

GENODE

Operating System Framework 24.11

Applications

Johannes Schlatow

Contents

1	Introduction	4
2	Getting started with Goa	5
2.1	Installation	6
2.2	A first example, using a plain old Makefile	7
2.3	A second example, using CMake	11
2.4	Running the scenario on Sculpt OS	16
3	Foundations	17
3.1	Genode's init component	18
3.2	Component API	20
3.2.1	Native Genode components	20
3.2.2	Libc components	20
3.2.3	POSIX components	21
3.3	C runtime and virtual file system	22
3.3.1	Libc configuration	22
3.3.2	VFS configuration	23
3.3.3	VFS plugins	24
3.4	Networking	29
3.4.1	TCP/IP stacks	29
3.4.2	NIC Router	30
3.4.3	Example: Virtual networking with Goa	32
3.4.4	Example: Cascaded NIC routers	33
3.5	Package management	35
3.6	Runtime configuration	37
3.7	Graphical User Interfaces	38
3.7.1	SDL	38
3.7.2	Qt (5/6)	38
3.7.3	Mobile SDK based on Ubuntu/Lomiri UI Toolkit	39
3.7.4	Light and Versatile Graphics Library (LVGL)	39
4	Development & Debugging	41
4.1	Adding debug info files	42
4.2	Using backtraces	43
4.3	Debugging with Goa on base-linux	48
4.4	Using Sculpt as a remote test target	53
4.5	Further reading	57
4.5.1	Using a VNC server on a remote test target	57
4.5.2	On-target debugging with GDB	57
4.5.3	Performance analysis	57

5 Tutorials	58
5.1 Sticking together a little Unix	59
5.2 Exporting and publishing	86
5.3 Writing a VFS plugin for network-packet access	97
5.4 Porting Lomiri Calculator App	119



This work is licensed under the Creative Commons Attribution + ShareAlike License (CC-BY-SA). To view a copy of the license, visit <http://creativecommons.org/licenses/by-sa/4.0/legalcode>

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://github.com/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-24-10>

2 Getting started with Goa

This section is based on Norman Feske's [Goa article](https://genodians.org)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/2019-11-24-go>

²<https://github.com/genodelabs/goa>

2.1 Installation

1. Install the latest Genode tool chain on a GNU/Linux OS on a 64-bit x86 PC. It is recommended to use the latest long-term support (LTS) version of Ubuntu. 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,
- *xmllint* for validating configurations,

Instructions for installing the Genode tool chain are available at <https://genode.org/download/tool-chain>.

2. Clone the Goa repository:

```
git clone https://github.com/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. 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
hello$ cd 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/2023-10-24.tar.xz
download genodelabs/api/base/2023-10-24.tar.xz.sig
```

Goa automatically downloaded the base API and installed it into a fresh depot at `var/depot/genodelabs/api/base/2023-10-24/`. 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
```


2.2 A first example, using a plain old Makefile

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. Let’s create one with the name “hello”:

```
hello$ mkdir -p pkg/hello
```

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. Create a file `pkg/hello/runtime` with the following content:

```
<runtime ram="1M" caps="100" binary="hello">
  <config/>
  <content>
    <rom label="hello"/>
  </content>
</runtime>
```

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/2023-10-24.tar.xz
download genodelabs/bin/x86_64/base-linux/2023-10-24.tar.xz.sig
download genodelabs/bin/x86_64/init/2023-10-24.tar.xz
download genodelabs/bin/x86_64/init/2023-10-24.tar.xz.sig
download genodelabs/src/base-linux/2023-10-24.tar.xz
download genodelabs/src/base-linux/2023-10-24.tar.xz.sig
download genodelabs/src/init/2023-10-24.tar.xz
download genodelabs/src/init/2023-10-24.tar.xz.sig
download genodelabs/api/os/2023-08-21.tar.xz
download genodelabs/api/os/2023-08-21.tar.xz.sig
download genodelabs/api/report_session/2023-05-26.tar.xz
download genodelabs/api/report_session/2023-05-26.tar.xz.sig
download genodelabs/api/sandbox/2023-10-03.tar.xz
download genodelabs/api/sandbox/2023-10-03.tar.xz.sig
download genodelabs/api/timer_session/2023-10-03.tar.xz
download genodelabs/api/timer_session/2023-10-03.tar.xz.sig
Genode sculpt-23.10
17592186044415 MiB RAM and 18997 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 2nd step of the excellent [CMake tutorial](#)¹. Let's call the project "cmake_step2". Instead of copying the code into the `cmake_step2/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_step2/import` with the following content. Please have a look at `goa help import` for a detailed explanation.

```
LICENSE := BSD
VERSION := master
DOWNLOADS := cmake_step2.sparse-git

URL(cmake_step2) := https://github.com/Kitware/CMake
REV(cmake_step2) := HEAD
DIR(cmake_step2) := src
SPARSE_PATH(cmake_step2) := Help/guide/tutorial/Step2
```

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

```
cmake_step2$ goa import
import download https://github.com/Kitware/CMake/trunk/Help/guide/tutorial/Step2
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_step2$ touch artifacts
cmake_step2$ goa build
```

¹<https://cmake.org/cmake-tutorial/>

```
[cmake_step2:cmake] -- The C compiler identification is GNU 12.3.0
[cmake_step2:cmake] -- The CXX compiler identification is GNU 12.3.0
[cmake_step2:cmake] -- Detecting C compiler ABI info
[cmake_step2:cmake] -- Detecting C compiler ABI info - done
[cmake_step2:cmake] -- Check for working C compiler:
      /usr/local/genode/tool/23.05/bin/genode-x86-gcc - skipped
[cmake_step2:cmake] -- Detecting C compile features
[cmake_step2:cmake] -- Detecting C compile features - done
[cmake_step2:cmake] -- Detecting CXX compiler ABI info
[cmake_step2:cmake] -- Detecting CXX compiler ABI info - done
[cmake_step2:cmake] -- Check for working CXX compiler:
      /usr/local/genode/tool/23.05/bin/genode-x86-g++ - skipped
[cmake_step2:cmake] -- Detecting CXX compile features
[cmake_step2:cmake] -- Detecting CXX compile features - done
[cmake_step2:cmake] -- Configuring done
[cmake_step2:cmake] -- Generating done
[cmake_step2:cmake] -- Build files have been written to: ../var/build/x86_64
[cmake_step2:cmake] Scanning dependencies of target Tutorial
[cmake_step2:cmake] [ 50%] Building CXX object
      CMakeFiles/Tutorial.dir/tutorial.cxx.obj
.../cmake_step2/src/tutorial.cxx:2:10: fatal error:
cmath: No such file or directory
    2 | #include <cmath>
      |           ^~~~~~
compilation terminated.
```

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_step2$ goa build
```

```
download genodelabs/api/libc/2023-10-03.tar.xz
download genodelabs/api/libc/2023-10-03.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/2023-10-24.tar.xz
download genodelabs/api/stdcxx/2023-10-24.tar.xz.sig
[cmake_step2:cmake] -- Configuring done
[cmake_step2:cmake] -- Generating done
[cmake_step2:cmake] -- Build files have been written to: .../var/build/x86_64
[cmake_step2:cmake] [ 50%] Building CXX object
                        CMakeFiles/Tutorial.dir/tutorial.cxx.obj
[cmake_step2:cmake] [100%] Linking CXX executable Tutorial
[cmake_step2:cmake] [100%] Built target Tutorial
```

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

```
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/cmake_step2/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/cmake_step2/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 name="dev">
        <log/>
      </dir>
    </vfs>
    <arg value="Tutorial"/>
    <arg value="24"/>
  </config>

  <content>
    <rom label="Tutorial"/>
    <rom label="posix.lib.so"/>
    <rom label="libc.lib.so"/>
    <rom label="libm.lib.so"/>
    <rom label="stdcxx.lib.so"/>
    <rom label="vfs.lib.so"/>
  </content>
</runtime>
```

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_step2/$ goa run
[cmake_step2:cmake] -- Configuring done
[cmake_step2:cmake] -- Generating done
[cmake_step2:cmake] -- Build files have been written to: .../var/build/x86_64
[cmake_step2:cmake] [100%] Built target Tutorial
download genodelabs/bin/x86_64/libc/2023-10-24.tar.xz
download genodelabs/bin/x86_64/libc/2023-10-24.tar.xz.sig
download genodelabs/bin/x86_64/posix/2023-10-24.tar.xz
download genodelabs/bin/x86_64/posix/2023-10-24.tar.xz.sig
download genodelabs/bin/x86_64/stdcxx/2023-10-24.tar.xz
download genodelabs/bin/x86_64/stdcxx/2023-10-24.tar.xz.sig
download genodelabs/bin/x86_64/vfs/2023-10-24.tar.xz
download genodelabs/bin/x86_64/vfs/2023-10-24.tar.xz.sig
...
download genodelabs/api/sandbox/2023-10-03.tar.xz.sig
Genode sculpt-23.10
17592186044415 MiB RAM and 18997 caps assigned to init
[init -> cmake_step2] The square root of 24 is 4.89898
[init] child "cmake_step2" exited with exit value 0
```

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_step2/$ goa run --target sculpt --target-opt-sculpt-server <sculpt-ip>
[cmake_step2:cmake] -- Configuring done (0.0s)
[cmake_step2:cmake] -- Generating done (0.0s)
[cmake_step2:cmake] -- Build files have been written to: .../var/build/x86_64
[cmake_step2:cmake] [100%] Built target Tutorial
uploaded libm.lib.so (local change)
uploaded stdcxx.lib.so (local change)
uploaded vfs.lib.so (local change)
uploaded Tutorial (local change)
uploaded posix.lib.so (local change)
uploaded libc.lib.so (local change)
uploaded config (local change)
Trying 10.0.1.1...
Connected to 10.0.1.1.
Escape character is '^]'.
[init -> cmake_step2] The square root of 24 is 4.89898
[init] child "cmake_step2" 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 XML. 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 name="fs_rom">
  <resource name="RAM" quantum="10M"/>
  <provides>
    <service name="ROM"/>
  </provides>
  <config/>
  <route>
    <service name="File_system"> <child name="vfs"/> </service>
    <any-service> <parent/> </any-service>
  </route>
</start>
```

- The *name* attribute of the `<start>` node refers the name of the child component and is identical to the binary name. A different binary name can be specified by adding a `<binary>` sub node.

3.1 Genode's init component

- The <resource> node 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)
{
    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` library looks for `<arg>` nodes in the component's config ROM in order to fill the `argv` array. 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 -->
  </pthread>
</libc>
```

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 name="tmp">
      <inline name="foobar">Hello!</inline>
      <rom name="config" binary="false"/>
      <ram/>
    </dir>
  </vfs>
</config>
```

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/> </vfs>
</config>
```

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* attribute that specifies the name of the file. If this attribute is omitted, the XML node type will be used as file name.

<inline name="inline"></inline>

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

<rom name="rom" 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.

<log name="log" 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="null"/>

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

<zero name="zero" 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="rtc"/>

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="terminal" 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="symlink" target=""/>

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

<block name="block" label="" block_buffer_count="1"/>

Makes a Block session available as a file. The *label* attribute specifies the optional session label. The *block_buffer_count* attribute sets the size of the internal block 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 *writable* 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 XML 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* attribute can be used to change the file name.

<jitterentropy name="jitterentropy"/>

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

<oss name="oss" 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 [README¹](#).

<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="capture" 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="tap" label="" mode="nic" mac=""..."/>

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 [README²](#).

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

<import overwrite="no"></import>

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=""..."/>

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

<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`

¹<https://github.com/genodelabs/genode/blob/master/repos/gems/src/lib/vfs/oss/README>

²<https://github.com/genodelabs/genode/blob/master/repos/os/src/lib/vfs/tap/README>

`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="">`

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 `fonts_fs` [raw archive](#)³.

`<lxip dhcp="false" ip_addr="..." netmask="..." gateway="..." nameserver="...">`

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.

`<lwip dhcp="false" label="lwip" ...>`

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. In addition, the *label* attribute can be used to change the NIC session label.

`<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.

`<fatfs/>`

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

¹<https://github.com/genodelabs/genode/blob/master/repos/gems/src/lib/vfs/pipe/README>

²<https://github.com/genodelabs/genode/blob/master/repos/gems/src/lib/vfs/trace/README>

³https://github.com/genodelabs/genode/blob/master/repos/gems/recipes/raw/fonts_fs/fonts_fs.config

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>

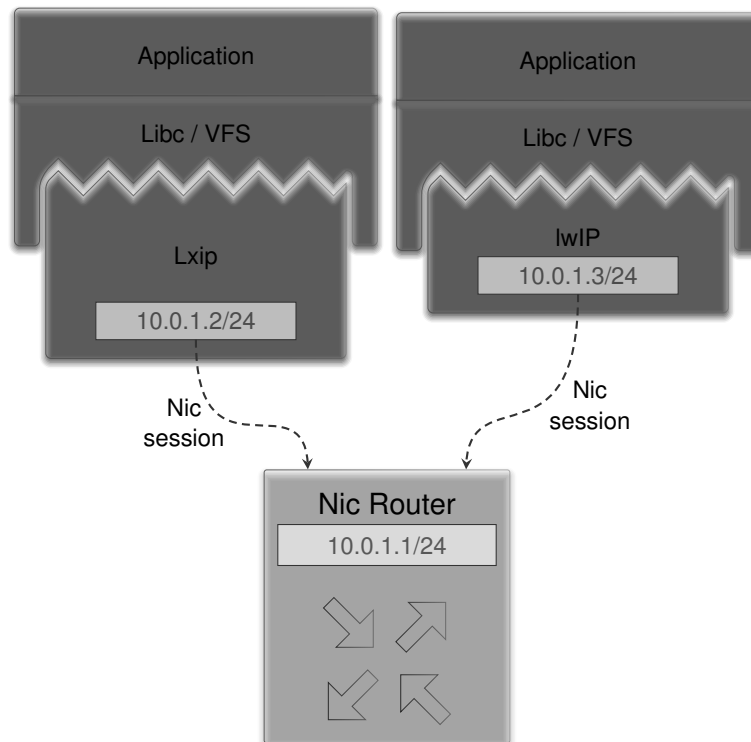
VFS examples in Goa repository

<https://github.com/genodelabs/goa/tree/master/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 name="...">
  <config>
    <libc socket="/sockets"/>
    <vfs>
      <dir name="sockets">
        <lwip dhcp="yes"/>
      </dir>
    </vfs>
  </config>
</start>
```

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.

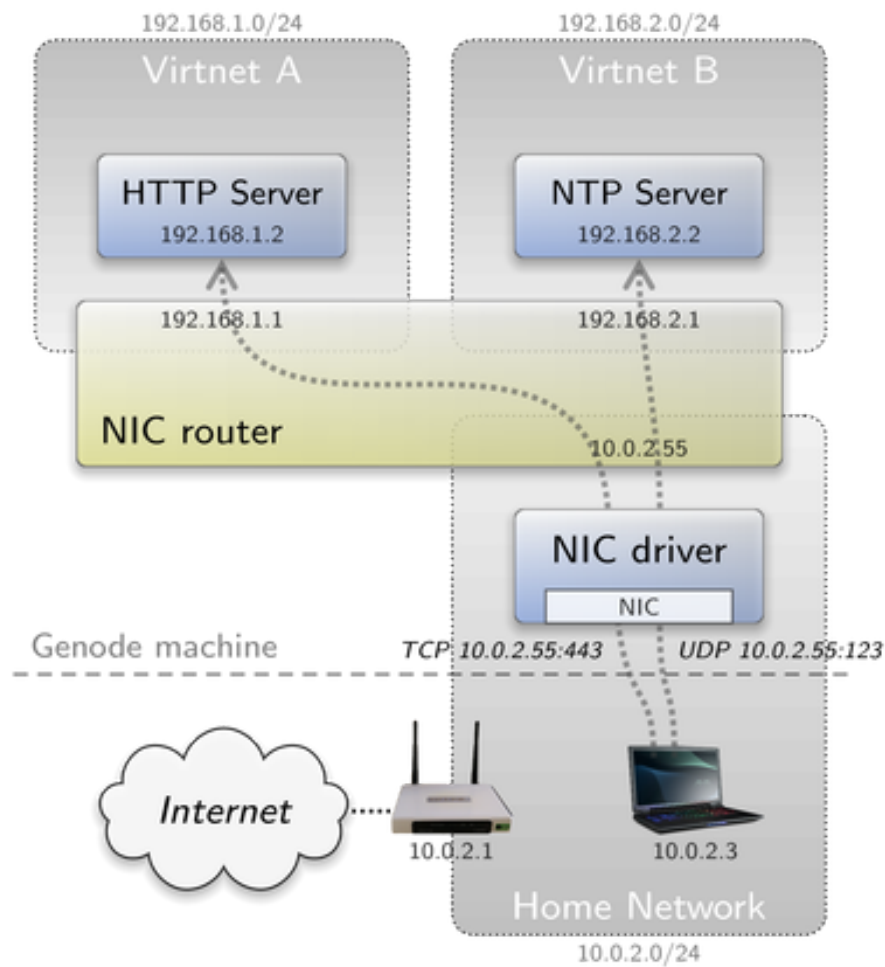


Figure 1

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 name="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>

  <domain name="virtnet_a" interface="192.168.1.1/24" />
  <domain name="virtnet_b" interface="192.168.2.1/24" />
</config>
```

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 details 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 NIC router and a Linux NIC driver. The latter connects to an existing Linux tap device, which is a virtual network interface.

You can add a `tap0` device with IP address 10.0.100.1 using the following commands:

```
$ sudo ip tuntap add dev tap0 mode tap user $(whoami)
$ sudo ip address flush dev tap0
$ sudo ip addr add 10.0.100.1/24 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=10.0.100.2,10.0.100.2,12h
```

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

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

In the `runtime` file of a Goa project, you are further able to set the name of the tap device and also specify additional domains and forwarding rules for the NIC router:

¹https://github.com/genodelabs/genode/blob/master/repos/os/src/server/nic_router/README

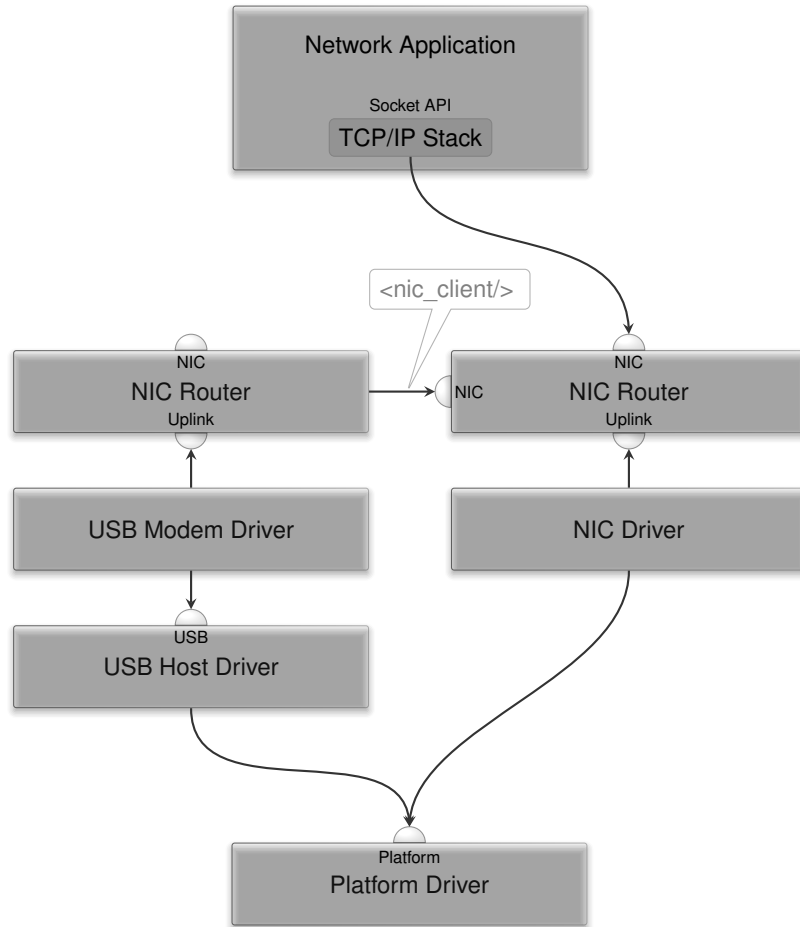

```
<runtime>
  <requires>
    <nic tap_name="tap0">
      <!-- additional NIC router <domain> and <policy> nodes -->
      <!-- <tcp-forward> and <udp-forward> nodes are inserted into
            uplink domain -->
    </nic>
  </requires>
</runtime>
```

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

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](https://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>
    <!-- required session interfaces -->
    <nic/>
  </requires>

  <provides>
    <!-- provided sessions interfaces -->
  </provides>

  <content>
    <!-- required ROM modules -->
    <rom label="..."/>
  </content>

  <config>
    <!-- component config -->
  </config>
</runtime>
```

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>` are the lower-case service 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](https://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://github.com/genodelabs/genode-world/>

Genode application examples

Port of numptyphysics

<https://github.com/nfeske/goa-playground/tree/master/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>

Qt6 examples and tutorials

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

¹<https://genodians.org/nfeske/2020-06-23-gui-stack>

Genode application examples

Falkon web browser

<https://github.com/genodelabs/genode-world/tree/master/recipes/pkg/falkon>

Qt5 textedit

https://github.com/genodelabs/genode/tree/master/repos/libports/recipes/pkg/qt5_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://github.com/genodelabs/genode-world/tree/master/recipes/pkg/ubuntu_ui_toolkit

Porting the calculator app from Lomiri UI Toolkit

see Section 5.4

UBports website

<https://ubports.com/>

Genode application examples

Morph browser

https://github.com/genodelabs/genode-world/tree/master/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 back-ends that interact with Genode's GUI session. Both libraries are still in experimental state.

LVGL documentation

<https://docs.lvgl.io/master/>

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 23.11¹](#) 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.

¹https://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, *genode-labs/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 sculpt-24.04
 17592186044415 MiB RAM and 18997 caps assigned to init
[init -> system_info] [Warn] (0.000, +0) lv_init: Style sanity checks [...]
[init -> system_info] [Warn] (0.000, +0) lv_style_init: Style might be [...]
[init -> system_info] backtrace "ep"
```

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 sculpt-24.04
17592186044415 MiB RAM and 18997 caps assigned to init
[init -> system_info] [Warn] (0.000, +0) lv_init: Style sanity checks [...]
[init -> system_info] [Warn] (0.000, +0) lv_style_init: Style might be [...]
[init -> system_info] backtrace "ep"
[init -> system_info] 403ff728      1003f7b
[init -> system_info] 403ff798      1003fd1
[init -> system_info] 403ff7b8      103a5ad
[init -> system_info] 403ff7e8      7ffff7fdedd0
```

The second column of the backtrace data shows the return addresses on the call stack. The first two addresses certainly belong to the *system_info* binary. The third 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:

```
system_info$ goa run --with-backtrace
Genode sculpt-24.04
17592186044415 MiB RAM and 18997 caps assigned to init
[init -> system_info] 0x1000000 .. 0x10ffffff: linker area
[init -> system_info] 0x40000000 .. 0x4fffffff: stack area
[init -> system_info] 0x50000000 .. 0x601b2fff: ld.lib.so
[init -> system_info] 0x10e1d000 .. 0x10ffffff: libc.lib.so
[init -> system_info] 0x10d79000 .. 0x10e1cfff: vfs.lib.so
[init -> system_info] 0x10d37000 .. 0x10d78fff: libm.lib.so
[init -> system_info] 0x101c000 .. 0x11f3fff: liblvglib.so
[init -> system_info] 0x10d2f000 .. 0x10d36fff: posix.lib.so
[init -> system_info] 0x11f4000 .. 0x120efff: liblvglib_support.lib.so
[init -> system_info] 0x120f000 .. 0x148cfff: stdcxx.lib.so
[init -> system_info] [Warn] (0.000, +0) lv_init: Style sanity checks [...]
[init -> system_info] [Warn] (0.000, +0) lv_style_init: Style might be [...]
[init -> system_info] backtrace "ep"
[init -> system_info] 403ff728      1003f7b
[init -> system_info] 403ff798      1003fd1
[init -> system_info] 403ff7b8      103a5ad
[init -> system_info] 403ff7e8      7ffff7fdedd0
```

The output confirms that the third address belongs to *liblvglib.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 sculpt-24.04
17592186044415 MiB RAM and 18997 caps assigned to init
[init -> system_info] 0x1000000 .. 0x10ffffff: linker area
[init -> system_info] 0x40000000 .. 0x4fffffff: stack area
[init -> system_info] 0x50000000 .. 0x601b2fff: ld.lib.so
[init -> system_info] 0x10e1d000 .. 0x10ffffff: libc.lib.so
[init -> system_info] 0x10d79000 .. 0x10e1cfff: vfs.lib.so
[init -> system_info] 0x10d37000 .. 0x10d78fff: libm.lib.so
[init -> system_info] 0x101c000 .. 0x11f3fff: liblvglib.so
[init -> system_info] 0x10d2f000 .. 0x10d36fff: posix.lib.so
[init -> system_info] 0x11f4000 .. 0x120efff: liblvglib_support.lib.so
[init -> system_info] 0x120f000 .. 0x148cfff: stdcxx.lib.so
[init -> system_info] [Warn] (0.000, +0) lv_init: Style sanity checks [...]
[init -> system_info] [Warn] (0.000, +0) lv_style_init: Style might be [...]
[init -> system_info] backtrace "ep"
[init -> system_info] 403ff728      1003f7b
[init -> system_info] 403ff798      1003fd1
[init -> system_info] 403ff7b8      103a5ad
[init -> system_info] 403ff7e8      7ffff7fdedd0
Expect: 'interact' received 'strg+c' and was cancelled
Scanned image system_info
Scanned image ld.lib.so
...
void Genode::log<Genode::Backtrace>(Genode::Backtrace&&)
  * 0x1003f7b: system_info:0x1003f7b W
  * /depot/genodelabs/api/base/2024-04-11/include/base/log.h:170

Info::Bar::_draw_part_event_cb(_lv_event_t*)
  * 0x1003fd1: system_info:0x1003fd1 W
  * [...] /var/build/x86_64/system_info.h:277 (discriminator 1)

event_send_core
  * 0x103a5ad: liblvglib.so:0x1e5ad t
  * [...] /goa-projects/lvgl/lvgl/src/src/core/lv_event.c:469

_end
  * 0x7ffff7fdedd0: liblvglib_support.lib.so:0x7ffff6deadd0 B
  * ??:0

```

The output shows that the first address on the stack points to the backtrace method itself. The second address points to the `_draw_part_event_cb()` in which we inserted the backtrace call. The third address points to `liblvglib` where the callback method was called, however, the backtrace stops here because the `lvgl` library was not built with frame-pointer information.

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

4.2 Using backtraces

```
lvgl$ goa export --debug --with-backtrace --depot-overwrite
...
[lvgl] exported [...] /depot/jschlatow/api/lvgl/2024-05-06
[lvgl] exported [...] /depot/jschlatow/src/lvgl/2024-05-06
[lvgl] exported [...] /depot/jschlatow/bin/x86_64/lvgl/2024-05-06
[lvgl] exported [...] /depot/jschlatow/dbg/x86_64/lvgl/2024-05-06

lvgl$ cd ../system_info
system_info$ goa backtrace
...
[init -> system_info] backtrace "ep"
[init -> system_info] 403ff6a8 1003f7b
[init -> system_info] 403ff718 1003fd1
[init -> system_info] 403ff738 103a57e
[init -> system_info] 403ff758 103a618
[init -> system_info] 403ff7a8 109d711
[init -> system_info] 403ff9a8 103a325
[init -> system_info] 403ff9c8 103a46a
[init -> system_info] 403ff9e8 103a618
[init -> system_info] 403ffa38 1048f10
[init -> system_info] 403ffab8 1048eb7
[init -> system_info] 403ffb38 1048eb7
[init -> system_info] 403ffbb8 1048eb7
[init -> system_info] 403ffc38 1048eb7
[init -> system_info] 403ffc8 1049611
[init -> system_info] 403ffd08 10496ff
[init -> system_info] 403ffe38 104ab23
[init -> system_info] 403ffec8 109a048
[init -> system_info] 403fff18 10f52382
```

```
Expect: 'interact' received 'strg+c' and was cancelled
Scanned image system_info
Scanned image ld.lib.so
...
void Genode::log<Genode::Backtrace>(Genode::Backtrace&&)
  * 0x1003f7b: system_info:0x1003f7b W
  * /depot/genodelabs/api/base/2024-04-11/include/base/log.h:170

Info::Bar::_draw_part_event_cb(_lv_event_t*)
  * 0x1003fd1: system_info:0x1003fd1 W
  * [...] /var/build/x86_64/system_info.h:277 (discriminator 1)

event_send_core
  * 0x103a57e: liblvgl.lib.so:0x1e57e t
  * [...] /goa-projects/lvgl/lvgl/src/src/core/lv_event.c:469

lv_event_send
  * 0x103a618: liblvgl.lib.so:0x1e618 T
  * [...] /goa-projects/lvgl/lvgl/src/src/core/lv_event.c:78

draw_indic
  * 0x109d711: liblvgl.lib.so:0x81711 t
  * [...] /goa-projects/lvgl/lvgl/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:


```
$ sudo 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 "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 2228
Attaching to process 2228
[New LWP 2237]
[New LWP 2246]
Reading symbols from [...]/depot/[...]/base-linux/2024-04-24/ld.lib.so...
(No debugging symbols found in [...]/base-linux/2024-04-24/ld.lib.so)
0x000000005009f74a in ?? ()
(gdb)
Thread 2 "ld.lib.so" stopped.
0x000000005009f74a in ?? ()

Thread 3 "ld.lib.so" stopped.
0x000000005009f74a 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/liblvglib.so.debug -o 0x101b000
add symbol table from file ".debug/liblvglib.so.debug" with all sections
offset by 0x101b000
(y or n) y
Reading symbols from .debug/liblvglib.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_verbosity="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 2228 "ld.lib.so" pseudo_end () at [...]/spec/x86_64/lx_syscall.S:29
   2   LWP 2237 "ld.lib.so" pseudo_end () at [...]/spec/x86_64/lx_syscall.S:29
   3   LWP 2246 "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 2237)]
(gdb) info threads
  Id  Target Id          Frame
   1   LWP 2228 "ld.lib.so" (running)
*  2   LWP 2237 "ld.lib.so" (running)
   3   LWP 2246 "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
Genode::Native_thread::Epoll::poll (this=0x401fffe8)
  at [...]/src/lib/base/native_thread.cc:82
82         if ((event_count == 1) && (events[0].events == POLLIN)) {
(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_periodic_timer = true,
86         .periodic_ms       = 5000,
87         .resize_callback   = &_amp;resize_callback,
88         .timer_callback    = &_amp;timer_callback,
89     };
90
91
92     void handle_resize()
93     {
94         Libc::with_libc([&] {
(gdb) break system_info.cc:94

Breakpoint 1 at 0x1000d50: system_info.cc:94. (2 locations)
Warning:
Cannot insert breakpoint 1.
Cannot access memory at address 0x1000d50
Cannot insert breakpoint 1.
Cannot access memory at address 0x1001c79
```

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 +132,7 @@ 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 `--run-as <user>` 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-jschlatow/src/base-linux 2024-06-27
```

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

```
(gdb) break system_info.h:272
Breakpoint 1 at 0x1001890: file [...]/var/build/x86_64/system_info.h, line 272
(gdb)
Thread 2 "ld.lib.so" hit Breakpoint 1, Info::Bar::_draw_part_event_cb
    at [...]/var/build/x86_64/system_info.h:272
272          lv_obj_draw_part_dsc_t * dsc = lv_event_get_draw_part_dsc(e);
```

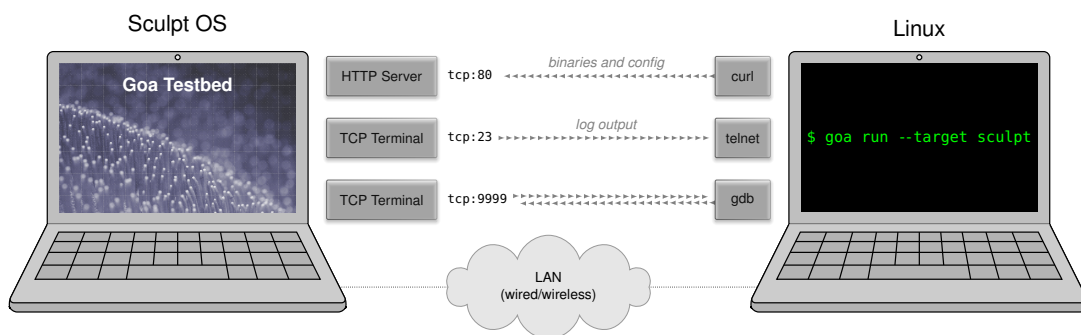
4.4 Using Sculpt as a remote test target

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 (remote change)
uploaded stdcxx.lib.so (remote change)
uploaded vfs.lib.so (remote change)
uploaded liblvgL.lib.so (local change)
uploaded system_info (local change)
uploaded posix.lib.so (remote change)
uploaded liblvgL_support.lib.so (local change)
uploaded libc.lib.so (remote change)
uploaded config (local change)
Trying 192.168.42.54...
Connected to 192.168.42.54.
Escape character is '^]'.
[monitor] monitor ready
[init -> 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 192.168.42.54 --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 /home/johannes/repos/genode/depot
set substitute-path /depot /home/johannes/repos/genode/depot
target extended-remote 192.168.42.54:9999
```

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 tool chain. Moreover, root privileges are not required.

```
$ genode-x86-gdb --command /path/to/project/var/project_name.gdb
GNU gdb (GDB) 13.1
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 192.168.42.54:9999)
```

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 initialization, 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 0x101b000
add symbol table from file ".debug/liblvgl.lib.so.debug" with all sections
offset by 0x101b000
(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:94
Breakpoint 2 at 0x1000d50: system_info.cc:94. (2 locations)
(gdb)
Thread 2 "ep" hit Breakpoint 2.1, Main::Resize_callback::operator()
(this=0x101abf8 <Libc::Component::construct(Libc::Env&)::main+8824>) at
system_info.cc:94
94          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-go-a-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>

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 tool chain and Goa installed (see [Section 2](#)).

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, publishing the result, and deploying it on top of Sculpt OS. 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 playground repository](#)².

Preparations Before continuing, please make sure to have installed the Goa tool, which is available at <https://github.com/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
```

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 unix
$ cd unix
unix$ mkdir -p pkg/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/unix/archives* file:

```
genodelabs/src/bash
```

The *pkg/unix/runtime* file:

```
<runtime ram="100M" caps="5000" binary="init">
  <content>
    <rom label="init"/>
  </content>
</runtime>
```

¹<https://genodians.org/nfeske/2019-12-13-go-a-unix-bash>

²<https://github.com/nfeske/goa-playground>

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

```
unix$ goa run

download genodelabs/bin/x86_64/bash/2023-10-24.tar.xz
download genodelabs/bin/x86_64/bash/2023-10-24.tar.xz.sig
download genodelabs/src/bash/2023-10-24.tar.xz
download genodelabs/src/bash/2023-10-24.tar.xz.sig
download genodelabs/api/libc/2023-10-03.tar.xz
download genodelabs/api/libc/2023-10-03.tar.xz.sig
download genodelabs/api/noux/2023-06-15.tar.xz
download genodelabs/api/noux/2023-06-15.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 in 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/unix/runtime* file:

```
<runtime ram="100M" caps="5000" binary="init">
  <config/>
  <content>
    <rom label="init"/>
  </content>
</runtime>
```

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/2023-10-24/

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/2023-10-24/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/unix/runtime*:

```
<runtime ram="100M" caps="5000" binary="init">
  <config/>
  <content>
    <rom label="init"/>
    <rom label="bash.tar"/>
  </content>
</runtime>
```

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

```
unix$ goa run
download genodelabs/bin/x86_64/base-linux/2023-10-24.tar.xz
download genodelabs/bin/x86_64/base-linux/2023-10-24.tar.xz.sig
download genodelabs/bin/x86_64/init/2023-10-24.tar.xz
download genodelabs/bin/x86_64/init/2023-10-24.tar.xz.sig
download genodelabs/src/base-linux/2023-10-24.tar.xz
download genodelabs/src/base-linux/2023-10-24.tar.xz.sig
download genodelabs/src/init/2023-10-24.tar.xz
download genodelabs/src/init/2023-10-24.tar.xz.sig
download genodelabs/api/base/2023-10-24.tar.xz
download genodelabs/api/base/2023-10-24.tar.xz.sig
download genodelabs/api/os/2023-08-21.tar.xz
download genodelabs/api/os/2023-08-21.tar.xz.sig
download genodelabs/api/report_session/2023-05-26.tar.xz
download genodelabs/api/report_session/2023-05-26.tar.xz.sig
download genodelabs/api/sandbox/2023-10-03.tar.xz
download genodelabs/api/sandbox/2023-10-03.tar.xz.sig
download genodelabs/api/timer_session/2023-10-03.tar.xz
download genodelabs/api/timer_session/2023-10-03.tar.xz.sig
Genode sculpt-23.10
17592186044415 MiB RAM and 18997 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/unix/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/unix/runtime* file by the following configuration.

```
<config>
  <parent-provides>
    <service name="ROM"/>
    <service name="LOG"/>
    <service name="RM"/>
    <service name="CPU"/>
    <service name="PD"/>
    <service name="Timer"/>
  </parent-provides>

  <start name="vfs" caps="100">
    <resource name="RAM" quantum="10M"/>
    <provides> <service name="File_system"/> </provides>
    <config>
      <vfs> <tar name="bash.tar"/> </vfs>
      <default-policy root="/" />
    </config>
    <route> <any-service> <parent/> </any-service> </route>
  </start>

</config>
```

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 sculpt-23.10
17592186044415 MiB RAM and 18997 caps assigned to init
[init -> unix] Error: vfs: environment ROM session denied ...
```

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/>
  </requires>
  ...
</runtime>
```

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/unix/archive*:

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

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

```
<content>
  <rom label="init"/>
  <rom label="bash.tar"/>
  <rom label="vfs"/>
</content>
```

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: Could not open ROM session for "vfs.lib.so"
```

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 *archives* file. So we can resolve this error by adding a corresponding `<rom>` entry to the *runtime* file. The `<content>` should now look as follows:

```
<content>
  <rom label="init"/>
  <rom label="bash.tar"/>
  <rom label="vfs"/>
  <rom label="vfs.lib.so"/>
</content>
```

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

```
unix$ goa run
```

```
Genode sculpt-23.10
17592186044415 MiB RAM and 18997 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/unix/runtime* with the following snippet:


```
<start name="vfs_rom" caps="100">
  <resource name="RAM" quantum="10M"/>
  <binary name="fs_rom"/>
  <provides> <service name="ROM"/> </provides>
  <config/>
  <route>
    <service name="File_system"> <child name="vfs"/> </service>
    <any-service> <parent/> </any-service>
  </route>
</start>
```

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 *archives* file and the *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 sculpt-23.10
17592186044415 MiB RAM and 18997 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 name="/bin/bash" caps="1000">
  <resource name="RAM" quantum="10M" />
  <config>
    <libc stdin="/dev/null" stdout="/dev/log" stderr="/dev/log"
      rtc="/dev/null"/>
    <vfs>
      <dir name="dev"> <null/> <log/> </dir>
    </vfs>
    <arg value="bash"/>
    <arg value="-c"/>
    <arg value="echo files at /dev: /dev/*"/>
  </config>
  <route>
    <service name="ROM" label_last="/bin/bash">
      <child name="vfs_rom"/> </service>
    <any-service> <parent/> </any-service>
  </route>
</start>
```

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 *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 *runtime* file's `<content>` node:

```
<content>
  <rom label="init"/>
  <rom label="bash.tar"/>
  <rom label="vfs"/>
  <rom label="vfs.lib.so"/>
  <rom label="fs_rom"/>
  <rom label="libc.lib.so"/>
  <rom label="libm.lib.so"/>
  <rom label="posix.lib.so"/>
</content>
```

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.

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 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">

  <content>
    <rom label="init"/>
    <rom label="bash.tar"/>
    <rom label="vfs"/>
    <rom label="vfs.lib.so"/>
    <rom label="fs_rom"/>
    <rom label="libc.lib.so"/>
    <rom label="libm.lib.so"/>
    <rom label="posix.lib.so"/>
    <rom label="unix.config"/>
  </content>

</runtime>
```

The *raw/unix.config* file:

```
<config verbose="yes">

  <parent-provides>
    <service name="ROM"/>
    <service name="LOG"/>
    <service name="RM"/>
    <service name="CPU"/>
    <service name="PD"/>
    <service name="Timer"/>
  </parent-provides>

  <start name="vfs" caps="100">
    <resource name="RAM" quantum="10M"/>
    <provides> <service name="File_system"/> </provides>
    <config>
      <vfs> <tar name="bash.tar"/> </vfs>
      <default-policy root="/" />
    </config>
    <route> <any-service> <parent/> </any-service> </route>
  </start>

  <start name="vfs_rom" caps="100">
    <resource name="RAM" quantum="10M"/>
    <binary name="fs_rom"/>
    <provides> <service name="ROM"/> </provides>
    <config/>
    <route>
      <service name="File_system"> <child name="vfs"/> </service>
      <any-service> <parent/> </any-service>
    </route>
  </start>

  <start name="/bin/bash" caps="1000">
    <resource name="RAM" quantum="10M" />
    <config>
      <libc stdin="/dev/null" stdout="/dev/log" stderr="/dev/log"
        rtc="/dev/null"/>
      <vfs>
        <dir name="dev"> <null/> <log/> </dir>
      </vfs>
      <arg value="bash"/>
      <arg value="-c"/>
      <arg value="echo files at /dev: /dev/*"/>
    </config>
    <route>
      <service name="ROM" label_last="/bin/bash">
        <child name="vfs_rom"/> </service>
      <any-service> <parent/> </any-service>
    </route>
  </start>

</config>
```

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/>
</requires>
```

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.

```
<parent-provides>
  ...
  <service name="Gui"/>
</parent-provides>
```

¹<https://github.com/genodelabs/genode/tree/master/repos/os/src/server/nitpicker>

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 greenish background 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 name="terminal" caps="110">
  <resource name="RAM" quantum="10M"/>
  <provides> <service name="Terminal"/> </provides>
  <route>
    <service name="ROM" label="config">
      <parent label="terminal.config"/> </service>
    <any-service> <parent/> </any-service>
  </route>
</start>
```

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/unix/archives`

```
genodelabs/src/terminal
```

The following two lines must be added to the runtime file’s `<content>`:

¹<https://github.com/genodelabs/genode/tree/master/repos/gems/src/server/terminal>


```
<content>
  ...
  <rom label="terminal"/>
  <rom label="terminal.config"/>
</content>
```

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: Uncaught exception of type
                          'Genode::Xml_node::Nonexistent_sub_node'
[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 name="VeraMono.ttf"/>
    <dir name="fonts">
      <dir name="monospace">
        <ttf name="regular" path="/VeraMono.ttf" size_px="16"/>
      </dir>
    </dir>
  </vfs>
</config>
```

Wait a minute. How is this a terminal configuration?

The terminal expects its font to be found at its local VFS at */fonts/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 */fonts/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:

```
[... unix -> terminal] Error: could not open ROM session for "VeraMono.ttf"
[... unix -> terminal] Error: failed to create <rom> VFS node
[... unix -> terminal] Error:      name="VeraMono.ttf"
[... unix -> terminal] Error: ROM-session creation failed (...)
[... unix -> terminal] Error: could not open ROM session for "vfs_ttf.lib.so"
[... unix -> terminal] Error: failed to create <ttf> VFS node
[... unix -> terminal] Error:      name="regular"
[... unix -> terminal] Error:      path="/VeraMono.ttf"
[... unix -> terminal] Error:      size_px="16"
```

The terminal configuration refers to two ROM modules 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 "vfs_ttf.lib.so" is the driver for the "ttf" pseudo file system. It is requested by the VFS when the *<ttf>* is encountered. The errors can be resolved by extending the *archives* file and the *runtime* file's *<content>* node accordingly.

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

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

The following lines must be added to the *<content>* node in *pkg/unix/runtime*

```
<content>
...
  <rom label="VeraMono.ttf"/>
  <rom label="vfs_ttf.lib.so"/>
</content>
```

Good news! On the next try of *goa run*, you can see the error gone and are greeted with a black 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:

```
<config>
  <vfs> <tar name="bash.tar"/> </vfs>
  <default-policy root="/" />
</config>
<route> <any-service> <parent/> </any-service> </route>
```

We change it to the following:

```
<config>
  <vfs>
    <tar name="bash.tar"/>
    <dir name="dev"> <terminal/> </dir>
  </vfs>
  <default-policy root="/" />
  <policy label_prefix="/bin/bash" root="/" writeable="yes" />
</config>
<route>
  <service name="Terminal"> <child name="terminal"/> </service>
  <any-service> <parent/> </any-service>
</route>
```

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 name="dev"> <null/> <log/> </dir>
  <fs/>
</vfs>
```

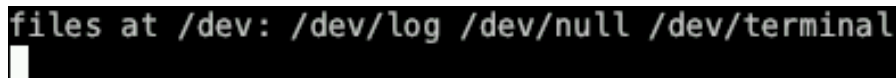
When the VFS of bash encounters the `<fs/>` node, it will request a session to a “File_system” service. To let this request reach the VFS server, we have to add a new entry to the `<route>` definition.

```
<route>
  <service name="File_system"> <child name="vfs"/> </service>
  ...
</route>
```

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 (just appending the `> /dev/terminal`):

```
<arg value="echo files at /dev: /dev/* > /dev/terminal"/>
```

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



```
files at /dev: /dev/log /dev/null /dev/terminal
```

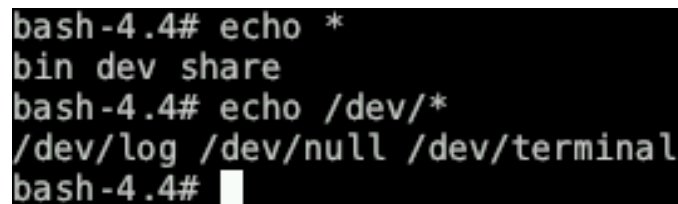
Figure 2

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:



```
bash-4.4# echo *
bin dev share
bash-4.4# echo /dev/*
/dev/log /dev/null /dev/terminal
bash-4.4#
```

Figure 3

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

```
bash-4.4# ls
bash: ls: command not found
bash-4.4#
```

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](https://www.gnu.org/software/coreutils/coreutils.html)¹ package.

The GNU coreutils package is readily available for Genode. We can add it by appending the following line to our `pkg/unix/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.

¹<https://www.gnu.org/software/coreutils/coreutils.html>

```
unix$ tar tf var/depot/genodelabs/bin/x86_64/coreutils/2023-10-24/coreutils.tar
./
./lib/
...
./share/
...
./bin/
./bin/uname
./bin/groups
./bin/dircolors
./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/unix/runtime* file's `<content>` node:

```
<content>
...
  <rom label="coreutils.tar"/>
</content>
```

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 name="bash.tar"/>
  <tar name="coreutils.tar"/>
  <dir name="dev"> <terminal/> </dir>
</vfs>
```

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](https://en.wikipedia.org/wiki/Union_mount)¹.

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

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

```
bash-4.4# ls
bash: ls: command not found
bash-4.4#
```

Figure 5

Let's give goa run another go now.

```
bash-4.4# ls
bash: /bin/ls: No such file or directory
bash-4.4#
```

Figure 6

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 `cd bin; echo *`. However, bash still failed to spawn the `ls` program. Genode's log output reveals why:

```
[init -> ...] Error: Could not open ROM session for "/bin/ls"
[init -> ...] Warning: execve: executable binary inaccessible as ROM module
```

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 name="File_system"> <child name="vfs"/> </service>
  <service name="ROM" label_last="/bin/bash">
    <child name="vfs_rom"/> </service>
  <any-service> <parent/> </any-service>
</route>
```

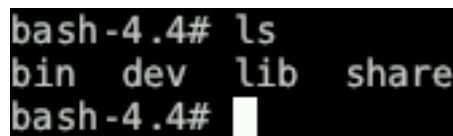
There is no valid route for a ROM service and the label `"/bin/ls"` yet. In principle, we could follow the pattern of the `"/bin/bash"` ROM. On the other hand, with many binaries installed at `/bin/`, the approach would become cumbersome. A better solution is adding a route that matches the label prefix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

```
<route>
  <service name="File_system"> <child name="vfs"/> </service>
  <service name="ROM" label_last="/bin/bash">
    <child name="vfs_rom"/> </service>
  <service name="ROM" label_prefix="/bin">
    <child name="vfs_rom"/> </service>
  <any-service> <parent/> <any-child/> </any-service>
</route>
```

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 key="PATH" value="/bin"/>
</config>
```

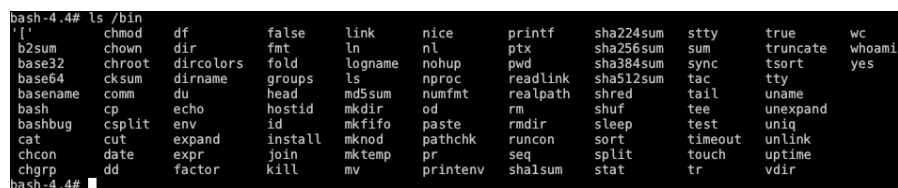
The next try of `goa run` yields the following result:



```
bash-4.4# ls
bin dev lib share
bash-4.4#
```

Figure 7

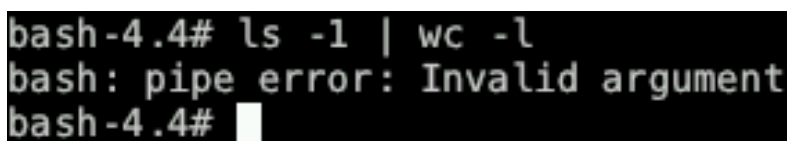
A look at `/bin/` reveals the wealth of commands that have just become available at our finger tips.



```
bash-4.4# ls /bin
['
b2sum      chmod      df          false      link       nice       printf     sha224sum  stty       true       wc
base32     chown      dir         fmt         ln         nl         ptx        sha256sum  sum        truncate  whoami
base64     chroot     dircolors  fold       logname    nohup      pwd        sha384sum  sync       tsort     yes
basename   cksum     dirname    groups     ls         nproc     readlink   sha512sum  tac        tty
cp         comm       du          head        md5sum     numfmt     realpath   shred      tail       uname
bash       cp         echo       hostid     mkdir      od         rm         shuf       tee        unexpand
bashbug    csplit    env        id          mkfifo     paste      rmdir     sleep     test       uniq
cat        cut       expand     install    knod       pathchk    runcon    sort      timeout   unlink
chcon     date     expr      join       mktemp     pr         seq       split     touch     uptime
chgrp     dd        factor    kill       mv          printenv   sha1sum   stat      tr        vdir
bash-4.4#
```

Figure 8

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



```
bash-4.4# ls -l | wc -l
bash: pipe error: Invalid argument
bash-4.4#
```


Figure 9

With our attempt of using a pipe, feeding the output of `ls -l` 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>` of the VFS server currently looks as follows.

```
<vfs>
  <tar name="bash.tar"/>
  <tar name="coreutils.tar"/>
  <dir name="dev"> <terminal/> </dir>
</vfs>
```

With the addition of the pipe pseudo file system, we change the `<dir name="dev">` node into this:

```
<dir name="dev">
  <terminal/>
  <dir name="pipe"> <pipe/> </dir>
</dir>
```

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

```
[init -> unix -> vfs] Error: Could not open ROM session for "vfs_pipe.lib.so"
[init -> unix -> vfs] Error: failed to create <pipe> VFS node
```

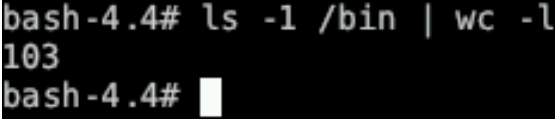
I'm sure, you guess what comes next. Let's enhance `pkg/unix/archives` with the following line:

```
genodelabs/src/vfs_pipe
```

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

```
<content>
...
<rom label="vfs_pipe.lib.so"/>
</content>
```

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



```
bash-4.4# ls -l /bin | wc -l
103
bash-4.4#
```

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/unix/archives*

```
genodelabs/src/vim
```

2. Add the “`vim.tar`” ROM to *pkg/unix/runtime*

```
<rom label="vim.tar"/>
```

3. Mount “`vim.tar`” at the VFS server

```
<tar name="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: Could not open ROM session for "ncurses.lib.so"
```

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/unix/archives* and `<rom label="ncurses.lib.so"/>` to *pkg/unix/runtime*.

¹<https://en.wikipedia.org/wiki/Ncurses>

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

```
E483: Can't get temp file name
E483: Can't get temp file name
Press ENTER or type command to continue
```

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 name="tmp"> <ram/> </dir>
```

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

```
Cannot execute shell sh
E79: Cannot expand wildcards
Cannot execute shell sh
E79: Cannot expand wildcards
Press ENTER or type command to continue
```

Figure 12

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 key="SHELL" value="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=
```

5.1 Sticking together a little Unix

Add a `<rom label="vimrc" />` node to the `<content>` of `pkg/unix/runtime`.
Mount the "vimrc" ROM as `/share/vim/vimrc` file at the VFS server:

```
<dir name="share"> <dir name="vim"> <rom name="vimrc" /> </dir> </dir>
```

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:

```
<env key="TERM" value="screen" />
```

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

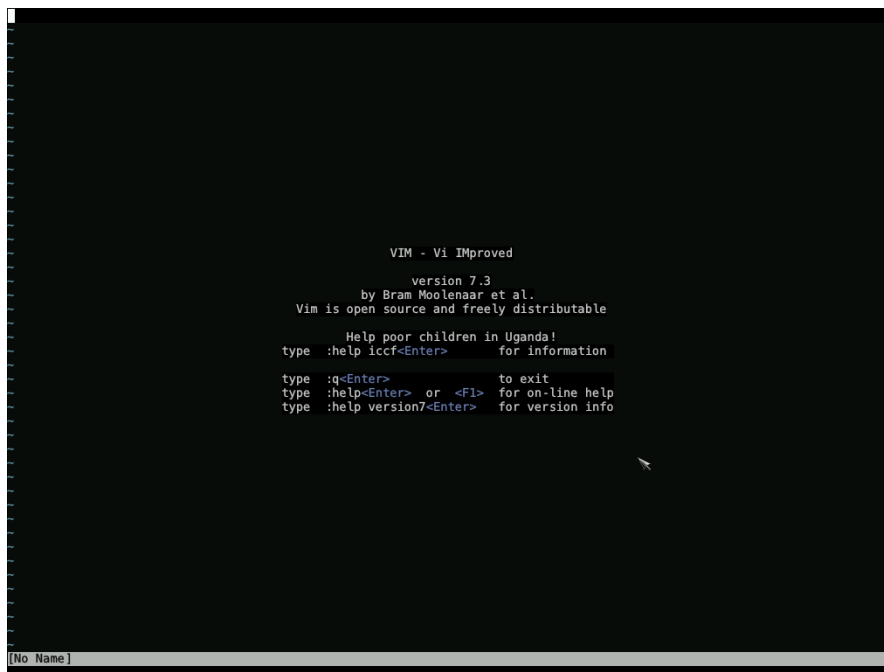


Figure 13

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 150 lines of XML:

```
$ wc -l raw/unix.config raw/terminal.config pkg/unix/runtime
 89 raw/unix.config
 10 raw/terminal.config
 28 pkg/unix/runtime
121 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 playground 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 playground repository

<https://github.com/nfeske/goa-playground>

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 Open-PGP 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 Open-PGP 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 Open-PGP key pair.

¹<https://genodians.org/nfeske/2020-01-16-go-publish>

²https://github.com/nfeske/goa-playground/tree/master/intro/unix_3rd

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-playground repository linked above, you would change to the `unix_3rd` directory.

```
$ git clone https://github.com/nfeske/goa-playground.git
```

```
$ cd goa-playground/intro/unix_3rd/
```

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

```
$ goa update-go
```

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/2023-10-24.tar.xz.sig
var/public/genodelabs/bin/x86_64/terminal/2023-10-24.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 Section 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
```

```
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 specifying the `'--depot-user <name>'` command-line argument.

As hinted by the error message, Goa needs to know the name of us as the software provider. The depot user name will correspond to the subdirectory within the depot that will host your content. Let us try the command again, but specifying the user name “john” this time.

```
$ goa export --depot-user john
```

```
Error: version for archive john/raw/unix_3rd undefined
```

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

This time, Goa seems to be happy about the depot user name, but it apparently 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. Hence, let’s write the version identifier into a new file called *version*:

```
$ echo 2023-11-15 > version
```

Let’s give `goa export` another try.

```
$ goa export --depot-user john
```

```
[unix_3rd] exported ../unix_3rd/var/depot/john/raw/unix_3rd/2023-11-15
Error: missing README file at pkg/unix_3rd/README
```

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

```
$ ls var/depot/john/raw/unix_3rd/2023-11-15/
```

```
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/unix_3rd/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 [GOSH¹](#) markup syntax, which is consistently used throughout Genode's documentation.

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

```
$ goa export --depot-user john
Error: archive john/raw/unix_3rd/2023-11-15 already exists in the depot
```

You may specify `'--depot-overwrite'` to replace 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. In this case, however, the message results from the partial success above that already exported *raw/unix_3rd*. So you are safe to specify the `--depot-overwrite` argument as suggested by Goa.

Note, you may alternatively use Goa's `bump-version` to change the version. This command sets the version file to the current date or, if this would not change the version, adds/increments an alphabetical suffix.

```
$ goa export --depot-user john --depot-overwrite
```

¹<https://github.com/nfeske/gosh>

```
[unix_3rd] exported ../unix_3rd/var/depot/john/raw/unix_3rd/2023-11-15
[unix_3rd] exported ../unix_3rd/var/depot/john/pkg/unix_3rd/2023-11-15
```

This time, the command succeeded. To celebrate the success, review the content of your part - john's part - of the depot.

```
$ find var/depot/john/
var/depot/john/
var/depot/john/raw
var/depot/john/raw/unix_3rd
var/depot/john/raw/unix_3rd/2023-11-15
var/depot/john/raw/unix_3rd/2023-11-15/terminal.config
var/depot/john/raw/unix_3rd/2023-11-15/vimrc
var/depot/john/raw/unix_3rd/2023-11-15/unix.config
var/depot/john/pkg
var/depot/john/pkg/unix_3rd
var/depot/john/pkg/unix_3rd/2023-11-15
var/depot/john/pkg/unix_3rd/2023-11-15/archives
var/depot/john/pkg/unix_3rd/2023-11-15/runtime
var/depot/john/pkg/unix_3rd/2023-11-15/README
```

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. This command implicitly executes the `goa export` command. So you need to specify all information that you supplied to export.

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

```
[unix_3rd] exported ../unix_3rd/var/depot/john/raw/unix_3rd/2023-11-15
[unix_3rd] exported ../unix_3rd/var/depot/john/pkg/unix_3rd/2023-11-15
Error: missing public key at ../unix_3rd/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" \  
    --gpg-user-id "a@b.cd" \  
    --depot-overwrite
```

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 `--gpg-user-id` can be any GPG user-ID string as understood by GPG. In the example above, we used the email address that we specified for the GPG key pair.

The `--depot-overwrite` argument is specified because Goa tries to prevent us from accidentally overwriting information of existing depot content, like the content you just created with the `goa export` command. 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
```

The content you extracted before is no more. Instead, 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 GPG keyring by using the specified GPG user ID.

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 --depot-overwrite
```

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

```
publish ../var/public/john/pkg/unix_3rd/2023-11-15.tar.xz  
publish ../var/public/john/raw/unix_3rd/2023-11-15.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/raw
var/public/john/raw/unix_3rd
var/public/john/raw/unix_3rd/2023-11-15.tar.xz.sig
var/public/john/raw/unix_3rd/2023-11-15.tar.xz
var/public/john/pkg
var/public/john/pkg/unix_3rd
var/public/john/pkg/unix_3rd/2023-11-15.tar.xz.sig
var/public/john/pkg/unix_3rd/2023-11-15.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](https://en.wikipedia.org/wiki/Rsync)¹ tool has proven to be useful for this purpose. You may use the following combination of arguments:

```
-rplt0vz --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 a launcher file at `/config/launcher/` 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 “Add” tab of 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 first need to export it from gpg in ASCII-armored form.

```
$ gpg --export John > pubkey
```

¹<https://en.wikipedia.org/wiki/Rsync>

The resulting *pubkey* needs to be placed alongside the *download* file that was created by Sculpt in the *depot/john/* directory when you typed in the URL via the Sculpt UI. You may use the “window manager” preset, which includes the “system shell” terminal application for this purpose. In the system shell, you find the user depot at */rw/depot/john/*.

Writing a launcher file Manually creating a launcher file is a good option 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 */config/launcher/unix* with the following content:

```
<launcher pkg="john/pkg/unix_3rd/2023-11-15">
  <route>
    <service name="Gui">
      <child name="wm"/>
    </service>
  </route>
</launcher>
```

The scenario merely requires a Gui service that we route to the “wm” component 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 “+” 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 playground repository, change into the *intro/* directory and create the following index file.

```
<index>
  <supports arch="x86_64"/>

  <index name="Tutorial">
    <pkg path="unix_3rd" info="Unix terminal from tutorial"/>
  </index>
</index>
```

¹https://genode.org/documentation/articles/sculpt-24-10#Runtime_management

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. You can therefore publish your depot index together with the `unix_3rd` package with a single command.

```
intro$ goa publish
[intro] exporting project ../intro/unix_3rd
[unix_3rd] exported ../intro/var/depot/john/raw/unix_3rd/2023-11-15
[unix_3rd] exported ../intro/var/depot/john/pkg/unix_3rd/2023-11-15
...
[intro] exported ../intro/var/depot/john/index/24.04
publish ../intro/var/public/john/pkg/unix_3rd/2023-11-15.tar.xz
publish ../intro/var/public/john/raw/unix_3rd/2023-11-15.tar.xz
publish ../intro/var/public/john/index/24.04.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-24-10>

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`, `stdc++`). 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 name="dev"> <log/> </dir>
  </vfs>
</config>
```

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:

¹<https://genodians.org/jschlatow/2022-03-01-vfs-tap>

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

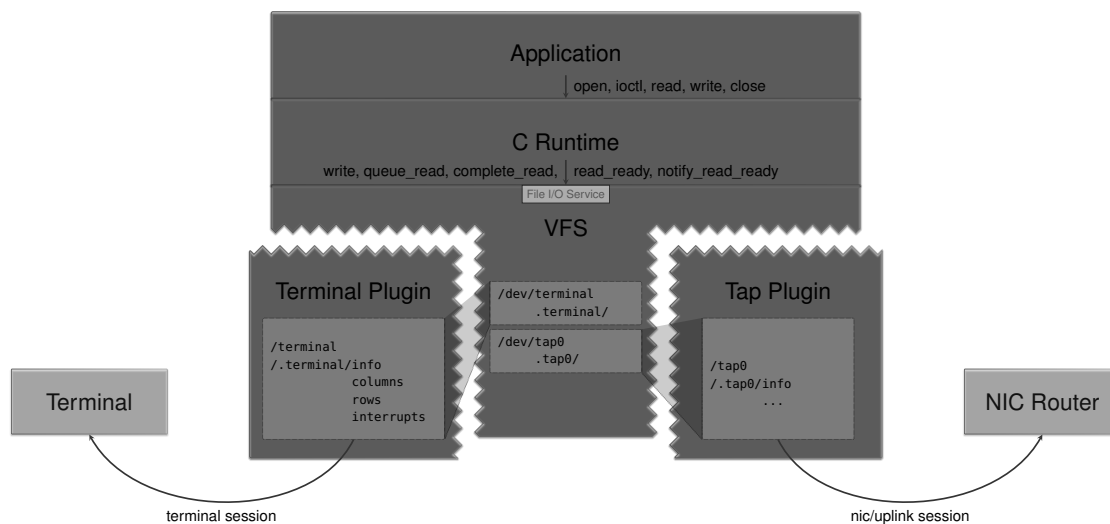
            if (i % 4 == 3)
                printf("\n");
        }

        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 XML report containing a single XML 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 name="tap0" mac_addr="02:02:02:02:02:02" />
```

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

²https://genode.org/documentation/release-notes/20.11#Streamlined_ioctl_handling_in_the_C_runtime__VFS

³<https://github.com/genodelabs/genode/commit/7ac32ea60>

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

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 name="dev">
      <tap name="tap0" label="tap"/>
    </dir>
  </vfs>
</config>
```

In the above example, we mount the plugin at the `/dev/tap0` file. The `name` attribute of the `<tap>` node is mandatory. The plugin shall use a NIC session to transmit the Ethernet frames to a NIC router. The `label` attribute can be used to distinguish multiple session requests at the 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 XML 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. Currently, Goa only supports library projects based on CMake. Thus, you need to 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 3-4 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*.

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

```
namespace Vfs {
    struct Tap_file_system;
}

struct Vfs::Tap_file_system
{
    using Name = String<64>;

    struct Compound_file_system;
    struct Local_factory;
    struct Data_file_system;
};

/* [...] see below */

extern "C" Vfs::File_system_factory *vfs_file_system_factory(void)
{
    struct Factory : Vfs::File_system_factory
    {
        Vfs::File_system
        *create(Vfs::Env &env, Genode::Xml_node config) override
        {
            return new (env.alloc())
                Vfs::Tap_file_system::Compound_file_system(env, config);
        }
    }

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

From the forward declarations, you can already see that the *Tap_file_system* is composed of three parts: 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. Note that you could also make `Tap_file_system` a namespace rather than a struct. The subtle difference here is that the struct emphasizes the inextensibility.

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 corresponding XML node (e.g. `<tap name="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_file_system::Compound_file_system
: private Local_factory, public Vfs::Dir_file_system
{
    private:

        typedef Tap_file_system::Name Name;

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

            Genode::Xml_generator xml(buf, sizeof(buf), "compound", [&] () {

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

                xml.node("dir", [&] () {
                    xml.attribute("name", Name(".", name));
                    xml.node("info",      [&] () {});
                });
            });

            return Config(Genode::CString(buf));
        }

    public:

        Compound_file_system(Vfs::Env &vfs_env, Genode::Xml_node node)
        :
            Local_factory(vfs_env, node),
            Vfs::Dir_file_system(vfs_env, Xml_node(
                _config(Local_factory::name(node)).string()
            ), *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 XML as we are used to when writing a component's `<vfs>` config. In this case, the static `_config()` method generates the following XML:

```
<compound>
  <data name="tap0"/>
  <dir name=".tap0">
    <info/>
  </dir>
</compound>
```

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>` XML node with a name attribute.

Note, the `type()` method is part of the `File_system` interface and must return the XML 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_file_system::Local_factory : File_system_factory
{
    Vfs::Env          &_env;

    Name              const _name;
    Data_file_system  _data_fs { _env.env(), _env.user(), _name };

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

In the first few lines of `Local_factory`, you 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 after we completed the `Local_factory`. Let’s first continue with the *info* file system:


```
struct Vfs::Tap_file_system::Local_factory : File_system_factory
{
    /* [...] see above */

    struct Info
    {
        Name const &_name;

        Info(Name const & name)
        : _name(name)
        { }

        void print(Genode::Output &out) const
        {
            char buf[128] { };
            Genode::Xml_generator xml(buf, sizeof(buf), "tap", [&] () {
                xml.attribute("name", _name);
            });
            Genode::print(out, Genode::CString(buf));
        }
    };

    Info                                     _info          { _name };
    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="...">`. 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 remaining fragment of the `Local_factory` comprises the constructor, an accessor for reading the device name from the `<tap>` node and the `File_system_factory` interface.

```

struct Vfs::Tap_file_system::Local_factory : File_system_factory
{
    /* [...] see above */

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

    static Name name(Xml_node config)
    {
        return config.attribute_value("name", Name("tap"));
    }

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

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

        return nullptr;
    }
};

```

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

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:

```

class Vfs::Tap_file_system::Data_file_system : public Vfs::Single_file_system
{
    private:

        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;
        Handle_registry         _handle_registry { };

    public:

        Data_file_system(Genode::Env      & env,
                        Vfs::Env::User  & vfs_user,
                        Name             const & name)
        :
            Vfs::Single_file_system(Node_type::TRANSACTIONAL_FILE, name.string(),
                                    Node_rwx::rw(), Genode::Xml_node("<data/>")),
            _env(env)
        { }

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

        /* [...] 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 `Single_file_system` constructor takes a node type, a name, an access mode and an `Xml_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 XML 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.

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

```

class Vfs::Tap_file_system::Data_file_system : public Vfs::Single_file_system
{
private:

    /* [...] see above */

public:

    /* [...] 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;

        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,
                                *this, *this, flags);
            return Open_result::OPEN_OK;
        }
        catch (Genode::Out_of_ram) {
            return Open_result::OPEN_ERR_OUT_OF_RAM; }
        catch (Genode::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.

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()`.

```

class Tap_vfs_handle : public Single_file_system::Single_vfs_handle
{
private:

    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();
        }
    }

public:

    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 an 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`.

```
class Tap_vfs_handle : public Single_file_system::Single_vfs_handle
{
    /* [...] (see above) */

    bool read_ready() const override
    {
        /* [...] */
    }

    bool write_ready() const override
    {
        /* [...] */
    }

    Read_result read(char *dst, file_size count,
                    file_size &out_count) override
    {
        if (!read_ready()) {
            _blocked = true;
            return Read_result::READ_QUEUED;
        }

        /* [...] */

        return Read_result::READ_OK;
    }

    Write_result write(char const *src, file_size count,
                      file_size &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 library 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) 2023-11-29
```

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:

```
vfs_tap$ mkdir -p test-vfs_tap/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
```

With these settings in place, you are able to export your *vfs_tap* archive. Let's assume your depot user is "john". 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 export --depot-user john
[vfs_tap] exported [...]var/depot/john/src/vfs_tap/2024-08-02
[vfs_tap] exported [...]var/depot/john/bin/x86_64/vfs_tap/2024-08-02
```

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-vfs_tap/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-vfs_tap/src/Makefile
vfs_tap$ echo "test-vfs_tap" > test-vfs_tap/artifacts
vfs_tap$ echo "genodelabs/api/libc" > test-vfs_tap/used_apis
vfs_tap$ echo "genodelabs/api/posix" >> test-vfs_tap/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 *runtime* file at *test-vfs_tap/pkg/test-vfs_tap*:

```
<runtime ram="20M" caps="1000" binary="init">
  <requires> <timer/> </requires>

  <config>
    <parent-provides>
      <service name="PD"/>
      <service name="CPU"/>
      <service name="LOG"/>
      <service name="ROM"/>
      <service name="Timer"/>
    </parent-provides>

    <default caps="100"/>
    <default-route>
      <service name="Nic"> <child name="nic_router"/> </service>
      <any-service>          <parent/>                      </any-service>
    </default-route>

    <start name="test-vfs_tap">
      <resource name="RAM" quantum="8M"/>
      <config>
        <libc stdout="/dev/log"/>
        <vfs>
          <dir name="dev"> <log/>
                                <tap name="tap0"/> </dir>
        </vfs>
      </config>
    </start>

    <start name="nic_router">
      <resource name="RAM" quantum="2M"/>
      <provides> <service name="Nic"/>
                  <service name="Uplink"/> </provides>
      <config verbose_domain_state="yes" verbose="yes">
        <default-policy domain="default"/>
        <domain name="default" interface="10.0.2.1/24"/>
      </config>
    </start>

    <start name="ping">
      <resource name="RAM" quantum="4M"/>
      <config interface="10.0.2.2/24" gateway="10.0.2.1"
              dst_ip="10.0.2.123" period_sec="5" verbose="no"/>
    </start>
  </config>

  <!-- [...] see below -->
</runtime>
```

```

<runtime>
  <!-- [...] see above -->

  <content>
    <rom label="test-vfs_tap"/>
    <rom label="libc.lib.so"/>
    <rom label="libm.lib.so"/>
    <rom label="posix.lib.so"/>
    <rom label="vfs.lib.so"/>
    <rom label="vfs_tap.lib.so"/>
    <rom label="ping"/>
    <rom label="nic_router"/>
  </content>

</runtime>

```

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 *archives* file at *test-vfs_tap/pkg/test-vfs_tap*:

```

genodelabs/src/init
genodelabs/src/libc
genodelabs/src/vfs
genodelabs/src/posix
genodelabs/src/nic_router
john/src/vfs_tap
jschlatow/src/ping/2024-04-11

```

Now, you can give the test scenario a try:

```

vfs_tap$ goa run -C test-vfs_tap/
Genode sculpt-24.04
17592186044415 MiB RAM and 18997 caps assigned to init
[init -> test-vfs_tap -> nic_router] [default] static IP config: interface ...
[init -> test-vfs_tap -> nic_router] [default] NIC sessions: 0
[init -> test-vfs_tap -> nic_router] [default] initiated domain
[init -> test-vfs_tap -> nic_router] [default] NIC sessions: 1
[init -> test-vfs_tap -> nic_router] [default] NIC sessions: 2
[init -> test-vfs_tap -> test-vfs_tap] MAC: 02:02:02:02:02:02
[init -> test-vfs_tap -> nic_router] [default] forward ARP request for local
  IP to all interfaces of the sender domain
[init -> test-vfs_tap -> test-vfs_tap] Received packet with 42 bytes
[init -> test-vfs_tap -> test-vfs_tap] ffffffff 0202ffff 01020202 01000608
[init -> test-vfs_tap -> test-vfs_tap] 04060008 02020100 01020202 0202000a
[init -> test-vfs_tap -> test-vfs_tap] ffffffff 000affff 027b

```

Excellent! The complete code of this tutorial is available on [github](#)¹. The official implementation of the `vfs_tap` plugin is part of the [Genode repository](#)².

¹https://github.com/jschlatow/goa-projects/tree/master/examples/vfs_tap

²<https://github.com/genodelabs/genode/tree/master/repos/os/src/lib/vfs/tap>

5.4 Porting Lomiri Calculator App

This section is based on the [article¹](https://genodians.org) at <https://genodians.org>.

Since the [port of Ubuntu UI Toolkit to Genode²](#), 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 can be found in [Sebastian Sumpf's depot³](#).

This section walks through the porting procedure of the [Lomiri Calculator App⁴](#) 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
```

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.

With the *import* file present, you are able to run `goa import`, which places the source code into the *src/* subdirectory.

¹<https://genodians.org/jschlatow/2024-01-11-lomiri-calculator-porting>

²https://genodians.org/ssumpf/2023-05-06-ubunutu_ui

³<https://depot.genode.org/ssumpf>

⁴<https://gitlab.com/ubports/development/apps/lomiri-calculator-app/>

```
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 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
...
CMake Error at CMakeLists.txt:16 (find_package):
  By not providing "FindQt5Core.cmake" in CMAKE_MODULE_PATH this project has
  asked CMake to find a package configuration file provided by "Qt5Core", but
  CMake did not find one.

Could not find a package configuration file provided by "Qt5Core" with any
of the following names:

  Qt5CoreConfig.cmake
  qt5core-config.cmake

Add the installation prefix of "Qt5Core" to CMAKE_PREFIX_PATH or set
"Qt5Core_DIR" to a directory containing one of the above files.  If
"Qt5Core" provides a separate development package or SDK, be sure it has
been installed.

[calculator:cmake] -- Configuring incomplete, errors occurred!
```


Apparently, CMake is unable to locate Qt5Core. The error message suggests providing a file named *FindQt5Core.cmake*. Goa is able to locate these files in the used API archives. Knowing that Qt5Core is part of the *qt5_base* archive, let's add this to the *used_apis* file.

```
calculator$ echo 'ssumpf/api/qt5_base' > used_apis
[lomiri-calculator-app] Error: no version defined for depot
archive 'ssumpf/api/qt5_base'
```

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:

```
calculator$ echo 'set version(ssumpf/api/qt5_base) 2024-04-18' >> goarc
calculator$ goa build
...
CMake Error at CMakeLists.txt:17 (find_package):
  By not providing "FindQt5Qml.cmake" in CMAKE_MODULE_PATH this project has
  asked CMake to find a package configuration file provided by "Qt5Qml", but
  CMake did not find one.

Could not find a package configuration file provided by "Qt5Qml" with any
of the following names:

  Qt5QmlConfig.cmake
  qt5qml-config.cmake

Add the installation prefix of "Qt5Qml" to CMAKE_PREFIX_PATH or set
"Qt5Qml_DIR" to a directory containing one of the above files.  If "Qt5Qml"
provides a separate development package or SDK, be sure it has been
installed.

[calculator:cmake] -- Configuring incomplete, errors occurred!
```

Qt5Qml is part of the *qt5_declarative* archive. Let's add the corresponding API archive and version information:

```
calculator$ echo 'ssumpf/api/qt5_declarative' >> used_apis
calculator$ echo 'set version(ssumpf/api/qt5_declarative) 2024-02-25' >> goarc
calculator$ goa build
...
[calculator:cmake] -- Build files have been written to:
                        /.../goa-projects/calculator/var/build/x86_64
...
[calculator:cmake] [ 98%] Built target com_ubuntu_calculator_translation_files
[calculator:cmake] [100%] Built target pofiles_84
[calculator:cmake] Install the project...
[calculator:cmake] -- Install configuration: ""
[calculator:cmake] -- Installing: //manifest.json
CMake Error at cmake_install.cmake:49 (file):
  file INSTALL cannot copy file
  "/.../goa-projects/calculator/var/build/x86_64/manifest.json" to
  "//manifest.json": Permission denied.

make: *** [Makefile:110: install] Error 1
[calculator] Error: install via cmake failed:
  child process exited abnormally
```

Yikes, the build succeeded but the installation failed writing to the file-system root. This is puzzling 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 `/`. Fortunately, Goa allows providing arbitrary arguments to CMake via a `cmake_args` file. Setting `CLICK_MODE=0` should do the trick:

```
calculator$ echo '-DCLICK_MODE=0' > cmake_args
calculator$ goa build
...
CMake Error at tests/autopilot/cmake_install.cmake:49 (file):
  file INSTALL cannot make directory
  "/usr/lib/python3.11/site-packages/ubuntu_calculator_app": Permission
  denied.
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: /.../goa-projects/calculator/var/build
/x86_64/install/bin/ubuntu-calculator-app
...
[calculator:cmake] -- Installing: /.../goa-projects/calculator/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 to the *artifacts* file. For more details, please refer to `goa help artifacts`:

```
ubuntu-calculator-app.tar: install/
```

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 the default package archive of a project to be named after the project, hence you need to create a *pkg/calculator* directory.

```
calculator$ mkdir pkg/calculator
```

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 label="mesa_gpu_drv.lib.so"/>
    <gpu/>
    <rtc/>
    <timer/>
    <report label="shape"/>
  </requires>

  <config>
    <vfs>
      <dir name="dev">
        <log/> <gpu/> <rtc/>
      </dir>
      <dir name=".local"> <ram/> </dir>
      <dir name="pipe"> <pipe/> </dir>
      <tar name="qt5_declarative_qml.tar"/>
      <tar name="qt5_dejavusans.tar"/>
      <tar name="qt5_graphicaleffects_qml.tar"/>
      <tar name="qt5_libqgenode.tar"/>
      <tar name="qt5_libqjpeg.tar"/>
      <tar name="qt5_libqsvg.tar"/>
      <tar name="ubuntu-ui-toolkit_qml.tar"/>
      <tar name="ubuntu-themes.tar"/>

      <!-- change to you projects tar file here -->
      <tar name="ubuntu-calculator-app.tar"/>

    </vfs>
    <libc stdout="/dev/log" stderr="/dev/log" pipe="/pipe" rtc="/dev/rtc"/>
    <arg value="ubuntu-ui-toolkit-launcher"/>

    <!-- add your startup QML file here -->
    <arg value="/share/ubuntu-calculator-app/ubuntu-calculator-app.qml"/>

    <env key="QT_SCALE_FACTOR" value="1"/>
  </config>

  <content>
    <!-- adjust to your tar -->
    <rom label="ubuntu-calculator-app.tar"/>
  </content>
</runtime>
```

With this *runtime* file at *pkg_calculator*, 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 label="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/calculator/archives
calculator$ echo 'set version(ssumpf/pkg/ubuntu_ui_toolkit) 2024-06-03' \
>> 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](https://ubports.com)¹.

In order to make Suru available on Genode, you need to create a separate Goa project.

Porting QtQuick Controls Suru Style Following the steps already taken for the calculator app, you create the project directory *qt5_quickcontrols2_suru/* with the following *import* file:

¹<https://gitlab.com/ubports/development/core/qqc2-suru-style>

```
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: could not find matching qt5_base API in depot
```

Goa detected that this is a qmake project and is therefore looking for a *qt5_base* API archive. However, you haven't defined this in the *used_apis* file yet. Let's fix this:

```
qt5_quickcontrols2_suru$ echo "genodelabs/api/qt5_base" > used_apis
```

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.

```
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>
      |           ^~~~~~
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:
  cannot find -l:qt5_component.lib.so: No such file or directory
/[...]/x86_64-pc-elf/bin/ld: cannot find
  /[...]/libQt5Quick.lib.so: No such file or directory
/[...]/x86_64-pc-elf/bin/ld: cannot find
  /[...]/libQt5QmlModels.lib.so: No such file or directory
/[...]/x86_64-pc-elf/bin/ld: cannot find
  /[...]/libQt5Qml.lib.so: No such file or directory
/[...]/x86_64-pc-elf/bin/ld: cannot find
  /[...]/libQt5QuickControls2.lib.so: No such file or directory
/[...]/x86_64-pc-elf/bin/ld: cannot find
  /[...]/libQt5QuickTemplates2.lib.so: No such file or directory
/[...]/x86_64-pc-elf/bin/ld: cannot find
  /[...]/libQt5Quick.lib.so: No such file or directory
/[...]/x86_64-pc-elf/bin/ld: cannot find
  /[...]/libQt5QmlModels.lib.so: No such file or directory
/[...]/x86_64-pc-elf/bin/ld: cannot find
  /[...]/libQt5Qml.lib.so: No such file or directory
collect2: error: ld returned 1 exit status
...

```

There are a bunch of library files missing. Goa creates these from the symbol files found in the used API archives. `ldso_so_support` is provided by `genodelabs/api/so`, `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 Goa 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. In a last step, you need to export the resulting archive into your depot. Let's assume your depot user is "john" and that you are using `~/depot` as a shared depot directory:

```
qt5_quickcontrols2_suru$ goa export --depot-user john --depot-dir ~/depot
[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 --depot-user john --depot-dir ~/depot
[qt5_quickcontrols2_suru] Error: version for
                                archive john/src/qt5_quickcontrols2_suru undefined
```

Create a 'version' file in your project directory, or define 'set version(jschlatow/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 --depot-user john --depot-dir ~/depot
[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. More precisely, add a `<tar>` node to the `vfs` and a `<rom>` node to the list of content ROM modules.

```
<config>
  <vfs>
    ...
    <tar name="qt5_quickcontrols2_suru_qml.tar"/>
    ...
  </vfs>
</config>

<content>
  ...
  <rom label="qt5_quickcontrol2_suru_qml.tar"/>
  ...
</content>
```

Before giving `goa run` a go, don't forget to add the corresponding depot archive to the `archives` file.

```
calculator$ echo "john/src/qt5_quickcontrols2_suru" >> pkg/calculator/archives
calculator$ goa run
...
[calculator] Error: no version defined for depot
                archive 'john/src/qt5_quickcontrols2_suru'
```

Goa is unable to find any version information for the archive. Instead of adding the version definition to a *goarc* file, you may use Goa's ability to locate the corresponding project directory in order to find its version information. By default, Goa uses the working directory as a starting point for locating those dependencies. This can be changed by adding a `--search-dir` argument or by setting the `search_dir` variable in a *goarc* file. Let's opt for the latter and also set the `depot_dir` variable to point Goa to the depot directory to which you exported the *qt5_quickcontrols2_suru* project.

```
calculator$ echo "set search_dir ../" >> goarc
calculator$ echo "set depot_dir ~/depot" >> goarc
calculator$ goa run
...
[init -> calculator] Error: ROM-session creation failed
                    (label="libqtquickcontrols2surustyleplugin.lib.so",...)
[init -> calculator] Error: could not open ROM session
                    for "libqtquickcontrols2surustyleplugin.lib.so"
[init -> calculator] QQmlComponent: Component is not ready
...
```

The library file is provided by the *qt5_quickcontrols2_suru* archive, however, the runtime error indicates that we missed adding it to the content section of the *runtime* file.

```
<content>
  ...
  <rom label="libqtquickcontrols2surustyleplugin.lib.so"/>
  ...
</content>
```

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
```

5.4 Porting Lomiri Calculator App

Apparently, we need a database driver. Fortunately, *qt5_libsqlite.tar* and *libsqlite.lib.so* are part of the *qt5_base* binary archive. Let's add them to the *runtime* file:

```
<config>
  <vfs>
    ...
    <tar name="qt5_libsqlite.tar"/>
    ...
  </vfs>
</config>

<content>
  ...
  <rom label="qt5_libsqlite.tar"/>
  <rom label="libsqlite.lib.so"/>
  ...
</content>
```

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.

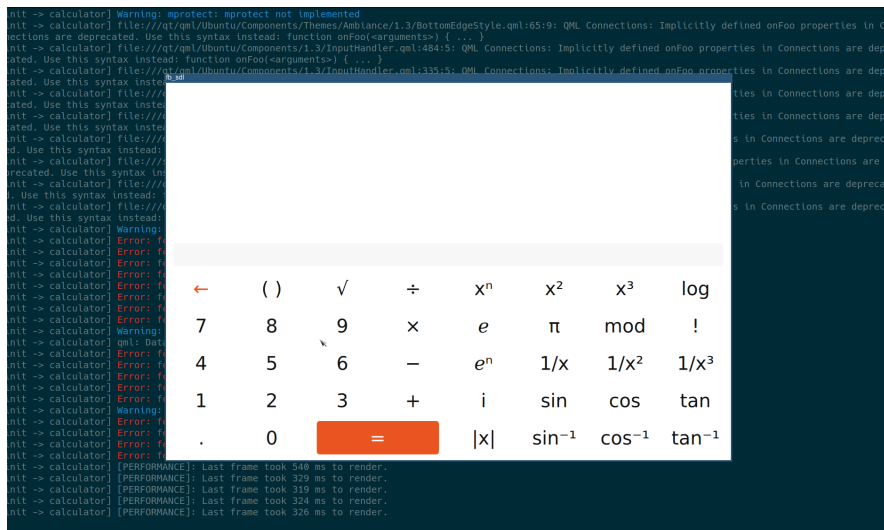


Figure 14: Calculator app running on base-linux with Goa

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://github.com/jschlatow/goa-projects/tree/master/lomiri>