating */dev/urandom*.

**+ oss <name> | play_enabled: yes | record_enabled: yes**

Makes Record and Play sessions available as a file suitable for emulation of */dev/dsp*. For more details, please consult its README[1].

**+ gpu**

Makes GPU session signalling available as file operations. This is currently used by the Mesa library. Any Mesa application must therefore have a */dev/gpu* in its VFS.

**+ capture <name> | label: <...>**

Provides access to a Capture session. Reading from this file delivers the pixel data of a 640x480 image with 4 bytes per pixel, which is mainly useful to receive images from a webcam. The optional `label` attribute specifies the session label.

**+ tap <name> | label: <...> | mode: nic | mac: 02:02:02:02:02:02**

Makes a NIC or Uplink session available as a file for emulation of */dev/tap* devices. The `label` attribute specifies an optional session label. When setting the `mode` attribute to "uplink", the plugin opens an Uplink session instead of a NIC session. In this case, the `mac` attribute should be used to set the default MAC address. For more details, please refer to Section 5.3 or the plugin's README[2].

Furthermore, the following plugins for subordinate file systems are available:

**+ import | overwrite: no**

The import plugin defines an entire temporary file system that is copied to the root of the main VFS. Existing files remain untouched unless the `overwrite` attribute has been set to "yes".

**+ audit | label: audit | path: /real/path**

The audit plugin relays all file system accesses to the path specified by the `path` attribute while writing a corresponding message to a LOG session. The plugin uses the value of the `label` attribute as LOG session label.

[1] https://codeberg.org/genodelabs/genode/src/branch/main/repos/gems/src/lib/vfs/oss/README

[2] https://codeberg.org/genodelabs/genode/src/branch/main/repos/os/src/lib/vfs/tap/README

**+ pipe**

The pipe plugin provides a VFS backend for supporting POSIX pipes and for inter-component communication. Named pipes can be created by adding `+ fifo <name>` nodes inside the `pipe` node. For more details, please refer to the plugin's README[1].

**+ trace | ram: <...>**

The trace plugin provides access to Genode's TRACE session. The mandatory `ram` attribute specifies the session quota. For more details, please refer to the plugin's README[2].

**+ ttf | path: <...> | size_px: 16.0 | cache: 0**

The ttf plugin provides the pixel data of a TTF font. The `path` attribute specifies the path to the ttf file inside the VFS. The `cache` attribute can be used to limit the maximum number of cached bytes. For a usage example, please have a look at the raw/font archive[3].

**+ lxip | dhcp: no | ip_addr: <...> | netmask: <...> | gateway: <...> | nameserver: <...> | mtu: <...> | label: <...>**

The lxip plugin provides a socket file system and maps its file operations to the Linux IP stack backed by a NIC session. The plugin either uses DHCP or a static configuration according to the provided attributes. The optional `mtu` attribute sets the MTU. In addition, the `label` attribute can be used to change the NIC session label.

**+ lwip | dhcp: no | ip_addr: <...> | netmask: <...> | gateway: <...> | nameserver: <...> | mtu: <...> | label: <...>**

The lwip plugin provides a socket file system and maps its file operations to the Lightweight IP stack backed by a NIC session. The plugin accepts the same attributes as the lxip plugin to enable DHCP or set a static IP configuration.

**+ rump | fs: <...> | ram: <...> | writeable: yes**

The rump plugin provides a persistent file system that is backed by a Block session. The `fs` attribute determines the type of the file system ("ext2fs", "msdos" or "cd9660"). The mandatory `ram` attribute limits the amount of RAM that is used by the plugin. The file system can be set to read-only via the `writeable` attribute.

[1] https://codeberg.org/genodelabs/genode/src/branch/main/repos/gems/src/lib/vfs/pipe/README

[2] https://codeberg.org/genodelabs/genode/src/branch/main/repos/gems/src/lib/vfs/trace/README

[3] https://codeberg.org/genodelabs/genode/src/branch/main/repos/gems/recipes/raw/font/font.config

**+ fatfs**

The fatfs plugin provides a persistent file system that is backed by a Block session. It currently supports FAT and exFAT file systems.

### Further reading

#### Unix tutorial

Section 5.1 demonstrates the use of the terminal and pipe plugins.

#### VFS plugin tutorial

Section 5.3 shows how to write VFS plugins.

#### VFS article series on genodians.org

https://genodians.org/m-stein/2021-06-21-vfs-1

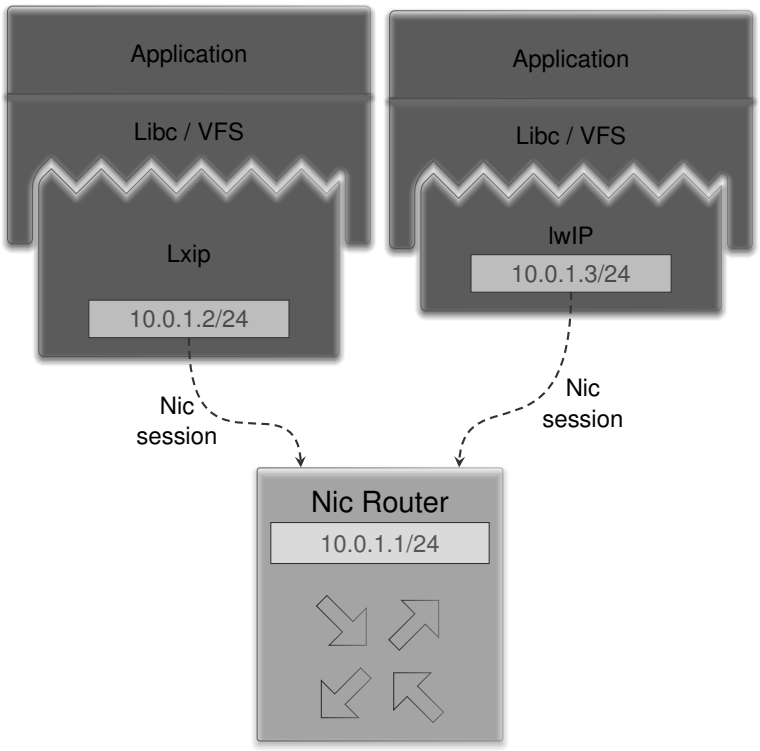
#### VFS examples in Goa repository

https://codeberg.org/genodelabs/goa/src/branch/main/examples/vfs

## 3.4 Networking

As a result of Genode's architecture, there is no centralized IP stack. Instead, every component (that requires network access) must have its own IP stack and IP address. Consequently, virtual-networking infrastructure is required for on-system routing, forwarding and network address translation. This is conducted by the NIC router.

### 3.4.1 TCP/IP stacks




In Genode, two different IP stacks are available as VFS plugins: the Linux TCP/IP stack (lxip) and the lightweight IP (lwIP) stack. These plugins implement a socket file system that translates file operations into network packets transmitted via a NIC session. By pointing Genode's C runtime to this socket file system, the BSD socket API becomes available to the application.

Below is a minimal configuration example. For more details, please refer to Section 3.3.

```
+ start ...
  + config
    + libc | socket: /sockets
    + vfs
      + dir sockets
```

```
        + lwip | dhcp: yes
```

### 3.4.2 NIC Router

The NIC router is a central building block of Genode's networking infrastructure. It acts as a resource multiplexer in order to provide multiple application components with a NIC session so that they can host their individual IP stacks. Moreover, driver components are able to connect via Uplink sessions to the NIC router as well. Having both, application components and driver components, act as client component has the benefit that the NIC router does not depend on any other component. As a consequence, driver components can be restarted or exchanged independently.

Internally, the NIC router performs network address translation and port forwarding according to its configuration. The below figure illustrates a configuration example with an NTP and HTTP server in separate virtual networks.

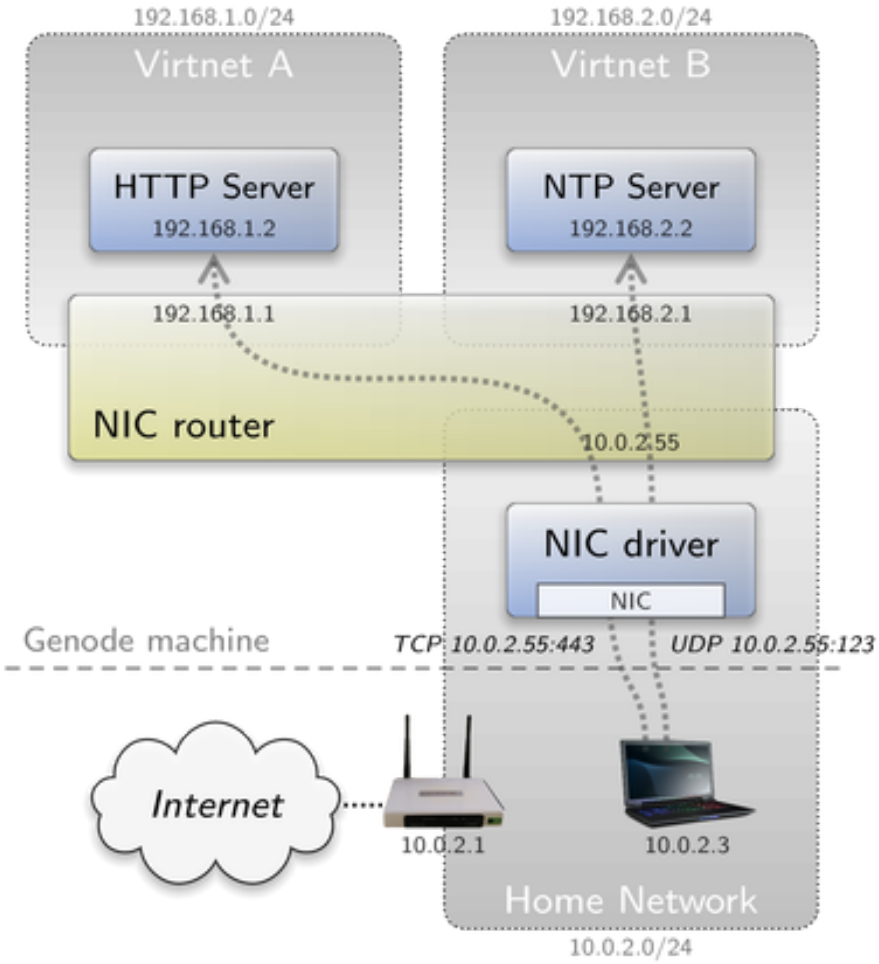


**Figure 1**

Here is the corresponding configuration snippet for the NIC router:

```
+ config
  + policy | label_prefix: virtnet_a | domain: virtnet_a
  + policy | label_prefix: virtnet_b | domain: virtnet_b

  + domain uplink | interface: 10.0.2.55/24 | gateway: 10.0.2.1
    + tcp-forward | port: 443 | domain: virtnet_a | to: 192.168.1.2
    + udp-forward | port: 123 | domain: virtnet_b | to: 192.168.2.2

  + domain virtnet_a | interface: 192.168.1.1/24
  + domain virtnet_b | interface: 192.168.2.1/24
```

The `domain` nodes define the virtual networks. The `policy` nodes assign the clients based on their session label to the defined domains. Each domain may further have its own port-forwarding rules. For a more detailed explanation on the NIC router configuration, please refer to the component's README[1].

### 3.4.3 Example: Virtual networking with Goa

By default, `goa run` executes the Genode binaries as Linux processes on the host system. For every NIC session required by the runtime, Goa starts a dedicated NIC router and a Linux-specific NIC driver. By default, Goa uses a libslirp-based driver (linux_slirp_nic) much like the user-mode networking known from QEMU. Each NIC router is set up as a DHCP server with a dedicated subnet for its clients starting from 10.0.10.0/24 and ending at 10.0.255.0/24. The first client of the first NIC router will therefore be offered the IP address 10.0.10.2.

In the *runtime* file of a Goa project, you are further able to specify additional domains and forwarding rules for the NIC router. The following example adds a TCP port forwarding from localhost:5555 on the Linux host to port 80 of the client that opens a NIC session with label "http". Similarly, it also adds a TCP port forwarding from localhost:2323 to port 23 of the client that opens a NIC session with label "telnet".

```
runtime
+ requires
  + nic http
    + tcp-forward | port: 5555      | to_port: 80
                  | domain: default | to: 10.0.10.2
  + nic telnet
```

[1] https://codeberg.org/genodelabs/genode/src/branch/main/repos/os/src/server/nic_router/README

```
    + tcp-forward | port: 2323      | to_port: 23
                  | domain: default | to: 10.0.11.2
```

Please refer to Section 3.6 for a more detailed explanation of the *runtime* file syntax.

Alternatively, Goa can be steered to use an existing tap device instead of running a SLIRP stack. This is achieved by adding a `tap_name` attribute to the `nic` node of the *runtime* file.

You can add a *tap0* device with IP address 10.1.10.1 using the following commands:

```
$ sudo ip tuntap add dev tap0 mode tap user $(whoami)
$ sudo ip address flush dev tap0
$ sudo ip addr add 10.1.10.1/24 dev tap0
$ sudo ip link set dev tap0 up
```

Since the Goa-managed NIC router issues DHCP requests to configure its uplink domain, you also require a DHCP server listening on the *tap0* device. There are several options for this depending on your Linux distribution. A lightweight DHCP server is *dnsmasq*. An exemplary configuration file *dnsmasq.conf* could look like this:

```
port=5353
interface=tap0
domain=lan
dhcp-range=10.1.10.2,10.1.10.2,12h
dhcp-option=6,1.1.1.1
```

With this file, you are able to start the DHCP server from the command line:

```
$ sudo dnsmasq -C dnsmasq.conf
```

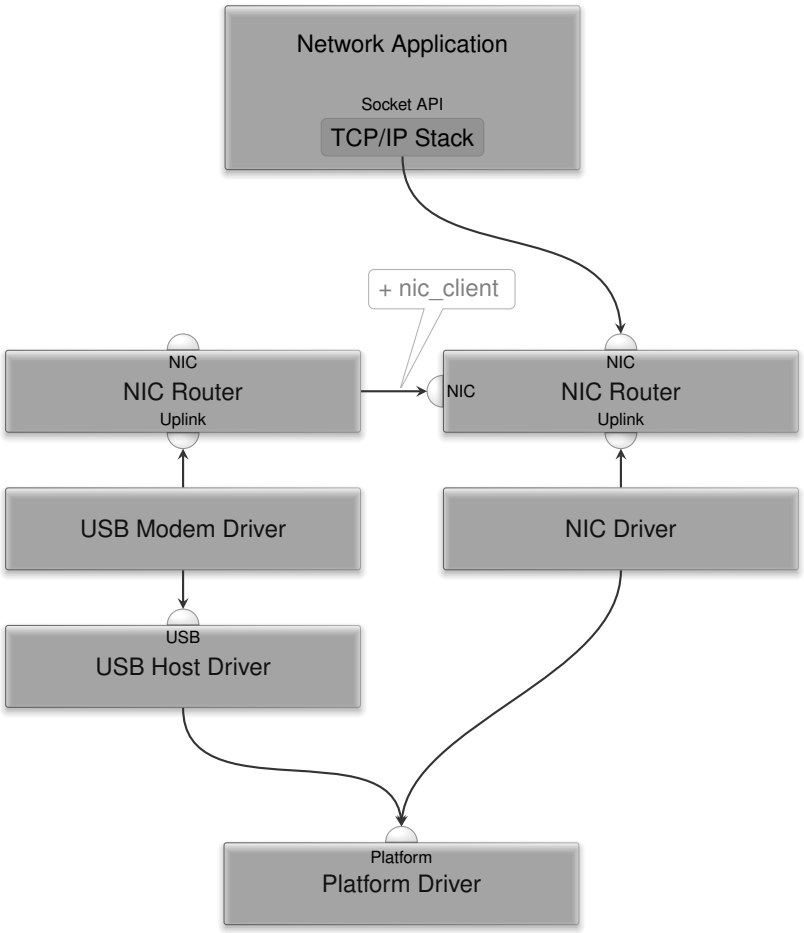
The DHCP server also announces a DNS server (1.1.1.1). In order to actually permit network traffic to the physical interface, you must enable IP forwarding and NATing:

```
$ sudo sysctl net.ipv4.ip_forward=1
$ sudo iptables -t nat -A POSTROUTING -j MASQUERADE
```

### 3.4.4 Example: Cascaded NIC routers

The NIC router can itself act as a NIC session client. This enables cascading router setups. For example, let's assume we start a subsystem with an LTE modem and its own NIC router. Now, we want to route network traffic from application components to the mobile network instead of a wired network. Application components that are already connected to another NIC router would, however, require a restart if we changed their service routing. By letting the NIC router in our subsystem act as a NIC client, we are

able to route network packets between the NIC routers. The figure below illustrates this setup. For a more detailed explanation, please refer to the corresponding article on genodians.org[1].




[1] https://genodians.org/jschlatow/2021-07-21-mobile-network

## 3.5 Package management

When speaking about "package management", one has to clarify what a "package" in the context of an operating system represents. Traditionally, a package is the unit of delivery of a bunch of "dumb" files, usually wrapped up in a compressed archive. A package may depend on the presence of other packages. Thereby, a dependency graph is formed. To express how packages fit with each other, a package is usually accompanied by meta data (description). Depending on the package manager, package descriptions follow certain formalisms (e. g., package-description language) and express more-or-less complex concepts such as versioning schemes or the distinction between hard and soft dependencies.

Genode's package management does not follow this notion of a "package". Instead of subsuming all deliverable content under one term, we distinguish different kinds of content, each in a tailored and simple form. To avoid the clash of the notions of the common meaning of a "package", we speak of "archives" as the basic unit of delivery. Archives are named with their version as suffix, appended via a slash. This results in the following scheme for architecture-independent archives:

```
<type>/<name>/<version>
```

Binary archives, on the other hand, are architecture-specific and adhere to a slightly different scheme that includes the target architecture:

```
<type>/<name>/<arch>/<version>
```

This section focuses on depot-archive management with Goa. For a more general explanation of archive categories, please refer to Section "Package management" in the Genode Foundations book.

With Goa, depot archives are created and published by the commands `goa export` and `goa publish`. Depending on the project-directory content, Goa creates the necessary depot archives. The project directory therefore follows the depot nomenclature as follows:

**raw/**

A raw-data archive contains arbitrary data that is independent of the processor architecture. If there is a *raw/* subdirectory, Goa takes its entire content to create a raw archive named after the project.

**src/**

Goa creates a source archive for a project if there exists a *src/* subdirectory. A source archive contains to-be-compiled source code. The directory content can either be manually managed or imported (see `goa help import`). Goa also creates

a corresponding, equally-named, binary (bin) archive containing the build artifacts as specified in the project's *artifacts* file (see `goa help artifacts`). Genode binaries are stripped from debug information. Instead, this information is made available in separate debug info files. Goa deals with downloading, exporting and publishing of the corresponding debug (dbg) archives when provided with the `--debug` switch.

**pkg/**

A package archive specifies what ingredients are needed to deploy and execute a certain scenario. It comprises three files: *archives*, *runtime* and *README*. The *archives* file lists the names of all required raw, source, or package archives. The *runtime* file describes the required/provided services and the subsystem configuration (see Section 3.6). Goa allows maintaining multiple package archives in the same project directory. It expects the content of each package archive in a *pkg/<name>/* subdirectory.

**api**

Goa creates an API archive if there is an *api* file in the project directory (see `goa help api`). An API archive is typically associated with a shared library and is meant to provide all the ingredients for building components that use this library. The archive contains header files and the library's binary interface in the form of an ABI-symbols file. Unless it is a header-only library, the API archive is accompanied by an equally-named source and binary archive.

**index**

Goa creates a depot index if there is an *index* file present in the project directory (see `goa help index`). A depot index describes the available package archives within a depot.

## 3.6 Runtime configuration

The *runtime* file of a package archive specifies the ingredients that are needed to deploy the archive on a Genode system. A *runtime* file has the following structure:

```
runtime | ram: ... | caps: ... | binary: ...
+ requires
  .
  . list of required session interfaces, e.g:
  .
  + gui
  + fs  <label>
  + rom <label>

+ provides
  .
  . list of provided session interfaces, e.g:
  .
  + report

+ content
  .
  . list of required ROM modules, e.g.:
  .
  + rom <module_name>
  + rom <module_name> | as: <name>

+ config | . component config
```

The runtime must define the amount of RAM, the number of capabilities and the binary name. It also lists the required and provided session interfaces. Note that the sub-nodes of the `requires` and `provides` nodes are the lower-case service names (with the exception of allowing `fs` as an abbreviation for `file_system`). Session labels can be specified as node names.

The `content` node contains a list of required ROM modules (e. g. binaries, libraries, config files). Furthermore, the component's config can be added via a `config` node. For more details, please consult Goa's built-in help:

```
$ goa help runtime
```

The *runtime* file is also evaluated by `goa run` in order to set up a suitable Genode environment on the host system. Section 3.4.3 has illustrated how Goa uses additional attributes and content of a `nic` node to set up virtual networking. Please consult Goa's built-in help for an explanation of how the other services are emulated by Goa.

```
$ goa help targets
```

## 3.7 Graphical User Interfaces

Since its first release, Genode came with its own low-level GUI stack centered around a component called *Nitpicker GUI server*. Nitpicker provides three types of session interfaces: GUI, Capture, and Event. Similar to the NIC router, Nitpicker is a resource multiplexer. It mediates between framebuffer driver, input drivers, and applications. Applications use the GUI session interface, which provides low-level access for writing to the framebuffer and receiving input events. Rather than sticking to low-level drawing methods, GUI frameworks provide a more suitable level of abstraction for application development.

This section provides an overview of the available GUI frameworks for Genode. For a more detailed explanation of Genode's low-level GUI stack, please refer to the corresponding article on genodians.org[1].

### 3.7.1 SDL

The Simple DirectMedia Layer (SDL) is a well-established cross-platform library often used by computer games. Ports of SDL 1.2 and SDL 2.0 are available in the genode-world repository. Additional SDL libraries such as SDL_image, SDL_ttf, SDL_net and SDL_mixer are also available.

#### Genode-world repository

https://codeberg.org/genodelabs/genode-world/

Examples of Genode applications making use of SDL are available in the following repositories:

#### Port of numptyphysics

https://codeberg.org/nfeske/goa-projects/src/branch/main/games/numptyphysics

### 3.7.2 Qt (5/6)

Qt is a popular cross-platform application development framework. Early versions of Genode already included a port of Qt4 that was later updated to Qt5 and, most recently, Qt6. Since Genode's port of the Falkon browser bases on Qt, and QtWebengine in particular, this is the best supported GUI framework for Genode applications.

#### Qt5 examples and tutorials

https://doc.qt.io/qt-5/qtexamplesandtutorials.html

[1] https://genodians.org/nfeske/2020-06-23-gui-stack

#### Qt6 examples and tutorials

https://doc.qt.io/qt-6/qtexamplesandtutorials.html

Examples of Genode applications making use of Qt:

#### Falkon web browser

https://codeberg.org/genodelabs/genode-world/src/branch/main/legacy/recipes/pkg/falkon

#### Qt6 textedit

https://codeberg.org/genodelabs/genode/src/branch/main/repos/libports/recipes/pkg/qt6_textedit

### 3.7.3 Mobile SDK based on Ubuntu/Lomiri UI Toolkit

The Ubuntu UI Toolkit bases on Qt5 and particularly targets touchscreen-optimized application development. Since UBports resumed the development for Ubuntu Touch after Canonical dropped support, the toolkit was renamed from *Ubuntu UI Toolkit* to *Lomiri UI Toolkit*.

#### Port of Ubuntu UI Toolkit

https://codeberg.org/genodelabs/genode-world/src/branch/main/legacy/recipes/pkg/ubuntu_ui_toolkit

#### Porting the calculator app from Lomiri UI Toolkit

see Section 5.4

#### UBports website

https://ubports.com/

Examples of Genode applications making use of the Ubuntu UI Toolkit:

#### Morph browser

https://codeberg.org/genodelabs/genode-world/src/branch/main/legacy/recipes/pkg/morph_browser

#### Linphone app

https://genodians.org/jws/2023-11-16-sip-client-for-genode

### 3.7.4 Light and Versatile Graphics Library (LVGL)

LVGL is a popular graphics library to create modern UIs for embedded devices. Being optimized for embedded devices, LVGL comes with a small memory footprint. This makes it a perfect fit for rather simple Genode applications.

Since LVGL targets embedded devices, it is typically used as a statically linked library and stripped down to the particular needs. For Genode, however, LVGL is available as a shared library (api/lvgl) with almost all features enabled. The LVGL library is accompanied by a support library (api/lvgl_support) providing the LVGL driver backends that interact with Genode's GUI session. Both libraries are still in experimental state.

#### LVGL documentation

https://docs.lvgl.io/

#### Dynamic desktop background "system info"

https://genodians.org/jschlatow/2024-02-07-system-info

## 4 Development & Debugging

This chapter describes how to prepare and build Genode executables for debugging. Furthermore, it shows how to debug a runtime scenario on a Linux host and on Sculpt OS.

## 4.1 Adding debug info files

Binary depot archives merely contain stripped binaries. Release 23.11[1] added the option to build and publish *dbg* archives that contain the corresponding debug info files along with the binary archives.

When provided with the `--debug` switch, Goa takes care of *dbg* archives. A `goa run --debug` will thus try downloading required *dbg* archives before running the scenario and link the debug info files into the *.debug* subdirectory of the project's run directory. Moreover, it will create debug info files for all binary artifacts of the project. When exporting/publishing a project, the `--debug` switch instructs Goa to create *dbg* archives along with the created *bin* archives.

[1] https://genode.org/documentation/release-notes/23.11#Debug_information_for_depot_binaries

## 4.2 Using backtraces

Genode's *os* API provides the utility function `Genode::backtrace()` to walk the stack and print the return addresses along the way. In order to use this function, *genodelabs/api/os* must be added to the *used_apis* file. The function is then made available by including the *os/backtrace.h* header. For demonstration, let's have a look at the *system_info* component (Section 3.7.4). After inserting a `Genode::backtrace()` in `Info::Bar::_draw_part_event_cb()` in *system_info.h* followed by an infinite loop, `goa run` produces the following output:

```
system_info$ goa run
Genode 26.02-309-g6ae975a32c8
17592186044415 MiB RAM and 19000 caps assigned to init
[init -> system_info] [Warn]    (0.000, +0) lv_init: Style sanity checks [...]
[init -> system_info] backtrace "ep"
[init -> system_info]   <no return address found>
```

This is obviously not very helpful. To assist the `backtrace()` function to parse stack frames correctly, the build system must be instructed to preserve frame-pointer information. Goa now provides the command-line switch `--with-backtrace` for this purpose. Let's give it a try:

```
system_info$ goa run --with-backtrace
Genode 26.02-309-g6ae975a32c8
17592186044415 MiB RAM and 19000 caps assigned to init
[init -> system_info] [Warn]    (0.000, +0) lv_init: Style sanity checks [...]
[init -> system_info] backtrace "ep"
[init -> system_info]   403ff708       1008f5a
[init -> system_info]   403ff7b8       103f8fd
[init -> system_info]   403ff7e8  7ffff7fd3d10
```

The second column of the backtrace data shows the return addresses on the call stack. The first address certainly belong to the *system_info* binary. The second address, however, looks as if it might already belong to a shared library. For evaluation of the backtrace, one needs to know to which addresses the shared libraries have been relocated. This information is acquired by adding the `ld_verbose: yes` attribute to the component's config. Let's try again with `ld_verbose`:

```
system_info$ goa run --with-backtrace
Genode 26.02-309-g6ae975a32c8
17592186044415 MiB RAM and 19000 caps assigned to init
[init -> system_info]   0x1000000 .. 0x16dfffff: linker area
[init -> system_info]   0x40000000 .. 0x4fffffff: stack area
[init -> system_info]   0x50000000 .. 0x601d3fff: ld.lib.so
```

```
[init -> system_info]   0x16bfa000 .. 0x16dfffff: libc.lib.so
[init -> system_info]   0x16b35000 .. 0x16bf9fff: vfs.lib.so
[init -> system_info]   0x16af3000 .. 0x16b34fff: libm.lib.so
[init -> system_info]   0x1021000 .. 0x1255fff: liblvgl.lib.so
[init -> system_info]   0x1256000 .. 0x1279fff: liblvgl_support.lib.so
[init -> system_info]   0x127a000 .. 0x1518fff: stdcxx.lib.so
[init -> system_info] [Warn]    (0.000, +0) lv_init: Style sanity checks [...]
[init -> system_info] backtrace "ep"
[init -> system_info]   403ff708       1008f5a
[init -> system_info]   403ff7b8       103f8fd
[init -> system_info]   403ff7e8  7ffff7fd3d10
```

The output confirms that the second address belongs to *liblvgl.lib.so*. For convenient interpretation of the backtrace data, Goa mirrors the *tool/backtrace* utility from the Genode repository. This utility translates the addresses from the backtrace into source code lines. The `goa backtrace` command executes a `goa run --debug --with-backtrace` and feeds the log output into the backtrace tool:

```
system_info$ goa backtrace
Genode 26.02-309-g6ae975a32c8
17592186044415 MiB RAM and 19000 caps assigned to init
[init -> system_info]   0x1000000 .. 0x16dfffff: linker area
[init -> system_info]   0x40000000 .. 0x4fffffff: stack area
[init -> system_info]   0x50000000 .. 0x601d3fff: ld.lib.so
[init -> system_info]   0x16bfa000 .. 0x16dfffff: libc.lib.so
[init -> system_info]   0x16b35000 .. 0x16bf9fff: vfs.lib.so
[init -> system_info]   0x16af3000 .. 0x16b34fff: libm.lib.so
[init -> system_info]   0x1021000 .. 0x1255fff: liblvgl.lib.so
[init -> system_info]   0x1256000 .. 0x1279fff: liblvgl_support.lib.so
[init -> system_info]   0x127a000 .. 0x1518fff: stdcxx.lib.so
[init -> system_info] [Warn]  (0.000, +0) lv_init: Style sanity checks [...]
[init -> system_info] backtrace "ep"
[init -> system_info]   403ff708       1008f5a
[init -> system_info]   403ff7b8       103f8fd
[init -> system_info]   403ff7e8  7ffff7fd3d10
Expect: 'interact' received 'strg+c' and was cancelled
Scanned image system_info
Scanned image ld.lib.so
Scanned image libc.lib.so
Scanned image vfs.lib.so
Scanned image libm.lib.so
Warning: /usr/local/genode/tool/23.05/bin/genode-x86-nm:
         '.debug/liblvgl.lib.so.debug': No such file
Warning: /usr/local/genode/tool/23.05/bin/genode-x86-nm:
         '.debug/liblvgl_support.lib.so.debug': No such file
```

```
Scanned image stdcxx.lib.so
Info::Bar::_draw_part_event_cb(_lv_event_t*)

    * 0x1008f5a: system_info:0x1008f5a W
    * /depot/genodelabs/api/base/2026-04-16/include/base/log.h:72

_prog_img_end

    * 0x103f8fd: system_info:0x103f8fd B
    * ??:0

_end

    * 0x7ffff7fd3d10: ld.lib.so:0x7ffff7fd3d10 b
    * ??:0
```

The output shows that the first address on the stack points to `_draw_part_event_cb()` in which we inserted the backtrace call. The second address presumably points to liblvgl where the callback method was called, however, as the warning message indicates, the corresponding debug symbols are missing.

Let's re-export liblvgl using the `--with-backtrace` switch and try again:

```
system_info$ goa -C ../../ports/lvgl export --debug --with-backtrace
...
[lvgl] exported [...]/depot/_/api/lvgl/2026-04-21
[lvgl] exported [...]/depot/_/src/lvgl/2026-04-21
[lvgl] exported [...]/depot/_/bin/x86_64/lvgl/2026-04-21
[lvgl] exported [...]/depot/_/dbg/x86_64/lvgl/2026-04-21
```

```
system_info$ goa backtrace
...
[init -> system_info] backtrace "ep"
[init -> system_info]   403ff648   1008f5a
[init -> system_info]   403ff6f8   103f8ce
[init -> system_info]   403ff728   103f978
[init -> system_info]   403ff778   10a5933
[init -> system_info]   403ff968   103f665
[init -> system_info]   403ff988   103f7aa
[init -> system_info]   403ff9b8   103f978
[init -> system_info]   403ffa08   104e180
[init -> system_info]   403ffa88   104e127
[init -> system_info]   403ffb08   104e127
[init -> system_info]   403ffb88   104e127
[init -> system_info]   403ffc08   104e127
```

```
[init -> system_info]   403ffc88   104e8aa
[init -> system_info]   403ffcd8   104e999
[init -> system_info]   403ffe08   104fe23
[init -> system_info]   403ffe98   10a21e8
[init -> system_info]   403ffee8  16d44782
Expect: 'interact' received 'strg+c' and was cancelled
Scanned image system_info
Scanned image ld.lib.so
Scanned image libc.lib.so
Scanned image vfs.lib.so
Scanned image libm.lib.so
Scanned image liblvgl.lib.so
Warning: /usr/local/genode/tool/23.05/bin/genode-x86-nm:
         '.debug/liblvgl_support.lib.so.debug': No such file
Scanned image stdcxx.lib.so
Info::Bar::_draw_part_event_cb(_lv_event_t*)

    * 0x1008f5a: system_info:0x1008f5a W
    * /depot/genodelabs/api/base/2026-04-16/include/base/log.h:72

event_send_core

    * 0x103f8ce: liblvgl.lib.so:0x1e8ce t
    * /depot/_/src/lvgl/2026-04-21/src/src/core/lv_event.c:469

lv_event_send

    * 0x103f978: liblvgl.lib.so:0x1e978 T
    * /depot/_/src/lvgl/2026-04-21/src/src/core/lv_event.c:78

draw_indic

    * 0x10a5933: liblvgl.lib.so:0x84933 t
    * /depot/_/src/lvgl/2026-04-21/src/src/widgets/lv_bar.c:506
...
```

Well, that looks much more helpful.

## 4.3 Debugging with Goa on base-linux

The Goa tool streamlines application development and testing as it allows executing a Genode runtime directly on the Linux host system. Goa leverages the ABI compatibility of Genode executables with all supported kernels. Genode executables can therefore be run as Linux processes (using base-linux).

Goa's default run target *linux* creates a *<project-name>.gdb* file in the project's var directory to assist with GDB's initialisation. Other run targets may copy this convention. As mentioned in Section 4.1, Goa should be provided with the `--debug` switch to prepare the run directory with additional debug info files:

```
system_info$ goa run --debug
```

Once the scenario of interest is running, you need to find the process ID (PID) of the to-be-debugged component (e. g. by using `pgrep -f`). With the PID at hand, you can start GDB and attach to the running process:

```
system_info$ sudo gdb --command ./var/system_info.gdb
GNU gdb (GDB) 17.1
Copyright (C) 2025 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) attach 18104
Attaching to process 18104
[New LWP 18119]
[New LWP 18114]
Reading symbols from [...]/base-linux/2026-04-20/ld.lib.so...
(No debugging symbols found in [...]/base-linux/2026-04-20/ld.lib.so)
0x00000000500ba8aa in ?? ()
(gdb)
Thread 3 "ld.lib.so" stopped.
0x00000000500ba8aa in ?? ()

Thread 2 "ld.lib.so" stopped.
```

```
0x00000000500ba8aa in ?? ()
```

On attach, GDB fails to load symbols from the binary because it does not know about the location of the corresponding debug info file. Moreover, GDB stops execution of all threads.

The *<project-name>.gdb* file instructs GDB to change into the run directory, where the debug info files are made available in the *.debug* subdirectory. GDB provides the commands *symbol-file* and *add-symbol-file* for symbol loading. The former is used for the main binary whereas the latter is intended for adding shared-library symbols. Let's give it a try:

```
(gdb) symbol-file .debug/system_info.debug
Reading symbols from .debug/system_info.debug...

(gdb) add-symbol-file .debug/ld.lib.so
add symbol table from file ".debug/ld.lib.so.debug"
(y or n) y
Reading symbols from .debug/ld.lib.so.debug...

(gdb) add-symbol-file .debug/liblvgl.lib.so.debug -o 0x1021000
add symbol table from file ".debug/liblvgl.lib.so.debug" with all sections
  offset by 0x1021000
(y or n) y
Reading symbols from .debug/liblvgl.lib.so.debug...
```

Except for the main binary and ld.lib.so, an offset address must be specified when loading symbols depending on where the libraries have been relocated. These addresses are shown by adding `ld_verbose: yes` to the component config.

With the symbols loaded, GDB's `info threads` command shows at which line each thread has been stopped:

```
(gdb) info threads
  Id   Target Id             Frame
* 1    LWP 18104 "ld.lib.so" pseudo_end () at [...]/spec/x86_64/lx_syscall.S:29
  2    LWP 18119 "ld.lib.so" pseudo_end () at [...]/spec/x86_64/lx_syscall.S:29
  3    LWP 18114 "ld.lib.so" pseudo_end () at [...]/spec/x86_64/lx_syscall.S:29
```

The selected thread is marked with an ∗. Let's continue all threads and switch to thread 2 (see Section 4.4 for more details):

```
(gdb) continue -a &
Continuing.
(gdb) thread 2
```

```
[Switching to thread 2 (LWP 18119)]
(gdb) info threads
  Id   Target Id            Frame
  1    LWP 18104 "ld.lib.so" (running)
* 2    LWP 18119 "ld.lib.so" (running)
  3    LWP 18114 "ld.lib.so" (running)
```

At this point, you are able to step through the individual threads:

```
(gdb) interrupt
Thread 2 "ld.lib.so" stopped.
pseudo_end () at [...]/src/lib/syscall/spec/x86_64/lx_syscall.S:29
29              ret             /* Return to caller.  */
(gdb) stepi
0x0000000050053228 in lx_recvmsg (sockfd=[...]
    [...]/src/lib/syscall/linux_syscalls.h:205
205             return (int)lx_syscall(SYS_recvmsg, sockfd.value, msg, flags);
(gdb)
```

Admittedly, navigating through the depth of ld.lib.so is a bit cumbersome. For serious debugging, you would ideally be using breakpoints. GDB provides the `list` command for showing source code. Let's peek into *system_info.cc* and insert a breakpoint in `handle_resize()`:

```
(gdb) list system_info.cc:90
85                                  .use_mouse          = false,
86                                  .use_periodic_timer = true,
87                                  .periodic_ms        = 5000,
88                                  .resize_callback    = &_resize_callback,
89                                  .timer_callback     = &_timer_callback,
90      };
91
92
93      void handle_resize()
94      {
(gdb) break system_info.cc:95

Breakpoint 1 at 0x1001150: system_info.cc:95. (2 locations)
Warning:
Cannot insert breakpoint 1.
Cannot access memory at address 0x1001150
Cannot insert breakpoint 1.
Cannot access memory at address 0x10013aa
```

Unfortunately, base-linux prevents inserting breakpoints at runtime by default. You may apply the following patch to base-linux in order to enable software breakpoints:

```
--- a/repos/base-linux/src/lib/base/region_map_mmap.cc
+++ b/repos/base-linux/src/lib/base/region_map_mmap.cc
@@ -132,7 +132,8 @@ Region_map_mmap::_map_local(Dataspace_capability ds,
        writeable = _dataspace_writeable(ds) && writeable;

        int  const  fd        = _dataspace_fd(ds);
-       int  const  flags     = MAP_SHARED | (overmap ? MAP_FIXED : 0);
+       int  const  flags     = (writeable ? MAP_SHARED : MAP_PRIVATE)
+                             | (overmap ? MAP_FIXED : 0);
        int  const  prot      = PROT_READ
                              | (writeable  ? PROT_WRITE : 0)
                              | (executable ? PROT_EXEC  : 0);
```

For providing the modified base-linux archive to Goa, you need to build pkg/goa and pkg/goa-linux and tell Goa not to use the *genodelabs* archives but your own archives by using the `--genodelabs-user <user>` argument and by adding the `--versions-from-genode-dir <path>` argument. Alternatively, you may edit Goa's linux.tcl file to pin only the base-linux archive to your depot.

Let's opt for the latter version and provide Goa with the corresponding version information using a `--version-...` argument:

```
system_info$ goa run --debug --version-_/src/base-linux 2025-05-18
```

After repeating the steps for symbol loading, breakpoints can be added successfully:

```
(gdb) break system_info.cc:95
Breakpoint 1 at 0x1001150: system_info.cc:95. (2 locations)
(gdb)
```

When resizing the fb_sdl window, the breakpoint hits:

```
Thread 3 "ld.lib.so" hit Breakpoint 1.1, Main::Resize_callback::operator()
    at system_info.cc:95
95                      Libc::with_libc([&] {
```

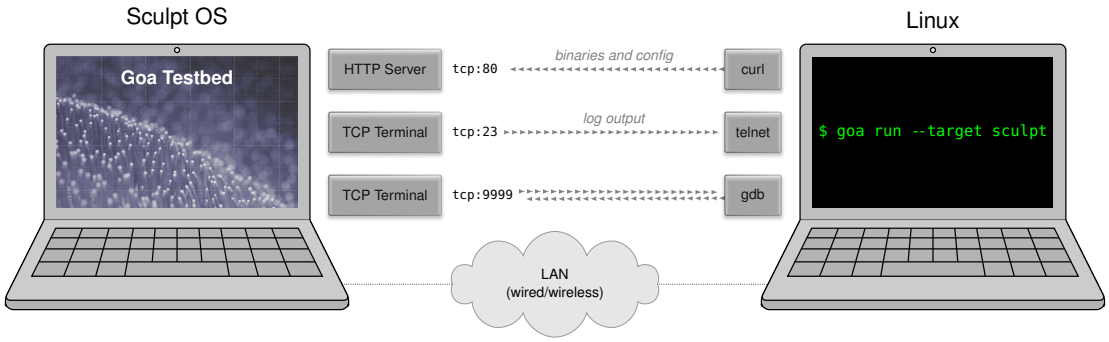
## 4.4 Using Sculpt as a remote test target

Running (and debugging) Genode applications with Goa on base-linux is typically the first step. For advanced runtime scenarios, Goa also supports using a Sculpt system as a remote test target, which eliminates the need for manually transferring depot archives.

Goa places all files required for running a scenario in the project's run directory. By transferring these files to the remote system, we are basically able to launch the scenario on that system. A specifically tailored subsystem called "goa testbed" is available as a preset since Sculpt 24.04. This subsystem hosts a *lighttpd* server with the *mod_webdav* module enabled. This allows Goa to use the server-provided HTTP ETags to identify what files from the run directory need to be (re-)uploaded via HTTP PUT.

In addition to lighttpd, the testbed runs a sub-init that reacts to changes to the config file from the synchronised run directory. Once all prerequisites have been synchronised, starting a scenario on the remote system comes down to uploading the config file. By deleting the config file from the remote system, the scenario is killed.

Log output is made available via telnet using an integrated TCP terminal component. Since Sculpt 24.10, the Goa testbed uses the debug monitor for the sub-init in order to support debugging via GDB. The debug monitor's terminal connection is made available via a separate TCP terminal. The below figure illustrates the resulting interplay between Goa and the Goa testbed.




In order to run a Goa project on a remote Sculpt system, you first need to launch goa_testbed, which is best done by enabling the built-in preset.

On the development system, you can switch the run target by adding the `--target sculpt` option to Goa's command line. The IP address of the remote system is specified by the `--target-opt-sculpt-server` argument (see `goa help targets`). Let's give the *system info* scenario a spin:

```
system_info$ goa run --target sculpt --target-opt-sculpt-server <sculpt-ip>
uploaded libm.lib.so (local change)
uploaded stdcxx.lib.so (local change)
uploaded vfs.lib.so (local change)
```

```
uploaded liblvgl.lib.so (local change)
uploaded system_info (local change)
uploaded posix.lib.so (local change)
uploaded liblvgl_support.lib.so (local change)
uploaded libc.lib.so (local change)
uploaded config (local change)
Trying 192.168.42.103...
Connected to 192.168.42.103.
Escape character is '^]'.
0.0 [monitor] monitor ready
51.1 [monitor -> system_info] [Warn]  (0.000, +0) lv_init: Style sanity checks [...]
Expect: 'interact' received 'strg+c' and was cancelled
deleted config
```

The app magically pops up on the target system and the log output is shown on the development system. When hitting ctrl+c, the config is deleted from the target system, which kills the app.

For starting a debugging session, you should add the `--debug` and `--target-opt-sculpt-kernel` arguments. The latter tells Goa what kernel the remote target is running so that the debug symbols of the corresponding ld.lib.so library can be made available:

```
system_info$ goa run --debug --target sculpt \
  --target-opt-sculpt-server 192.168.42.103 --target-opt-sculpt-kernel nova
```

The *sculpt* run target follows the lead of the *linux* target and also generates a *<project-name>.gdb* to assist GDB initialisation. Let's peek into the file:

```
$ cat [...]/var/system_info.gdb
cd [...]/var/run
set non-stop on
set substitute-path /data/depot-sculpt /home/johannes/repos/genode/depot
set substitute-path /data/depot /home/johannes/repos/genode/depot
set substitute-path /depot /home/johannes/repos/genode/depot
target extended-remote 192.168.42.103:9999
```

The file instructs GDB to change into the project's run directory and sets GDB into non-stop mode. Moreover, GDB must be pointed to the correct depot location on the host system. The paths from the debug info files typically refer to files at */data/depot* or */depot*. These paths can be relocated by using the `set substitute-path` command. The last line instructs GDB to connect to the remote target using the address provided via the `--target-opt-sculpt-server` argument and the port provided by `--target-opt-sculpt-port-gdb`.

Let's start GDB with this file. In contrast to debugging on Linux, you should use the gdb binary from the Genode toolchain. Moreover, root privileges are not required.

```
$ genode-x86-gdb --command /path/to/project/var/project_name.gdb
GNU gdb (GDB) 14.2
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-pc-linux-gnu --target=x86_64-pc-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
(gdb) warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
(gdb) info inferiors
  Num  Description       Connection                              Executable
* 1    process 1         1 (extended-remote 192.168.42.103:9999)
```

GDB's `info inferiors` command lists a single process. Let's have a look at the threads:

```
(gdb) info threads
  Id   Target Id                  Frame
  1    Thread 1.1 "system_info"   (running)
  2    Thread 1.2 "ep"            (running)
* 3    Thread 1.3 "signal handler" (running)
```

Fortunately, we are provided with the thread names. The "system_info" thread is the initial thread set up by Genode's init component. After component initialisation, however, the entrypoint thread "ep" becomes the most interesting thread for Genode components. Let's therefore switch to thread 2 as we did in the previous section without giving any explanation:

```
(gdb) thread 2
[Switching to thread 2 (Thread 1.2)](running)
```

As before, symbols must be loaded manually:

```
(gdb) symbol-file .debug/system_info.debug
Reading symbols from .debug/system_info.debug...

(gdb) add-symbol-file .debug/ld.lib.so.debug
add symbol table from file ".debug/ld.lib.so.debug"
(y or n) y
Reading symbols from .debug/ld.lib.so.debug...

(gdb) add-symbol-file .debug/liblvgl.lib.so.debug -o 0x1021000
add symbol table from file ".debug/liblvgl.lib.so.debug" with all sections
  offset by 0x1021000
(y or n) y
Reading symbols from .debug/liblvgl.lib.so.debug...
```

With the most essential symbols available, you can insert a software breakpoint in the `handle_resize()` method and trigger it by resizing the window on the target system:

```
(gdb) break system_info.cc:95
Breakpoint 2 at 0x1001150: system_info.cc:95. (2 locations)
(gdb)
Thread 2 "ep" hit Breakpoint 2.1, Main::Resize_callback::operator()
[...] at system_info.cc:95
95                      Libc::with_libc([&] {
```

Perfect. Note that in some occasions, it can be helpful to insert breakpoints at compile time to halt the execution before an error condition occurs. On x86, this can be achieved by inserting an `asm volatile ("int3")` at the point of interest. Happy debugging!

## 4.5 Further reading

### 4.5.1 Using a VNC server on a remote test target

Goa's ability to run applications on a remote Sculpt system comes in handy for testing. However, switching between keyboards to control the remote-running application can be a tiny inconvenience. The following article demonstrates how a VNC server can be put into use for remote accesses to GUI applications.

**Using a headless Sculpt as a remote test target**

https://genodians.org/jschlatow/2024-06-04-goa-sculpt-vnc

### 4.5.2 On-target debugging with GDB

Live debugging of Sculpt runtime components is a built-in feature since version 24.04. Instructions and live demo are available on genodians.org:

**On-target debugging with GDB on Sculpt OS 24.04**

https://genodians.org/chelmuth/2024-05-17-on-target-debugging

### 4.5.3 Performance analysis

For an introduction to pragmatic performance analysis and tracing, please refer to these articles at genodians.org.

**Performance analysis made easy**

https://genodians.org/nfeske/2021-04-07-performance

**Identifying network-throughput bottlenecks with trace recording**

https://genodians.org/jschlatow/2022-08-29-trace-recorder

In addition, Goa integrates support for a pragmatic function profiling library via the `goa backtrace` command. A usage example is available in the *examples/profiling* directory of the Goa repository. For further details, please consult the README[1] file of the *profile* library.

[1] https://codeberg.org/genodelabs/genode/src/branch/main/repos/os/src/lib/profile/README

## 5 Tutorials

This section provides a collection of tutorials that focus on certain aspects during application development for Genode. As preparatory steps, make sure you have the latest Genode toolchain and Goa installed (see Section2).

The following listing serves as an orientation on what you will learn in the individual tutorials.

**Section 5.1** Sticking together a little Unix

- Run `bash` in a graphical terminal.
- Configuring the VFS.
- Split complex scenarios into multiple configuration files by using the `hid` tool.
- Use the `ttf` VFS plugin for providing font glyphs and metadata via a pseudo file system.
- Make use of the `pipe` VFS plugin.
- Run `vim` inside the graphical terminal.

**Section 5.2** Exporting and publishing

- Create a PGP keypair using GnuPG or Sequoia PGP.
- Export a Goa project into a Genode depot.
- Sign and publish the exported depot archive.
- Use the `goa published` and `goa bump-version --if-needed` commands.
- Deploy the published archive on Sculpt OS.
- Publish a depot index of your archives.

**Section 5.3** Writing a VFS plugin for network-packet access

- Write and build a shared library to be used as a VFS plugin.
- Learn about ioctl-emulation in Genode.
- Create a test project.
- Share a depot directory between the main project and the test project.

**Section 5.4** Porting Lomiri Calculator App

- Port a cmake-based application from the Ubuntu UI Toolkit.
- Specify archive versions and the path to a LICENSE file in a *goarc* file.
- Pass additional arguments to CMake.

- Make use of the wildcard depot user "_".
- Run the ported application using the *ubuntu-ui-toolkit-launcher*.
- Port a qmake-based QtQuick style plugin as a shared library.
- Integrate the ported plugin library into the runtime scenario.

**Section 5.5** Porting the curl command-line tool and library

- Port the autoconf-based curl binary and shared library.
- Apply patches to the imported source code.
- Pass additional arguments to `configure`.
- Use version information from a depot user's index.
- Learn about Goa's support for pkg-config
- Specify the path to a LICENSE file in a *goarc* file.
- Import example source code from curl into a separate project.
- Add a .pc file to the exported archive for pkg-config support.

## 5.1 Sticking together a little Unix

*This section is based on Norman Feske's article series[1] at https://genodians.org.*

This tutorial takes you on a ride of creating a small Unix OS out of Genode's ready-to-use building blocks. It shows the fun and productive way of crafting component compositions out of Genode's readily available building blocks. What could be a better example than building an old-school operating system - Unix - that we all know and love? You can find the results of this tutorial in Norman's Goa-projects repository[2].

**Preparations** Before continuing, please make sure to have installed the Goa tool, which is available at https://codeberg.org/genodelabs/goa. If you have it installed already, please make sure the tool is up to date. You can issue the following command to update Goa to the latest version:

```
$ goa update-goa 26.04
```

**Hello bash** As the first step, we want to get a life sign of the bash shell. We start with a new Goa project appropriately named `unix` that hosts a runtime package but no source code.

```
$ mkdir -p unix/pkg
$ cd unix
```

Let's pretend we don't know what we are doing and create an *archives* file with only bash listed, and an almost empty *runtime* file. The runtime starts the binary `init`, which is supposed to be a ROM module. Please have a look at `goa help runtime` for more details on how to write runtime files.

The *pkg/archives* file:

```
genodelabs/src/bash
```

The *pkg/runtime* file:

```
runtime | ram: 100M | caps: 5000 | binary: init
+ content
  + rom init
-
```

Let's see what happens when issuing the `run` command:

[1] https://genodians.org/nfeske/2019-12-13-goa-unix-bash
[2] https://codeberg.org/nfeske/goa-projects

```
unix$ goa run
download genodelabs/bin/x86_64/bash/2026-03-11.tar.xz
download genodelabs/bin/x86_64/bash/2026-03-11.tar.xz.sig
download genodelabs/src/bash/2026-03-11.tar.xz
download genodelabs/src/bash/2026-03-11.tar.xz.sig
download genodelabs/api/libc/2026-03-11.tar.xz
download genodelabs/api/libc/2026-03-11.tar.xz.sig
download genodelabs/api/noux/2025-10-16.tar.xz
download genodelabs/api/noux/2025-10-16.tar.xz.sig
download genodelabs/api/posix/2020-05-17.tar.xz
download genodelabs/api/posix/2020-05-17.tar.xz.sig
[unix] Error: runtime lacks a configuration

 You may declare a 'config' attribute for the 'runtime' node, or
 define a 'config' node inside the 'runtime' node.
```

Let's follow the advice by adding an empty `config` node to our *pkg/runtime* file:

```
runtime | ram: 100M | caps: 5000 | binary: init
+ config
+ content
  + rom init
-
```

Besides the error message, you could see that Goa automatically downloaded bash along with its dependencies such as the libc. Besides the binaries, it also fetches all source codes. You can find all the downloads at *var/depot/*. One particularly interesting directory is the binary archive for bash:

```
unix$ ls var/depot/genodelabs/bin/x86_64/bash/2026-03-11/

bash.tar
```

It contains a single tar archive, which in turn, contains all installation files of bash. Let's take a look inside:

```
unix$ tar tf var/depot/genodelabs/bin/x86_64/bash/2026-03-11/bash.tar
./
./share
./share/doc
...
./bin/bashbug
./bin/bash
```

Of course, the most interesting bit is the `bash` executable at *bin/bash*. When using the binary archive, the whole *bash.tar* is supplemented to Genode as a single ROM module. Let's add it to the `content` of the *pkg/runtime*:

```
runtime | ram: 100M | caps: 5000 | binary: init
+ config
+ content
  + rom init
  + rom bash.tar
-
```

After issuing `goa run` again, Goa downloads the additional packages needed to run our *pkg/runtime* on Linux, integrates the scenario, and starts it.

```
unix$ goa run
download genodelabs/bin/x86_64/base-linux/2026-04-20.tar.xz
download genodelabs/bin/x86_64/base-linux/2026-04-20.tar.xz.sig
download genodelabs/bin/x86_64/init/2026-04-29.tar.xz
download genodelabs/bin/x86_64/init/2026-04-29.tar.xz.sig
download genodelabs/src/base-linux/2026-04-20.tar.xz
download genodelabs/src/base-linux/2026-04-20.tar.xz.sig
download genodelabs/src/init/2026-04-29.tar.xz
download genodelabs/src/init/2026-04-29.tar.xz.sig
download genodelabs/api/base/2026-04-16.tar.xz
download genodelabs/api/base/2026-04-16.tar.xz.sig
download genodelabs/api/os/2026-04-16.tar.xz
download genodelabs/api/os/2026-04-16.tar.xz.sig
download genodelabs/api/report_session/2024-08-28.tar.xz
download genodelabs/api/report_session/2024-08-28.tar.xz.sig
download genodelabs/api/sandbox/2025-12-11.tar.xz
download genodelabs/api/sandbox/2025-12-11.tar.xz.sig
download genodelabs/api/timer_session/2025-10-12.tar.xz
download genodelabs/api/timer_session/2025-10-12.tar.xz.sig
Genode 26.02-309-g6ae975a32c8
17592186044415 MiB RAM and 19000 caps assigned to init
```

You can find *bash.tar* added to the *var/run/* directory, which comprises all the ROM modules of our Genode system.

Of course, we cannot start a TAR archive. It is not an executable after all. We rather need to access the content of the archive. Here, the combination of three Genode components namely VFS, `fs_rom`, and init come to the rescue.

1. The VFS server is able to mount a TAR archive locally as a virtual file system and offer its content as a *file-system service*.

2. The `fs_rom` component provides a ROM service by fetching the content of ROM modules from a file system. By connecting the `fs_rom` with the VFS component, the files of the *bash.tar* archives become available as ROM modules. With the bash executable binary accessible, we can execute it.

3. The init component allows us to stick components together and let the result appear to the surrounding system as a single component. We can use it to host the composition of the VFS, `fs_rom`, and bash.

Note that our *pkg/runtime* refers to Genode's init component in the attribute `binary: init`. So as a whole, our subsystem will be an instance of init. Internally, init will host several child components and manage their resources and relationships according to its configuration. Let's start with a fresh init that hosts only the VFS server by replacing our empty `config` in our *pkg/runtime* file by the following configuration.

```
+ config
  + parent-provides
    + service ROM
    + service LOG
    + service RM
    + service CPU
    + service PD
    + service Timer
  + start vfs | caps: 100 | ram: 10M
    + provides
    | + service File_system
    + config
    | + vfs
    |   + tar bash.tar
    | + default-policy | root: /
    + route
      + any-service
        + parent
```

The `default-policy` expresses that any client should be able to access the root of the virtual file system in a read-only fashion.

When trying to `run` the scenario now, you see a bunch of messages:

```
unix$ goa run
[unix] config 'parent-provides' mentions a timer service;
       consider adding 'timer' as a required runtime service
Genode 26.02-309-g6ae975a32c8
17592186044415 MiB RAM and 19000 caps assigned to init
[init -> unix] Error: vfs: environment ROM session denied (label="vfs", ...)
```

The first message points out that init's `parent-provides` declaration refers to a service that should better also be announced as a requirement in the *runtime* file. This can be done by adding the following `requires` node inside the `runtime` node.

```
runtime
...
+ requires
  + timer
...
-
```

The subsequent "Error:" messages tell us that init requested the ROM module `vfs` that is not available to the scenario, yet. To make this ingredient available to our scenario, we have to declare it in the *archives* and as `content` in the *runtime* file. While we are at it, let's also capture the need for `init` because our entire scenario is based on this component. Let's add the following lines to *pkg/archives*:

```
genodelabs/src/vfs
genodelabs/src/init
```

Also make sure to have the ROM modules listed as `content` in the *pkg/runtime* so that it looks as follows:

```
+ content
  + rom init
  + rom bash.tar
  + rom vfs
```

When issuing `goa run` again, we can see Goa downloading the additional components. On the attempt to start the scenario, we are confronted with another error message:

```
[init -> unix -> vfs] Error: stop because parent denied ROM-session:
                      label="vfs.lib.so", , ram_quota=10K, cap_quota=3
```

This message tells us that the VFS server requests another ROM module, which is a shared library. The *vfs.lib.so* contains the actual implementation of the virtual file system. It comes in the form of a library to enable its use either locally by an individual application or via the VFS server. The library is part of the `genodelabs/src/vfs` archive that is already listed in our *pkg/archives* file. So we can resolve this error by adding a corresponding `rom` entry to the *pkg/runtime* file. The `content` should now look as follows:

```
+ content
  + rom init
  + rom bash.tar
  + rom vfs
  + rom vfs.lib.so
```

When running the scenario again, we see a sign of hope:

```
unix$ goa run
Genode 26.02-309-g6ae975a32c8
17592186044415 MiB RAM and 19000 caps assigned to init
```

No further errors! That means that the VFS server is running and has presumably mounted the *bash.tar* archive. On a second terminal, you can indeed observe the VFS server showing up.

```
$ ps u
... [Genode] init
... [Genode] init -> timer
... [Genode] init -> unix
... [Genode] init -> unix -> vfs
```

The second piece of the puzzle is the `fs_rom` server, which can be added to the `config` node of *pkg/runtime* with the following snippet:

```
+ start vfs_rom | caps: 100 | ram: 10M
  + binary fs_rom
  + provides | + service ROM
  + config
  + route
    + service File_system | + child vfs
    + any-service         | + parent
```

By using the `binary` node, we can label the component in a meaningful way, calling it "vfs_rom". The first entry of the `route` node defines that the request for a file-system session should be routed to the "vfs" component.

On the next attempt to issue `goa run`, we face an error message:

```
[init -> unix] Error: vfs_rom: environment ROM session denied (...)
```

By now, I'm sure you know how to resolve this one. Corresponding entries to your *pkg/archives* file and the *pkg/runtime* file's `content` are added swiftly. The fs_rom component gives us no life sign, which is normal. If you want to get a little bit more action

on screen, you may add the `verbose: yes` attribute to init's `config` node. Another try of `goa run` reveals the following output.

```
unix$ goa run
Genode 26.02-309-g6ae975a32c8
17592186044415 MiB RAM and 19000 caps assigned to init
[init -> unix] parent provides
[init -> unix]   service "ROM"
[init -> unix]   service "LOG"
[init -> unix]   service "RM"
[init -> unix]   service "CPU"
[init -> unix]   service "PD"
[init -> unix]   service "Timer"
[init -> unix] child "vfs"
[init -> unix]   RAM quota:  9992K
[init -> unix]   cap quota:  66
[init -> unix]   ELF binary: vfs
[init -> unix]   priority:   0
[init -> unix]   provides service File_system
[init -> unix] child "vfs_rom"
[init -> unix]   RAM quota:  9992K
[init -> unix]   cap quota:  66
[init -> unix]   ELF binary: fs_rom
[init -> unix]   priority:   0
[init -> unix]   provides service ROM
[init -> unix] child "vfs" announces service "File_system"
[init -> unix] child "vfs_rom" announces service "ROM"
```

That looks promising. Now with the bash executable available as ROM module, let's give the bash shell a spin:

```
+ start /bin/bash | caps: 1000 | ram: 10M
  + config
  | + libc | stdin:  /dev/null
  |        | stdout: /dev/log
  |        | stderr: /dev/log
  |        | rtc:    /dev/null
  | + vfs
  |   + dir dev
  |     + null
  |     + log
  | + arg | : bash
  | + arg | : -c
  | + arg | : echo files at /dev: /dev/*
  + route
```

```
    + service ROM | label_last: /bin/bash | + child vfs_rom
    + any-service                         | + parent
```

The following parts are worth highlighting:

- The bash has its own VFS! This has nothing to do with the VFS server we started above. In fact, bash's VFS - as configured by the `vfs` node - merely contains the two pseudo files */dev/null* and */dev/log*. The latter one is a LOG connection that enables the bash to write messages to the outside world.
- The `libc` node contains the configuration of the C runtime used by bash. Here we say how the standard output should go, or that the C runtime should obtain its "real-time-clock" information from */dev/null*. No time for you this time!
- Via the sequence of `arg` nodes, we execute the command

  ```
  echo files at /dev: /dev/*
  ```

  It uses the shell's file-globbing mechanism to obtain the list of files matching the pattern "/dev/*" and prints it via the `echo` built-in command.
- The `route` rules explicitly tell init that the binary of the component should be obtained from the "vfs_rom" component.

When trying to `goa run` the scenario now, we have to add a few more entries to our *pkg/archives* and `content`, specifically because bash uses the C runtime (libc and libm) as well as the posix library. The full list of *archives* now looks as follows:

```
genodelabs/src/bash
genodelabs/src/vfs
genodelabs/src/init
genodelabs/src/fs_rom
genodelabs/src/libc
genodelabs/src/posix
```

For reference, the `rom` modules listed in the *pkg/runtime* file's `content` node:

```
+ content
  + rom init
  + rom bash.tar
  + rom vfs
  + rom vfs.lib.so
  + rom fs_rom
  + rom libc.lib.so
  + rom libm.lib.so
```

```
  + rom posix.lib.so
```

Once these stumbling blocks are out of the way, `goa run` greets us with the following output:

```
...
[init -> unix] child "vfs" announces service "File_system"
[init -> unix] child "vfs_rom" announces service "ROM"
[init -> unix -> /bin/bash] files at /dev: /dev/log /dev/null
[init -> unix] child "/bin/bash" exited with exit value 0
```

The message "files at /dev: /dev/log /dev/null" is the output of the bash command we have hoped for!

**Some reorg is in order** The scenario we just built was quite small. For such small scenarios, defining the `config` node right in the *runtime* file is quite handy. Once the subsystem becomes bigger, however, it's better to move the `config` into a dedicated ROM module. Let us create a new directory named *raw/* inside the project directory, and move the `config` node from the *runtime* file to a new file *raw/unix.config*. Goa will pick up all files contained in the *raw/* directory and supply them as ROM modules to the Genode scenario.

For moving the `config` node, you may use the `hid` tool, which you'll find at *share/goa/hid* in the Goa repository:

```
unix$ hid subnodes 'runtime | + config' pkg/runtime > raw/unix.config
unix$ hid -i remove 'runtime | + config' pkg/runtime
```

Since there is no longer a `config` provided in the *runtime* file, we tell the runtime to use the "unix.config" as configuration by changing the `runtime` node as follows:

```
runtime | ram: 100M | caps: 5000 | binary: init | config: unix.config
```

Since *unix.config* is expected to be present as a ROM module, we have to declare it via a `rom` node in the *runtime* file. For reference, the *pkg/unix/runtime* file should now look as follows:

```
runtime | ram: 100M | caps: 5000 | binary: init | config: unix.config
+ requires | + timer
+ content
  + rom init
  + rom bash.tar
  + rom vfs
  + rom vfs.lib.so
```

```
  + rom fs_rom
  + rom libc.lib.so
  + rom libm.lib.so
  + rom posix.lib.so
  + rom unix.config
-
```

The *raw/unix.config* file:

```
config | verbose: yes
+ parent-provides
  + service ROM
  + service LOG
  + service RM
  + service CPU
  + service PD
  + service Timer
+ start vfs | caps: 100 | ram: 10M
  + provides
  | + service File_system
  + config
  | + vfs
  |   + tar bash.tar
  | + default-policy | root: /
  + route
    + any-service
      + parent
+ start vfs_rom | caps: 100 | ram: 10M
  + binary fs_rom
  + provides | + service ROM
  + config
  + route
    + service File_system | + child vfs
    + any-service         | + parent
+ start /bin/bash | caps: 1000 | ram: 10M
  + config
  | + libc | stdin:  /dev/null
  |        | stdout: /dev/log
  |        | stderr: /dev/log
  |        | rtc:    /dev/null
  | + vfs
  |   + dir dev
  |     + null
  |     + log
  | + arg | : bash
  | + arg | : -c
```

```
  | + arg | : echo files at /dev: /dev/*
  + route
    + service ROM | label_last: /bin/bash | + child vfs_rom
    + any-service                         | + parent
-
```

This reorganization has two advantages. First, we save one indentation level for the `config` node. Second, by separating the *unix.config* from the *runtime* in the form of a dedicated ROM module, we can later reuse the same ROM module for other *runtime* files. It is always good to have reusable building blocks.

You may give the new version a try by issuing `goa run`. The output should look familiar.

**GUI stack** Goa supports interactive system scenarios by looking at the requirements stated in the *runtime* file. Right now, the runtime file merely states the amount of RAM and caps as a requirement. We can add the presence of a GUI service as an additional requirement by adding a `gui` node inside the `runtime` node:

```
+ requires
  + gui
  + timer
```

When Goa processes the `goa run` command, it evaluates this information. The `gui` node tells Goa that the scenario will request a session to a GUI server. When running the scenario on Linux, Goa will automatically integrate the components needed for such a GUI server. This includes a pseudo graphics driver, a pseudo input driver, and the nitpicker GUI server[1].

Let's try `goa run` after having added the `requires` definition to our *runtime*. Goa responds with the following message:

```
[unix] Error: runtime requires 'gui',
              which is not mentioned in 'parent-provides'
```

It points out the fact that the runtime file pretends to require a `gui` service but according to init configuration in *unix.config* no such service is actually obtained from the parent. So either the `requires` definition is superfluous or the init configuration is wrong or incomplete. To satisfy this sanity check, let's add the following line to the `parent-provides` declarations in the *raw/unix.config* file.

[1] https://codeberg.org/genodelabs/genode/src/branch/main/repos/os/src/server/nitpicker

```
+ parent-provides
  + service Gui
...
```

Upon the next `goa run`, we can see that Goa automatically downloads the basic components of the GUI stack. Not only that. When starting the scenario, a new window with a blue-ish screen pops up. When hovering the mouse over the window, you can see a small mouse pointer. If you are curious how the GUI stack is assembled in detail, please have a look at *var/run/config*. Yet, from the perspective of our Unix scenario, these exact details are not of interest. The only important point is that our scenario is now officially able to request a "Gui" and a "Timer" service from the underlying system.

With these preconditions in place, we can start a graphical terminal in our *unix.config* by adding the following `start` node:

```
+ start terminal | caps: 110 | ram: 10M
  + provides | + service Terminal
  + route
    + service ROM | label: config | + parent | label: terminal.config
    + any-service                 | + parent
```

The "terminal"[1] uses a GUI service to create a graphical terminal and provides the textual input and output in the form of a "Terminal" service. In the routing rules of the terminal, you can see that the terminal's configuration is fetched from a dedicated ROM module called "terminal.config". We have no such ROM module defined yet. However, let's still give it a try:

```
[init -> unix] Error: terminal: environment ROM session denied
               (label="terminal" ...)
...
```

That's not surprising as we have not added `terminal` to our *archives* nor have we stated the `rom` modules in the runtime file's `content`. Let's do this now. While we are at it, let's also add a `rom` node for the "terminal.config" ROM.

The following line must be added to *pkg/archives*

```
genodelabs/src/terminal
```

The following two lines must be added to the runtime file's `content`:

[1] https://codeberg.org/genodelabs/genode/src/branch/main/repos/gems/src/server/terminal

```
+ content
  ...
  + rom terminal
  + rom terminal.config
```

When trying `goa run` again, we see that we exchanged the previous errors with a new one. Let's call it progress:

```
[unix] Error: Unable to find content ROM module 'terminal.config'.

 You either need to add it to the 'raw/' directory
 or add the corresponding dependency to the 'archives' file.
```

The error is easy to explain. We have configured the "terminal" start node to fetch its configuration from a ROM called *terminal.config* but have not defined the ROM module so far. Let's add a new file *raw/terminal.config* with an empty `config` node:

```
config
-
```

With the file added, our next call of `goa run` is answered as follows.

```
[init -> unix -> terminal] Error: VFS not configured
[init -> unix -> terminal] Error: failed to watch
                           '/font/monospace/regular/glyphs'
[init -> unix -> terminal] Error: Uncaught exception of type
                           'Genode::Directory::Nonexistent_directory'
[init -> unix -> terminal] Warning: abort called - thread: ep
```

Well, the terminal seems underwhelmed by us serving an empty `config` as configuration. It is time to become more specific. Let's change the content of the *raw/terminal.config* to something meaningful:

```
config
+ vfs
  + rom VeraMono.ttf
  + dir font
    + dir monospace
      + ttf regular | path: /VeraMono.ttf | size_px: 16
-
```

Wait a minute. How is this a terminal configuration?

The terminal expects its font to be found at its local VFS at */font/monospace*. The font has the form of a pseudo file system that provides the pixel data of the glyphs along

with the font meta data as a bunch of pseudo files. So here, we mount a TrueType font with the `ttf` file-system driver at */font/monospace*. The font file is specified as `path` attribute, which refers to */VeraMono.ttf*". This file, in turn, is backed by a `rom` session that requests the ROM module named "VeraMono.ttf".

With this configuration in place, the next attempt of `goa run` yields a quite predictable result:

```
[init -> unix -> terminal] Error: stop because parent denied ROM-session:
                           label="VeraMono.ttf", , ram_quota=10K, [...]
```

The terminal configuration refers to a ROM module that we haven't yet included into the scenario. The "VeraMono.ttf" is the TrueType font data we tried to mount as `rom` node. The error can be resolved by extending the *archives* file and the *runtime* file's `content` node accordingly:

The following line must be added to *pkg/archives*

```
genodelabs/raw/ttf-bitstream-vera-minimal
```

The following line must be added to the `content` node in *pkg/runtime*

```
+ content
  ...
  + rom VeraMono.ttf
```

Let's have another try with `goa run`:

```
[init -> unix -> terminal] Error: stop because parent denied ROM-session:
                           label="vfs_ttf.lib.so", , ram_quota=10K, [...]
```

Once again, the configuration is missing a ROM module. The "vfs_ttf.lib.so" is the driver for the "ttf" pseudo file system. It is requested by the VFS when the `ttf` node is encountered. As before, this is resolved by extending the *archives* and *runtime* files.

The following lines must be added to *pkg/archives*

```
genodelabs/src/vfs_ttf
```

The following lines must be added to the `content` node in *pkg/runtime*

```
+ content
  ...
  + rom vfs_ttf.lib.so
```

Good news! On the next try of `goa run`, you can see the error gone and are greeted with a black-ish screen instead. The log output of `/bin/bash` looks as usual.

**Connecting bash with the terminal** With the current scenario, bash and the GUI stack are running peacefully side by side, but they do not interact with each other. To connect them, we do the following:

1. Mount a terminal session to the VFS of the VFS server at /dev/terminal.

   This can be done by changing the content of the `start` node of the VFS server. As a reminder, this is how it looks so far:

   ```
   + start vfs | caps: 100 | ram: 10M
     + provides | + service File_system
     + config
     | + vfs | + tar bash.tar
     | + default-policy | root: /
     + route | + any-service | + parent
   ```

   We change it to the following:

   ```
   + start vfs | caps: 100 | ram: 10M
     + provides | + service File_system
     + config
     | + vfs
     |   + tar bash.tar
     |   + dir dev | + terminal
     | + default-policy | root: /
     | + policy | label_prefix: /bin/bash | root: / | writeable: yes
     + route
       + service Terminal | + child terminal
       + any-service      | + parent
   ```

   The `vfs` node gained the configuration of */dev/terminal*. When the VFS encounters the `terminal` node upon initialization, it will request a session to a "Terminal" service. The added route tells init to route the terminal session towards the "terminal" component. The added `policy` node defines that a file-system client labeled as "/bin/bash" is allowed to access the entirety of the VFS in a writeable fashion.

2. Mount the file system as provided by the VFS server into the VFS of the bash shell. This way, all files provided by the VFS server become visible in the file name space of bash. This can be done by extending the `vfs` of bash by adding an `fs` node:

   ```
   + vfs
     + dir dev
       + null
   ```

```
    + log
  + fs
```

When the VFS of bash encounters the `fs` node, it will request a session to a "File_system" service. To let this request reach the VFS server, we have to add a new entry to the `route` definition.

```
+ route
  + service File_system | + child vfs
```

To have a visible effect, let's redirect the output of the "echo" command executed by bash to the pseudo file */dev/terminal*. Change the bash argument to the following (appending the "> /dev/terminal" and adding a while loop):

```
+ arg | : while true; do echo files at /dev: /dev/* > /dev/terminal; done
```

Note that executing the `echo` command in an infinite loop circumvents startup effects of the terminal component, which discards inputs as long as the GUI window has not been initialized.

Upon the next attempt of `goa run`, magic happens:


**Figure 2**

We have just redirected the output of the bash command to our terminal, which used our TrueType pseudo-file-system driver to render glyphs on a pixel buffer that, in turn, was blitted by the nitpicker GUI server to screen. Could our day become any better? Sure! How about interacting with bash directly?

Change the `libc` configuration of bash to the following:

```
+ libc | stdin:  /dev/terminal | stdout: /dev/terminal
       | stderr: /dev/terminal | rtc:    /dev/null
```

This change wires up the standard input and output of bash with */dev/terminal*. Let's also drop the `-c` arguments from the bash `config` so that bash will wait for a command typed in via stdin. The next `goa run` will greet us with a shell prompt where we can type in bash commands like echo:

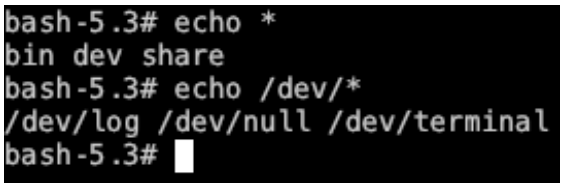


**Figure 3**

Of course, we feel a sudden urge to also execute the `ls` command.


**Figure 4**

The `ls` command is a separate Unix command that is not yet part of our scenario. It is covered by the following section.

**Adding GNU coreutils** The `ls` command - along with most others we commonly associate with Unix - are actually little programs that are spawned by the shell each time when used. When typing `ls`, bash doesn't actually know the purpose of `ls`. It merely looks up a program named `ls` and executes it. The program `ls`, in turn, has the single purpose of printing directory contents. When executed, it takes a look at the file system, prints the gathered information, and exits. The `ls` command together with its friends `cp`, `mkdir`, `sort`, and many others are the Unix core utilities. On a regular GNU/Linux system, they are provided by the GNU coreutils[1] package.

The GNU coreutils package is readily available for Genode. We can add it by appending the following line to our *pkg/archives* file:

```
genodelabs/src/coreutils
```

After adding this line, the next invocation of `goa run` will download the source code along with a ready-to-use binaries to *var/depot/*. In particular, you can find the binary at *var/depot/genodelabs/bin/x86_64/coreutils/<version>/*. Analogous to the bash package, described in the beginning of this section, there is a single TAR archive containing all the files that comprise the coreutils installation.

```
unix$ tar tf var/depot/genodelabs/bin/x86_64/coreutils/2026-03-11/coreutils.tar
./
./lib/
...
./share/
...
./bin/
./bin/uname
./bin/groups
./bin/dircolors
```

[1] https://www.gnu.org/software/coreutils/coreutils.html

```
./bin/chcon
./bin/nproc
./bin/true
./bin/mv
...
```

We follow the same pattern as previously used for integrating the *bash.tar* archive.

1. Declaring the use of `coreutils.tar` as ROM module in the *pkg/runtime* file's `content` node:

   ```
   + content
     ...
     + rom coreutils.tar
   ```

2. Mounting the `coreutils.tar` as file system into the VFS of the VFS server. The VFS server's `vfs` should now look as follows:

   ```
   + vfs
     + tar bash.tar
     + tar coreutils.tar
     + dir dev | + terminal
   ```

   As you can see, the VFS supports the mounting any number of file systems side by side as overlays, which is commonly known as union mounting[1].

Remember from the end of the previous section that our attempt to issue `ls` resulted in the following message:


**Figure 5**

Let's give `goa run` another go now.

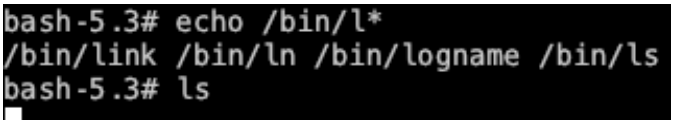



**Figure 6**

[1] https://en.wikipedia.org/wiki/Union_mount

Unlike before, bash has actually found the `ls` binary on the file system. We mounted `coreutils.tar` into the VFS after all, which you can easily reaffirm via `echo /bin/l*`. However, bash still failed to spawn the `ls` program. Genode's log output reveals why:

```
[init -> unix -> /bin/bash -> 1] Error: stop because parent denied ROM-session:
                                 label="/bin/ls", , ram_quota=10K, [...]
```

Remember that we have to make a program's binary available as ROM module in order to execute it. We have accomplished this via the `fs_rom` server handing out file-system content as ROM modules, and directing bash's request for the "/bin/bash" ROM module to `fs_rom`. To recap, we defined the `route` rules for `bash` as follows:

```
+ route
  + service File_system                 | + child vfs
  + service ROM | label_last: /bin/bash | + child vfs_rom
  + any-service                         | + parent
```

There is no valid route for a ROM service and the label "/bin/ls" yet. In principle, we could follow the pattern of the "/bin/bash" ROM. On the other hand, with many binaries installed at */bin/*, the approach would become cumbersome. A better solution is adding a route that matches the label prefix "/bin". Changing the `route` of "/bin/bash" as follows does the trick (pay attention to the third `service` node).

```
+ route
  + service File_system                 | + child vfs
  + service ROM | label_last: /bin/bash | + child vfs_rom
  + service ROM | label_prefix: /bin/   | + child vfs_rom
  + any-service                         | + parent
```

In the following, we don't want to refer to the Unix commands using their full paths but by their names. So let us set the `PATH` environment variable in the `config` of bash's `start` node.

```
+ config
  ...
  + env PATH | : /bin
```

The next try of `goa run` yields the following result:

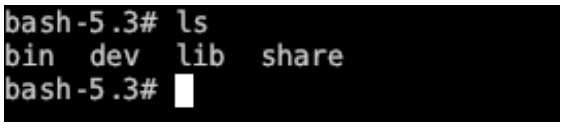



**Figure 7**

A look at */bin/* reveals the wealth of commands that have just become available at our finger tips.

```
bash-5.3# ls bin/
'['         chmod     df          false     link      nice      printenv  sha1sum   stat      tr        users
 b2sum      chown     dir         fmt       ln        nl        printf    sha224sum stty      true      vdir
 base32     chroot    dircolors   fold      logname   nohup     ptx       sha256sum sum       truncate  wc
 base64     cksum     dirname     groups    ls        nproc     pwd       sha384sum sync      tsort     who
 basename   comm      du          head      md5sum    numfmt    readlink  sha512sum tac       tty       whoami
 bash       cp        echo        hostid    mkdir     od        realpath  shred     tail      uname     yes
 bashbug    csplit    env         id        mkfifo    paste     rm        shuf      tee       unexpand
 cat        cut       expand      install   mknod     pathchk   rmdir     sleep     test      uniq
 chcon      date      expr        join      mktemp    pinky     runcon    sort      timeout   unlink
 chgrp      dd        factor      kill      mv        pr        seq       split     touch     uptime
bash-5.3#
```

**Figure 8**

**Plumbing pipes** Let us try to count'em via the `wc -l` command (`wc -l` counts the number of lines).

```
bash-5.3# ls -l | wc -l
bash: pipe error: No error: 0
bash-5.3#
```

**Figure 9**

With our attempt of using a pipe, feeding the output of `ls -1` via the `|` symbol as input into `wc -l`, we seem to hit another brick wall. But that one isn't too bad. Until now, we haven't yet configured the C runtime of "/bin/bash" (and its child processes) for the use of a pipe mechanism. We can do so by adding a `pipe` attribute to the `libc` node:

```
+ libc | stdin:  /dev/terminal | stdout: /dev/terminal
       | stderr: /dev/terminal | rtc:    /dev/null
       | pipe:   /dev/pipe
```

But */dev/pipe* does not exist, you ask! Thanks for paying attention. On traditional Unix systems, the pipe mechanism is provided by the kernel. On Genode, we provide it via a pseudo file system that is shared by both ends of the pipe. The path */dev/pipe/* is the location of this pseudo file system. To make it easily available to all Unix processes, we have to mount it into the VFS of the VFS server. As a reminder, the `vfs` node of the VFS server currently looks as follows.

```
+ vfs
  + tar bash.tar
  + tar coreutils.tar
  + dir dev | + terminal
```

With the addition of the pipe pseudo file system, we change the `+ dir dev` node into this:

```
+ vfs
  + tar bash.tar
  + tar coreutils.tar
  + dir dev
    + terminal
    + dir pipe | + pipe
```

As usual after making such changes, the repeated use of `goa run` guides us forward:

```
[init -> unix -> vfs] Error: stop because parent denied ROM-session:
                      label="vfs_pipe.lib.so", , ram_quota=10K, [...]
```

I'm sure, you guess what comes next. Let's enhance *pkg/archives* with the following line:

```
genodelabs/src/vfs_pipe
```

Also declare the "vfs_pipe.lib.so" ROM in our *pkg/runtime* file:

```
+ content
  ...
  + rom vfs_pipe.lib.so
```

With these minor tweaks in place, `goa run` starts up successfully again. This time, our attempt to combine `ls` with `wc` works as intended!

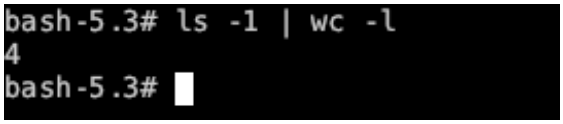



**Figure 10**

**Life is not complete without Vim** To wrap up the Unix experience, let's add the Vim text editor to the scenario. The process is rather straight forward and follows exactly the pattern of the addition of coreutils. That is

1. Add vim to *pkg/archives*

   ```
   genodelabs/src/vim
   ```

2. Add the "vim.tar: ROM to *pkg/runtime*

   ```
   + rom vim.tar
   ```

3. Mount "vim.tar" at the VFS server

```
+ tar vim.tar
```

Another try of `goa run` downloads the needed depot content and starts the scenario. The attempt to start `vim` results in an error message in the Genode log:

```
[init -> ...] Error: stop because parent denied ROM-session:
              label="ncurses.lib.so", , ram_quota=10K, cap_quota=3
```

Vim is the first Unix program that requires ncurses[1], which is a library for interactive terminal applications. To make it available to our system, add `genodelabs/src/ncurses` to *pkg/archives* and `+ rom ncurses.lib.so` to *pkg/runtime*.

The next test run looks much better. Vim starts up successfully but is not entirely happy:

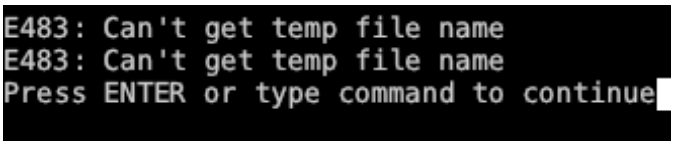



**Figure 11**

Vim relies on the presence of a */tmp/* directory. We can satisfy it by mounting a memory-backed `ram` file system in our VFS server by adding the following line to its `vfs` configuration:

```
+ dir tmp | + ram
```

Upon the next test run, we are greeted with another error message:

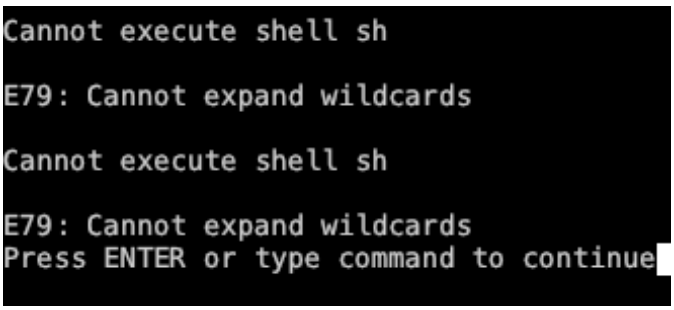



**Figure 12**

[1] https://en.wikipedia.org/wiki/Ncurses

For some tasks like file globbing, Vim spawns a shell as child process and expects the shell being available as */bin/sh*. This default can be overridden via the `SHELL` environment variable. We can set the `SHELL` environment variable to the value "bash" by adding the following line to `config` of the "/bin/bash" `start` node:

```
+ env SHELL | : bash
```

Furthermore, we can tame Vim by overriding its default configuration.
Create a file *raw/vimrc* with the following content:

```
set noloadplugins
set hls
set nocompatible
set laststatus=2
set noswapfile
set viminfo=
```

Add a `+ rom vimrc` node to the `content` of *pkg/runtime*.
Mount the "vimrc" ROM as */share/vim/vimrc* file at the VFS server:

```
+ dir share | + dir vim | + rom vimrc
```

Finally, we can make ncurses aware of the actual terminal protocol implemented by Genode's graphical terminal by setting the environment variable `TERM`. This enables the use of colors in Vim. Add the following line to the `config` of the "/bin/bash" `start` node:

```
+ key TERM | : screen
```

With these changes, we are greeted with the following screen when starting `vim` from the bash shell in our little Unix environment:

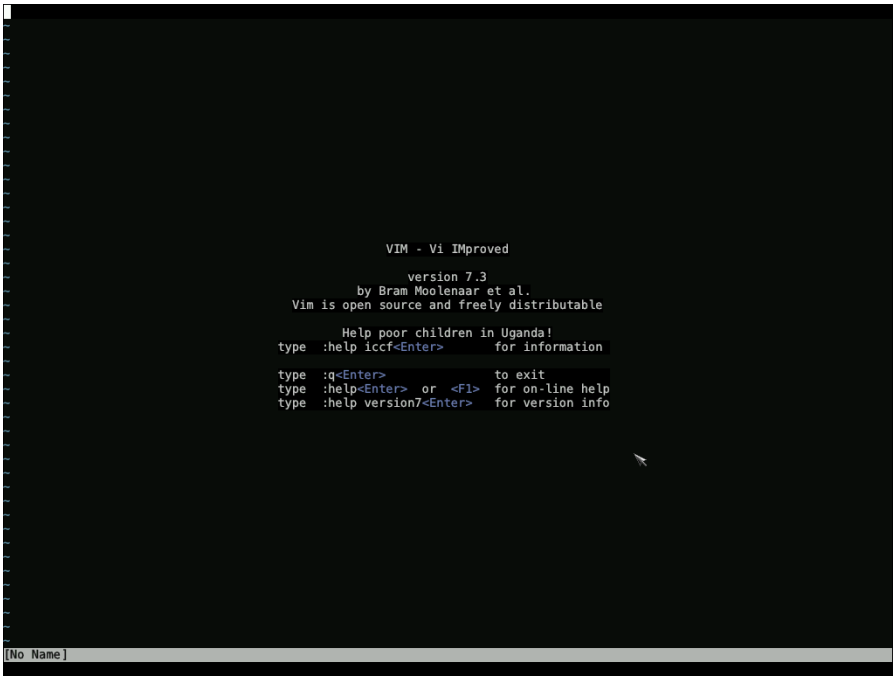



**Figure 13**

We have just crafted a little Unix out of Genode's generic building blocks. The result allows us to work with the time-tested and loved Unix core utilities, combine them with pipes, and edit files with the full comfort of Vim. All that has become possible with less than 100 lines of HID:

```
$ wc -l raw/unix.config raw/terminal.config pkg/runtime
 52 raw/unix.config
  8 raw/terminal.config
 39 pkg/runtime
 99 total
```

## 5.2 Exporting and publishing

*This section is based on Norman Feske's article Goa - publishing packets[1] at https://genodians.org.*

Let's follow up on the Unix tutorial from Section 5.1 and make the scenario available in form of a ready-to-use depot.

In Norman's Goa-projects repository, you can find the results of the Unix tutorial in the *intro* directory. This section uses the unix_3rd[2] subdirectory as the basis for the steps described below.

### Norman's Goa projects repository

https://codeberg.org/nfeske/goa-projects

**Software-publishing prerequisites** In order to provide packaged software to other Genode users, you will need the following prerequisites:

1. A publicly accessible place on the web where users can download your software packages from.
2. A PGP key pair to protect the end-to-end integrity of your packages.

This article does not cover the first point as there are so many options when it comes to web hosting. However, the use of PGP deserves an explanation.

Genode's depot tools use OpenPGP signatures to ensure that the packages created by you are bit-for-bit identical to the packages arrived at the user's system. It works like this: You as the software provider create an OpenPGP key pair consisting of a private key and a matching public key. The private key must remain your secret. The public key should be made publicly available.

You can use your private key to put *your* digital signature on a package. Nobody else can forge your signature because the private key is known only to you. Once a user has downloaded the package, the signature attached to the package can be tested against the public key. If the package was mutated on the way to the user's machine, e. g., the web server was compromised by an attacker, this check would ultimately fail. The user is saved from the risk of running non-genuine or randomly broken software. Vice versa, if the signature check succeeds, the user can be certain to have obtained a bit-for-bit identical copy of the package created by the owner of the private key - the software provider.

Since you are an aspiring software provider, you ought to have an OpenPGP key pair. Goa integrates support for GnuPG as well as Sequoia PGP.

[1] https://genodians.org/nfeske/2020-01-16-goa-publish
[2] https://codeberg.org/nfeske/goa-projects/src/branch/main/intro/unix_3rd

**Creating a key pair using Sequoia PGP** Sequoia PGP is a modular implementation of the OpenPGP standard in Rust, focusing on security and robustness. If your GNU/Linux distribution ships Sequoia PGP 1.2.0 or later, you may choose to create your key pair with Sequoia PGP instead of GnuPG.

To create a new key pair you can use the following command, inserting your name and email address:

```
$ sq --cli-version 1.2.0 key generate --expiration never --own-key \
  --name [...] --email [...]
Please enter the password to protect key (press enter to not use a password):
                                                    Please repeat the password:
```

You are greeted with a dialog asking for a new passphrase. This passphrase is used to encrypt your private key before storing it in a file. In the event of a leak of this file, your private key remains still a secret unless your passphrase becomes known. Hence, you should better not write down your passphrase but keep it in your head only. Once you supplied your passphrase, Sequoia confirms the creation of the new key pair with a message like this:

```
...
Transferable Secret Key.

      Fingerprint: E169976BFE66C741AC2CEBF286426500B4A49FAE
  Public-key algo: EdDSA
  Public-key size: 256 bits
       Secret key: Encrypted
    Creation time: 2026-05-20 11:52:33 UTC
        Key flags: certification
...
Hint: You can export your certificate as follows:

  $ sq cert export --cert=E169976BFE66C741AC2CEBF286426500B4A49FAE

Hint: Once you are happy you can upload it to public directories using:

  $ sq network keyserver publish --cert=E169976BFE66C741AC2CEBF286426500B4A49FAE
```

**Creating a key pair using GnuPG** GnuPG is the go-to implementation of the OpenPGP standard. It is usually installed by default on GNU/Linux distributions. If you are already using GPG for encrypting/signing email, you may, in principle, use your existing key pair. If so, you may skip this section.

To create a new key pair, you can use the following command:

```
$ gpg --full-generate-key
gpg (GnuPG) 2.2.4; Copyright (C) 2017 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
Your selection?
```

Stick to the default (RSA) by hitting enter. Next, you are asked for the key size.

```
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (3072)
```

GnuPG suggests a default key size of 3072 bits. You can add a safety margin by raising the size to 4096. Next, you are asked to decide for how long you want to use this key.

```
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0)
```

For our use case, there is no point in limiting the key's lifetime. Press enter to let the key never expire.

```
Key does not expire at all
Is this correct? (y/N)
```

The tool apparently wants to have us think twice about it. Well, typing `y` gives it the assurance it desires.

Next, the question about your real name. Well, for the purpose of this tutorial, let's use "John K.".

```
Real name: John K.
```

When asked for the email address, it's technically fine to just fill-in some place holder.

Should you intend to widely publish your public key, e. g., by uploading it to a key server, please consider using your real identity. You want to be trusted by the users of your software after all, don't you? A real identity is certainly more trustworthy than a random internet person hiding behind a pseudonym.

```
Email address: a@b.cd
```

Next, you can leave a comment or leave it blank by pressing enter.

```
Comment:
You selected this USER-ID:
    "John K. <a@b.cd>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit?
```

After pressing `o`, you are greeted with a dialog asking for a new passphrase. This passphrase is used to encrypt your private key before storing it in a file. In the event of a leak of this file, your private key remains still a secret unless your passphrase becomes known. Hence, you should better not write down your passphrase but keep it in your head only. Once you supplied your passphrase, GPG confirms the creation of the new key pair with a message like this:

```
...
public and secret key created and signed.

pub   rsa4096 2020-01-16 [SC]
      96541E89AA71BAA88DF56C538ADB04B1F162AF2D
uid                      John K. <a@b.cd>
sub   rsa4096 2020-01-16 [E]
```

When inspecting the GPG keyring via the command `gpg --list-secret-keys`, you can see the new key listed:

```
$ gpg --list-secret-keys
...
sec   rsa4096 2020-01-16 [SC]
      96541E89AA71BAA88DF56C538ADB04B1F162AF2D
uid           [ultimate] John K. <a@b.cd>
ssb   rsa4096 2020-01-16 [E]
```

**A quick look back at the project we wish to publish** To publish the depot content for a given Goa project, first change to the project directory. For example, within the `goa-projects` repository linked above, you would change to the `unix_3rd` directory.

```
$ git clone https://codeberg.org/nfeske/goa-projects.git
$ cd goa-projects/intro/unix_3rd/
```

Before proceeding, please make sure to use the latest version of the Goa tool.

```
$ goa update-goa 26.04
```

It is always a good idea to give the project a quick try before publishing it.

```
$ goa run
```

Goa will download all the components needed to build the scenario, and execute it directly on the GNU/Linux development machine. You should see a terminal window with a bash prompt.

When peeking at the *var/public/* directory now, you see the downloaded archives and signatures. For example,

```
$ find var/public/genodelabs/bin/x86_64/terminal
var/public/genodelabs/bin/x86_64/terminal
var/public/genodelabs/bin/x86_64/terminal/2026-04-28.tar.xz.sig
var/public/genodelabs/bin/x86_64/terminal/2026-04-28.tar.xz
```

The `sig` file is the signature that was created via the private PGP key of Genode Labs when terminal was originally published. After downloading, Goa verifies the signature using Genode Labs' public key that is provided at *var/depot/genodelabs/pubkey*.

When taking a look at the *var/depot/* directory, you see the depot content extracted from the corresponding `tar.xz` archives.

**Exporting the project to a Genode depot** Genode's package management organizes software in a so-called depot, which is a directory with a special structure explained in Secion 3.5. To create depot content for a project, Goa features the `goa export` command. Let's give it a try without a second thought.

```
$ goa export
[unix_3rd] Error: version for archive _/raw/unix_3rd undefined

 Create a 'version' file in your project directory, or
 define 'set version(_/raw/unix_3rd) <version>' in your goarc file,

 or specify '--version-_/raw/unix_3rd <version>' as argument.
```

Apparently, Goa misses any version information about the project. Indeed, while following the steps of Section 5.1, we did not talk or think about versions at all. Now

it is time to make up our minds about a suitable version identifier. In principle, any character string will do, as long as it does not contain anything fancy like whitespace. It is generally a good practice to just use the current date. You may use the `goa bump-version` command to write the version identifier to the *version* file.

```
$ goa bump-version
```

Let's give `goa export` another try.

```
$ goa export
[unix_3rd] exported [...]/var/depot/_/raw/unix_3rd/2026-05-20
[unix_3rd] Error: missing README file at [...]/unix_3rd/pkg/README
```

This looks like a partial success! When inspecting *var/depot/* now, you can indeed find content that looks pretty familiar.

```
$ ls var/depot/_/raw/unix_3rd/2026-05-20/
terminal.config  unix.config  vimrc
```

However, let's pay attention to the `Error:` part of the message. By convention, each depot package features a README file, and Goa nags us to follow this convention. We have to give in. Create a file at *pkg/README* with content of your choice. The README should contain a short description of the purpose of the package, along with instructions for using it. Note that future versions of Sculpt OS will present README texts nicely formatted to the user. We therefore recommend following the GOSH[1] markup syntax, which is consistently used throughout Genode's documentation.

With the README file in place, let's try again:

```
$ goa export
[unix_3rd] exported [...]/var/depot/_/raw/unix_3rd/2026-05-20
[unix_3rd] exported [...]/var/depot/_/pkg/unix_3rd/2026-05-20
```

This time, the command succeeded. Note that Goa uses the wildcard user "_" by default for exporting. The depot of the wildcard user is meant for development and testing purposes. By exporting to the wildcard user's depot, you are able to use the corresponding archives in other Goa projects without the need to publish and upload untested versions to a web server.

Let's have a brief look at the content of the wildcard depot:

```
$ find var/depot/_/
var/depot/_/
```

[1] https://codeberg.org/genodelabs/gosh

```
var/depot/_/pkg
var/depot/_/pkg/unix_3rd
var/depot/_/pkg/unix_3rd/2026-05-20
var/depot/_/pkg/unix_3rd/2026-05-20/README
var/depot/_/pkg/unix_3rd/2026-05-20/archives
var/depot/_/pkg/unix_3rd/2026-05-20/runtime
var/depot/_/raw
var/depot/_/raw/unix_3rd
var/depot/_/raw/unix_3rd/2026-05-20
var/depot/_/raw/unix_3rd/2026-05-20/terminal.config
var/depot/_/raw/unix_3rd/2026-05-20/unix.config
var/depot/_/raw/unix_3rd/2026-05-20/vimrc
```

You can nicely see here how the *version* file defines the name of the subdirectory of the content.

**Signing and archiving** Even though the depot content looks good, it has not yet a suitable form for distributing it. We ultimately need to wrap the content in archive files and apply our digital signature to these archives. Fortunately, you don't need to do these steps manually since Goa assists with the `publish` command.

```
$ goa publish
[unix_3rd] exported [...]/var/depot/_/raw/unix_3rd/2026-05-20
[unix_3rd] exported [...]/var/depot/_/pkg/unix_3rd/2026-05-20

[unix_3rd] Error: missing definition of depot user

 You can define your depot user name by setting the 'depot_user'
 variable in a goarc file, or by specifing the '--depot-user <name>'
 command-line argument.
```

This command implicitly executed the `goa export` command. As hinted by the error message, Goa needs to know the name of us as the software provider for the publish step. Let us try the command again, but specifying the user name "john" this time.

```
$ goa publish --depot-user john
[unix_3rd] exported [...]/var/depot/_/raw/unix_3rd/2026-05-20
[unix_3rd] exported [...]/var/depot/_/pkg/unix_3rd/2026-05-20

[unix_3rd] Error: missing public key at [...]/var/depot/john/pubkey

 You may use the 'goa add-depot-user' command.
 To learn more about this command:
```

```
goa help add-depot-user
```

Goa cannot know which key to use for signing the depot content. It only knows the name of our made-up depot user "john". But you have not yet drawn the connection to the PGP key pair you have created at the beginning of this article. The `goa add-depot-user` command closes the circle.

```
$ goa add-depot-user john --depot-url "https://your-domain/and/url" \
                          --user-id "a@b.cd"
```

The URL specified as `--depot-url` argument should point to the designated location of the archives on your web server. For reference, Genode Labs' depot URL is https://depot.genode.org/. Note that the URL points to the root of the depot directory structure, not the depot user's subdirectory.

The `--user-id` points to the email or name that we specified when creating the key pair. Alternatively, you may use the `--fingerprint` argument to be more specific.

It is interesting to take a look at the content of the depot user "john" now.

```
$ find var/depot/john/
var/depot/john/
var/depot/john/pubkey
var/depot/john/download
```

There is a fresh subdirectory *john* with the information you supplied to the `goa add-depot-user` command. Take the time to look into both files. Goa extracted the ASCII-armored *pubkey* from the GnuPG (or Sequoia) keyring by using the specified user ID (or fingerprint).

With the connection between the depot user "john" and his key pair drawn, let us give Goa another chance to publish the project.

```
$ goa publish --depot-user john
```

This time, Goa is able to proceed, as indicated by the following messages:

```
publish [...]/var/public/john/pkg/unix_3rd/2026-05-20.tar.xz
publish [...]/var/public/john/raw/unix_3rd/2026-05-20.tar.xz
```

You are also asked by GPG for your passphrase for decrypting your private key.

Once the command completed, you can find the archived and signed depot content at *var/public/john/:*

```
$ find var/public/john
var/public/john/
var/public/john/pkg
var/public/john/pkg/unix_3rd
var/public/john/pkg/unix_3rd/2026-05-20.tar.xz.sig
var/public/john/pkg/unix_3rd/2026-05-20.tar.xz
var/public/john/raw
var/public/john/raw/unix_3rd
var/public/john/raw/unix_3rd/2026-05-20.tar.xz.sig
var/public/john/raw/unix_3rd/2026-05-20.tar.xz
```

Out of curiosity, let's run `goa publish` once again:

```
$ goa publish --depot-user john
[unix_3rd] Error: archive john/raw/unix_3rd/2026-05-20 already exists
                  in the depot

 You may specify '--depot-overwrite' to replace or
 '--depot-retain' to keep the existing version
```

Goa wants to save us from accidentally overwriting existing depot content, which can happen, for example, if you made changes in the project but forgot to adjust the *version* file. You can actually double-check for relevant changes by running `goa published`:

```
$ goa published --depot-user john
[unix_3rd] Exported archives are up-to-date
```

The message tells us that we have already published the archives of this project. Let's make a small change to *raw/vimrc* and run the command again:

```
$ echo "syn on" > raw/vimrc
$ goa published --depot-user john
[unix_3rd] john/raw/unix_3rd/2026-05-20 differs from current project state
[unix_3rd] Version update recommended
```

Goa's `bump-version` command helps us with the version update. When combined with the `--depot-user` and `--if-needed` arguments, the version update is omitted if we have not made any changes:

```
$ goa bump-version --if-needed --depot-user john
[unix_3rd] john/raw/unix_3rd/2026-05-20 differs from current project state
[unix_3rd] Version update required

$ goa publish --depot-user john
```

```
[unix_3rd] exported [...]/var/depot/john/raw/unix_3rd/2026-05-20-a
[unix_3rd] exported [...]/var/depot/john/pkg/unix_3rd/2026-05-20-a

Please enter password to decrypt the PGP key for john:
publish [...]/var/public/john/pkg/unix_3rd/2026-05-20-a.tar.xz
publish [...]/var/public/john/raw/unix_3rd/2026-05-20-a.tar.xz
```

**Syncing the public depot content to the web server** The entirety of the *var/public/john* directory can now be copied as is to the web server. The way of how this content is uploaded is up to you.

The fantastic rsync[1] tool has proven to be useful for this purpose. You may use the following combination of arguments:

```
-rpltOvz --checksum --chmod=Dg+s,ug+w,o-w,+X
```

Please use `man rsync` to decrypt this information.

**Deployment on Sculpt OS** Now that you have published your first Goa project in your depot, you probably want to give it a spin on Sculpt OS. There are two practical options for this: You can either create an option file at */model/option/* or you may publish a depot index referring to your depot package.

For both options, you need to let Sculpt OS know about from where to download your depot archives. For a quick test, you may type in your depot URL in the "+" menu. Be aware, however, that this circumvents any integrity checks of the downloaded archives as your public key still remains unknown to Sculpt.

In order to add your public key to Sculpt, you can copy the *pubkey* file from the Goa-managed depot (e. g. *var/depot/john/pubkey*) to a USB stick. On Sculpt, the *pubkey* file needs to be placed alongside the *download* file that was created by Sculpt in the *depot/john* directory of the used file system when you typed in the URL via the "+" menu.

You may inspect both file systems, i. e. the USB stick on which you copied the *pubkey* file and the file system used by Sculpt, and then copy the file via Sculpt's Inspect view.

**Writing an option file** Manually creating an option file is a good way for testing. The file captures the integration of the deployed component into Sculpt and makes it easy to adapt the archive version.

Using the system shell or the Inspect view in Sculpt, you can create the file */model/option/unix* with the following content:

[1] https://en.wikipedia.org/wiki/Rsync

```
option
+ child unix_3rd | pkg: john/pkg/unix_3rd/2026-05-20
  + connect
    + gui | + child wm
```

The scenario merely requires a Gui service that we route to the "wm" child that is deployed by the "window manager" preset. Once the launcher file is in place, the scenario can be enabled/disabled in the "Options" tab of the "System" menu.

For more details, please consult the Sculpt OS documentation[1].

**Publishing a depot index** A user's depot index is a curated list of the packages and their versions provided by the user. Sculpt OS downloads the index and presents the users with a UI for deploying the referred packages.

Fortunately, Goa assists with managing and publishing a depot index. Let's give it a try! In the Goa-projects repository, change into the *intro/* directory and create the following index file.

```
index
+ supports | arch: x86_64
+ index Tutorial
  + pkg unix_3rd | info: Unix terminal from tutorial
-
```

This file almost represents your depot index as expected by Sculpt but misses the user and version information. Goa takes care of adding this information. Please consult `goa help index` for more details on the structure of index files.

Placing the index file above the *unix_3rd/* directory in the hierarchy enables Goa to look up the version information from the *version* file and publish the referenced Goa projects if necessary. Goa simply scans the subdirectories of the current working directory for looking up related Goa projects. After adding your depot user. You can therefore publish your depot index together with the unix_3rd package with a single command.

```
intro$ goa add-depot-user john --depot-url ... --fingerprint ...

intro$ goa publish --depot-user john --depot-overwrite
[intro] exporting project .../intro/unix_3rd
[unix_3rd] exported .../intro/var/depot/john/raw/unix_3rd/2026-05-20
[unix_3rd] exported .../intro/var/depot/john/pkg/unix_3rd/2026-05-20
...
[intro] exported .../intro/var/depot/john/index/26.04
```

[1] https://genode.org/documentation/articles/sculpt-26-04#Runtime_management

```
publish .../intro/var/public/john/pkg/unix_3rd/2026-05-20.tar.xz
publish .../intro/var/public/john/raw/unix_3rd/2026-05-20.tar.xz
publish .../intro/var/public/john/index/26.04.xz
```

After syncing your depot content to the web server. Users are able to install your unix_3rd package via the Sculpt UI. Please refer to the Sculpt documentation for more details.

**Sculpt OS documentation**

https://genode.org/documentation/articles/sculpt-26-04

## 5.3 Writing a VFS plugin for network-packet access

*This section reproduces a minimal implementation of the original VFS tap plugin[1] with Goa. The complete implementation is available in the Genode repository.*

In Linux and FreeBSD, the kernel provides virtual TAP devices as an interface for sending/receiving raw Ethernet frames. This section demonstrates how this functionality can be added to Genode's VFS by means of a dedicated plugin.

When porting software from the Unix world to Genode, we try to keep modifications of the 3rd-party code to a minimum. An essential part of this consists in providing the required libraries (e. g., libc, stdcxx). But, even with all libraries in place, we also need to bridge the gap between the Unix viewpoint of "everything is a file (descriptor)" and the Genode world of session interfaces. This is where the VFS comes into play: Genode's C runtime (libc) maps file operations to the component's VFS. Let's have a look at a common example:

```
config
+ libc | stdout: /dev/log
+ vfs
+ dir dev
  + log
-
```

This component config tells the libc to use */dev/log* for stdout and use the built-in log plugin of the VFS to "connect" */dev/log* to a LOG session. Section 3.3 provides an overview of libc and VFS configuration.

For writing a VFS plugin for raw network-packet access, let's first sketch an overview on how TAP devices are used on FreeBSD/Linux and how this maps to the VFS architecture.

**TAP-device foundations** Genode's C runtime is based on a port of FreeBSD's libc. On FreeBSD, we simply open an existing TAP device (e. g. */dev/tap0*) and are able to write/read to the acquired file descriptor afterwards. In addition, there are a few I/O control operations (ioctl), by which we can get/set the MAC address or get the device name for instance. Let's look at an example:

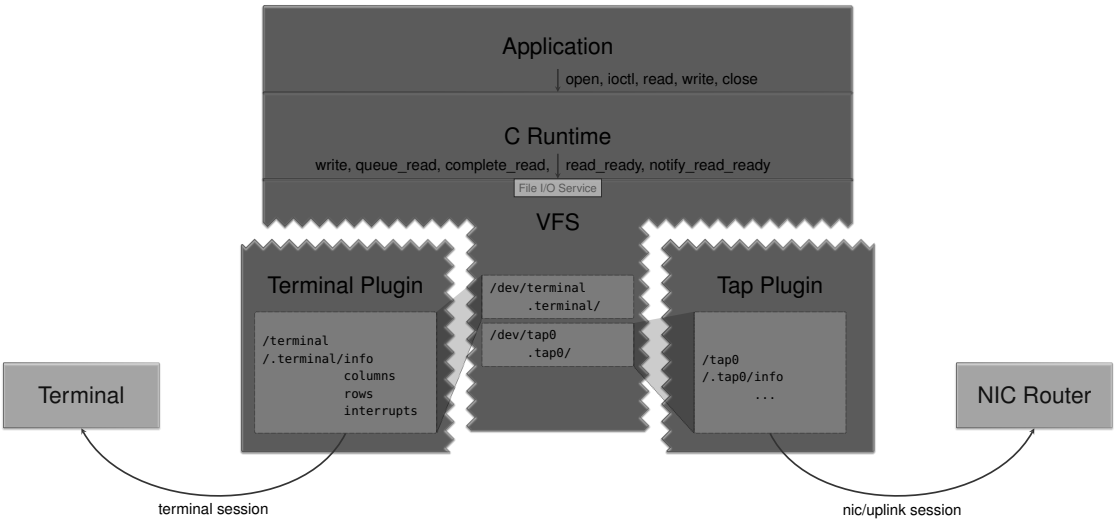
```
#include <net/if.h>
#include <net/if_tap.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
```

[1] https://genodians.org/jschlatow/2022-03-01-vfs-tap

```
#include <sys/ioctl.h>
#include <stdint.h>

int main()
{
   int fd0 = open("/dev/tap0", O_RDWR);
   if (fd0 == -1) {
      printf("Error: open(/dev/tap0) failed\n");
      return 1;
   }

   char mac[6];
   memset(mac, 0, sizeof(mac));
   if (ioctl(fd0, SIOCGIFADDR, (void *)mac) < 0) {
      printf("Error: Could not get MAC address of /dev/tap0.\n");
   } else {
      printf("MAC: %02x:%02x:%02x:%02x:%02x:%02x\n", mac[0], mac[1], mac[2],
                                                     mac[3], mac[4], mac[5]);
   }

   enum { BUFFLEN = 1500 };
   char buffer[BUFFLEN];
   while (1) {
      ssize_t received = read(fd0, buffer, BUFFLEN);
      if (received < 0) {
         close(fd0);
         return 1;
      }

      printf("Received packet with %d bytes\n", received);
      size_t i=0;
      uint32_t *words = (uint32_t*)buffer;
      for (; i < received / 4; i++) {
         printf("%08x ", *words++);

         if (i % 4 == 3)
            printf("\n");
      }

      uint8_t *bytes = (uint8_t*)&buffer[i*4];
      for (i*=4; i < received; i++)
         printf("%02x", *bytes++);

      printf("\n");
   }
}
```

This application code receives Ethernet frames from tap0 and prints out the data. For demonstrative purpose, there is also an ioctl call for getting the MAC address of tap0. A detailed description of TAP devices in FreeBSD is given in the corresponding man page[1].

**Architecture** Before diving into the VFS, let's draw a high-level picture of how Genode's C runtime maps file operations to the VFS.




The figure above illustrates the plugin structure of the VFS. A plugin provides one or multiple files (e. g. */dev/tap0*) that are incorporated into the directory tree of the VFS. The application is then able to perform the standard file operations on these files. The VFS plugin typically translates these operations into operations on a particular session interface. The C runtime also emulates ioctl by mapping these to read/write accesses of pseudo files (e. g. */dev/.tap0/...*) as described in the corresponding release notes[2] and commit message[3].

By convention, an info file (e. g. */dev/.tap0/info*) hosts an HID report containing a single node named after the plugin type. The node may comprise any number of attributes to specify parameters needed by the C runtime to implement the particular ioctl, e. g.:

```
+ tap tap0 | mac_addr: 02:02:02:02:02:02
```

In case parameters shall be modifiable, the info file can be accompanied by a separate (writeable) file for each modifiable parameter.

[1] https://man.freebsd.org/cgi/man.cgi?tap(4)

[2] https://genode.org/documentation/release-notes/20.11#Streamlined_ioctl_handling_in_the_C_runtime___VFS

[3] https://codeberg.org/genodelabs/genode/commit/7ac32ea60

The C runtime takes care of emulating the blocking semantics of read/write operations. Internally, the C runtime uses the non-blocking `Vfs::File_io_service` interface to perform read/write accesses on the VFS. The `write()` operation returns an error if writing cannot be performed immediately. Reads are split into `queue_read()` and `complete_read()` methods. In order to avoid futile polling, the latter are accompanied by a `read_ready()` method, which returns `true` if there is readable data, and a `notify_read_ready()` method by which one is able to announce interest in receiving read-ready signals. Moreover, a `write_ready()` method propagates the saturation of I/O buffers to the VFS user, which becomes important when using non-blocking file descriptors.

**Usage preview** Before we start coding, let's envision how we want to use the plugin:

```
config
+ vfs
  + dir dev
    + tap tap0
```

In the above example, we mount the plugin at the */dev/tap0* file. The name of the `tap` node is mandatory. The plugin shall use a NIC session to transmit the Ethernet frames to a NIC router.

**Creating a new Goa project** Let's start with preparing the stage for the plugin by starting a new Goa project.

```
$ mkdir -p vfs_tap/src
$ cd vfs_tap
```

The VFS library uses the type of the HID node to determine the name of the plugin library to probe. More precisely, when adding a `tap` node to the config, the VFS tries to load a *vfs_tap.lib.so*. Hence, we need to tell the build system to create a shared library with this name. Goa supports library projects for most supported commodity build systems. Let's use CMake for this one and, thus, create the file *src/CMakeLists.txt* with the following content:

```
cmake_minimum_required(VERSION 3.10)
project(vfs_tap)

set(LIB_SRCS vfs_tap.cc)
add_library(vfs_tap SHARED ${LIB_SRCS})

set_target_properties(vfs_tap PROPERTIES PREFIX "")
```

The first two lines are mandatory for CMake. Line 4-5 define the library build target and *vfs_tap.cc* as the only source file. The last line removes the default "lib" prefix. Without this line, the build artifact would be named *libvfs_tap.lib.so* instead of *vfs_tap.lib.so*

**Writing the vfs_tap plugin** Now, let's add the first few lines to *src/vfs_tap.cc*:

```
namespace Vfs_tap {

  using namespace Genode;
  using namespace Genode::Vfs;

  using Name  = String<64>;

  struct Local_factory;
  struct Data_file_system;
  struct Compound_file_system;

}

/* [...] see below */

extern "C" Genode::Vfs::File_system_factory *vfs_file_system_factory(void)
{
  using namespace Genode;

  struct Factory : Vfs::File_system_factory
  {
    Vfs::File_system *create(Vfs::Env &env, Node const &config) override
    {
     return new (env.alloc())
      Vfs_tap::Compound_file_system(env, config);
    }
  };

  static Factory f;
  return &f;
}
```

First, you can see that the plugin lives in its own namespace `Vfs_tap`. The namespace declares three objects: a `Compound_file_system`, a `Local_factory` and a `Data_file_system`. This is a scheme that we commonly apply when writing VFS plugins. Let's walk through each of those step by step.

The plugin's entrypoint is the `vfs_file_system_factory` method that returns a `File_system_factory` by which the VFS is able to create a `File_system` from the cor-

responding HID node (e. g. `+ tap tap0`). We return a `Compound_file_system` which serves as a top-level file system and which is able to instantiate arbitrary sub-directories and files on its own by using VFS primitives. Let's have a closer look:

```
class Vfs_tap::Compound_file_system : private Local_factory,
                                      public  Vfs::Dir_file_system
{
  private:

    using Name = Vfs_tap::Name;

    using Config = String<200>;
    static Config _config(Name const &name)
    {
      char buf[Config::capacity()] { };

      /*
       * By not using the node type "dir", we operate the
       * 'Dir_file_system' in root mode, allowing multiple sibling nodes
       * to be present at the mount point.
       */
      Generator::generate({ buf, sizeof(buf) }, "compound",
        [&] (Generator &g) {

          g.node("data", [&] () {
            g.attribute("name", name); });

          g.node("dir", [&] () {
            g.attribute("name", Name(".", name));
            g.node("info");
          });
      }).with_error([] (Buffer_error) {
        warning("VFS-tap compound exceeds maximum buffer size");
      });

      return Config(Cstring(buf));
    }

  public:

    Compound_file_system(Vfs::Env &vfs_env, Node const &node)
    :
      Local_factory(vfs_env, node),
      Vfs::Dir_file_system(vfs_env,
                           Node(_config(Local_factory::name(node))),
                           *this)
```

```
    { }

    static const char *name() { return "tap"; }

    char const *type() override { return name(); }
};
```

The `Compound_file_system` is a `Dir_file_system` and a `Local_factory`. The former allows us to create a nested directory structure from HID as we are used to when writing a component's `vfs` config. In this case, the static `_config()` method generates the following HID:

```
compound
+ data tap0
+ dir .tap0
  + info
-
```

The type of the root node has no particular meaning, yet, since it is not "dir", it instructs the `Dir_file_system` to allow multiple sibling nodes to be present at the mount point. In particular, this is a *data* file system and a subdirectory containing an *info* file system. The latter has a static name, whereas the subdirectory and data file system are named after what the implementation of `Local_factory::name()` returns (e. g. "tap0"). Already knowing how the C runtime interacts with the VFS, we can identify that the data file system shall provide read/write access to our virtual TAP device whereas the subdirectory is used for ioctl support. The *info* file system follows the aforementioned convention and provides a file containing a `tap` node with a `name` and `mac_addr` attribute.

Note, the `type()` method is part of the `File_system` interface and must return the HID node type to which the plugin responds.

Next, we must implement the `Local_factory`. As the name suggest, it is responsible for instantiating the file systems that we used in the `Compound_file_system`, i. e. the *data* and *info* file system:

```
struct Vfs_tap::Local_factory : File_system_factory, Device_update_handler
{
  Vfs::Env              &_env;

  Name             const _name;
  Net::Mac_address       _mac { };

  Data_file_system   _data_fs { _env.env(), _env.user(), _name, *this };
```

```
  /* [...] see below */
```

In the first few lines of `Local_factory`, you see that the object stores the name and MAC address. You also see the instantiation of the *data* file system. You have already seen the forward declaration of `Data_file_system` in the beginning. We will come back to this - and to the peculiar `Device_update_handler` - after we completed the `Local_factory`. Let's first continue with the *info* file system:

```
struct Vfs_tap::Local_factory : File_system_factory, Device_update_handler
{
   /* [...] see above */

  struct Info
  {
    Name             const &_name;
    Net::Mac_address const &_mac;

    Info(Name             const & name,
         Net::Mac_address const & mac)
    : _name(name),
      _mac(mac)
    { }

    void print(Output &out) const
    {
      char buf[128] { };
      Generator::generate({ buf, sizeof(buf) }, "tap",
        [&] (Generator &g) {
          g.attribute("mac_addr", String<20>(_mac));
          g.attribute("name", _name);
      }).with_error([&] (Buffer_error) {
        warning("VFS-tap info exceeds maximum buffer size"); });

      Genode::print(out, Cstring(buf));
    }
  };

  Info                                 _info          { _name,  _mac };
  Readonly_value_file_system<Info>     _info_fs       { "info", _info };

   /* [...] see below */
```

For the *info* file system, we use the `Readonly_value_file_system` template from *os-/include/vfs/readonly_value_file_system.h*. As the name suggests, it provides a file system with a single read-only file that contains the value of the given type. More precisely,

the string representation of its value. In case of the *info* file system, we want to fill the file with `+ tap | name: ... | mac_addr: ...`. Knowing that we are able to convert any object to `Genode::String` by defining a `print(Genode::Output)` method, we can use the `Info` struct as a type for `Readonly_value_file_system` and customize its string representation at the same time.

The next fragment of the `Local_factory` comprises the `File_system_factory` interface, an accessor for reading the device name from the `tap` node and the constructor.

```
struct Vfs_tap::Local_factory : File_system_factory, Device_update_handler
{
   /* [...] see above */

  /***********************
   ** Factory interface **
   ***********************/

  Vfs::File_system *create(Vfs::Env&, Node const &node) override
  {
    if (node.has_type("data"))      return &_data_fs;
    if (node.has_type("info"))      return &_info_fs;

    return nullptr;
  }

  /***********************
   ** Constructor, etc. **
   ***********************/

  static Name name(Node const &config)
  {
    return config.attribute_value("name", Name("tap"));
  }

  Local_factory(Vfs::Env &env, Node const &config)
  :
    _name(name(config)),
    _env(env)
  { }

  /* [...] see below */
```

The `create()` method is the more interesting part. Here, it returns either the *data* or *info* file system depending on the node type. The function is called by the `Dir_file_system` on the config defined by the `Compound_file_system`.

Note that mutable parameters need to be provided as additional writeable files along with the info file. For this purpose, you may use the `Value_file_system` template from *os/include/vfs/value_file_system.h* together with `Genode::Watch_handler` to react to file modifications. Since both, the name and MAC address, are supposed to be read-only via ioctl, this is not part of this tutorial, though.

The remaining part of the `Local_factory` concerns the `Device_update_handler` interface that we have disregarded so far:

```
namespace Vfs_tap
{
   /* [...] see above */

   struct Device_update_handler;
}

struct Vfs_tap::Device_update_handler : Interface
{
  virtual void device_state_changed() = 0;
};

struct Vfs_tap::Local_factory : File_system_factory, Device_update_handler
{
   /* [...] see above */

  /*****************************
   ** Device update interface **
   *****************************/

  void device_state_changed() override
  {
    /* update mac address */
    _data_fs.with_handle([&] (auto &h) { _mac = h.mac_address(); });

    /* propagate changes to info_fs */
    _info_fs.value(_info);
  }
};
```

The `Device_update_handler` defines an interface used by the `Data_file_system` for notifying the `Local_factory` about a changed device state. We are merely using this to read the MAC address that is assigned to the NIC session by the NIC router, and update the *info* file system accordingly.

The last missing piece of our puzzle is the `Data_file_system`. Luckily, there is no need to take a deep dive into the VFS internals because `Vfs::Single_file_system` comes to the rescue. It already implements big parts of the `Directory_service` and

the File_io_service interface, and leaves only a handful methods to be implemented by Data_file_system. Let's have a look at the first fragment:

```
struct Vfs_tap::Data_file_system : public Single_file_system
{
    struct Tap_vfs_handle : Single_vfs_handle
    {
       /* [...] see below */
    };

    using Registered_handle = Genode::Registered<Tap_vfs_handle>;
    using Handle_registry   = Genode::Registry<Registered_handle>;
    using Open_result       = Directory_service::Open_result;

    Genode::Env            &_env;
    Vfs::Env::User         &_vfs_user;
    Name                   &_name;
    Device_update_handler  &_device_update_handler;
    Handle_registry         _handle_registry { };

    Data_file_system(Genode::Env            & env,
                     Vfs::Env::User         & vfs_user,
                     Name             const & name,
                     Device_update_handler  & handler)
    :
      Single_file_system(Node_type::TRANSACTIONAL_FILE, name.string(),
                         Node_rwx::rw(), Node()),
      _env(env), _vfs_user(vfs_user), _name(name),
      _device_update_handler(handler)
    { }

    static const char *name()   { return "data"; }
    char const *type() override { return "data"; }

    template <typename FN>
    void with_handle(FN && fn)
    {
      _handle_registry.for_each([&] (Tap_vfs_handle &handle) {
        fn(handle); });
    }

    /* [...] see below */
```

Let's skip the details of Tap_vfs_handle for the moment. You see that we use a Genode::Registry to manage the Tap_vfs_handle. The Data_file_system constructor stores a few references for later use. The Single_file_system constructor takes a node

type, a name, an access mode and a `Node` as arguments. For the node type, you can choose between `CONTINUOUS_FILE` and `TRANSACTIONAL_FILE`. Since a network packet is supposed to be written as a whole and not in arbitrary chunks, we must choose `TRANSACTIONAL_FILE` here. The file name is determined from the provided HID node by looking up a *name* parameter. Here, we pass an empty `data` node, in which case, the `Single_file_system` uses the second argument as a file name instead. The fragment also shows the implementation of the `with_handle()` method that is used by the previously shown `device_state_changed()` method.

Let's continue with completing the `Directory_service` interface:

```
struct Vfs_tap::Data_file_system : public Single_file_system
{
    /* [...] see above */

    /*********************************
     ** Directory service interface **
     *********************************/

    Open_result open(char const  *path, unsigned flags,
                     Vfs_handle **out_handle,
                     Allocator   &alloc) override
    {
      if (!_single_file(path))
        return Open_result::OPEN_ERR_UNACCESSIBLE;

      /* return error if already opened. */
      unsigned handles = 0;
      _handle_registry.for_each([&handles] (Tap_vfs_handle const &) {
        handles++;
      });
      if (handles) return Open_result::OPEN_ERR_EXISTS;

      try {
        *out_handle = new (alloc)
          Registered_handle(_handle_registry, _env, _vfs_user, alloc,
                            _name.string(), *this, *this, flags);
        _device_update_handler.device_state_changed();
        return Open_result::OPEN_OK;
      }
      catch (Out_of_ram)  { return Open_result::OPEN_ERR_OUT_OF_RAM; }
      catch (Out_of_caps) { return Open_result::OPEN_ERR_OUT_OF_CAPS; }
    }
}
```

The only method of the `Directory_service` interface not implemented by `Single_file_system` is the `open()` method. First, we use a helper method `_single_file` to check whether the correct path was given. Second, we ensure that the file has not been opened yet since the FreeBSD man page says that a TAP device is exclusive-open. Third, we allocate a new `Tap_vfs_handle`, which is conveniently put into the `_handle_registry` by using the `Genode::Registered` wrapper. The latter also takes care that the handle is removed from the registry on destruction. Once created, we call the `device_state_changed()` method.

The read and write operations are part of the `File_io_service` interface. This interface is already implemented by `Single_file_system`, which forwards most methods to `Single_vfs_handle`. Let's thus look at `Tap_vfs_handle`, which implements the read and write operations and translates them to the NIC session interface (details omitted for conciseness). Note that `Single_file_system` forwards `complete_read()` to the handle's `read()` method and always returns `true` for `queue_read()`.

```
struct Tap_vfs_handle : Single_vfs_handle
{
  using Read_result    = File_io_service::Read_result;
  using Write_result   = File_io_service::Write_result;
```

```
  Genode::Io_signal_handler<Tap_vfs_handle> _read_avail_handler {
        _env.ep(), *this, &Tap_vfs_handle::_handle_read_avail };

  bool _notifying = false;
  bool _blocked   = false;

  void _handle_read_avail()
  {
     if (!read_ready()) return;

     if (_blocked) {
        _blocked = false;
        _vfs_user.wakeup_vfs_user();
     }

     if (_notifying) {
        _notifying = false;
        read_ready_response();
     }
  }

  Tap_vfs_handle(Genode::Env            &env,
                 Vfs::Env::User         &vfs_user,
                 Allocator              &alloc,
```

```
                Directory_service      &ds,
                File_io_service        &fs,
                int                     flags)
  : Single_vfs_handle  { ds, fs, alloc, flags },
    _env(env), _vfs_user(vfs_user), _nic(/* ... */)
  {
     _nic.rx_channel()->sigh_ready_to_ack(_read_avail_handler);
     _nic.rx_channel()->sigh_packet_avail(_read_avail_handler);
  }

  bool notify_read_ready() override
  {
     _notifying = true;
     return true;
  }

  /* [...] (see below) */
};
```

The `Tap_vfs_handle` defines a signal-handler method `_handle_read_avail()` that notifies the C runtime or the VFS server of any progress. There are two types of progress notifications: I/O progress and read ready. The latter we have already come across when mentioning the `notify_read_ready()` method of the `File_io_service`. In this implementation, we issue a read-ready response whenever the `notify_read_ready()` was called before on this file handle. Similarly, we keep track of whether a `read()` operation is unable to complete via the `_blocking` member variable. By calling `wakeup_vfs_user()`, the C runtime is notified of the fact that there was I/O progress, and it may retry the read operation. Note that the C runtime stalls any application-level signals when in a blocking operation, hence the `_read_avail_handler` must be declared as `Io_signal_handler`.

```
struct Tap_vfs_handle : Single_vfs_handle
{
  /* [...] (see above) */

  bool read_ready() const override
  {
     /* [...] */
  }

  bool write_ready() const override
  {
     /* [...] */
  }
```

```
  Read_result read(Byte_range_ptr const &dst, size_t &out_count) override
  {
     if (!read_ready()) {
        _blocked = true;
        return Read_result::READ_QUEUED;
     }

     /* [...] */

     return Read_result::READ_OK;
  }

  Write_result write(Const_byte_range_ptr const &src,
                     size_t                     &out_count) override
  {
     if (!_nic.tx()->ready_to_submit())
        return Write_result::WRITE_ERR_WOULD_BLOCK;

     /* [...] */

     return Write_result::WRITE_OK;
  }
};
```

The last ingredient is inserting the proper result types: While `READ_OK` and `WRITE_OK` are self-explanatory, there are two common result types for unsuccessful reads/writes. On the one hand, `READ_QUEUED` indicates that a previously queued read cannot be completed. On the other hand, we may return `WRITE_ERR_WOULD_BLOCK` if, e. g., the submit queue of the NIC session's transmit channel is full.

**Building the VFS plugin with Goa** With the source code in place, you can try building the plugin with Goa. For this purpose, Goa needs to know what APIs are used by the source code. This is achieved by listing them in the *used_apis* file. It's a good practice to start with the most obvious ones.

```
vfs_tap$ cat used_apis
genodelabs/api/base
genodelabs/api/os
genodelabs/api/vfs
genodelabs/api/nic_session
```

Now, create an *artifacts* file mentioning *vfs_tap.lib.so* and try `goa build`:

```
vfs_tap$ echo "vfs_tap.lib.so" > artifacts
vfs_tap$ goa build
[vfs_tap] Error: no version defined for depot archive
  'genodelabs/api/nic_session'
```

Apparently, Goa lacks any version information for the NIC session API. This information can be added by the following line in a *goarc* file.

```
set version(genodelabs/api/nic_session) 2026-04-16
```

Now, give `goa build` another try:

```
vfs_tap$ goa build
...
[vfs_tap:cmake] [100%] Linking CXX shared library vfs_tap.lib.so
[...]/ld: cannot find -l:ldso_so_support.lib.a: No such file or directory
```

Oh yes, building a shared library requires adding the *so* API to the *used_apis* file.

```
vfs_tap$ echo "genodelabs/api/so" >> used_apis
vfs_tap$ goa build
[vfs_tap:cmake] -- Configuring done (0.0s)
[vfs_tap:cmake] -- Generating done (0.0s)
[vfs_tap:cmake] -- Build files have been written to: [...]/var/build/x86_64
[vfs_tap:cmake] [ 50%] Building CXX object
  CMakeFiles/vfs_tap.dir/vfs_tap.cc.obj
[vfs_tap:cmake] [100%] Linking CXX shared library vfs_tap.lib.so
[vfs_tap:cmake] [100%] Built target vfs_tap
[vfs_tap] Error: missing symbols file 'vfs_tap'

 You can generate this file by running 'goa extract-abi-symbols'.
```

Well, Goa noticed that you are building a shared library object and expects a symbols file. Usually, when we create a library with Goa, we also want to export an API archive which comprises the header files and the exported symbols to allow linking against the library's ABI. This, however, is not needed for a VFS plugin library. You may therefore use an empty symbols file to satisfy Goa.

```
vfs_tap$ mkdir symbols
vfs_tap$ touch symbols/vfs_tap
vfs_tap$ goa build
...
[vfs_tap:cmake] [100%] Built target vfs_tap
```

Yay, you've successfully built the VFS plugin.

**Testing the plugin** Let's create a simple test application that uses the VFS plugin. For this, you need a separate project directory with a *src* subdirectory. Let's create one as a subdirectory in our *vfs_tap* project:

```
vfs_tap$ mkdir -p test/src
```

You'll also need to use the same depot dir, which can be achieved by adding the following lines to the *goarc* file:

```
vfs_tap$ echo "set depot_dir  ./var/depot"  >> goarc
vfs_tap$ echo "set public_dir ./var/public" >> goarc
```

Now, you need to add some code for the test application. Simply use the example code from the very beginning of this section and place it in the file *test/src/test-vfs_tap.cc*. Also add a *Makefile*, an *artifacts* file and a *used_apis* file:

```
vfs_tap$ echo "test-vfs_tap: test-vfs_tap.cc" > test/src/Makefile
vfs_tap$ echo "test-vfs_tap" > test/artifacts
vfs_tap$ echo "genodelabs/api/libc"  >  test/used_apis
vfs_tap$ echo "genodelabs/api/posix" >> test/used_apis
```

In order to run the test application, you need to define a runtime scenario. The Genode repository contains a ping application that you can use for generating some network traffic. When both, the ping component and the test application connect to the same domain of a NIC router, you should be able to see some output of the test application. For this purpose, create the following *test/pkg/runtime* file:

```
runtime | ram: 20M | caps: 1000 | binary: init
+ requires
  + timer

+ content
  + rom test-vfs_tap
  + rom libc.lib.so
  + rom libm.lib.so
  + rom posix.lib.so
  + rom vfs.lib.so
  + rom vfs_tap.lib.so
  + rom ping
  + rom nic_router

+ config
  + parent-provides
  | + service PD
```

```
  | + service CPU
  | + service LOG
  | + service ROM
  | + service Timer
  + default-route
  | + service Nic
  | | + child nic_router
  | + any-service
  |   + parent

  + default | caps: 100

  + start nic_router | ram: 2M
  | + provides
  | | + service Nic
  | | + service Uplink
  | + config | verbose_domain_state: yes | verbose: yes
  |   + default-policy | domain: default
  |   + domain default | interface: 10.0.2.1/24

  + start test-vfs_tap | ram: 8M
  | + config
  |   + libc | stdout: /dev/log
  |   + vfs
  |     + dir dev
  |       + log
  |       + tap tap0

  + start ping | ram: 4M
    + config
             interface:  10.0.2.2/24
             gateway:    10.0.2.1
             dst_ip:     10.0.2.123
             period_sec: 5
             verbose:    no
-
```

Goa also needs to know in what archives it can find the content ROM modules mentioned in the *runtime* file. This is achieved by the following *test/pkg/archives* file:

```
genodelabs/src/init
genodelabs/src/libc
genodelabs/src/vfs
genodelabs/src/posix
genodelabs/src/nic_router
```

```
_/src/vfs_tap
jschlatow/src/ping/2026-04-16
```

With these settings in place, you are able to execute `goa run -C test`, which automatically exports the vfs_tap archive as the wildcard depot user "_"". Don't forget to initialize the version file and to add a *LICENSE* file. You may start with an empty file for testing:

```
vfs_tap$ goa bump-version
vfs_tap$ touch LICENSE
vfs_tap$ goa run -C test
...
[vfs_tap] exported [...]/var/depot/_/src/vfs_tap/2026-05-20
[vfs_tap] exported [...]/var/depot/_/bin/x86_64/vfs_tap/2026-05-20

Genode 26.02-309-g6ae975a32c8
17592186044415 MiB RAM and 19000 caps assigned to init
[init -> test -> nic_router] [default] static IP config: interface ...
[init -> test -> nic_router] [default] NIC sessions: 0
[init -> test -> nic_router] [default] initiated domain
[init -> test -> nic_router] [default] NIC sessions: 1
[init -> test -> nic_router] [default] NIC sessions: 2
[init -> test -> test-vfs_tap] MAC: 02:02:02:02:02:02
[init -> test -> nic_router] [default] forward ARP request for local IP to
                                       all interfaces of the sender domain
[init -> test -> test-vfs_tap] Received packet with 42 bytes
[init -> test -> test-vfs_tap] ffffffff 0202ffff 01020202 01000608
[init -> test -> test-vfs_tap] 04060008 02020100 01020202 0202000a
[init -> test -> test-vfs_tap] ffffffff 000affff 027b
```

Excellent! The complete code of this tutorial is available on Codeberg[1]. The official implementation of the vfs_tap plugin is part of the Genode repository[2].

[1] https://codeberg.org/jschlatow/goa-projects/src/branch/main/examples/vfs_tap
[2] https://codeberg.org/genodelabs/genode/src/branch/main/repos/os/src/lib/vfs/tap

## 5.4 Porting Lomiri Calculator App

*This section is based on the article[1] at https://genodians.org.*

Since the port of Ubuntu UI Toolkit to Genode[2], Ubuntu Touch apps can be ported to Genode. After Canonical dropped support for Ubuntu Touch, the toolkit was adopted by *UBports* as a community project and renamed to *Lomiri UI Toolkit*.

The port of the toolkit is available in the genode-world repository (see Section 3.7.3). Ready-to-use depot archives for x86_64 are available in Sebastian's depot[3].

This section walks through the porting procedure of the Lomiri Calculator App[4] and thereby serves as a blueprint for porting other apps from the toolkit.

**Creating a new Goa project** Every Goa project resides in a separate directory (they can be nested, though). Goa automatically determines whether a directory is a project directory based on its content. Goa uses the name of the directory as a project name.

Starting a new Goa project merely consists in creating a separate directory at an arbitrary location and supplementing directory content that is considered by Goa.

```
$ mkdir calculator
calculator$
```

**Importing the source code** As a first step, you need to import the app's source code. For this, you simply create an *import* file with the following content:

```
LICENSE   := GPLv3
VERSION   := 3.3.7
DOWNLOADS := calc.archive

APPS_URL  := https://gitlab.com/ubports/development/apps/
BASE_URL  := $(APPS_URL)/lomiri-calculator-app/-/archive/
URL(calc) := $(BASE_URL)/v$(VERSION)/lomiri-calculator-app-v$(VERSION).tar.gz
SHA(calc) := 821f045e9cdb5f26145f60c53bf92f96ba81a563c0a0fec72ee1cdfccc0a9f88
DIR(calc) := src
```

Syntactically, the file is a Makefile that merely defines a couple of variables documented by `goa help import`. With the above definitions, you are importing the source code from a tar archive. The tool also supports git and svn. Note that the app version 3.3.7 was the last version before Ubuntu UI Toolkit got renamed to Lomiri.

[1] https://genodians.org/jschlatow/2024-01-11-lomiri-calculator-porting
[2] https://genodians.org/ssumpf/2023-05-06-ubunutu_ui
[3] https://depot.genode.org/ssumpfs
[4] https://gitlab.com/ubports/development/apps/lomiri-calculator-app/

With the *import* file present, you are able to run `goa import`, which places the source code into the *src/* subdirectory.

```
calculator$ goa import
import  download https://gitlab.com/ubports/[...]/lomiri-calculator-app
                /-/archive//v3.3.7/lomiri-calculator-app-v3.3.7.tar.gz
import  extract lomiri-calculator-app-v3.3.7.tar.gz (calc)
import  generate import.hash
```

In case the source code needs some adaptations, Goa is able to apply patches during import (see `goa help import` for more details). For convenience, `goa diff` lets you easily create a patch for your local modifications.

**Building the application** Goa supports various commodity build systems such as GNU Make, autoconf, CMake, qmake, Meson and Cargo (see `goa help build-systems` for more details). Fortunately, the calculator app is based on CMake, hence let's try running `goa build`:

```
calculator$ goa build
[calculator] Error: [...] has a 'src' directory but lacks an 'artifacts' file.
             You may start with an empty file.
```

As mentioned in Section 3.5, Goa requires an *artifacts* file to build a binary archive so let's do as suggested and create an empty one.

```
calculator$ touch artifacts
calculator$ goa build
[calculator] CMake module Qt5Core is not provided by any api archive please
  consider adding the corresponding api archive to the used_apis file.
[calculator] CMake module Qt5Qml is not provided by any api archive please
  consider adding the corresponding api archive to the used_apis file.
[calculator] CMake module Qt5Quick is not provided by any api archive please
  consider adding the corresponding api archive to the used_apis file.
CMake Error at CMakeLists.txt:1 (cmake_minimum_required):
  Compatibility with CMake < 3.5 has been removed from CMake.

  Update the VERSION argument <min> value.  Or, use the <min>...<max> syntax
  to tell CMake that the project requires at least <min> but has been updated
  to work with policies introduced by <max> or earlier.

  Or, add -DCMAKE_POLICY_VERSION_MINIMUM=3.5 to try configuring anyway.


[calculator:cmake] -- Configuring incomplete, errors occurred
```

Apparently, CMake is unable to locate Qt5Core, Qt5Qml and Qt5Quick since we have not added a *used_apis* file yet. Moreover, the CMake version installed on the host system has removed compatibility with CMake < 3.5. Let's first focus on getting the *used_apis* right. Qt5Core is part of the *qt5_base* archive whereas Qt5Qml and Qt5Quick are part of *qt5_declarative*. The *used_apis* file therefore looks like this:

```
genodelabs/api/qt5_base
ssumpf/api/qt5_declarative
```

```
calculator$ goa build
...
[lomiri-calculator-app] Error: no version defined for depot
                        archive 'ssumpf/api/qt5_declarative'
```

Well, since Goa only comes with the version information of official archives from the genodelabs depot, you have to provide the version information. This is achieved by adding the version definition in a *goarc* file. You may use the wildcard depot user "_" to define the version of the api archive for all depot users:

```
calculator$ echo 'set version(_/api/qt5_declarative) 2024-02-25' >> goarc
calculator$ goa build
...
CMake Error at CMakeLists.txt:1 (cmake_minimum_required):
  Compatibility with CMake < 3.5 has been removed from CMake.

  Update the VERSION argument <min> value.  Or, use the <min>...<max> syntax
  to tell CMake that the project requires at least <min> but has been updated
  to work with policies introduced by <max> or earlier.

  Or, add -DCMAKE_POLICY_VERSION_MINIMUM=3.5 to try configuring anyway.
```

This leaves us with the issue of the incompatible `cmake_minimum_required` statement in the *src/CMakeList.txt* file. Let's do as suggested and add the `-DCMAKE_POLICY_VERSION_MINIMUM=3.5` argument. Fortunately, Goa allows providing arbitrary argument to CMake via a *cmake_args* file.

```
calculator$ echo '-DCMAKE_POLICY_VERSION_MINIMUM=3.5' > cmake_args
calculator$ goa build
...
[calculator:cmake] -- Build files have been written to:
                      [...]/calculator/var/build/x86_64
...
[calculator:cmake] [100%] Built target com_ubuntu_calculator_translation_files
[calculator:cmake] Install the project...
```

```
[calculator:cmake] -- Install configuration: ""
[calculator:cmake] -- Installing: //manifest.json
[calculator:cmake] -- Installing: //ubuntu-calculator-app.apparmor
[calculator:cmake] -- Installing: /share/qml/graphics/ubuntu-calculator-app.svg
...
```

Yikes, the build succeeded and apparently installed files into the file-system root. Luckily, no file system was harmed because Goa executes `cmake install` in a sandboxed environment, which only has write access to the build directory. It is puzzling, though, it tried writing to the file-system root in the first place because Goa calls `cmake install` with `CMAKE_INSTALL_PREFIX=/.../var/build/<arch>/install/`. Looking at *src/CMakeFile.txt* reveals the `CLICK_MODE` option, which sets `CMAKE_PREFIX_PATH` to `/`. Setting `CLICK_MODE=0` should do the trick:

```
calculator$ echo '-DCLICK_MODE=0' >> cmake_args
calculator$ goa build
...
[calculator:cmake] -- Installing: /usr/lib/python3.14/[...]
CMake Error at tests/autopilot/cmake_install.cmake:49 (file):
  file INSTALL cannot make directory
  "/usr/lib/python3.14/site-packages/ubuntu_calculator_app": Read-only file
  system.
Call Stack (most recent call first):
  tests/cmake_install.cmake:42 (include)
  cmake_install.cmake:59 (include)
```

Apparently, there are some testing-related python files to be installed. Looking at *src/CMakeFiles.txt* again reveals that unsetting the `INSTALL_TESTS` options prevents this.

```
calculator$ echo '-DINSTALL_TESTS=0' >> cmake_args
calculator$ goa build

...
[calculator:cmake] -- Up-to-date: [...]/var/build/x86_64
  /install/bin/ubuntu-calculator-app
...
[calculator:cmake] -- Installing: [...]/var/build
  /x86_64/install/share/locale/zh_TW/LC_MESSAGES/com.ubuntu.calculator.mo
```

Very nice! You got past all build and installation errors. The environment for Ubuntu UI Toolkit apps is set up by the *ubuntu-ui-toolkit-launcher*, which expects the application files in its VFS. Since the VFS allows importing files from a tar archive, wrapping the application files into a tar archive is the best option. You can achieve this by adding the following line in the *artifacts* file. For more details, please refer to `goa help artifacts`:

```
ubuntu-calculator-app.tar: install/
```

After running `goa build` again, you will find the .tar-file in *var/bin/x86_64/*:

```
calculator$ goa build
...
calculator$ ls var/bin/x86_64
ubuntu-calculator-app.tar
```

Next task is defining the runtime scenario.

**Writing the package runtime** In order to run the just built component with Goa or on Sculpt, you need a corresponding package archive defining the runtime. Goa expects this to be defined by the *runtime* file in the *pkg/* subdirectory.

```
calculator$ mkdir pkg
```

Since the *runtime* file for Ubuntu UI Toolkit applications comprises mostly boilerplate code, you may use any existing application as blueprint and modify a few lines as indicated by the inline comments:

```
runtime | ram: 200M | caps: 1000 | binary: ubuntu-ui-toolkit-launcher
+ requires
  + gui
  + rom mesa_gpu.lib.so
  + gpu
  + rtc
  + timer
  + report shape

+ config
  + libc | stdout: /dev/log | stderr: /dev/log | pipe: /pipe | rtc: /dev/rtc
  + vfs
  | + dir dev
  |   + log
  |   + gpu
  |   + rtc
  | + dir .local
  |   + ram
  | + dir pipe
  |   + pipe
  | + tar qt5_declarative_qml.tar
  | + tar qt5_dejavusans.tar
  | + tar qt5_graphicaleffects_qml.tar
```

```
  | + tar qt5_libqgenode.tar
  | + tar qt5_libqjpeg.tar
  | + tar qt5_libqsvg.tar
  | + tar ubuntu-ui-toolkit_qml.tar
  | + tar ubuntu-themes.tar
  | + tar ubuntu-calculator-app.tar | . change to your projects tar file here

  + arg | : ubuntu-ui-toolkit-launcher
  + arg | : /share/ubuntu-calculator-app/ubuntu-calculator-app.qml
        | . add your startup QML file here
  + env QT_SCALE_FACTOR | : 1

+ content
  + rom ubuntu-calculator-app.tar | . adjust to your tar
-
```

With this *pkg/runtime* file set up, you are able to execute `goa run`. Note that Goa automatically executes all the required stages such as importing and building so that you don't need to worry about invoking these manually.

```
calculator$ goa run
...
[calculator] Error: Binary 'ubuntu-ui-toolkit-launcher' not mentioned as
             content ROM module.

 You either need to add '+ rom ubuntu-ui-toolkit-launcher' to the
 content ROM list
 or add a pkg archive to the 'archives' file from which to inherit.
```

Oops! We missed putting the Ubuntu UI Toolkit package archive into the *archives* file. Let's amend this:

```
calculator$ echo "ssumpf/pkg/ubuntu_ui_toolkit" > pkg/archives
calculator$ echo 'set version(ssumpf/pkg/ubuntu_ui_toolkit) 2026-04-22' \
>> goarc
calculator$ goa run
...
[init -> calculator] QQmlComponent: Component is not ready
[init -> calculator] file:///[...]/ubuntu-calculator-app.qml:23
                     module "QtQuick.Controls.Suru" is not installed
[init -> calculator]
[init -> calculator] QThread: Destroyed while thread is still running
[init -> calculator] Error: raise(ABRT)
[init] child "calculator" exited with exit value -1
```

Alright, Goa was actually able to start the scenario, yet the component seems to miss a QtQuick style module. The Suru style package is available at UBports[1].

In order to make Suru available on Genode, you need to create a separate Goa project.

**Porting QtQuick Controls Suru Style** As before you need to create a new project directory. Since Goa is able to look up dependencies from the current working directory, it makes perfect sense to create the new project as a subdirectory to the calculator project. Thus, create the project directory *calculator/qt5_quickcontrols2_suru/* with the following *import* file:

```
LICENSE   := GPLv2
VERSION   := main
DOWNLOADS := suru.git

URL(suru) := https://gitlab.com/ubports/development/core/qqc2-suru-style.git
REV(suru) := c0cf2007
DIR(suru) := src
```

These definitions create a clone of the specified git repository at the *src/* subdirectory during import. Create an empty *artifacts* file and give `goa run` a try:

```
qt5_quickcontrols2_suru$ touch artifacts
qt5_quickcontrols2_suru$ goa build
import  download https://gitlab.com/ubports/[...]/qqc2-suru-style.git
import  git Cloning into 'src'...
import  update src
import  generate import.hash
[qt5_quickcontrols2_suru] Error: build via qmake failed: unable to detect
                                 Qt version

Please add qt5_base or qt6_base to your 'used_apis' file.
```

Goa detected that this is a qmake project and is therefore looking for a corresponding API archive. Let's do as suggested:

```
qt5_quickcontrols2_suru$ echo "genodelabs/api/qt5_base" > used_apis
qt5_quickcontrols2_suru$ goa build
...
/[...]/depot/genodelabs/api/qt5_base/[...]/include/QtCore/qglobal.h:45:12:
      fatal error: type_traits: No such file or directory
   45 | #  include <type_traits>
      |            ^~~~~~~~~~~~
```

[1] https://gitlab.com/ubports/development/core/qqc2-suru-style

```
compilation terminated.
make[1]: *** [Makefile.suru:1175: .obj/qquicksurustyle.o] Error 1
make[1]: *** Waiting for unfinished jobs....
make[1]: *** [Makefile.suru:1425: .obj/qquicksuruanimations.o] Error 1
make[1]: *** [Makefile.suru:1598: .obj/qquicksuruunits.o] Error 1
make[1]: *** [Makefile.suru:1370: .obj/qquicksurutheme.o] Error 1
make: *** [Makefile:47: sub-qqc2-suru-suru-pro-make_first] Error 2
[qt5_quickcontrols2_suru] Error: build via qmake failed:
 child process exited abnormally
```

The build failed with the above error, which reminds us of adding `genodelabs/api/stdcxx` and `genodelabs/api/libc` to the *used_apis* file. Note that this may require adding the `--rebuild` argument to `goa build` to force Goa and qmake to re-create the build directory:

```
qt5_quickcontrols2_suru$ echo "genodelabs/api/stdcxx" >> used_apis
qt5_quickcontrols2_suru$ echo "genodelabs/api/libc" >> used_apis
qt5_quickcontrols2_suru$ goa build --rebuild
...
/[...]/depot/genodelabs/api/qt5_base/[...]/include/QtGui/qopengl.h:141:13:
      fatal error: GL/gl.h: No such file or directory
  141 | #   include <GL/gl.h>
      |             ^~~~~~~~
compilation terminated.
make[1]: *** [Makefile.suru:1370: .obj/qquicksurutheme.o] Error 1
make: *** [Makefile:47: sub-qqc2-suru-suru-pro-make_first] Error 2
[qt5_quickcontrols2_suru] Error: build via qmake failed:
 child process exited abnormally
```

Alright, this looks like we also need `genodelabs/api/mesa`:

```
qt5_quickcontrols2_suru$ echo "genodelabs/api/mesa" >> used_apis
qt5_quickcontrols2_suru$ goa build --rebuild
[qt5_quickcontrols2_suru:qmake] Info: creating stash file /[...]/.qmake.stash
/[...]/x86_64-pc-elf/bin/ld:
  cannot find -l:ldso_so_support.lib.a: No such file or directory
/[...]/x86_64-pc-elf/bin/ld:
...
```

Right, since we are building a shared library, we need `genodelabs/api/so`:

```
qt5_quickcontrols2_suru$ echo "genodelabs/api/so" >> used_apis
qt5_quickcontrols2_suru$ goa build --rebuild
...
[...]/ld: cannot find -l:qt5_component.lib.so: [...]
```

```
[...]/ld: cannot find [...]/qmake_root/lib/libQt5Quick.lib.so: [...]
[...]/ld: cannot find [...]/qmake_root/lib/libQt5QmlModels.lib.so: [...]
[...]/ld: cannot find [...]/qmake_root/lib/libQt5Qml.lib.so: [...]
[...]/ld: cannot find [...]/qmake_root/lib/libQt5QuickControls2.lib.so: [...]
[...]/ld: cannot find [...]/qmake_root/lib/libQt5QuickTemplates2.lib.so: [...]
[...]/ld: cannot find [...]/qmake_root/lib/libQt5Quick.lib.so: [...]
[...]/ld: cannot find [...]/qmake_root/lib/libQt5QmlModels.lib.so: [...]
[...]/ld: cannot find [...]/qmake_root/lib/libQt5Qml.lib.so: [...]
```

There are a bunch of library files missing. Goa creates these from the symbol files found in the used API archives. `qt5_component` is provided by `genodelabs/api/qt5_component`, and the Qt5 libraries are provided by `ssumpf/api/qt5_declarative` and `ssumpf/api/qt5_quickcontrols2`. The resulting *used_apis* file should therefore look like this:

```
genodelabs/api/qt5_base
genodelabs/api/stdcxx
genodelabs/api/libc
genodelabs/api/mesa
genodelabs/api/so
genodelabs/api/qt5_component
ssumpf/api/qt5_declarative
ssumpf/api/qt5_quickcontrols2/2023-05-26
```

Note that you can benefit from the version information already specified in the calculator's *goarc* file. Goa reads all *goarc* files it finds along the path from the project directory to your home directory. You may thus move the *goarc* file in the directory hierarchy to share it between both projects. Goa also allows specifying version information directly in the *used_apis* file as done for `ssumpf/api/qt5_quickcontrol2`.

After executing `goa build` successfully, you may have a look at the build directory at *var/build/x86_64* to identify the build artifacts. For QML modules, we need the qml files in a tar archive to be able to populate the ubuntu-ui-toolkit-launcher's VFS. Moreover, we need the **.lib.so* file. Your *artifacts* file should look like this:

```
qt5_quickcontrols2_suru_qml.tar/qt/: qmake_root/qml
qmake_root/qml/QtQuick/Controls.2/Suru/libqtquickcontrols2surustyleplugin.lib.so
```

Let's give `goa build` another try:

```
qt5_quickcontrols2_suru$ goa build
[qt5_quickcontrols2_suru] Error: missing symbols file
                          'libqtquickcontrols2surustyleplugin'
```

```
You can generate this file by running 'goa extract-abi-symbols'
```

Goa recognized that you are building a library and therefore expects a symbol file. Let's follow the advice given by Goa:

```
qt5_quickcontrols2_suru$ goa extract-abi-symbols
The following library symbols file(s) were created:
  > 'symbols/libqtquickcontrols2surustyleplugin
Please review the symbols files(s) and add them to your repository.
```

After removing the comment from the generated symbol file, you should be able to run `goa build` successfully. Let's try `goa export`:

```
qt5_quickcontrols2_suru$ goa export
[qt5_quickcontrols2_suru] Error: cannot export src archive because the
                          license is undefined

 Create a 'LICENSE' file for the project, or
 define 'set license <path>' in your goarc file, or
 specify '--license <path>' as argument.
```

Fortunately, Goa reminds us of adding a *LICENSE* file. Since the file is already present in the *src/* directory, you point Goa to it using this *goarc* line:

```
set license src/LICENSE.GPL-2
```

Let's run `goa export` again:

```
qt5_quickcontrols2_suru$ goa export
[qt5_quickcontrols2_suru] Error: version for
                          archive _/src/qt5_quickcontrols2_suru undefined

 Create a 'version' file in your project directory, or
 define 'set version(_/src/qt5_quickcontrols2_suru) <version>'
 in your goarc file.
```

Goa features a `bump-version` command to create/update the version file. It simply sets the version to the current date or appends/increases a letter suffix if the version was already set to this date.

```
qt5_quickcontrols2_suru$ goa bump-version
qt5_quickcontrols2_suru$ goa export
[qt5_quickcontrols2_suru] exported [...]/src/qt5_quickcontrols2_suru/...
[qt5_quickcontrols2_suru] exported [...]/bin/x86_64/qt5_quickcontrols2_suru/...
```

All done, back to the calculator project...

**Revising the package runtime** In order to utilize the just created Suru module, you need to add the tar file to the calculator *runtime* file. More precisely, add a `tar` node to the vfs and a `rom` node to the list of content ROM modules.

```
+ config
  + vfs
    ...
    + tar qt5_quickcontrols2_suru_qml.tar

+ content
    ...
  + rom qt5_quickcontrols2_suru_qml.tar
```

Before giving `goa run` a go, don't forget to add the corresponding depot archive to the *archives* file. You can use the wildcard depot user. When Goa is provided with the `--depot-user` argument, e. g. in the course of a `goa publish`, Goa will replace the wildcard depot user with the provided name:

```
calculator$ echo "_/src/qt5_quickcontrols2_suru" >> pkg/archives
calculator$ goa run
...
[qt5_quickcontrols2_suru] exported
  [...]/calculator/var/depot/_/src/qt5_quickcontrols2_suru/2026-05-21
[qt5_quickcontrols2_suru] exported
  [...]/calculator/var/depot/_/bin/x86_64/qt5_quickcontrols2_suru/2026-05-21
...
[init -> calculator] Error: stop because parent denied ROM-session:
 label="libqtquickcontrols2surustyleplugin.lib.so", [...]
```

Goa automatically exported the qt5_quickcontrols2_suru project into the *var/depot/* of the calculator project. When running the calculator, however, it fails to find the .lib.so file of the *qt5_quickcontrols2_suru*. Adding a corresponding `rom` node to the `content` in the *pkg/runtime* file should fix this:

```
+ content
  ...
  + rom libqtquickcontrols2surustyleplugin.lib.so
```

Giving `goa run` another shot reveals another issue:

```
calculator$ goa run
...
```

```
[init -> calculator] QSqlDatabase: QSQLITE driver not loaded
[init -> calculator] QSqlDatabase: available drivers:
[init -> calculator] Warning: chmod: chmod not implemented
[init -> calculator] QSqlQuery::prepare: database not open
[init -> calculator] file:///[...]/engine/CalculationHistory.qml:82:
                     Error: Driver not loaded Driver not loaded
```

Apparently, we need a database driver. Fortunately, *qt5_libsqlite.tar* and *libqsqlite.lib.so* are part of the *qt5_base* binary archive. Let's add them to the *runtime* file:

```
+ config
  + vfs
    ...
    + tar qt5_libqsqlite.tar

+ content
  ...
  + rom qt5_libqsqlite.tar
  + rom libqsqlite.lib.so
```

Finally, `goa run` is able to start up the calculator app successfully. The *fb_sdl* window may remain white though. A random mouse click into the window, however, lets the GUI pop up as shown below. That's good enough for now.


**Figure 14:** Calculator app running on base-linux with Goa

Unfortunately, you will notice substantial delays when interacting with the GUI due to the lack of hardware acceleration with Goa on Linux. Please refer to Section 4.4 for running Goa scenarios on a remote Sculpt target to mitigate this limitation.

The complete code is available in Johannes' goa-projects repository.

### Ported Lomiri Calculator App

https://codeberg.org/jschlatow/goa-projects/src/branch/main/ports/lomiri-calculator-app

## 5.5 Porting the curl command-line tool and library

*This section is based on an article[1] originally published at https://genodians.org.*

For more than a decade, there is a port of the curl library for Genode available. Yet, with the use of Sculpt OS as a daily driver, as well as the rise of Goa, porting the command-line tool is very appealing.

**Importing the curl source code** As always, let's start with a new project directory:

```
$ mkdir -p ports/curl
```

Since Goa mirrors the import tool from the Genode repository, you can take the existing *import* file as a blueprint. After updating the version and eliminating a few specifics of Genode's build system, your *import* file should look like this:

```
$ cat ports/curl/import
LICENSE   := curl
DOWNLOADS := curl.archive
VERSION   := 8.20.0

URL(curl) := https://curl.se/download/curl-$(VERSION).tar.gz
SHA(curl) := fc5819cad3f9f5482669adcdc49a782c15f36d2a0715b395b06d9173593d2dc0
DIR(curl) := src

PATCHES   := $(addprefix patches/,max_write_size.patch)
```

The patch file is taken from the Genode repository and is best placed into the *patches/* subdirectory. The file looks as follows:

```
ports/curl$ cat patches/max_write_size.patch
+++ src/include/curl/curl.h
@@ -247,6 +247,10 @@
 #define CURL_MAX_WRITE_SIZE 16384
 #endif

+/* Genode: override the default to foster the batching of network packets */
+#undef  CURL_MAX_WRITE_SIZE
+#define CURL_MAX_WRITE_SIZE 262144
+
 #ifndef CURL_MAX_HTTP_HEADER
 /* The only reason to have a max limit for this is to avoid the risk of a bad
    server feeding libcurl with a never-ending header that will cause reallocs!
```

[1] https://genodians.org/jschlatow/2025-03-07-porting-curl

With these prerequisites, you should be able to run `goa import`:

```
ports/curl$ goa import
import  extract curl-8.20.0.tar.gz (curl)
import  apply patches/max_write_size.patch
import  generate import.hash
```

**First successful build** Goa tries to detect the used build system by evaluating the presence of specific files in the *src/* directory (e. g. *Makefile*, *CMakeList.txt*). According to the build instructions for curl[1], the recommended way is to use `./configure` but there is also (partial) CMake support. In order to take the beaten track, let's remove the *CMakeList.txt* file and give `goa build` a try:

```
ports/curl$ rm src/CMakeList.txt
ports/curl$ goa build
Error: ports/curl has a 'src' directory but lacks an 'artifacts' file.
  You may start with an empty file.
```

Doing as suggested and retrying:

```
ports/curl$ touch artifacts
ports/curl$ goa build
[...]
configure: error: select TLS backend(s) or disable TLS with --without-ssl.

Select from these:

  --with-amissl
  --with-bearssl
  --with-gnutls
  --with-mbedtls
  --with-openssl (also works for BoringSSL and libressl)
  --with-rustls
  --with-schannel
  --with-secure-transport
  --with-wolfssl
```

Alright, curl requires us to select a TLS backend. Genode's port of OpenSSL is restricted to version 1.1.1, which is no longer supported by curl. However, an up-to-date port of GnuTLS is available in the genode-world repository. You may either build it yourself or use the prebuilt archives from jschlatow's depot. In the scope of this tutorial, let's opt for the latter option.

[1] https://curl.se/docs/install.html

For reference, exporting the gnutls archives as user "john" into the *var/depot* of your curl project is done by the following commands:

```
$ git clone https://codeberg.org/genodelabs/genode-world
$ cd genode-world
genode-world$ goa export -C network/gnutls \
  --depot-dir /path/to/curl/var/depot \
  --public-dir /path/to/curl/var/public \
  --depot-user john
...
[gnutls] exporting project [...]/genode-world/crypto/nettle
[nettle] exported curl/var/depot/john/api/nettle/3.7-2026-05-22
[nettle] exported curl/var/depot/john/src/nettle/3.7-2026-05-22
[nettle] exported curl/var/depot/john/bin/x86_64/nettle/3.7-2026-05-22
...
[gnutls] exported curl/var/depot/john/api/gnutls/3.6.16-2026-05-22
[gnutls] exported curl/var/depot/john/src/gnutls/3.6.16-2026-05-22
[gnutls] exported curl/var/depot/john/bin/x86_64/gnutls/3.6.16-2026-05-22
```

As you can see, Goa will conveniently export the nettle project as a dependency of gnutls as well.

Back in the *curl* project directory, let's instead add *jschlatow/api/gnutls* and *jschlatow/api/nettle* along with the official *libc* and *posix* archives to the *used_apis* file:

```
ports/curl$ echo "genodelabs/api/libc"  > used_apis
ports/curl$ echo "genodelabs/api/posix" >> used_apis
ports/curl$ echo "jschlatow/api/gnutls" >> used_apis
ports/curl$ echo "jschlatow/api/nettle" >> used_apis
```

You also need to guide curl to use GnuTLS. In Goa, you can supply command-line arguments to `./configure` by adding them to a *configure_args* file:

```
ports/curl$ echo "--with-gnutls" > configure_args
ports/curl$ goa build
[curl] Error: no version defined for depot archive 'jschlatow/api/gnutls'
```

Goa needs to know the version of the *gnutls* archive. Luckily, Goa is able to lookup version information from a depot user's index. Let's tell Goa to download depot index files:

```
ports/curl$ goa build --update-index
download jschlatow/index/26.04.xz
download jschlatow/index/26.04.xz.sig
download jschlatow/api/gnutls/3.6.16-2026-05-22.tar.xz
```

```
download jschlatow/api/gnutls/3.6.16-2026-05-22.tar.xz.sig
download jschlatow/api/nettle/3.7-2026-05-22.tar.xz
download jschlatow/api/nettle/3.7-2026-05-22.tar.xz.sig
...
[curl:autoconf] checking for psl_builtin in -lpsl... no
configure: error: libpsl libs and/or directories were not found
                  where specified!
```

Curl is unable to locate libpsl. Let's try telling it to not use it:

```
ports/curl$ echo "--without-libpsl" >> configure_args
ports/curl$ goa build
[...]
vtls/gtls.c: In function 'gtls_client_init':
vtls/gtls.c:927:46: error: 'GNUTLS_NO_END_OF_EARLY_DATA' undeclared [...]
 927 | init_flags |= GNUTLS_ENABLE_EARLY_DATA | GNUTLS_NO_END_OF_EARLY_DATA;
     |                                         ^~~~~~~~~~~~~~~~~~~~~~~~~~~
     |                                         GNUTLS_ENABLE_EARLY_DATA
```

This error looks strange. After a bit of investigation, it turns out that `GNUTLS_NO_END_OF_EARLY_DATA` was introduced with GnuTLS 3.7.2 whereas the current port in genode-world is based on 3.6.16. Apparently, the following version check in curl's *lib/vtls/gtls.c* is a bit off:

```
#if GNUTLS_VERSION_NUMBER >= 0x03060d
#define CURL_GNUTLS_EARLY_DATA
#endif
```

Note that this bug has been reported to the curl project and will therefore be fixed in the next release. In the meantime, you can correct this by replacing the `0x03060d` by `0x030702`. Let's try again after that:

```
ports/curl$ goa build
[...]
In file included from tool_operate.c:77:
tool_xattr.h:34:12: fatal error: sys/extattr.h: No such file or directory
   34 | #  include <sys/extattr.h>
```

Looking at *src/src/tool_xattr.h*, you may notice the following lines:

```
#elif (defined(__FreeBSD_version) && (__FreeBSD_version > 500000)) || \
      defined(__MidnightBSD_version)
#  include <sys/types.h>
#  include <sys/extattr.h>
```

```
#  define USE_XATTR
```

In spite of being based on FreeBSD's libc, Genode's C-runtime does not have a *sys/extattr.h*. Let's try again after removing the lines above.

```
ports/curl$ goa build
[curl:make] Making all in lib
[curl:make] Making all in docs
[curl:make] Making all in .
[curl:make] Making all in cmdline-opts
[curl:make] Making all in libcurl
[curl:make] Making all in opts
[curl:make] Making all in src
[curl:make]   CC       curl-tool_operate.o
[curl:make]   CC       curl-tool_stderr.o
[curl:make]   CC       curl-tool_strdup.o
[curl:make]   CC       curl-tool_urlglob.o
[curl:make]   CC       curl-tool_util.o
[curl:make]   CC       curl-tool_vms.o
[curl:make]   CC       curl-tool_writeout.o
[curl:make]   CC       curl-tool_writeout_json.o
[curl:make]   CC       curl-tool_xattr.o
[curl:make]   CC       curl-var.o
[curl:make]   HUGE     tool_hugehelp.c
[curl:make]   CC       ../lib/curl-base64.o
/bin/sh: line 2: tool_hugehelp.c: Read-only file system
/bin/sh: line 3: tool_hugehelp.c: Read-only file system
/bin/sh: line 4: tool_hugehelp.c: Read-only file system
```

Oops, apparently there is a write access to *tool_hugetable.c*. Since the file is present in the *src/* directory, it is provided by Goa as a read-only file to the sandboxed build environment. Looking into the code clarifies that the file is auto-generated, hence you can simply remove it from *src/*

```
ports/curl$ rm src/src/tool_hugehelp.c
ports/curl$ goa build
[curl:make] Making all in lib
[curl:make] Making all in docs
[curl:make] Making all in .
[curl:make] Making all in cmdline-opts
[curl:make] Making all in libcurl
[curl:make] Making all in opts
[curl:make] Making all in src
[curl:make]   CC       ../lib/curl-curl_get_line.o
[curl:make]   CC       ../lib/curl-curl_multibyte.o
```

```
[curl:make]   CC       ../lib/curl-dynbuf.o
[curl:make]   CC       ../lib/curl-nonblock.o
[curl:make]   CC       ../lib/curl-strtoofft.o
[curl:make]   CC       ../lib/curl-timediff.o
[curl:make]   CC       ../lib/curl-version_win32.o
[curl:make]   CC       ../lib/curl-warnless.o
[curl:make]   HUGE     tool_hugehelp.c
[curl:make]   CC       curl-tool_ca_embed.o
[curl:make]   CC       curl-tool_hugehelp.o
[curl:make]   CCLD     curlinfo
[curl:make]   CCLD     curl
[curl:make]   CCLD     libcurltool.la
[curl:make] Making all in scripts
[curl:make] Making install in lib
...
```

**Refining the configuration** Having completed the first successful build, it's time to review the configuration. The output of `goa build --rebuild` contains the status summary of `./configure`:

```
[curl:autoconf]   curl version:     8.20.0
[curl:autoconf]   SSL:              enabled (GnuTLS)
[curl:autoconf]   SSH:              no       (--with-{libssh,libssh2})
[curl:autoconf]   zlib:             no       (--with-zlib)
[curl:autoconf]   brotli:           no       (--with-brotli)
[curl:autoconf]   zstd:             no       (--with-zstd)
[curl:autoconf]   GSS-API:          no       (--with-gssapi)
[curl:autoconf]   GSASL:            no       (--with-gsasl)
[curl:autoconf]   TLS-SRP:          enabled
[curl:autoconf]   resolver:         POSIX threaded
[curl:autoconf]   IPv6:             enabled
[curl:autoconf]   Unix sockets:     enabled
[curl:autoconf]   IDN:              no       (--with-{libidn2,winidn})
[curl:autoconf]   Build docs:       enabled  (--disable-docs)
[curl:autoconf]   Build libcurl:    Shared=no, Static=yes
[curl:autoconf]   Built-in manual:  enabled
[curl:autoconf]   --libcurl option: enabled  (--disable-libcurl-option)
[curl:autoconf]   Type checking:    enabled  (--disable-typecheck)
[curl:autoconf]   Verbose errors:   enabled  (--disable-verbose)
[curl:autoconf]   Code coverage:    disabled
[curl:autoconf]   SSPI:             no       (--enable-sspi)
[curl:autoconf]   ca native:        no
[curl:autoconf]   ca cert bundle:   no
[curl:autoconf]   ca cert path:     no
[curl:autoconf]   ca cert embed:    no
```

```
[curl:autoconf]   ca fallback:      no
[curl:autoconf]   LDAP:             no       (--enable-ldap /
                                              --with-ldap-lib /
                                              --with-lber-lib)
[curl:autoconf]   LDAPS:            no       (--enable-ldaps)
[curl:autoconf]   IPFS/IPNS:        enabled
[curl:autoconf]   RTSP:             enabled
[curl:autoconf]   PSL:              no       (--with-libpsl)
[curl:autoconf]   Alt-svc:          enabled  (--disable-alt-svc)
[curl:autoconf]   Headers API:      enabled  (--disable-headers-api)
[curl:autoconf]   HSTS:             enabled  (--disable-hsts)
[curl:autoconf]   HTTP1:            enabled  (internal)
[curl:autoconf]   HTTP2:            no       (--with-nghttp2)
[curl:autoconf]   HTTP3:            no       (--with-ngtcp2
                                              --with-nghttp3,
                                              --with-quiche)
[curl:autoconf]   ECH:              no      (--enable-ech)
[curl:autoconf]   HTTPS RR:         no      (--enable-httpsrr)
[curl:autoconf]   SSLS-EXPORT:      no      (--enable-ssls-export)
[curl:autoconf]   Protocols:        dict file ftp ftps gopher gophers http
                                    https imap imaps ipfs ipns mqtt mqtts
                                    pop3 pop3s rtsp smtp smtps telnet tftp
                                    ws wss
[curl:autoconf]   Features:         alt-svc AsynchDNS HSTS HTTPS-proxy
                                    IPv6 Largefile SSL threadsafe TLS-SRP
                                    UnixSockets
```

Apparently, the CA cert bundle and path are missing. Having a TLS-enabled binary would be much more useful if there was a way for curl to use a CA bundle. Calling `./configure --help` in the *src/* subdirectory clarifies what arguments to use for this. Let's add them to the *configure_args* file:

```
ports/curl$ echo "--with-ca-path=/etc/ssl/certs" >> configure_args
ports/curl$ echo "--with-ca-bundle=/etc/ssl/certs/ca-certificates.crt" \
  >> configure_args
```

In order to apply the modified arguments, you need to force Goa into recreating the build directory by adding the `--rebuild` switch:

```
ports/curl$ goa build --rebuild
[...]
[curl:autoconf]   ca cert bundle:   /etc/ssl/certs/ca-certificates.crt
[curl:autoconf]   ca cert path:     /etc/ssl/certs
[...]
```

There is another optional dependency that may raise your attention: zlib. Since there already exist a Genode port for this library, simply add the corresponding dependency to the *used_apis* file and supplement the *configure_args*:

```
ports/curl$ echo "genodelabs/api/zlib" >> used_apis
ports/curl$ echo "--with-zlib" >> configure_args
ports/curl$ goa build --rebuild
[curl:autoconf]
[curl:autoconf]   curl version:     8.20.0
[curl:autoconf]   SSL:              enabled (GnuTLS)
[curl:autoconf]   SSH:              no       (--with-{libssh,libssh2})
[curl:autoconf]   zlib:             enabled
[curl:autoconf]   brotli:           no       (--with-brotli)
[curl:autoconf]   zstd:             no       (--with-zstd)
[curl:autoconf]   GSS-API:          no       (--with-gssapi)
[curl:autoconf]   GSASL:            no       (--with-gsasl)
...
```

Note that `./configure` makes use of pkg-config in order to detect certain library dependencies. The *genodelabs/api/zlib* archive therefore includes a *zlib.pc* file, which indicates the availability of the zlib library in the build environment set up by Goa.

```
ports/curl$ cat var/depot/genodelabs/api/zlib/2026-04-16/zlib.pc
Name: zlib
Description: zlib compression library
Version: 1.3.2
Libs: -l:zlib.lib.so
```

**Capturing local changes as patches** Having modified the content of *src/*, you also need to capture these changes as patches so that `goa import` reproduces the same result. Goa conveniently provides the `goa diff` command to inspect the changes. The easiest way would be to redirect the entire output to a single patch file and modify the *import* file to include this patch.

```
ports/curl$ goa diff > patches/all_changes.patch
```

For better structure, however, you may split the changes into separate patch files.

**Exporting the binary archive** Before exporting a binary archive, you need to define what artifacts shall be included. For command-line tools that not only come as a singular binary, you can package all required files into a tar container. Goa assists selecting the required files by automatically executing `make install` (if available), which places these files into the *install/* subdirectory of the build directory. For curl, let's include *in-*

*stall/bin* and *install/share* into a *curl.tar* container as follows (see `goa help artifacts` for more details):

```
ports/curl$ echo "curl.tar: install/bin" > artifacts
ports/curl$ echo "curl.tar: install/share" >> artifacts
```

Now, you can try to export the archive:

```
ports/curl$ goa export
[...]
[curl] Error: cannot export src or api archive because the license is undefined

 Create a 'LICENSE' file for the project, or
 define 'set license <path>' in your goarc file, or
 specify '--license <path>' as argument.
```

Since curl presents its license in the *COPYING* file, you may define the license path in the project's *goarc* file:

```
ports/curl$ echo "set license src/COPYING" >> goarc
ports/curl$ goa export
[...]
[curl] Error: version for archive _/src/curl undefined

 Create a 'version' file in your project directory, or
 define 'set version(_/src/curl) <version>' in your goarc file,
 or specify '--version-_/src/curl <version>' as argument
```

Goa comes with the `goa bump-version` command to populate the *version* file:

```
ports/curl$ goa bump-version
ports/curl$ goa export
[...]
[curl] exported ports/curl/var/depot/_/src/curl/2026-05-22
[curl] exported ports/curl/var/depot/_/bin/x86_64/curl/2026-05-22
```

The exported binary archive contains the *curl.tar* container that can be imported into a VFS using a `tar` node. Along with a CA bundle at */etc/ssl/certs/ca-certificates.crt*, you can run a bash on this VFS and use the curl command-line tool. Note that this also requires the enablement of network for the bash. An example is available in Johannes Schlatow's unix_shell runtime[1].

[1] https://codeberg.org/jschlatow/goa-projects/src/branch/main/apps/unix_shell

**Building the shared library** This is where the porting-ride becomes a bit rough. Looking at *install/lib/* in the project's build directory indicates that `goa build` has not built a shared library, yet. Moreover, the following line of the build output stands out:

```
[curl:autoconf]   Build libcurl:    Shared=no, Static=yes
```

Unfortunately, adding `--enable-shared` to *configure_args* does not change this. Looking at and debugging *src/configure* reveals that the script is unable to determine the dynamic linker. The following patch rectifies this:

```
    shlibpath_var=LD_LIBRARY_PATH
    ;;
.
+genode*)
+  dynamic_linker="Genode ld.lib.so"
+  shrext_cmds=.lib.so
+  libname_spec='$name'
+  library_names_spec='$libname$shared_ext'
+  ;;
+
  *)
    dynamic_linker=no
    ;;
```

By default, Goa provides the `./configure` command with `--host=x86_64-pc-elf`. As this is not known to the script, it lands in the `*)` case, setting `dynamic_linker=no`. In order to set the dynamic linker correctly, the patch adds the `genode*)` case and also sets the library name so that the build creates the file *curl.lib.so*. In order to make use of this modification, you must also add `--host=x86_64-pc-genode` to the *configure_args*. You also need to extend the *used_apis* with "genodelabs/api/so":

```
ports/curl$ echo "-host=x86_64-pc-genode" >> configure_args
ports/curl$ echo "genodelabs/api/so" >> used_apis
ports/curl$ goa build --rebuild
[...]
[curl:autoconf]   Build libcurl:    Shared=yes, Static=yes
[...]
ports/curl$ ls var/build/x86_64/install/lib/
curl.a  curl.lib.so  libcurl.la  pkgconfig
```

Wow, it built successfully. That almost seems too easy. Anyway, let's add the library to the *artifacts* file, create an *api* file and export the new archive:

```
ports/curl$ echo install/lib/curl.lib.so >> artifacts
ports/curl$ echo install/include/curl/ > api
ports/curl$ goa export
[...]
[curl] Error: missing symbols file 'curl'

 You can generate this file by running 'goa extract-abi-symbols'
```

Well, we forgot to create the symbol file. Fortunately, Goa reminds us of this fact as it detected a shared library file in the build artifacts.

```
ports/curl$ goa extract-abi-symbols
The following library symbols file(s) were created:
  > 'symbols/curl
Please review the symbols files(s) and add them to your repository.
```

After reviewing the file and removing the comment at the very first line, you can give `goa export` a try:

```
ports/curl$ goa export
[...]
[curl] exported ports/curl/var/depot/_/api/curl/2026-05-22
[curl] exported ports/curl/var/depot/_/src/curl/2026-05-22
[curl] exported ports/curl/var/depot/_/bin/x86_64/curl/2026-05-22
```

Unfortunately, when testing the command-line tool, you will notice that building the shared library broke the command-line tool because it was linked against *../lib/.libs/curl.lib.so*:

```
ports/curl$ ldd var/build/x86_64/install/bin/curl
       linux-vdso.so.1 (0x0000799381ee4000)
       ../lib/.libs/curl.lib.so (0x0000799381e39000)
[...]
```

Executing this binary, the runtime fails to open a ROM session with label "../lib/.libs/curl.lib.so". You can either apply label-rewriting or resort to fixing the root cause by tricking libtool into preferring static linking for the libtool-managed libraries. The following patch of *src/src/Makefile.in* adds the `-static-libtool-libs` option to the corresponding command:

```
ports/curl$ goa diff
+++ src/src/Makefile.in
@@ -366,7 +366,7 @@
 curl_DEPENDENCIES = $(top_builddir)/lib/libcurl.la
```

```
 curl_LINK = $(LIBTOOL) $(AM_V_lt) --tag=CC $(AM_LIBTOOLFLAGS) \
   $(LIBTOOLFLAGS) --mode=link $(CCLD) $(AM_CFLAGS) $(CFLAGS) \
-  $(curl_LDFLAGS) $(LDFLAGS) -o $@
+  $(curl_LDFLAGS) $(LDFLAGS) -static-libtool-libs -o $@
 am_curlinfo_OBJECTS = curlinfo.$(OBJEXT)
 curlinfo_OBJECTS = $(am_curlinfo_OBJECTS)
 curlinfo_LDADD = $(LDADD)
```

**A simple test project** In order to give the newly exported library a spin, you may create a test project that imports the *simple.c* from curl. Creating a new subdirectory *test/* with the following *import* file does the trick:

```
LICENSE   := curl
DOWNLOADS := curl.archive
VERSION   := 8.20.0

URL(curl) := https://curl.se/download/curl-$(VERSION).tar.gz
SHA(curl) := fc5819cad3f9f5482669adcdc49a782c15f36d2a0715b395b06d9173593d2dc0
DIR(curl) := tmp

PATCHES   := $(addprefix patches/,Makefile.patch \
                                  no_ssl.patch)

DIRS := src
DIR_CONTENT(src) := tmp/docs/examples/simple.c
```

Note that it uses a somewhat quirky trick of the import tool: Instead of unpacking the source code into *src/*, it uses *tmp/* and manually defines the directory content of *src/*. This way, you are able to only extract the *simple.c*. Since this example uses SSL, which we do not want to deal with at the moment, you may simply change the URL in *example.c* (done by *no_ssl.patch*). You can also add a custom Makefile via a separate patch. The Makefile looks as follows:

```
ports/curl$ cat test/src/Makefile
test-curl: simple
  @mv simple test-curl
```

You are able to build the test application with the following *used_apis* and *artifacts* files:

```
ports/curl$ cat test/used_apis
genodelabs/api/libc
genodelabs/api/posix
```

```
_/api/curl
```

```
ports/curl$ cat test/artifacts
test-curl
```

Let's try running the test application from within the *curl/* directory:

```
ports/curl$ goa run -C test
[test] exporting project ../curl
[curl] Error: no version defined for depot archive 'jschlatow/api/gnutls'
[test] Error: failed to export depot archive _/api/curl/2026-05-22
```

Goa detected that `_/api/curl` is provided by the project in our *curl/* directory and tries to export it into the *test/var/depot/* subdirectory. However, it failed to get the version information for `jschlatow/api/gnutls`. Let's try again with `--update-index`:

```
ports/curl$ goa run -C test --update-index
[...]
[curl] exported [...]/curl/test/var/depot/_/api/curl/2026-05-22
[curl] exported [...]/curl/test/var/depot/_/src/curl/2026-05-22
[curl] exported [...]/curl/test/var/depot/_/bin/x86_64/curl/2026-05-22
[test] Error: no runtime defined at [...]/curl/test/pkg
```

Of course, we need to define the runtime scenario by adding a *test/pkg/archives* and a *test/pkg/runtime* file.

The *test/pkg/archives* file:

```
genodelabs/src/vfs
genodelabs/src/vfs_lxip
genodelabs/src/vfs_pipe
genodelabs/src/libc
genodelabs/src/posix
genodelabs/src/zlib
genodelabs/src/gmp
jschlatow/src/gnutls
jschlatow/src/nettle
_/src/curl
```

The *test/pkg/runtime* file:

```
runtime | ram: 12M | caps: 300 | binary: test-curl
+ requires
  + nic
  + timer
```

```
+ config
  + libc | stdout: /log | stderr: /log | socket: /sockets
         | pipe: /pipe  | rtc: /rtc
  + vfs
    + log
    + dir pipe
    | + pipe
    + dir sockets
      + lxip | dhcp: yes
    + inline rtc | : 2026-05-22 00:00
+ content
  + rom | label: test-curl
  + rom | label: libc.lib.so
  + rom | label: libm.lib.so
  + rom | label: gmp.lib.so
  + rom | label: posix.lib.so
  + rom | label: curl.lib.so
  + rom | label: gnutls.lib.so
  + rom | label: nettle.lib.so
  + rom | label: lxip.lib.so
  + rom | label: vfs.lib.so
  + rom | label: vfs_lxip.lib.so
  + rom | label: vfs_pipe.lib.so
  + rom | label: zlib.lib.so
-
```

Let's give this a spin:

```
ports/curl$ goa run -C test
[...]
[init -> test] Using DHCP for interface configuration.
[init -> test] Sending DHCP requests ., OK
[init -> test] IP-Config: Got DHCP answer from 10.0.10.1 ...
[init -> test] IP-Config: Complete:
[init -> test]      device=eth0, hwaddr=02:02:02:02:02:01,
                    ipaddr=10.0.10.2, mask=255.255.255.0, gw=10.0.10.1
[init -> test]      host=10.0.10.2, domain=, nis-domain=
[init -> test]      bootserver=10.0.10.1, rootserver=10.0.10.1,
                    rootpath=nameserver0=10.1.10.3
[init -> test] Warning: Libc RNG not configured
[init -> test] curl_easy_perform() failed: Could not connect to server
[init] child "test" exited with exit value 0
```

Our test program got an IP address but failed to connect to the server. Unfortunately, curl is not very verbose about the reason. You can change this by adding this `curl_easy_setopt()` call to *test/src/simple.c*:

```
curl_easy_setopt(curl, CURLOPT_VERBOSE, 1L);
```

Let's run it again:

```
ports/curl$ goa run -C test
[...]
[init -> test] * Host example.com:80 was resolved.
[init -> test] * IPv6: (none)
[init -> test] * IPv4: 104.20.23.154, 172.66.147.243
[init -> test] * setsockopt enable SO_NOSIGPIPE: Protocol not available
[init -> test] * connect to  port 0 from  port 0 failed: No error: 0
[init -> test] * setsockopt enable SO_NOSIGPIPE: Protocol not available
[init -> test] * connect to  port 0 from  port 0 failed: No error: 0
[init -> test] * Failed to connect to example.com port 80 after 5845 ms:
                 Could not connect to server
[init -> test] * closing connection #0
[init -> test] curl_easy_perform() failed: Could not connect to server
```

It looks as if `SO_NOSIGPIPE` is not supported on Genode. Looking into the curl code reveals that *src/lib/curl_setup.h* defines `USE_SO_NOSIGPIPE`. You can prevent this with the following patch:

```
+++ src/lib/curl_setup.h
@@ -477,7 +477,7 @@
 #define USE_EVENTFD
 #endif

-#ifdef SO_NOSIGPIPE
+#if defined(SO_NOSIGPIPE) && !defined(__GENODE__)
 #define USE_SO_NOSIGPIPE
 #endif
```

In order to re-run the test scenario, you must make sure that the curl library is re-build and exported with the modified source code. The curl library was initially exported as a dependency of the test project and exported into the depot directory used by the latter. If an archive is already present in the depot, Goa will not re-build the corresponding project. Bumping the version is an easy way to trigger this.

Note that regular Goa users typically use a depot directory that is shared between multiple projects by defining the `depot_dir` and `public_dir` variables in a *~/goarc* file. With a shared depot directory, a `goa export && goa run -C test` will also do the trick. As you have not set up a shared depot directory in this tutorial, let's use the version bump:

```
ports/curl$ goa bump-version
ports/curl$ goa run -C test
[...]
[init -> test] <!doctype html><html lang="en"><head>
               <title>Example Domain</title>
               <meta name="viewport" content=[...]>
               <style>[...]</style></head>
               <body><div><h1>Example Domain</h1>
               <p>This domain is for use in documentation examples
                  without needing permission. Avoid use in operations.</p>
               [...]
[init] child "test" exited with exit value 0
```

There it is! The test application using curl.lib.so works perfectly.

**Tweaking the API archive** In order to work as a replacement for the official *genode-labs/api/curl* archive, you may add a *libcurl.pc* file. This file assist pkg-config in detecting the presence of the curl library whenever it's mentioned in the *used_apis* file. The *libcurl.pc* is actually built along with the library based on the *libcurl.pc.in* file. You may patch the file to discard irrelevant information...

```
@@ -22,10 +22,6 @@
 #
 ###########################################################################
.
-prefix=@prefix@
-exec_prefix=@exec_prefix@
-libdir=@libdir@
-includedir=@includedir@
 supported_protocols="@SUPPORT_PROTOCOLS@"
 supported_features="@SUPPORT_FEATURES@"
.
@@ -35,7 +31,4 @@
 Version: @CURLVERSION@
 Requires: @LIBCURL_PC_REQUIRES@
 Requires.private: @LIBCURL_PC_REQUIRES_PRIVATE@
-Libs: -L${libdir} -lcurl @LIBCURL_PC_LIBS@
+Libs: -lcurl @LIBCURL_PC_LIBS@
-Libs.private: @LIBCURL_PC_LDFLAGS_PRIVATE@ @LIBCURL_PC_LIBS_PRIVATE@
-Cflags: -I${includedir} @LIBCURL_PC_CFLAGS@
-Cflags.private: @LIBCURL_PC_CFLAGS_PRIVATE@
```

...and add the file to the *api* artifacts:

```
ports/curl$ echo "libcurl.pc" >> api
```

Congratulations! The complete code is available in Johannes Schlatow's goa-projects repository.

#### Ported Curl executable and binary

https://codeberg.org/jschlatow/goa-projects/src/branch/main/ports/curl

## 5.6 Further reading

More tutorials and stories around Goa are available on the federated blog *genodians.org* or, maybe, you're ready to share you own story by now...

### Goa-related articles on genodians.org

http://genodians.org/topics-goa

## 6 Changelog

### 26.04

- Updated Sculpt version and URLs.
- Translated XML into HID.
- Updated repository URLs due to migration from GitHub to Codeberg.
- Applied flattened project structure (no subdirectories under *pkg/*)
- Section 2: Updated cmake_step2 (now: cmake_step3).
- Section 3.3.3: Updated block plugin, added ram_log.
- Section 3.4: Add libslirp-based virtual networking.
- Section 4: Mention *profile* library.
- Section 5: List content for every tutorial.
- Section 5.1: Minor updates in tutorial.
- Section 5.2: Sequoia PGP instructions, export with wildcard user, moved `--depot-overwrite` instructions to the end, updated Sculpt instructions.
- Section 5.3: Applied `Vfs` namespace reorganisation.
- Section 5.4: Make use of wildcard depot user and subprojects.
- Section 5.5: Updated to curl 8.20.0, changed instructions to use GnuTLS instead of OpenSSL.

### 25.04

- Updated Sculpt version and URLs.
- Section 2: Updated install instruction (bubblewrap and optional toolchain). Minor updates in tutorial.
- Section 3.1: Added ram attribute.
- Section 3.3.3: Updated lxip and lwip, added xoroshiro.
- Section 3.4: Added example for IP forwarding and NATing on host system.
- Section 4: Changed depot user to “john”.
- Section 5: Added “Further Reading” subsection.

- Section 5.1: Minor updates in tutorial.
- Section 5.2: Minor updates in tutorial.
- Section 5.3: Updated to Sculpt 25.04.
- Section 5.4: Updated tutorial due to Goa sandboxing.
- Added Section 5.5.

### 24.11 Initial version