UBports Documentation

Marius Gripsgard

Apr 09, 2019

About

Welcome to the official documentation of the UBports project!

UBports develops the mobile phone operating system Ubuntu Touch. Ubuntu Touch is a mobile operating system focused on ease of use, privacy, and convergence.

On this website you find guides to *install Ubuntu Touch on your mobile phone*, *use Ubuntu Touch*, *develop Ubuntu Touch apps*, *port Ubuntu Touch to an Android handset* and *learn more about system components*. If this is your first time here, please consider reading our *introduction*.

If you want to help improving this documentation, *the Documentation contribute page* will get you started.

You may view this documentation in the following languages:

- English
- Català
- Français
- Deutsch
- Italiano
- Română
- Türkçe
- Español

CHAPTER 1

Introduction

This is the documentation for the UBports project. Our goal is to create a Free and open-source (GPL if possible) mobile operating system that converges and respects your freedom.

1.1 About UBports

The UBports project was founded by Marius Gripsgard in 2015 and in its infancy was a place where developers could share ideas and educate each other in hopes of bringing the Ubuntu Touch platform to more mobile devices.

After Canonical suddenly announced their plans to terminate support of Ubuntu Touch in April 2017, UBports and its sister projects began work on the open-source code, maintaining and expanding its possibilities for the future.

1.2 About the Documentation

This documentation is always improving thanks to the members of the UBports community. It is written in ReStructuredText and converted into this readable form by Sphinx, ReCommonMark, and Read the Docs. You can start contributing by checking out the *Documentation contribution document*.

1.3 Attribution

This documentation was heavily modeled after the Godot Engine's Documentation, attribution to Juan Linietsky, Ariel Manzur and the Godot community.

CHAPTER 2

Processes

This section of the documentation details standardized processes for different teams.

Note: The process definitions are still a work in progress and need to be completed by the respective teams.

2.1 Issue-Tracking Guidelines

This document describes the standard process of dealing with new issues in UBports projects. Not to be confused with the *guide on writing a good bugreport*.

2.1.1 Where are bugs tracked?

Since our quality assurance depends heavily on the community, we try to track issues where the user would expect them, instead of separated by repository. This means, that issues of almost all components that are distributed as with the system-image are tracked in the Ubuntu Touch tracker. An exception of this are click-apps that can be updated independently through the OpenStore.

Most other repositories track issues locally. If you're unsure whether a repository uses its own tracker or not, consult the README.md file. Repositories that don't track issues locally have their bugtracker disabled.

This page is mainly about the Ubuntu Touch tracker, but most principles apply to other projects as well.

Note: Practicality beats purity! Exceptions might apply and should be described in the projects README.md file.

2.1.2 GitHub projects

To increase transparency and communication, GitHub projects (Kanban-Boards) are used wherever practical. In case of github.com/ubports/ubuntu-touch, a single project is used for all issues. Projects support filtering by labels, so that

only issues that belong to a specific team or affect a specific device can be viewed.

These are the standard columns:

- **None (awaiting triage)**: The issue has been approved by a member of the QA team and is awaiting review from the responsible development team. if the issue is a bug, instructions to reproduce are included in the issue description. if the issue is a feature request, it has passed a primary sanity check by the qa-team but has not yet been accepted by the responsible development-team.
- **Accepted**: The issue has been accepted by the responsible development-team. If the issue is a bugreport, the team has decided that it should be fixable and accepts the responsibility. If the issue is a feature request, the team thinks it should be implemented as described.
- **In Development**: A patch is in development. Usually means that a developer is assigned to the issue.
- **Quality Assurance**: The patch is completed and has passed initial testing. The QA team will now review it and provide feedback. If problems are found, the issue is moved back to "Accepted".
- **Release Candidate**: The patch has passed QA and is ready for release. In case of deb-packages that are included in the system-image, the patch will be included in the next over-the-air update on the rc channel, and, if everything goes well, in the next release of the stable channel.
- **None (removed from the project)**: If the issue is open and labeled "help wanted", community contributions are required to resolve the issue. If it's closed, that means that either a patch has been released on the stable channel (a comment on the issue should link to the patch) or the issue has been rejected (labeled "wontfix").

2.1.3 Labels

All issues - even closed ones - should be labeled to allow the use of GitHub's global filtering. For example, these are all of the issues labeled 'enhancement' inside @ubports. Consult the GitHub help pages for more information on search and filtering.

Here's a list of labels that are normally used by all repositories.

- **needs confirmation**: The bug needs confirmation and / or further information from affected users
- **bug**: This issue is a confirmed bug. If it's reproducible, reproduction steps are described.
- **opinion**: This issue needs further discussion.
- **enhancement**: This issue is a feature request.
- **question**: This issue is a support request or general question.
- **invalid**: This issue can not be confirmed or was reported in the wrong tracker.
- **duplicate**: This has already been reported somewhere else. Please provide a link and close.
- **help wanted**: This issue is ready to be picked up by a community developer.
- **good first issue**: This issue is not critical and trivial to fix. It is reserved for new contributors as a place to start.
- **wontfix**: It does not make sense to fix this bug, since it will probably resolve itself, it will be too much work to fix it, it's not fixable, or an underlying component will soon change.

Additional special labels can be defined. As an example, here's the labels used in the Ubuntu Touch tracker:

- **critical (devel)**: This critical issue that only occurs on the devel channel is blocking the release of the next rc image.
- **critical (rc)**: This critical issue that only occurs on the devel and rc channel is blocking the release of the next stable release. Usually, issues that can not simply be moved to a different release and have the power to postpone the release are labeled this.

- **device: [DEVICE CODENAME]**: This issue affects only the specified device(s).
- **team: [TEAM NAME]**: This issue is falls under the responsibility of a specific team (hal, middleware, UI).

Note: If a repository that tracks issues locally defines it's own labels, they should be documented in the README.md.

2.1.4 Milestones

Milestones are used for stable OTA releases only. In general, milestones for the work-in-progress OTA and the next OTA are created. The ETA is set, once the work on the release starts (that is 6 weeks from start date), but can be adjusted afterwards. See the *release-schedule* for more info.

2.1.5 Assignees

To make it transparent who's working on an issue, the developer should be assigned. This also allows the use of GitHub's global filtering as a type of TODO list. For example, this is everything assigned to mariogrip in @ubports.

Developers are encouraged to keep their list short and update the status of their issues.

2.1.6 Examples

Bug Lifecycle

Note: The same principle applies to feature requests. The only difference is, that instead of the label **bug**, the label **enhancement** is used. The **needs confirmation** label is not applicable for feature requests.

- A *User* files a new bug using the issue-template.
- The *QA-Team* adds the label **needs confirmation** and tries to work with the user to confirm the bug and add potentially missing information to the report. Once the report is complete a **team-label** will be added to the issue, the issue will be put on the **awaiting-triage-list** of the project and the label needs confirmation will be replaced with **bug**.
- The affected *Team* will triage the issue and either reject (label **wontfix**, close and remove from the project) or accept the issue. The team decides if it they will fix the issue in-house (move to "Accepted" and assign a team member) or wait for a community developer to pick it up (Label **help wanted**, remove from the project board and provide hints on how to resolve the issue and further details on the how the fix should be implemented if necessary). For non-critical issues that are trivial to fix, the label **good first issue** can be added as well.
- Once a *developer* is assigned and starts working on the issue, it is moved to "In Development". As soon as they have something to show for, the issue is closed and automatically moved to "Quality Assurance" for feedback from the QA team. If necessary, the developer will provide hints on how to test his patch in a comment on the issue.
- The *QA-Team* tests the fix on all devices and provides feedback to the developer. If problems are found, the issue is re-opened and goes back to "Accepted", else it's moved to "Release Candidate" to be included in the next release.
- If not done already, the issue is added to the next milestone. Once the milestone is released, the issue is removed from the project board.

2.2 Release Schedule

OTA updates usually follow this rhythm:

- devel: daily builds
- rc: weekly if no critical issue exists in the devel channel
- stable: every six through eight weeks, if no critical issue exists in the rc channel

This is not a definitive cycle. Stable releases are ready when they're ready and should not introduce new bugs or ship very incomplete features. Since the OTA update can not be released before all critical issues are closed, the ETA might have to be moved by making an educated guess for when all the issues can be handled. If there are too many issues added to a milestone, they are either removed or added to the next milestone.

2.3 repo.ubports.com

This is the package archive system for UBports projects. It hosts various PPAs containing all deb-components of Ubuntu Touch.

2.3.1 Repository naming convention

Native packages

Native packages are repositories which contain a `debian/` directory **with** the source used to create the Debian source package (`.dsc`).

The name of the Debian source package generated from the repository and the name of the git-repository should be the same.

Packaging repositories

Packaging repositories contain a `debian/` directory **without** the source used to create the Debian source package (`.dsc`). They also contain instructions to tell `debhelper` how to get the source used to create the source package.

The repository should have the name of the source package it generates with `-packaging` appended to the end. For example, a packaging repository that generates a source package called `sandwich` would be called `sandwich-packaging`.

2.3.2 Creating new PPAs

New PPAs can be created dynamically by the CI server using a special *git-branch naming convention*. The name of the branch translates literally to the name of the PPA: `http://repo.ubports.com/dists/[branch name]`

Non-standard PPAs (i.e. not `xenial`, `vivid` or `bionic`) are kept for three months. In case they need to be kept for a longer time, a file with the name `ubports.keep` can be created in the root of the repository, containing the date until which the PPA shall be kept open in the form of `YYYY-MM-dd`. If this file is empty, the PPA will be kept for two years after the last build.

2.4 Branch-naming convention

Our branch-naming conventions ensure that software can be built by our CI and tested easily by other developers.

Every Git repository's README file should state which branch-naming convention is used and possible deviations from the norm.

2.4.1 Click-Packages

Software that is exclusively distributed as a click-package (and not also as a deb) only uses one `master` branch that is protected. Separate temporary development branches with arbitrary descriptive names can be created and merged into master when the time comes. For marking and archiving milestones in development history, ideally Git tags or GitHub releases should be used.

2.4.2 Deb-Packages

To make most efficient use of our CI system, a special naming convention for git-branches is used.

For pre-installed Ubuntu Touch components, deb-packages are used wherever possible. This includes Core Apps, since they can still be independently updated using click-package downloads from the OpenStore. This policy allows us to make use of the powerful Debian build system to resolve dependencies.

Every repository that uses this convention will have branches for the actively supported Ubuntu releases referenced by their codenames (`bionic`, `xenial`, `vivid`, etc.). These are the branches that are built directly into the corresponding images and published on *repo.ubports.com*. If no separate versions for the different Ubuntu bases are needed, the repository will just have one `master` branch and the CI system will still build versions for all actively supported releases and resolve dependencies accordingly.

Branch-extensions

To build and publish packages based on another repository, an extension in the form of `xenial_-_some-descriptive_extension` can be used. The CI system will then resolve all dependencies using the `xenial_-_some-descriptive_extension` branch of other repositories or fall back on the normal `xenial` dependencies, if that doesn't exist. These special dependencies are not built into the image but still pushed to on *repo.ubports.com*.

Multiple branch extensions can be chained together in the form of `xenial_-_dependency-1_-_dependency-2_-_dependen`
This means that the CI system will look for dependencies in the following repositories:

```
xenial
xenial_-_dependency-1
xenial_-_dependency-1_-_dependency-2
xenial_-_dependency-1_-_dependency-2_-_dependency-3
```

Note: There is no prioritization, so the build system will always use the package with the highest version number or the newest build if the version is equal.

Dependency-file

For complex or non-linear dependencies, a `ubports.depends` file can be created in the root of the repository to specify additional dependencies. The branch name will be ignored if this file exists.

```
xenial
xenial_-_dependency-1_-_dependency-2_-_dependency-3
xenial_-_something-else
```

Note: The `ubports.depends` file is an **exclusive list**, so the build system will not resolve dependencies linearly like it does in a branch name! Every dependency has to be listed. You will almost always want to include your base release (i.e. `xenial`).

CHAPTER 3

Install Ubuntu Touch

Installing Ubuntu Touch is easy, and a lot of work has gone in to making the installation process less intimidating to less technical users. The UBports Installer is a nice graphical tool that you can use to install Ubuntu Touch on a supported device from your Linux, Mac or Windows computer.

Warning: If you're switching your device over from Android, you will not be able to keep any data that is currently on the device. Create an external backup if you want to keep it.

Go to the download page and download the version of the installer for your operating system:

- Windows: `ubports-installer-<version-number>.exe`
- macOS: `ubports-installer-<version-number>.dmg`
- Ubuntu or Debian: `ubports-installer-<version-number>.deb`
- Other Linux distributions: `ubports-installer-<version-number>.snap` or `ubports-installer-<version-number>.AppImage`

Start the installer and follow the on-screen instructions that will walk you through the installation process. That's it! Have fun exploring Ubuntu Touch!

Note: If your device is not detected automatically when using the snap installer, then unplug your device, open a terminal and enter the following command:

```
sudo snap connect ubports-installer:raw-usb
```

Then restart the installer, plug in your device and give it another try.

If you're an experienced android developer and want to help us bring Ubuntu Touch to more devices, visit the *porting section*.

3.1 Install on legacy Android devices

While the installation process is fairly simple on most devices, some legacy Bq and Meizu devices require special steps. This part of the guide does not apply to other devices.

Note: This is more or less uncharted territory. If your device's manufacturer does not want you to install an alternative operating system, there's not a lot we can do about it. The instructions below should only be followed by experienced users. While we appreciate that lots of people want to use our OS, flashing a device with OEM tools shouldn't be done without a bit of know-how and plenty of research.

Meizu devices are pretty much stuck on Flyme. While the MX4 can be flashed successfully in some cases, the Pro5 is Exynos-based and has its own headaches.

Warning: BE VERY CAREFUL! You are responsible for your own actions!

1. Disconnect all devices and non-essential peripherals from your PC. Charge your device on a wall-charger (not your PC) to at least 40 percent
2. Download the Ubuntu Touch ROM for your device:
   - Bq E4.5 (*krillin*)
   - Bq E5 HD (*vegetahd*)
   - Bq M10 HD (*cooler*)
   - Bq M10 FHD (*frieza*)
   - Meizu MX4 (*arale*)
3. Download SP flash tool for Linux.

On Ubuntu 17.10, there are issues with flash_tool loading the shared library 'libpng12', so this can be used as a workaround:

```
wget -q -O /tmp/libpng12.deb http://mirrors.kernel.org/ubuntu/pool/main/libp/libpng/
↪libpng12-0_1.2.54-1ubuntu1_amd64.deb \
&& sudo dpkg -i /tmp/libpng12.deb \
&& rm /tmp/libpng12.deb
```

You will also need to use the latest version of the tool.

4. Extract the zip files
5. Start the tool with `sudo`
6. Select the `*Android_scatter.txt` file from the archive you downloaded in the first step as the scatter-loading file
7. Choose "Firmware Upgrade"
8. Double-check you chose "Firmware Upgrade" and not "Download" or "Format All"

Warning: If you select DOWNLOAD rather than FIRMWARE UPGRADE, you will end up with a useless brick rather than a fancy Ubuntu Touch device. Be sure to select FIRMWARE UPGRADE.

9. Turn your device completely off, but do not connect it yet

10. Press the button labeled “Download”
11. Perform a final sanity-check that you selected the “Firmware Upgrade” option, not “Download”
12. Make sure your device is off and connect it to your PC. Don’t use a USB 3.0 port, since that’s known to cause communication issues with your device.
13. Magic happens. Your device will boot into a super old version of Ubuntu Touch.
14. Congratulations! Your device will now boot into a very old version of Ubuntu Touch. You can now use the UBports Installer to proceed.

CHAPTER 4

Daily use

This section of the documentation details common tasks that users may want to perform while using their Ubuntu Touch device.

4.1 Run desktop applications

Libertine allows you to use standard desktop applications in Ubuntu Touch.

To display and launch applications you need the *Desktop Apps Scope* which is available in the Open Store. To install applications you need to use the commandline as described below.

4.1.1 Manage containers

Create a container

The first step is to create a container where applications can be installed:

```
libertine-container-manager create -i CONTAINER-IDENTIFIER
```

You can add extra options such as:

- `-n name` name is a more user friendly name of the container
- `-t type` type can be either `chroot` or `lxc`. Default is `chroot` and is compatible with every device. If the kernel of your device supports it then `lxc` is suggested.

The creating process can take some time, due to the size of the container (some hundred of megabytes).

Note: The `create` command shown above cannot be run directly in the terminal app, due apparmor restrictions. You can run it from another device using either `adb` or `ssh` connection. Alternatively, you can run it from the terminal app using a loopback ssh connection running this command: `ssh localhost`.

List containers

To list all containers created run:

```
libertine-container-manager list
```

Destroy a container

```
libertine-container-manager destroy -i CONTAINER-IDENTIFIER
```

4.1.2 Manage applications

Once a container is set up, you can list the installed applications:

```
libertine-container-manager list-apps
```

Install a package:

```
libertine-container-manager install-package -p PACKAGE-NAME
```

Remove a package:

```
libertine-container-manager remove-package -p PACKAGE-NAME
```

Note: If you have more than one container, then you can use the option `-i CONTAINER-IDENTIFIER` to specify for which container you want to perform an operation.

4.1.3 Files

Libertine applications do have access to these folders:

- Documents
- Music
- Pictures
- Downloads
- Videos

4.1.4 Tips

Locations

For every container you create there will be two directories created:

- A root directory `~/.cache/libertine-container/CONTAINER-IDENTIFIER/rootfs/` and
- a user directory `~/.local/share/libertine-container/user-data/CONTAINER-IDENTIFIER/`

Shell access

There are 2 options for executing commands inside the container.

The first option is based on `libertine-container-manager exec`. It lets you run your commands as root. The drawback is that the container is not completely set up. So far we know that the folders mentioned above (Documents, Music, . . .) are not mounted i.e., the `/home/phablet/` directory is empty. Likewise the directory referenced in `TMPDIR` is not available what may lead to problems with software that tries to create temporary files or directories. You may use this option e.g., for installing packages.

To execute a command you can use the following pattern:

```
libertine-container-manager exec -i CONTAINER-IDENTIFIER -c "COMMAND-LINE"
```

For example run:

```
libertine-container-manager exec -i CONTAINER-IDENTIFIER -c "apt-get --help"
```

To get a shell into your container as `root` run:

```
libertine-container-manager exec -i CONTAINER-IDENTIFIER -c "/bin/bash"
```

The second option is based on `libertine-launch`. It will execute your commands as user phablet in a completely set up container. So you may use this option to modify your files using installed packages.

To execute a command you can use the following pattern:

```
libertine-launch -i CONTAINER-IDENTIFIER COMMAND-LINE
```

For example run:

```
libertine-launch -i CONTAINER-IDENTIFIER ls -a
```

To get a shell as user `phablet` run:

```
DISPLAY= libertine-launch -i CONTAINER-IDENTIFIER /bin/bash
```

Note: When you launch bash in this way you will not get any specific feedback to confirm that you are now *inside* the container. You can check `ls /` to confirm for yourself that you are inside the container. The listing of `ls /` will be different inside and outside of the container.

Accessing SD card

In order to access your SD-card or any other part of the regular filesystem from inside your libertine container you must create a bind mount.

In order to add a bind mount use:

```
libertine-container-manager configure -i CONTAINER-IDENTIFIER -b add -p /media/
↪phablet/ID-OF-SD
```

You can also make deep links in case you only want parts of your SD-card available in the container. In this case just the entire path to the directory you want to bind mount:

```
libertine-container-manager configure -i CONTAINER-IDENTIFIER -b add -p /media/
↪phablet/ID-OF-SD/directory/you/want
```

This will not allow the container access to any of the directories earlier in the path for anything other than accessing your mounted directory.

In order to use the SD-card as extra space for your container, make sure first to format it using ext4 or similar. There is a mis-feature in udisk2 that mounts SD-cards (showexec) that ensures only files ending in .bat, .exe or .com can be executed from the drive if it is (v)fat formatted. This has been changed in other distributions allowing any file to have execute priviliges, but not ubuntu. The reccomended workaround is to add a udev rule to control how to mount a card with a given id, but since the udev rules are on the read only port on touch, this is not possible.

Shortcuts

If you want, you can add aliases for command line tools. Add lines like the following ones to your `~/.bash_aliases`:

```
alias git='libertine-launch -i CONTAINER-IDENTIFIER git'
alias screenfetch='libertine-launch -i CONTAINER-IDENTIFIER screenfetch'
```

4.1.5 Background

A display server coordinates input and output of an operating system. Most Linux distributions today use the X server. Ubuntu Touch does not use X, but a new display server called Mir. This means that standard X applications are not directly compatible with Ubuntu Touch. A compatibility layer called XMir resolves this. Libertine relies on XMir to display desktop applications.

Another challenge is that Ubuntu Touch system updates are released as OTA images. A consequence of this is that the root filesystem is read only. Libertine provides a container with a read-write filesystem to allow the installation of regular Linux desktop applications.

4.2 Android apps

Anbox is a minimal Android container and compatibility layer that allows you to run Android apps on GNU/Linux operating systems such as UBports.

Note:

- When "host" is used in this document, it refers to another device which you can connect your Ubuntu Touch device to. Your host device must have `adb` and `fastboot` installed.

4.2.1 Supported devices

Make sure your device is supported:

- Meizu Pro 5 (codename: `turbo`, name of the boot partition: `bootimg`)
- Fairphone 2 (codename: `FP2`, name of the boot partition: `boot`)
- OnePlus One (codename: `bacon`, name of the boot partition: `boot`)

- Nexus 5 (codename: `hammerhead`, name of the boot partition: `boot`)
- BQ M10 HD (codename: `cooler`, name of the boot partition: `boot`)
- BQ M10 FHD (codename: `frieza`, name of the boot partition: `boot`)

You will need the device codename and the name of your boot partition for the installation.

4.2.2 How to install

Warning: Installing Anbox is only recommended for experienced users.

- Make sure your supported device runs on 16.04 (Anbox doesn't work on 15.04)
- Open a terminal on your host and set some device specific variables by running `export CODENAME="turbo" && export PARTITIONNAME="bootimg"`, but replace the part between the quotes respectively with the codename and name of the boot partition for your device. See the above list.
- Activate developer mode on your device.
- Connect the device to your host and run the following commands from your host (same terminal you ran the `export` command in):

```
adb shell
sudo reboot -f bootloader # 'adb shell' will exit after this command
wget http://cdimage.ubports.com/anbox-images/anbox-boot-$CODENAME.img
sudo fastboot flash $PARTITIONNAME anbox-boot-$CODENAME.img
sudo fastboot reboot
rm anbox-boot-$CODENAME.img
exit
```

- Wait for the device to reboot, then run this from your host:

```
adb shell
sudo mount -o rw,remount /
sudo apt update
sudo apt install anbox-ubuntu-touch
anbox-tool install
exit
```

- Done! You might have to refresh the apps scope (pull down from the center of the screen and release) for the new Android apps to show up.

4.2.3 How to install new APKs

- Copy the APK to `/home/phablet/Downloads`, then run the following from your host:

```
adb shell
sudo mount -o rw,remount /
sudo apt update
sudo apt install android-tools-adb
adb install /home/phablet/Downloads/my-app.apk
exit
```

- Done! You might have to refresh the apps scope (pull down from the center of the screen and release) for the new Android apps to show up.

4.2.4 Keep your apps up to date

- To keep your apps up to date you can use of F-Droid or ApkTrack. If you want to install any of the above apps you can find them here:
- F-Droid: https://f-droid.org/
- ApkTrack: https://f-droid.org/packages/fr.kwiatkowski.ApkTrack/

4.2.5 How to uninstall apps

- This is a example of the app-list installed apps on your device
- To uninstall apps, run `adb uninstall [APP_ID]` from your Ubuntu Touch device:

```
adb shell
sudo mount -o rw,remount /
adb uninstall [APP_ID]
exit
```

- Done! You might have to refresh the apps scope (pull down from the center of the screen and release) for the new Android apps to show up.

4.2.6 Troubleshooting

- If installing `anbox-ubuntu-touch` or `android-tools-adb` on the device fails with an error about unsufficient space, try this:

```
adb shell
sudo mount -o rw,remount /
sudo rm -r /var/cache/apt      # delete the apt cache; frees space on system image
sudo tune2fs -m 0 /dev/loop0   # space reserved exclusively for root user on␣
↪system image set to zero
sudo apt update                # recreate apt cache to install Anbox and adb
sudo apt install anbox-ubuntu-touch android-tools-adb
sudo mount -o ro,remount /
exit
```

- When you want to install an apk but get the error `Invalid APK file` that error could also mean "file not found"
  - Check that you typed the file name correctly
  - If the APK does not reside in the current folder where you execute adb, you have to specify the full path, e.g. `/home/phablet/Downloads/my-app.apk` instead of just `my-app.apk`

4.2.7 Reporting bugs

Please *report any bugs* you come across. Bugs with Ubuntu Touch 16.04 are reported in the normal Ubuntu Touch tracker and issues with Anbox are reported on our downstream fork. Thank you!

CHAPTER 5

Advanced use

This section of the documentation details advanced tasks that power users may want to perform on their Ubuntu Touch device.

Note: Some of these guides involve making your system image writable, which may break OTA updates. The guides may also reduce the overall security of your Ubuntu Touch device. Please consider these ramifications before hacking on your device too much!

5.1 Shell access via adb

You can put your UBports device into developer mode and access a Bash shell from your PC. This is useful for debugging or more advanced shell usage.

5.1.1 Install ADB

First, you'll need ADB installed on your computer.

On Ubuntu:

```
sudo apt install android-tools-adb
```

On Fedora:

```
sudo dnf install android-tools
```

And on MacOS with Homebrew:

```
brew install android-platform-tools
```

For Windows, grab the command-line tools only package from here.

5.1.2 Enable developer mode

Next, you'll need to turn on Developer Mode.

1. Reboot your device
2. Place your device into developer mode (Settings - About - Developer Mode - check the box to turn it on)
3. Plug the device into a computer with adb installed
4. Open a terminal and run `adb devices`.

Note: When you're done using the shell, it's a good idea to turn Developer Mode off again.

If there's a device in the list here (The command doesn't print "List of devices attached" and a blank line), you are able to use ADB successfully. If not, continue to the next section.

5.1.3 Add hardware IDs

ADB doesn't always know what devices on your computer it should or should not talk to. You can manually add the devices that it does not know how to talk to.

Just run the command for your selected device if it's below. Then, run `adb kill-server` followed by the command you were initially trying to run.

Fairphone 2:

```
printf "0x2ae5 \n" >> ~/.android/adb_usb.ini
```

Oneplus One:

```
printf "0x9d17 \n" >> ~/.android/adb_usb.ini
```

5.2 Shell access via ssh

You can use ssh to access a shell from your PC. This is useful for debugging or more advanced shell usage.

You need a ssh key pair for this. Logging in via password is disabled by default.

5.2.1 Create your public key

If not already created, create your public key, default choices should be fine for LAN, you can leave empty password if you don't want to deal with it each time:

```
ssh-keygen
```

5.2.2 Copy the public key to your device

You need then to transfer your public key to your device. There are multiple ways to do this. For example:

- Connect the ubports device and the PC with a USB cable. Then copy the file using your filemanager.

- Or transfer the key via the internet by mailing it to yourself, or uploading it to your own cloud storage, or webserver, etc.
- You can also connect via *adb* and use the following command to copy it:

```
adb push ~/.ssh/id_rsa.pub /home/phablet/
```

5.2.3 Configure your device

Now you have the public key on the UBports device. Let's assume it's stored as `/home/phablet/id_rsa.pub`. Use the terminal app or and adb connection to perform the following steps on your phone.

```
mkdir /home/phablet/.ssh
chmod 700 /home/phablet/.ssh
cat /home/phablet/id_rsa.pub >> /home/phablet/.ssh/authorized_keys
chmod 600 /home/phablet/.ssh/authorized_keys
chown -R phablet.phablet /home/phablet/.ssh
```

Now start the ssh server:

```
sudo android-gadget-service enable ssh
```

5.2.4 Connect

Now everything is set up and you can use `ssh`

```
ssh phablet@<ip-address>
```

Of course you can now also use `scp` or `sshfs` to transfer files.

5.2.5 References

- askubuntu.com: How can I access my Ubuntu phone over ssh?
- gurucubano: BQ Aquaris E 4.5 Ubuntu phone: How to get SSH access to the ubuntu-phone via Wifi

5.3 Switch release channels

5.4 Screen Casting your UT device to your computer

Ubuntu Touch comes with a command line utility called `mirscreencast` which dumps screen frames to a file. The idea here is to stream UT display to a listening computer over the network or directly trough adb so that we can either watch it live or record it to a file.

5.4.1 Using adb

You can catch output directly from `adb exec-out` command and forward it to mplayer:

```
adb exec-out timeout 120 mirscreencast -m /run/mir_socket --stdout --cap-interval 2 -
↪s 384 640 | mplayer -demuxer rawvideo -rawvideo w=384:h=640:format=rgba -
```

NB: `timeout` here is used in order to kill process properly on device (here 120 seconds). Otherwise process still continuing even if killed on computer. You can reduce or increase frame per second with‘‘–cap-interval‘‘ (1 = 60fps, 2=30fps, . . .) and size of frames `384 640` means width=384 height=640

5.4.2 Via network

On receiver

For real time casting:

Prepare your computer to listen to a tcp port(here 1234) and forward raw stream to a video player (here mplayer) with a frame size of 384x640:

```
nc -l -p 1234 | gzip -dc | mplayer -demuxer rawvideo -rawvideo␣
↪w=384:h=640:format=rgba -
```

For stream recording:

Prepare your computer to listen to a tcp port(here 1234), ungzip and forward raw stream to a video encoder (here mencoder):

```
nc -l -p 1234 | gzip -dc | mencoder -demuxer rawvideo -rawvideo␣
↪fps=60:w=384:h=640:format=rgba -ovc x264 -o out.avi -
```

On device

Forward and gzip stream with 60fps (–cap-interval 1) and frame size of 384x640 to computer 10.42.0.209 on port 1234

```
mirscreencast -m /run/mir_socket --stdout --cap-interval 1 -s 384 640 | gzip -c | nc␣
↪10.42.0.209 1234
```

Example script

This script allows you to screen cast remote UT device to your local PC (must have ssh access to UT and mplayer installed on PC), run it on your computer:

```
#!/bin/bash
SCREEN_WIDTH=384
SCREEN_HEIGHT=640
PORT=1234
FORMAT=rgba


if [[ $# -eq 0 ]] ; then
    echo 'usage: ./mircast.sh UT_IP_ADDRESS , e.g: ./mircast.sh 192.168.1.68'
    exit 1
fi

IP=$1

LOCAL_COMMAND='nc -l -p $PORT | gzip -dc | mplayer -demuxer rawvideo -rawvideo w=
↪$SCREEN_WIDTH:h=$SCREEN_HEIGHT:format=$FORMAT -'
```

(continues on next page)

(continued from previous page)

```
REMOTE_COMMAND="mirscreencast -m /run/mir_socket --stdout --cap-interval 1 -s $SCREEN_
↪WIDTH $SCREEN_HEIGHT | gzip -c | nc \$SSH_CLIENT $PORT"
ssh -f phablet@$IP "$REMOTE_COMMAND"
eval $LOCAL_COMMAND
```

You can download it here: `files/mircast.sh`

5.4.3 References

- initial source: https://wiki.ubuntu.com/Touch/ScreenRecording
- demo: https://www.youtube.com/watch?v=HYm4RUwwo5Q

5.5 Reverse tethering

Some users may not have an available wifi connection for their phone nor a data subscription from their mobile provider. This short tutorial will help you to connect your Ubuntu Touch to your computer to access internet.

Prerequisite: Phone is connected to the computer with usb and developer mode enabled.

5.5.1 Steps

1. On phone: `android-gadget-service enable rndis`
2. On computer: get your rndis ip address e.g:`hostname -I`
3. On phone:

   add gateway: `sudo route add default gw YOUR_COMPUTER_RNDIS_IP`

   add nameservers: `echo "nameserver 8.8.8.8" | sudo tee /etc/resolv.conf`

   refresh dns table: `resolvconf -u`
4. On computer:

   enable ip forwarding: `echo 1 | sudo tee /proc/sys/net/ipv4/ip_forward`

   apply NAT: `sudo iptables -t nat -A POSTROUTING -s 10.0.0.0/8 -o eth0 -j MASQUERADE`

5.5.2 References

- askubuntu: https://askubuntu.com/questions/655321/ubuntu-touch-reverse-tethering-and-click-apps-updates

5.6 CalDAV and CardDAV synchronization

CalDAV and CardDAV are protocols to synchronize calendars and contacts with a remote server. Many email-hosters provide a CalDAV and CardDAV interface.

Note: CalDAV Sync can also be set up in using the calendar app. Open the app, click on the little calendar icon in the top right corner and select “Add internet calendar > Generic CalDAV”. Enter your calendar URL as well as your username and password to complete the process.

At the moment, there is no carddav implementation directly accessible from the Ubuntu Touch graphical user-interface, so the only way to sync carddav is by using syncevolution + cron. However, there is a simple way to do that with a script that you can run in the terminal or via phablet SSH connection. These instructions work for caldav as well.

1) Follow this guide to activate Developer Mode and ADB (or SSH) connection.
2) Download this script (let's call it dav.sh) and edit the following variables:
   - server side : CAL_URL, **CONTACTS_URL**, USERNAME, PASSWORD (of your ownCloud/nextCloud/baikal/SOGO/... server)
   - CONTACT and CALENDAR _ NAME / VISUAL_NAME / CONFIG_NAME (it's more cosmetic)
   - CRON_FREQUENCY (for the frequency of synchronization)
   - Line 61: write `sudo sh -c "echo '$COMMAND_LINE' > /sbin/sogosync"` , instead of `sudo echo "$COMMAND_LINE" > /sbin/sogosync`, to avoid permission denied error
3) Move the file to your Ubuntu Touch device, either by file manager or with adb:

```
adb push dav.sh /home/phablet
```

4) Connect with the phablet shell (`adb shell`) or directly on the phone Terminal app and type the following:

```
chmod +x dav.sh
./dav.sh
```

5.6.1 Sources:

- https://askubuntu.com/questions/616081/ubuntu-touch-add-contact-list-and-calendars/664834#664834
- https://gist.github.com/boTux/069b53d8e06bdb9b9c97
- https://gist.github.com/tcarrondo
- https://gist.github.com/bastos77
- https://askubuntu.com/questions/601910/ssh-ubuntu-touch

CHAPTER 6

Contributing to UBports

Welcome! You're probably here because you want to contribute to UBports. The pages you'll find below here will help you do this in a way that's helpful to both the project and yourself.

If you're just getting started, we always need help with *thorough bug reporting*. If you are multilingual, *translations* are also a great place to start.

If those aren't enough for you, see our contribute page for an introduction of our focus groups.

6.1 Bug reporting

This page contains information to help you help us by reporting an actionable bug for Ubuntu Touch. It does NOT contain information on reporting bugs in apps, most of the time their entry in the OpenStore will specify where and how to do that.

6.1.1 Get the latest Ubuntu Touch

This might seem obvious, but it's easy to miss. Go to (Settings - Updates) and make sure that your device doesn't have any Ubuntu updates available. If not, continue through this guide. If so, update your device and try to reproduce the bug. If it still occurs, continue through this guide. If not, do a little dance! The bug has already been fixed and you can continue using Ubuntu Touch.

6.1.2 Check if the bug is already reported

Open up the bug tracker for ubuntu-touch.

First, you'll need to make sure that the bug you're trying to report hasn't been reported before. Search through the bugs reported. When searching, use a few words that describe what you're seeing. For example, "Lock screen transparent" or "Lock screen shows activities".

If you find that a bug report already exists, select the "Add your Reaction" button (it looks like a smiley face) and select the +1 (thumbs up) reaction. This shows that you are also experiencing the bug.

If the report is missing any of the information specified later in this document, please add it yourself to help the developers fix the bug.

6.1.3 Reproduce the issue you've found

Next, find out exactly how to recreate the bug that you've found. Document the exact steps that you took to find the problem in detail. Then, reboot your phone and perform those steps again. If the problem still occurs, continue on to the next step. If not...

6.1.4 Getting Logs

We appreciate as many good logs as we can get when you report a bug. In general, /var/log/dmesg and the output of /android/system/bin/logcat are helpful when resolving an issue. I'll show you how to get these logs.

To get set ready, follow the steps to *set up ADB*.

Now, you can get the two most important logs.

dmesg

The **dmesg** (*display message* or *driver message*) command displays debug messages from the kernel.

1. Using the steps you documented earlier, reproduce the issue you're reporting
2. cd to a folder where you're able to write the log
3. Run the command: *adb shell dmesg > UTdmesg.txt*

This log should now be located at UTdmesg.txt under your working directory, ready for uploading later.

logcat

The **logcat** (*log concatenator*) command displays debug information from various parts of the underlying android system.

1. cd to a folder where you're able to write the log
2. Run the command: *adb shell /android/system/bin/logcat -d > UTlogcat.txt*
3. Using the steps you documented earlier, reproduce the issue you're reporting

This log will be located at UTlogcat.txt in your current working directory, so you'll be able to upload it later.

6.1.5 Making the bug report

Now it's time for what you've been waiting for, the bug report itself! Bug reports need to be filed in English.

First, pull up the bug tracker and click "New Issue". Log in to GitHub if you haven't yet.

Next, you'll need to name your bug. Pick a name that says what's happening, but don't be too wordy. Four to eight words should be enough.

Now, write your bug report. A good bug report includes the following:

- What happened: A synopsis of the erroneous behavior
- What I expected to happen: A synopsis of what should have happened, if there wasn't an error

- Steps to reproduce: You wrote these down earlier, right?
- Logs: Attach your logs by clicking and dragging them into your GitHub issue.
- Software Version: Go to (Settings - About) and list what appears on the "OS" line of this screen. Also include the release channel that you used when you installed Ubuntu on this phone.

Once you're finished with that, post the bug. You can't add labels yourself, so please don't forget to state the device you're experiencing the issue on in the description so a moderator can easily add the correct tags later.

A developer or QA-team member will confirm and triage your bug, then work can begin on it. If you are missing any information, you will be asked for it, so make sure to check in often!

6.2 Quality Assurance

This page explains how to help the UBports QA team, both as an official member or a new contributor. Please also read the *issue tracking* and *bugreporting* guides to better understand the workflow. For real-time communication, you can join our telegram group.

6.2.1 Smoke testing

To test the core functionality of the operating system, we have compiled a set of standardized tests. Run these tests on your device to *find and report bugs and regressions*. It's usually run on all devices before a new release to make sure no new issues were introduced.

6.2.2 Confirming bug reports

Unconfirmed bugreports are labeld **needs confirmation** to enable global filtering. Browse through the list, read the bugreports and try to reproduce the issues that are described. If necessary, add *missing information or logs, or improve the quality of the report by other means*. Leave a comment stating your device, channel, build number and whether or not you were able to reproduce the issue.

If you have write-access to the repository, you can replace the **needs confirmation** label with **bug** (to mark it confirmed) or **invalid** (if the issue is definitely not reproducible). In that case it should be closed.

If you find two issues describing the same problem, leave a comment and try to find their differences. If they are in fact identical, close the newer one and label it **duplicate**.

6.2.3 Testing patches

Pull-requests can be tested using the QA scripts. Run `ubports-qa -h` for usage information.

Once the pull-request has been merged, the issue it fixes is moved to the quality assurance column of the GitHub project. Please check if the issues in this column are still present in the latest update on the devel channel, then see if anything else has broken in the update. Check if the developer mentioned specific things to look out for when testing and leave a comment detailing your experience. If you have write-access to the repository, you can move the issue back to **In Development** (and reopen it) or forward to **Release Candidate** as specified by the *issue tracking guidelines*.

6.2.4 Initial triaging of issues

Initial triaging of new issues is done by QA-team members with write-access to the repository. If a new issue is filed, read the report and add the correct labels as specified by the *issue tracking guidelines*. You can also immediately start confirming the bugreport.

If the new issue has already been reported elsewhere, label it **duplicate** and close it.

6.3 Documentation

Tip: Documentation on this site is written in ReStructuredText, or RST for short. Please check the RST Primer if you are not familiar with RST.

This page will guide you through writing great documentation for the UBports project that can be featured on this site.

6.3.1 Documentation guidelines

These rules govern how you should write your documentation to avoid problems with style, format, or linking. If you don't follow these guidelines, we will not accept your document.

Title

All pages must have a document title that will be shown in the table of contents (left sidebar) and at the top of the page.

Titles should be sentence cased rather than Title Cased. For example:

```
Incorrect casing:
    Writing A Good Bug Report
Correct casing:
    Writing a good bug report
Correct casing when proper nouns are involved:
    Installing Ubuntu Touch on your phone
```

There isn't a single definition of title casing that everyone follows, but sentence casing is easy. This helps keep capitalization in the table of contents consistent.

Page titles are underlined with equals signs. For example, the markup for *Bug reporting* includes the following title:

```
Bug reporting
=============
```

Note that:

1. The title is sentence cased
2. The title is underlined with equals signs
3. The underline spans the title completely without going over

Incorrect examples of titles include:

- Incorrect casing

  ```
  Bug Reporting
  =============
  ```

- Underline too short

```
Bug reporting
=====
```

- Underline too long

```
Bug reporting
================
```

Headings

There are several levels of headings that you may place on your page. These levels are shown here in order:

```
Page title
==========

Level one
---------

Level two
^^^^^^^^^

Level three
"""""""""""
```

Each heading level creates a sub-section in the global table of contents tree available when the documentation is built. In the primary (web) version of the documentation, this only shows four levels deep from the top level of the documentation. Please refrain from using more heading levels than will show in this tree as it makes navigating your document difficult. If you must use this many heading levels, it is a good sign that your document should be split up into multiple pages.

Table of contents

People can't navigate to your new page if they can't find it. Neither can Sphinx. That's why you need to add new pages to Sphinx's table of contents.

You can do this by adding the page to the `index.rst` file in the same directory that you created it. For example, if you create a file called "newpage.rst", you would add the line marked with a chevron (>) in the nearest index:

```
.. toctree::
    :maxdepth: 1
    :name: example-toc

    oldpage
    anotheroldpage
>   newpage
```

The order matters. If you would like your page to appear in a certain place in the table of contents, place it there. In the previous example, newpage would be added to the end of this table of contents.

Warnings

Your edits must not introduce any warnings into the documentation build. If any warnings occur, the build will fail and your pull request will be marked with a red 'X'. Please ensure that your RST is valid and correct before you create a

pull request. This is done automatically (via sphinx-build crashing with your error) if you follow *our build instructions* below.

Line length

There is no restriction on line length in this repository. Please do not break lines at an arbitrary line length. Instead, turn on word wrap in your text editor.

6.3.2 Contribution workflow

The following steps will help you to make a contribution to this documentation after you have written a document.

Note: You will need a GitHub account to complete these steps. If you do not have one, click here to begin the process of making an account.

Forking the repository

You can make more advanced edits to our documentation by forking ubports/docs.ubports.com on GitHub. If you're not sure how to do this, check out the excellent GitHub guide on forking projects.

Building the documentation

If you'd like to build this documentation *before* sending a PR (which you should), follow these instructions on your *local copy* of your fork of the repository.

The documentation can be built by running `./build.sh` in the root of this repository. The script will also create a virtual build environment in `~/ubportsdocsenv` if none is present.

If all went well, you can enter the `_build/html` directory and open `index.html` to view the UBports documentation.

If you have trouble building the docs, the first thing to try is deleting the build environment. Run `rm -r ~/ubportsdocsenv` and try the build again. Depending on when you first used the build script, you may need to run the `rm` command with `sudo`.

6.3.3 Alternative methods to contribute

Translations

You may find the components of this document to translate at its project in UBports Weblate.

Writing documents not in RST format

If you would like to write documents for UBports but are not comfortable writing ReStructuredText, please write it without formatting and post it on the UBports Forum in the relevant section (likely General). Someone will be able to help you revise your draft and write the required ReStructuredText.

Uncomfortable with Git

If you've written a complete document in ReStructuredText but aren't comfortable using Git or GitHub, please post it on the UBports Forum in the relevant section (likely General). Someone will be able to help you revise your draft and submit it to this documentation.

6.3.4 Current TODOs

This section lists the TODOs that have been included in this documentation. If you know how to fix one, please send us a Pull Request to make it better!

To create a todo, add this markup to your page:

```
.. todo::

   My todo text
```

6.4 Translations

Although English is the official base language for all UBports projects we believe you have the right to use it in any language you want.

We are working hard to meet that goal, and you can help as well.

6.4.1 Tools for Translation

- For everyone: A web based translation tool called Weblate. This is the recommended way.
- For advanced users: Working directly on .po files with the editor of your choice, and a GitHub account. The .po files for each project are in their repository on our GitHub organization.

We also have a Translation Forum to discuss translating Ubuntu Touch and its core apps.

6.4.2 How-To

UBports Weblate

You can go to UBports Weblate, click on "Dashboard" button, go to a project, and start making anonymous suggestions without being registered. If you want to save your translations, you must be logged in.

For that, go to UBports Weblate and click on the "Register" button. Once in the "Registration" page, you'll find two options:

- Register using a valid email address, a username, and your full name. You'll need to resolve an easy control question too.
- Register using a third party registration. Currently the system supports accounts from openSUSE, GitHub, Fedora, and Ubuntu.

Once you're logged in, the site is self-explanatory and you'll find there all the options and customization you can do.

Now, get on with it. The first step is to search if your language already exists in the project of your choice.

If your language is not available for a specific project, you can add it yourself.

You decide how much time you can put into translation. From minutes to hours, everything counts.

.po file editor

As was said up above, you need a file editor of your choice and a GitHub account to translate .po files directly.

There are online gettext .po editors and those you can install in your computer.

You can choose whatever editor you want, but we prefer to work with free software only. There are too many plain text editors and tools to help you translate .po files to put down a list here.

If you want to work with .po files directly you know what you're doing for sure.

6.4.3 Translation Team Communication

The straightforward and recommended way is to use the forum category that UBports provides for this task.

To use it you need to register, or login if you're registered already.

The only requirement is to start your post putting down your language in brackets in the "Enter your topic title here" field. For example, `[Spanish] How to translate whatever?`

Just for your information, some projects are using Telegram groups too, and some teams are still using the Ubuntu Launchpad framework.

In your interactions with your team you'll find the best way to coordinate your translations.

6.4.4 License

All the translation projects, and all your contributions to this project, are under a Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) license that you explicitly accept by contributing to the project.

Go to that link to learn what this exactly means.

6.5 Preinstalled apps

This page will help you get started with developing the apps which are included with Ubuntu Touch.

6.5.1 Core apps

Core apps are applications which are included with UBports distributions of Ubuntu Touch **and** placed in the OpenStore for updates. Core apps are a good "first development" experience within UBports. Many are built with Clickable, an easy to use build and test system for Ubuntu Touch apps. Most core apps can even be built and run without an Ubuntu Touch device!

Which core apps currently exist?

A full list of the Ubuntu Touch core apps follows.

Application Name	Store page	Code repository
Calculator	Calculator on OpenStore	calculator-app on GitLab
Calendar	Calendar on OpenStore	calendar-app on GitLab
Clock	Clock on OpenStore	clock-app on GitLab
File Manager	File Manager on OpenStore	filemanager-app on GitLab
Gallery	Gallery on OpenStore	gallery-app on GitLab
Music	Music on OpenStore	music-app on GitLab
Notes	Notes on OpenStore	notes-app on GitLab
OpenStore	OpenStore… on OpenStore	openstore-app on GitLab
Terminal	Terminal on OpenStore	terminal-app on GitLab
UBports Welcome	UBports Welcome on OpenStore	ubports-app on GitLab
Weather	Weather on OpenStore	weather-app on GitLab

6.5.2 Other preinstalled apps

The following applications are preinstalled in Ubuntu Touch but are not considered core apps. Most of the time these projects must be updated with the system because they use many system services which do not necessarily have a stable API.

These apps may be more difficult to work with, but their repository should contain a document stating how to build and run them on a device.

- Browser (morph-browser on GitHub)
- Contacts (address-book-app on GitHub)
- Camera (camera-app on GitLab)
- External Drives (ciborium on GitHub)
- Media Player (mediaplayer-app on GitHub)
- Messaging (messaging-app on GitHub)
- Phone (dialer-app on GitHub)
- System Settings (system-settings on GitHub)

Instructions for contributing to these applications can be found in their respective repositories.

6.6 Monetary support

You can help us keep the lights on at UBports by becoming a patron on Liberapay or Patreon!

Your contribution finances our server infrastructure and debug services, and helps covering additional costs.

CHAPTER 7

App development

Welcome to an open source and free platform under constant scrutiny and improvement by a vibrant global community, whose energy, connectedness, talent and commitment is unmatched. Ubuntu is also the third most deployed desktop OS in the world.

7.1 Get started

Ubuntu Touch is a fun and vibrant platform for development. Whether you're a new developer or an experienced one, we have several resources to help you get started.

Ubuntu Touch supports several different types of apps. *Native apps* are apps with interfaces made using QML or HTML with their behavior defined in JavaScript, C++, Python, or Go. *Web apps* are special containers to run websites in.

7.1.1 Native applications

7.1.2 Web applications

Ubuntu Webapps are a great way to deliver online web applications into Ubuntu.

The Ubuntu platform provides an advanced web engine container to run online applications on the Ubuntu client devices.

Web applications are hosted online. They can be as simple as a website, like an online news site, or they can distribute content like videos. They can also have a rich user interface or use the WebGL extension to deliver games online.

Note: Ubuntu webapps and Ubuntu HTML5 apps are similar but not identical. The main difference is that the content of a webapp is provided through a URL, whereas HTML5 apps install their content (and usually provide an Ubuntu HTML5 GUI). Webapps also have restricted access to platform APIs.

Guide

This guide targets webapps for converged Ubuntu, that is, Ubuntu for Devices (phones and tablets). The Ubuntu Desktop has additional webapps support not covered here. Support for webapps on converged Ubuntu will continue to grow, and of course the future of Ubuntu is convergence, so stay tuned.

Guide

How the webapp fits into the shell

A web app displays in a webview inside a webapp-container that runs as an Ubuntu app in the Ubuntu/Unity shell.

Taking a closer look:

> At the innermost level, there is a website that the developer identifies by URL. The website is rendered and runs in an Oxide webview. Oxide is a Blink/Chrome webview that is customized for Ubuntu. The Oxide webview runs and displays in the webapp-container. The webapp-container is the executable app runtime that is integrated with the Ubuntu/unity shell.

Launching

You can launch a webapp from the terminal with::

```
webapp-container URL
```

For example::

```
webapp-container http://www.ubuntu.com
```

This simple form works, but almost every webapp also uses other features, such as URL containment with URL patterns as described below.

User interface

A webapp generally fills the entire app screen space, without the need of the UI controls generally found on standard browsers.

In some cases some navigation controls are appropriate, such as Back and Forward buttons, or a URL address bar. These are added as command line arguments:

- `--enable-back-forward` Enable the display of the back and forward buttons in the toolbar (at the bottom of the webapp window)
- `--enable-addressbar` Enable the display of the address bar (at the bottom of the webapp window)

URL patterns

Webapp authors often want to contain browsing to the target website. That is, the developer wants to control the URLs that can be opened in the webapp (all other URLs are opened in the browser). This is done with URL patterns as part of the webapp command line.

However, many web apps use pages that are hosted over multiple sites or that use external resources and pages.

However, both containment and access to specified external URLs are implemented with URL patterns provided as arguments at launch time. Let's take a closer look.

Uncontained by default

By default, there is no URL containment. Suppose you launch a webapp without any patters and only a starting URL like this::

```
webapp-container http://www.ubuntu.com
```

The user can navigate to any URL without limitation. For example, if they click the Developer button at the top, they navigate to developer.ubuntu.com, and it displays in the webapp.

Tip: You can see the URL of the current page by enabling the address bar with `--enable-addressbar`.

Simple containment to the site

One often wants to contain users to the site itself. That is, if the website is www.ubuntu.com, it may be useful to contain webapp users only to subpages of www.ubuntu.com. This is done by adding a wildcard URL pattern to the launch command, as follows::

```
webapp-container --webappUrlPatterns=http://www.ubuntu.com/* http://www.ubuntu.com
```

`--webappUrlPatterns=` indicates a pattern is next http://www.ubuntu.com/* is the pattern The asterisk is a wildcard that matches any valid sequence of trailing (right-most) characters in a URL

With this launch command and URL pattern, the user can navigate to and open in the webapp any URL that starts with http://www.ubuntu.com/. For example, they can click on the Phone button (http://www.ubuntu.com/phone) in the banner and it opens in the webapp, or the Tablet button (http://www.ubuntu.com/tablet). But, clicking Developer opens the corresponding URL in the browser.

Tip: Make sure to fully specify the subdomain in your starting URL, that is, use http://www.ubuntu.com instead of www.ubuntu.com. Not specifying the subdomain would create an ambiguous URL and thus introduces security concerns. More complex wildcard patterns

You might want to limit access to only some subpages of your site from within the webapp. This is easy with wildcard patterns. (Links to other subpages are opened in the browser.) For example, the following allows access to www.ubuntu.com/desktop/features and www.ubuntu.com/phone/features while not allowing access to www.ubuntu.com/desktop or www.ubuntu.com/phone:

```
webapp-container --webappUrlPatterns=http://www.ubuntu.com/*/features http://www.
↪ubuntu.com
```

Multiple patterns

You can use multiple patterns by separating them with a comma. For example, the following allows access only to www.ubuntu.com/desktop/features and www.ubuntu.com/phone/features::

```
webapp-container --webappUrlPatterns=http://www.ubuntu.com/desktop/features,http://
↪www.ubuntu.com/phone/features  http://www.ubuntu.com
```

Tip: Multiple patterns are often necessary to achieve the intended containment behavior.

Adding a specific subdomain

Many URLs have one or more subdomains. (For example, in the following, "developer" is the subdomain: developer.ubuntu.com.) You can allow access to a single subdomain (and all of its subpages) with a pattern like this::

```
--webappUrlPatterns=http://developer.ubuntu.com/*
```

However, one usually wants the user to be able to navigate back to the starting URL (and its subpages). So, if the starting URL is http://www.ubuntu.com, a second pattern is needed::

```
--webappUrlPatterns=http://developer.ubuntu.com/*,http://www.ubuntu.com/*
```

Putting these together, here's an example that starts on http://www.ubuntu.com and allows navigation to http://developer.ubuntu.com and subpages and back to http://www.ubuntu.com and subpages::

```
webapp-container --webappUrlPatterns=http://developer.ubuntu.com/*,http://www.ubuntu.
↪com/*  http://www.ubuntu.com
```

Adding subdomains with a wildcard

Some URLs have multiple subdomains. For example, www.ubuntu.com has design.ubuntu.com, developer.ubuntu.com and more. You can add access to all subdomains with a wildcard in the subdomain position::

```
webapp-container --webappUrlPatterns=http://*.ubuntu.com/*  http://www.ubuntu.com
```

Note: An asterisk in the subdomain position matches any valid single subdomain. This single pattern is sufficient to enable browsing to any subdomain and subpages, including back to the starting URL (http://www.ubuntu.com) and its subpages.

Adding https

Sometimes a site uses https for some of its URLs. Here is an example that allows https and https as access within the webapp to www.launchpad.net (and all subpages due to the wildcard)::

```
webapp-container --webappUrlPatterns=https?://http://www.launchpad.net/* http://www.
↪launchpad.net
```

Note: the question mark in https?. This means the preceding character (the 's') is optional. If https is always required, omit the question mark.

Command line arguments

The webapp-container accepts many options to fine tune how it hosts various web applications.

See all help with::

```
webapp-container --help
```

Note: Only the following options apply to converged Ubuntu.:

```
--fullscreen Display full screen
--inspector[=PORT] Run a remote inspector on a specified port or 9221 as the default␣
↪port
--app-id=APP_ID Run the application with a specific APP_ID
--name=NAME Display name of the webapp, shown in the splash screen
--icon=PATH Icon to be shown in the splash screen. PATH can be an absolute or path␣
↪relative to CWD
--webappUrlPatterns=URL_PATTERNS List of comma-separated url patterns (wildcard␣
↪based) that the webapp is allowed to navigate to
--accountProvider=PROVIDER_NAME Online account provider for the application if the␣
↪application is to reuse a local account.
--accountSwitcher Enable switching between different Online Accounts identities
--store-session-cookies Store session cookies on disk
--enable-media-hub-audio Enable media-hub for audio playback
--user-agent-string=USER_AGENT Overrides the default User Agent with the provided one.
```

Chrome options (if none specified, no chrome is shown by default)::

```
--enable-back-forward Enable the display of the back and forward buttons (implies --
↪enable-addressbar)
--enable-addressbar Enable the display of a minimal chrome (favicon and title)
```

Note: The other available options are specific to desktop webapps. It is recommended to not use them anymore.

User-Agent string override

Some websites check specific portions of the web engine identity, aka the User-Agent string, to adjust their presentation or enable certain features. While not a recommended practice, it is sometimes necessary to change the default string sent by the webapp container.

To change the string from the command line, use the following option::

```
--user-agent-string='<string>' Replaces the default user-agent string by the string␣
↪specified as a parameter
```

Browser data containment

The webapp experience is contained and isolated from the browser data point of view. That is webapps do not access data from any other installed browser, such as history, cookies and so on. Other browser on the system do not access the webapp's data. Storage

W3C allows apps to use local storage, and Oxide/Webapp-container supports the main standards here: LocalStorage, IndexedDB, WebSQL.

Quick start

There are several tools to help you package and deploy your webapp to your device:

- Webapp creator application available from the openstore
- Clickable CLI

Debugging your webapp

This guide give you some tips to help you debug your webapp.

Debug web application

Most web-devs will probably want do most of their coding and debugging in the usual browser environment. The Ubuntu Touch browser is compliant with modern web standards, and most webapps will just work without further changes.

For those (hopefully) rare cases where further debugging is needed, there are two ways to gain further information on the failure.

Watch the logs

If you are comfortable in a CLI environment, most Javascript errors will leave an entry in the app log file:

```
.cache/upstart/application-click-[YOUR_APP_NAME.AUTHOR_NAME..].log
```

Debugging in the browser

The default Ubuntu Touch browser is based on the Blink technology that is also used in Chrome/Chromium. By starting the browser in a special mode, you have access to the regular Chrome-style debugger.

On your phone, start the browser in inspector mode::

```
ubuntu-app-launch webbrowser-app --inspector
```

Now on your computer, launch Chrome/Chromium browser, and point address to `http://YOUR_UT_IP_ADDRESS:9221`

7.2 Community

Have questions that aren't answered in the docs or want to chat with other Ubuntu Touch developers? Join our app dev Telegram group or chat with us on the UBports app dev forum.

7.3 Tools

Clickable is a meta-build system for Ubuntu Touch applications that allows you to compile, build, test and publish `click` packages and provides various templates to get you started with app development. It is currently the easiest and most convenient way of building click packages for Ubuntu Touch. You can use any code editor or IDE that you choose and build your apps from the commandline with Clickable.

Alternatively there is the old Ubuntu SDK IDE. Be aware that it is no longer supported by Canonical, and UBports has chosen to not support it either due to lack of manpower.

You can still install the SDK IDE in Ubuntu 16.04, but it is not guaranteed to work correctly. You can use the following commands to install:

```
sudo add-apt-repository ppa:ubuntu-sdk-team/ppa
sudo apt update && sudo apt dist-upgrade
sudo apt install ubuntu-sdk
sudo reboot # or logout/login
```

7.4 Guides

Get started building your first app or learn about advanced concepts with our *Developer guides*.

7.4.1 Developer guides

We are currently working on expanding our list of developer guides. In the mean time we recommend the Ubuntu Touch Programming Course.

If you are interested in helping create developer guides check out our GitLab Project.

7.4.2 App development cookbook

Unofficial resources

In this section you will find links to external resources about creating applications for Ubuntu Touch.

- Ubuntu Touch programming book
- Application Templates by fulvio

Playground

In a completely free and open source community, it is natural to have community members exploring the limits of the platform in many many directions. In this section you will find links to external resources that do exactly that: Explore. The purpose of this list is to show the unlimited possibilities of an open platform like Ubuntu Touch.

Note: The resources listed here do not necessarily represent the officially endorsed way of developing applications for Ubuntu Touch, but are interesting experiments.

- Free Pascal development for Ubuntu Touch
- Lazarus development for Ubuntu Touch
- Geany on Ubuntu Touch device as text editor, source code editor, debugger and compiler for multiple languages

7.4.3 The content hub - tips and tricks

On Ubuntu Touch the apps are confined. The way of sharing files between them is through the Content Hub, a part of the system that takes care of file import, export and sharing.

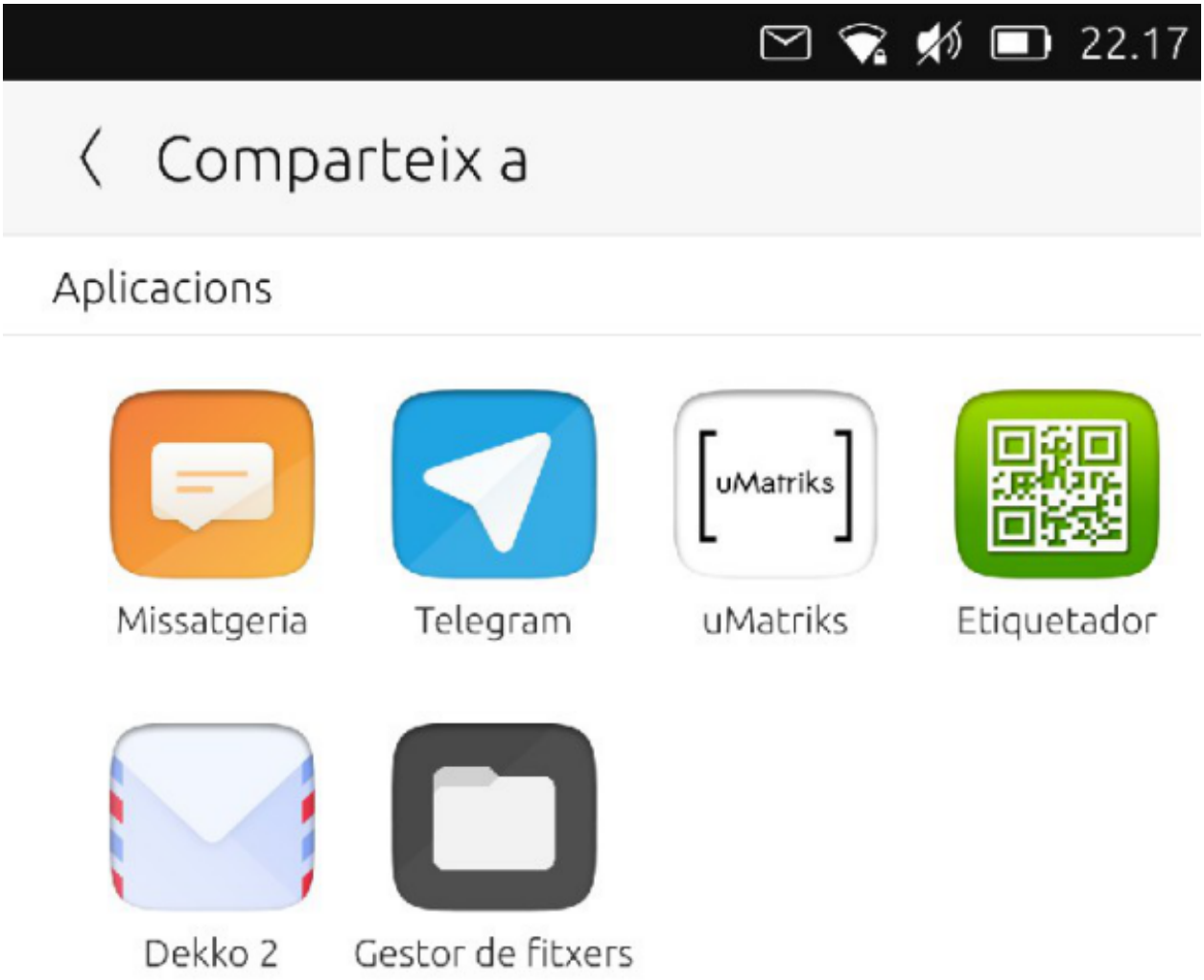


Fig. 1: Content Hub Share Page

Different ways of sharing the content

As we can see in the Content Hub documentation, there are several ways of handling the file to be shared:

- `ContentHandler.Source` (The app selected is going to be the source of the file imported)
- `ContentHandler.Destination` (The app selected is going to be the destination of the file exported)
- `ContentHandler.Share` (The app selected is going to be the app from witch the file is going to be shared from)

Importing

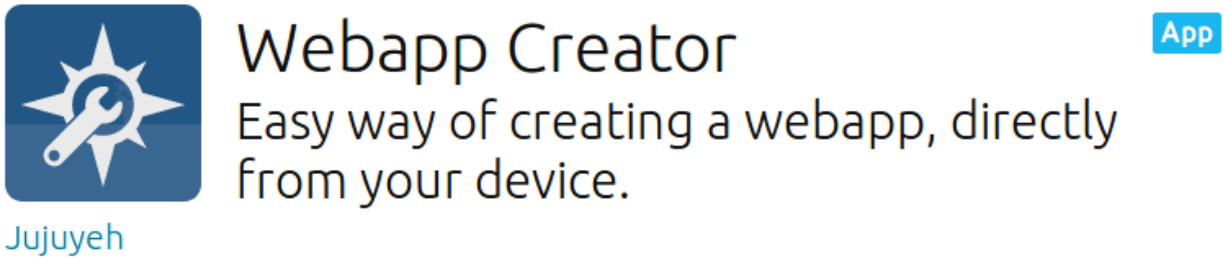


Fig. 2: Webapp Creator on the OpenStore

Looking into the code of Webapp Creator, we'll find the code to import an image to be used as icon. Tapping on the place holder will open the Content Hub that will let us choose from where to import the image (see the Webapp Creator source code)

```
ContentPeerPicker {
    anchors { fill: parent; topMargin: picker.header.height }
    visible: parent.visible
    showTitle: false
    contentType: picker.contentType //ContentType.Pictures
    handler: picker.handler //ContentHandler.Source
```

`ContentPeerPicker` is the element that shows the apps.

```
var importPage = mainPageStack.push(Qt.resolvedUrl("ImportPage.qml"),{"contentType":␣
↪ContentType.Pictures, "handler": ContentHandler.Source})
```

`contentType` is passed in Main.qml as ContentType.Pictures. So, we will only see apps from which we only can import images. `handler` is passed in the same line as ContentHandler.Source. As we want to import an image from the app selected in the Content Hub.

Exporting

In Gelek, we are going to end with some saved games that we want to save in our device or share with ourselves (in Telegram and then save them to our computer).

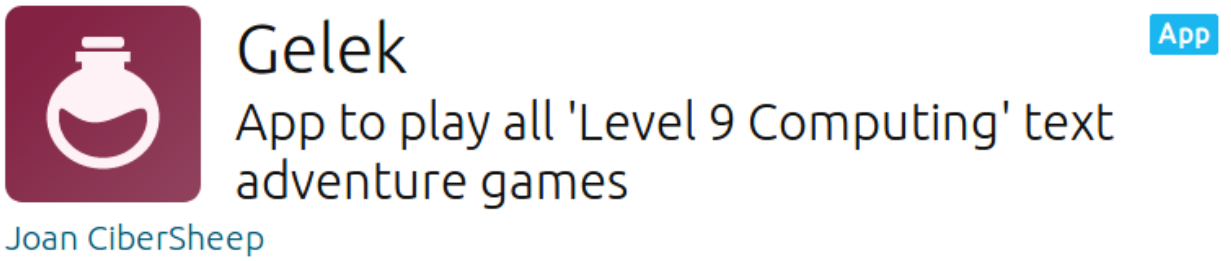
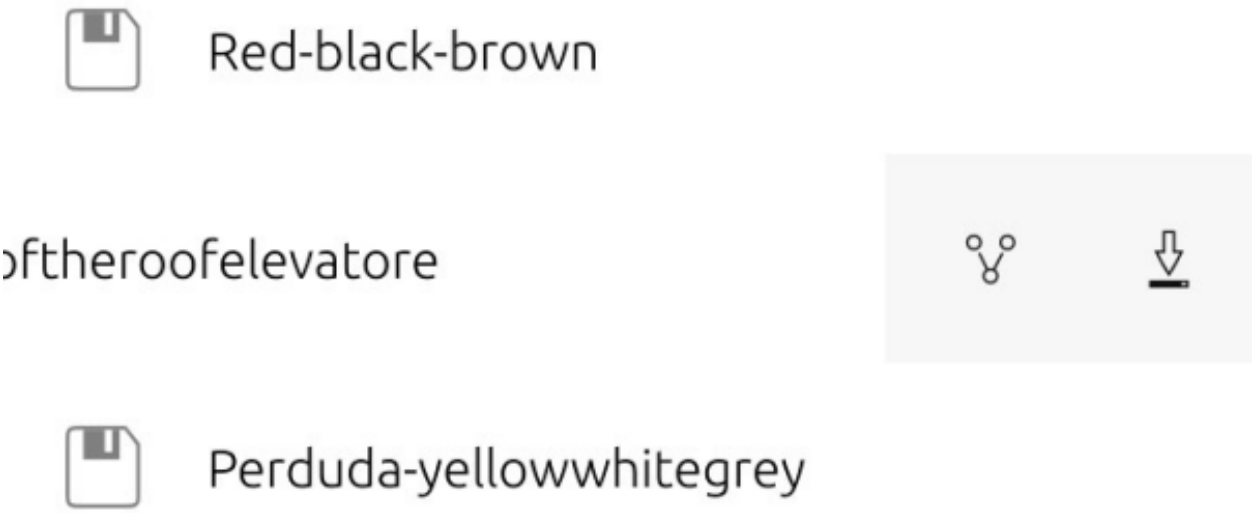


Fig. 3: Gelek in the OpenStore



Tapping on the download icon we will get a Content Hub to *save* the game file (which is actually an export).

The game file is a file of type `glksave`. We will tell Content Hub that we are sending a file of type `All` (see the Install Page code).

```
ContentPeerPicker {
    anchors { fill: parent; topMargin: picker.header.height }
    visible: parent.visible
    showTitle: false
    contentType: ContentType.All
    handler: ContentHandler.Destination

    onPeerSelected: {
```

`contentType` is `ContentType.All`, so we will only see apps which are able to receive unmarked file types. `handler` is `ContentHandler.Destination`, so the app selected should store the saved game.

Tapping on the File Manager we will save the saved game in the folder we choose.

Sharing

Similarly, tapping on the share icon will allow us to send the saved game through Telegram to ourselves (see the Webapp Creator Import Page source code).

```
ContentPeerPicker {
    anchors { fill: parent; topMargin: picker.header.height }
    visible: parent.visible
    showTitle: false
    contentType: picker.contentType //ContentType.Pictures
    handler: picker.handler //ContentHandler.Source

    onPeerSelected: {
```

The only difference between this and the previous code is that `handler` is `ContentHandler.Share`

Wait a minute. Why the different apps?

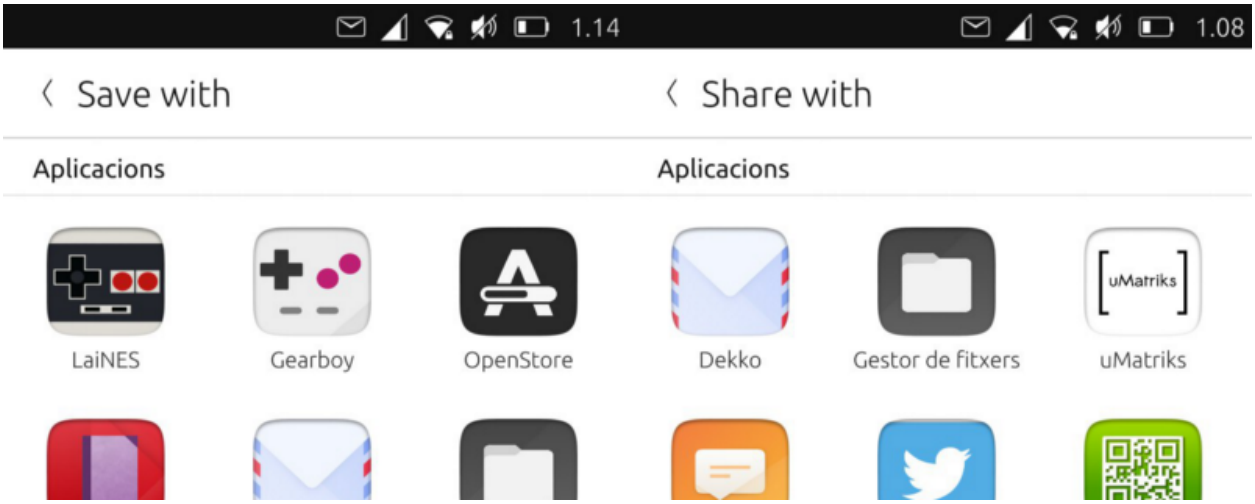


Fig. 4: Content Hub: Export vs Share

Each developer can decide the rules each app would follow in relation to the Content Hub. Why the OpenStore is shown as the destination of an export?

Let's check its manifest.json

```
"hooks": {
    "openstore": {
        "apparmor": "openstore/openstore.apparmor",
        "desktop": "openstore/openstore.desktop",
        "urls": "openstore/openstore.url-dispatcher",
        "content-hub": "openstore/openstore-contenthub.json"
    }
},
```

The above code defines that the hooks for the app named `"openstore"` in relation to the `"content-hub"` should follow the rules defined in `openstore-contenthub.json`

```
{
    "destination": [
        "all"
    ]
}
```

This means, the OpenStore will be the destination for *all* `ContentTypes`.

What about uMatriks? Let's see its `content-hub.json`

```
{
    "destination": [
        "pictures",
        "documents",
        "videos",
        "contacts",
        "music"
    ],
    "share": [
        "pictures",
        "documents",
        "videos",
        "contacts",
        "music"
    ],
    "source": [
        "pictures",
        "documents",
        "videos",
        "contacts",
        "music"
    ]
}
```

So, with this example, uMatriks will be able to be the destination, source and share app for all kinds of `ContentType`. What about the other hooks in the `manifest.json`? That is discussed in the next guide.

7.4.4 Importing from Content Hub and URLdispatcher

```
"hooks": {
    "openstore": {
        "apparmor": "openstore/openstore.apparmor",
        "desktop": "openstore/openstore.desktop",
        "urls": "openstore/openstore.url-dispatcher",
        "content-hub": "openstore/openstore-contenthub.json"
    }
},
```

In the previous guide we have seen a little bit about how Content Hub works. In this guide we will see how `URLdispatcher` works and how to handle imported data from the Content Hub.

Handle data from the Content Hub

One of the easiest ways of testing an app, is to send a test click to yourself on Telegram, and opening that click file with the OpenStore through the Content Hub:

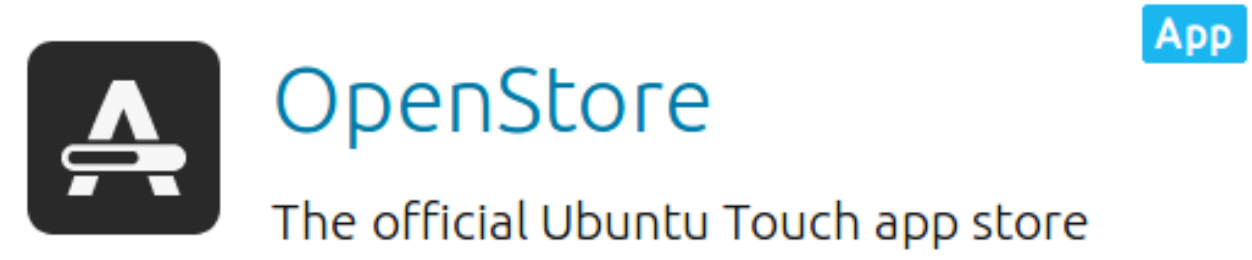
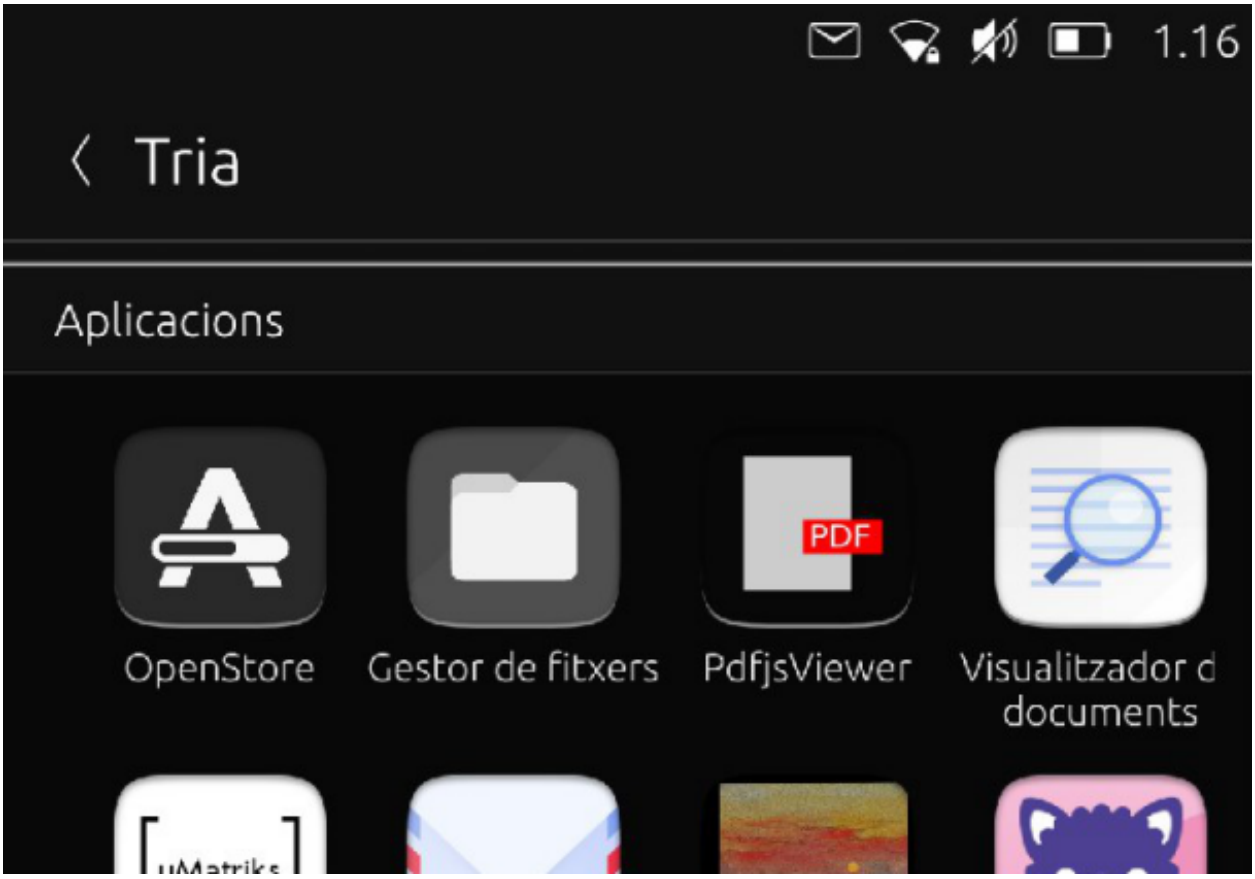


Fig. 5: OpenStore app from open-store.io



If we tap on the OpenStore app, it will be opened and it will ask if we want to install the click file. Let's take a look into the Main.qml code of the app to see how it is done:

```
Connections {
        target: ContentHub

        onImportRequested: {
            var filePath = String(transfer.items[0].url).replace('file://', '')
            print("Should import file", filePath)
            var fileName = filePath.split("/").pop();
            var popup = PopupUtils.open(installQuestion, root, {fileName: fileName});
            popup.accepted.connect(function() {
                contentHubInstallInProgress = true;
```

(continues on next page)

(continued from previous page)

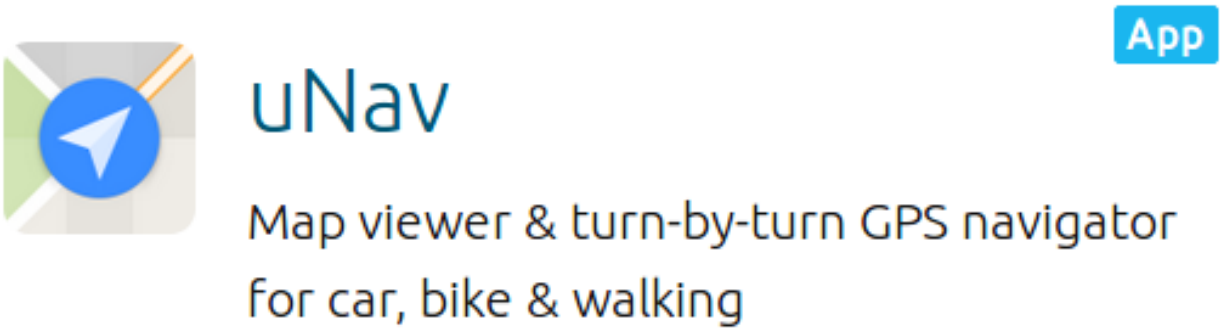
```
                PlatformIntegration.clickInstaller.installPackage(filePath)
            })
        }
}
```

Do you see that Connections element that targets the ContentHub? When it receives the signal onImportRequested, it will take the url of the data sent from the Content Hub (`transfer.items[0].url` is the url of the first data sent) and open a `PopUp` element to let the user install the click.

What about the URLdispatcher?

The URL dispatcher is a piece of software, similar to the Content Hub, that makes our life easier trying to choose the correct app for a certain protocol. For example: We probably want to open the web browser when tapping on an http protocol. If we tap on a map link it is handy to open it with uNav or to open a twitter link in the Twitter app! How does that work?

The `URLdispatcher` selects which app (according to their `manifest.json`) will open a certain link.



Let's see how our navigation app looks inside. uNav's manifest shows special hooks for the `URLdispatcher` in its manifest.json code:

```
1  [
2      {
3          "protocol": "http",
4          "domain-suffix": "map.unav.me"
5      },
6      {
7          "protocol": "http",
8          "domain-suffix": "unav-go.github.io"
9      },
10     {
11         "protocol": "geo"
12     },
13     {
14         "protocol": "http",
15         "domain-suffix": "www.openstreetmap.org"
16     },
17     {
18         "protocol": "http",
19         "domain-suffix": "www.opencyclemap.org"
20     },
21     {
```

(continues on next page)

(continued from previous page)

```
22         "protocol": "https",
23         "domain-suffix": "maps.google.com"
24     }
25 ]
```

This means that a link that looks like http://map.unav.me/xxxxx,xxxxx will be opened in uNav. And that's defined in lines 2 and 3, where it looks for protocol http followed by map.unav.me.

Also, a URI formatted geo:xxx,xxx should open in uNav, as it's defined in line 11.

And how do we manage the received URL?

After the `URLdispatcher` sends the link to the correspondent app, we need to handle that URL or URI in the targeted app. Let's see how to do that:

In the main qml file, we need to add some code to know what to do with the dispatched URL. First add an Arguments element that holds the URL, as is done, for example, in the Linphone app. Also, we add connection to the URI Handler with a Connection element with `UriHandler` as a target.

```
Arguments {
    id: args

    Argument {
        name: 'url'
        help: i18n.tr('Incoming Call from URL')
        required: false
        valueNames: ['URL']
    }
}

Connections {
    target: UriHandler

    onOpened: {
        console.log('Open from UriHandler')

        if (uris.length > 0) {
            console.log('Incoming call from UriHandler ' + uris[0]);
            showIncomingCall(uris[0]);
        }
    }
}
```

This code will manage a URI in the form `linphone://sip:xxx@xxx.xx` when the app is opened. But what do we need to do to handle this link when the app is closed?

We need to add a little bit extra code that will cover two cases: 1) We receive one URL 2) We receive more than one URL

```
Component.onCompleted: {
    //Check if opened the app because we have an incoming call
    if (args.values.url && args.values.url.match(/^linphone/)) {

        console.log("Incoming Call on Closed App")
        showIncomingCall(args.values.url);
```

(continues on next page)

(continued from previous page)

```
    } else if (Qt.application.arguments && Qt.application.arguments.length > 0) {

        for (var i = 0; i < Qt.application.arguments.length; i++) {
            if (Qt.application.arguments[i].match(/^linphone/)) {
                showIncomingCall(Qt.application.arguments[i]);
            }
        }
    }

    //Start timer for Registering Status
    checkStatus.start()
}
```

What happens if more than one app has the same URL type defined?

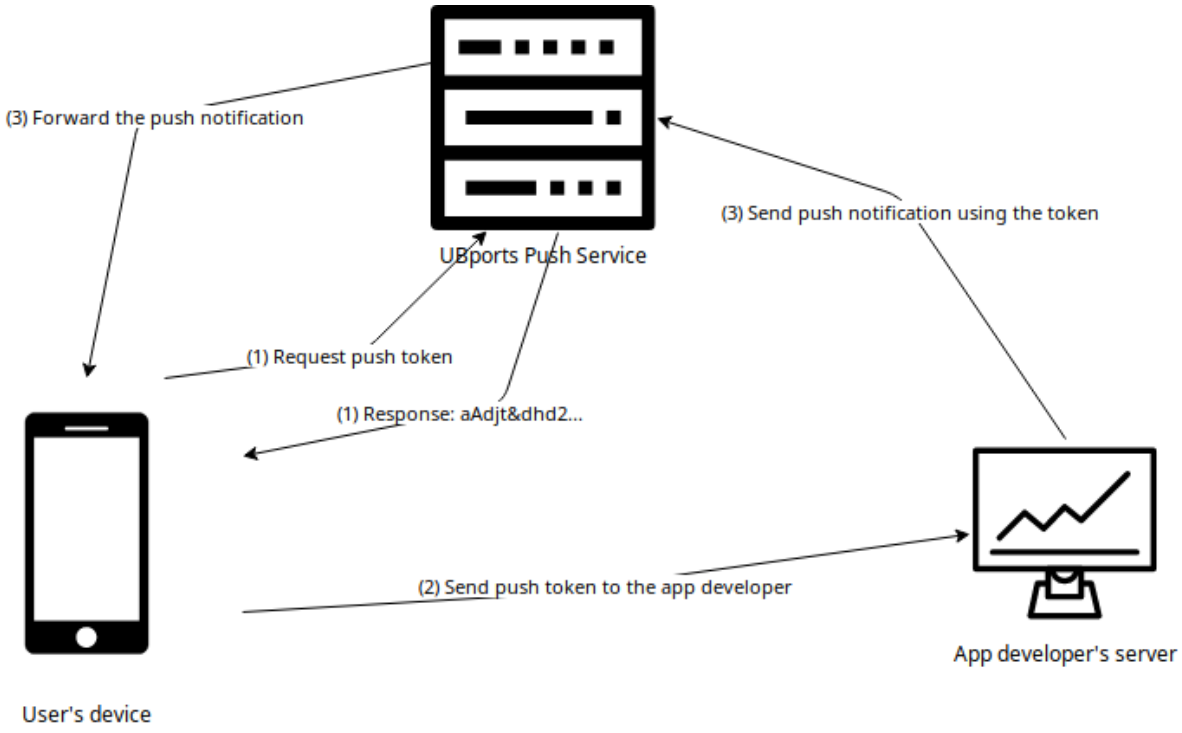
A very good question. What happens if we tap on a Twitter link? How is such a URL handled by the `URLdispatcher` as protocol `http` or the protocol `http://twitter`?

What happens if two apps have the same defined protocol?

Now it's time to do some tests and share the results in the next guide.

7.4.5 Push notifications

Let's assume that you have an app created with Clickable and published on the OpenStore. But now you want to be able to send Push Notifications to your users. First of all, you need to understand how this is working:



1. Every app which uses push notifications has got a unique token. This token identifies the user, the device and the app itself. The token is generated by the UBports Push Service.

2. You will need the token to send a push notification. So the app sends its token to the app developer's server.
3. With the token you can send a HTTP request to the UBports Push Server which will forward the notification the user's device.

Let's practice this step-by-step.

Note: In the following example we will not implement a server. Also the communication between your app and your server is up to you. Please inform the user about the communication with your server by providing a privacy policy!

Make the app ready for push notifications

Implementing the PushClient

First we need to add the policy group "push-notification-client". Your apparmor file could look like this:

```
    {
        "policy_groups": [
            "networking",
            "push-notification-client"
        ],
        "policy_version": 16.04
    }
```

In the next step we need to modify the Qml parts. We need to add a pushclient component:

```
//...

import Ubuntu.PushNotifications 0.1

//...

PushClient {
        id: pushClient
        appId: "myapp.yourname_hookname"
        onTokenChanged: console.log("", pushClient.token)
}
```

You need to set the correct appId! If the app name in your manifest file is myapp.yourname and the name of the main hook (the one which handles the .desktop file) is hookname, then the appId is: myapp.yourname_hookname. When we now start the app, it will get a token and print this token in the logs. With clickable logs we will be able to copy this token out of the terminal. But the app is not yet ready to receive a push notification. For this we need something called a pushhelper!

Implementing the pushhelper

The pushhelper is a part of the app which will receive all push notifications and process them before sending them to the system notification center. It will receive a json-file and must output another json-file in the correct format. The pushhelper is seperated from the app. So we need a new hook in the manifest. It could look like this:

```
    {
        //...
```

(continues on next page)

(continued from previous page)

```
        "title": "myapp",
        "hooks": {
            "myapp": {
                "apparmor": "myapp.apparmor",
                "desktop":  "myapp.desktop"
            },
            "push": {
                "apparmor": "push-apparmor.json",
                "push-helper": "push.json"
            }
        },

        //...
    }
```

It should be clear that we now need a different apparmor file and a different executable file. The **push-apparmor.json** file must only contain the policy group push-notification-client and should look like this:

```
{
    "template": "ubuntu-push-helper",
    "policy_groups": [
        "push-notification-client"
    ],
    "policy_version": 16.04
}
```

The **push.json** is for redirecting to the executable file:

```
{
    "exec": "pushexec"
}
```

In our tutorial we will use python to create a executable named **pushexec** which will forward the notification without changing anything:

```
#!/usr/bin/python3

import sys

f1, f2 = sys.argv[1:3]

open(f2, "w").write(open(f1).read())
```

We also need to add this new files to the **CMakeLists.txt** and make the pushexec executable:

```
[...]

install(FILES pushexec PERMISSIONS OWNER_EXECUTE OWNER_WRITE OWNER_READ DESTINATION $
↪{DATA_DIR})
install(FILES push.json DESTINATION ${DATA_DIR})
install(FILES push-apparmor.json DESTINATION ${DATA_DIR})

[...]
```

Now the app is ready to receive and process push notifications!

Using the Push Service API

So now you have the token and the app is ready to receive and process push notifications. To send a notification, you need to send a HTTP request to this address: https://push.ubports.com/notify The content-type must be application/json and it must fit in the correct format. A example in javascript could look like this:

```
var req = new XMLHttpRequest();
req.open("post", "https://push.ubports.com/notify", true);
req.setRequestHeader("Content-type", "application/json");
req.onreadystatechange = function() {
        if ( req.readyState === XMLHttpRequest.DONE ) {
                        console.log(" Answer:", req.responseText);
        }
}
var approxExpire = new Date ();
approxExpire.setUTCMinutes(approxExpire.getUTCMinutes()+10);
req.send(JSON.stringify({
        "appid" : "appname.yourname_hookname",
        "expire_on": approxExpire.toISOString(),
        "token": "aAnqwiFn$DF%2",
        "data": {
                "notification": {
                        "card": {
                                "icon": "notification",
                                "summary": "Push Notification",
                                "body": "Hello world",
                                "popup": true,
                                "persist": true
                        },
                "vibrate": true,
                "sound": true
                }
        }
}));
```

Push Notification Object

Parameter	Type	Description
appid	string	Required. ID of the application that will receive the notification, as described in the client side documentation.
expire_on	string	Required. Expiration date/time for this message, in ISO8601 Extendendformat.
token	string	Required. The token identifying the user+device to which the message is directed, as described in the client side documentation.
clear_pending	bool	Discards all previous pending notifications. Usually in response to getting a "too-many-pending" error. Defaults to false.
replace_tag	string	If there's a pending notification with the same tag, delete it before queuing this new one.
data	Data	A JSON object. The contents of the data field are arbitrary. We can use it to send any data to the app.

Data

Parameter	Type	Description
notification	Notification	A JSON object which defines how this notification will be presented.
message	object	A JSON object that is passed as-is to the application via PopAll.

Notification

Parameter	Type	Description
tag	string	The tag of the push notification.
sound	bool or string	This is either a boolean (play a predetermined sound) or the path to a sound file. The user can disable it, so don't rely on it exclusively. Defaults to empty (no sound). The path is relative, and will be looked up in (a) the application's `.local/share/<pkgname>`, and (b) standard xdg dirs.
vibrate	bool or Vibrate	The notification can contain a vibrate field, causing haptic feedback, which can be either a boolean (if true, vibrate a predetermined way) or an Vibrate object.
emblem-counter	Emblem-counter	A JSON object, which defines how to display the emblem counter.
card	Card	A JSON object with information about the notification card.

Card

Parameter	Type	Description
summary	string	Required. A title. The card will not be presented if this is missing.
body	string	Longer text, defaults to empty.
actions	array	If empty (the default), a bubble notification is non-clickable. If you add a URL, then bubble notifications are clickable and launch that URL. One use for this is using a URL like `appid://com.ubuntu.developer.ralsina.hello` which will switch to the app or launch it.
icon	string	An icon relating to the event being notified. Defaults to empty (no icon); a secondary icon relating to the application will be shown as well, regardless of this field.
timestamp	integer	Seconds since the unix epoch, only used for persist for now. If zero or unset, defaults to current timestamp.
persist	bool	Whether to show in notification centre; defaults to false.
popup	bool	Whether to show in a bubble. Users can disable this, and can easily miss them, so don't rely on it exclusively. Defaults to false.

Vibrate

Parameter	Type	Description
pattern	array	A list of integers describing a vibration pattern (duration of alternating vibration/no vibration times, in milliseconds).
repeat	integer	Number of times the pattern has to be repeated (defaults to 1, 0 is the same as 1).

Emblem-Counter

Parameter	Type	Description
count	integer	A number to be displayed over the application's icon in the launcher.
visible	bool	Set to true to show the counter, or false to hide it.

7.4.6 User metrics

What are user metrics?

If you look on the lock screen, you will see a circle. Inside the circle is text. Look closer, and you'll notice that the text contains data regarding the user's activity. Double tap on the middle of the circle, and you will see more "metrics" about the user.

Fig. 6: This shows "7 text messages sent today." How did it know?

For the most part, these messages are quite clearly state what they are counting, and which app is related. But where do these metrics come from?

How can I use user metrics in my application?

All of the following information will be based on the code for nCounter.

First, you will need to import the module in the QML file that will handle the User Metrics:

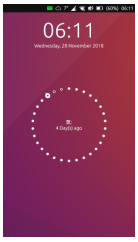


Fig. 7: This is from a 3rd-party application (nCounter) that makes use of the User Metrics feature.

```
import UserMetrics 0.1
```

(There may be updated versions of this above 0.1)

Next, the specific Metric must be defined in the code as an object:

```
Metric { // Define the Metric object.
    property string circleMetric // Create a string-type variable called "circleMetric
↪". This is so you can update it later from somewhere else.
    id: metric // A name to reference the metric elsewhere in the code. i.e. when␣
↪updating format values below.
    format: circleMetric // This is the metric/message that will display "today".␣
↪Again it uses the string variable that we defined above
    emptyFormat: i18n.tr("Check nCounter") // This is the metric/message for tomorrow.
↪ It will "activate" once the day roles over and replaces "format". Here I have use␣
↪a simple translatable string instead of a variable because I didn't need it to␣
↪change.
    domain: "ncounter.joe" // This is the appname, based on what you have in your app␣
↪settings. Presumably this is how the system lists/ranks the metrics to show on the␣
↪lock screen.
}
```

Now that the metric is created, we can update the "format" or "emptyFormat" when an event takes place by referencing the variables in the Metric object.

```
onButtonPressed: {
        metric.circleMetric = "New Metric Message"
        metric.update(0)
        console.log("Metric updated")
}
```

Here we assign a new value to the circleMetric string variable that's inside the Metric object:

(Remember that circleMetric is the variable value assigned to format)

> Metric Id [dot] Variable Name [equals] New variable information
>
> `metric.circleMetric = "New Metric Message"`

We then tell the lock screen to update the metric in ZERO milliseconds i.e. immediately

> Metric ID [dot] update (Number of milliseconds)
>
> `metric.update(0)`

We have now updated the metric for today. When the time rolls over to tomorrow, the metric will be reset to whatever is in `emptyFormat`.

For most apps, this defaults to 0 counts for messages, calls, etc.

How do user metrics work?

User metrics are made up of two "formats"

- metrics/messages for today (`format`)
- metrics/messages for tomorrow (`emptyFormat`)

The value of `emptyFormat` is what displays on the lock screen when no value has been stored in `format`. In order to display a new value of `format` the metric must be updated with `metric.update(x)` (where x is the number of milliseconds until the update takes place). The metric will reset back to the value stored in `emptyFormat` each day.

Applications make use of this system, but setting and updating the user metric "formats" by running a certain code whenever a certain event takes place. e.g. When you press send in Telegram, or when you receive a phone call. The application may store the data for manipulation, but generally the data is stored in the system (/var/lib/usermetrics).

Limitations and wonders

Based on how the "formats" are set up, it seems that it is difficult to maintain a running tally beyond one day. It also doesn't seem to truly reset a counted variable. Instead it reverts to a default setting. This would not normally allow for long-term data interpretation without some kind of database logging.

In the case of the nCounter app. I wanted to count the number of days, but since the metric "resets" each day, that presents a problem. I create a workaround that updates the metric every time the application is opened. Thus, the `emptyFormat` (default) tells the user to open the application. This, however, nearly defeats the purpose of the user metric entirely, other than having a neat stat reminder for the day.

There must be a way for a process to run independently in the background (e.g. cron) to retrieve data from a specific app code. One lead is the Indicator Weather app. This runs a process every X minutes to update the weather indicator automatically without having to open the app.

7.4.7 Writable directories

App confinement is part of the Ubuntu Touch security concept. Data can be exchanged between apps only according to the AppArmor policies, mainly using the *ContentHub*. This being said, apps can only read and write files that are located in one of three app specific directories explained in this guide.

Standard Paths

Besides the write access to the app directories explained below, the app can write debug messages into the app log file located at `/home/phablet/.cache/upstart/application-click-<fullappname>_<appname>_<version>.log`. To append messages to the log file, use the Qt debug functions.

Config

Path: `/home/phablet/.config/<fullappname>/`

This is the place to store configurations files to. The music app for example stores its configuration to `/home/phablet/.config/com.ubuntu.music/com.ubuntu.music.conf`.

Cache

Path: `/home/phablet/.cache/<fullappname>/`

This is the place to cache data for later use. The cache is also used by the Content Hub. Files that have been shared with the music app for example can be found in `/home/phablet/.cache/com.ubuntu.music/HubIncoming/`.

App data

Path: `/home/phablet/.local/share/<fullappname>/`

This is where your app stores any data. The music app for example stores its data bases to `/home/phablet/.local/share/com.ubuntu.music/Databases/`.

Using Standard Paths in C++

The Qt header `QStandardPaths` provides the app's writable locations:

```
#include <QStandardPaths>
...
QString configPath =␣
↪QStandardPaths::writableLocation(QStandardPaths::AppConfigLocation);
QString cachePath = QStandardPaths::writableLocation(QStandardPaths::CacheLocation);
QString appDataPath =␣
↪QStandardPaths::writableLocation(QStandardPaths::AppDataLocation);
...
```

Using Standard Paths in QML

The Qt module Qt.labs.platform provides the QStandardPaths in QML. Unfortunately it is not available on Ubuntu Touch, yet. Probably you don't want to ship that module with your app. Instead, you could simply let your C++ plugin provide the paths to your QML part. Therefore, add these methods to your plugin class:

```
#include <QStandardPaths>
...
Q_INVOKABLE QString configPath()
{
    return QStandardPaths::writableLocation(QStandardPaths::AppConfigLocation);
}
Q_INVOKABLE QString cachePath()
{
    return QStandardPaths::writableLocation(QStandardPaths::CacheLocation);
}
Q_INVOKABLE QString appDataPath()
{
    return QStandardPaths::writableLocation(QStandardPaths::AppDataLocation);
}
...
```

Assuming the name of your plugin is MyPlugin, you can access these paths in QML:

```
Label
{
    text: MyPlugin.configPath()
}
Label
{
    text: MyPlugin.cachePath()
}
Label
{
    text: MyPlugin.appDataPath()
}
```

7.4.8 Handle C++ dependencies with Clickable

If an app depends on a library that is not pre-installed in Ubuntu Touch, the app needs to ship it inside the click package. This guide shows how this can be done with Clickable.

Building

Often libraries are available pre-built, So you need to decide whether you want your build setup to download the library source and compile it or download the pre-built version. The following sections describe the options with its pros and cons.

Compilation

Compiling the library gives you control over the result, which brings a lot of advantages. For instance, you may disable components your app doesn't need and link the library statically, resulting in a much smaller package size.

Put the libraries source code at `libs/LIBNAME`, because this is where clickable will look for it by default. If the dependency source code is available as a git repository, it is a good idea to add it as a git submodule. Otherwise you might want to add a script to download the sources.

Add a libraries section to your clickable.json, like this:

```
{
  "template": "cmake",
  "dir": "build/app",
  "libraries": {
    "LIBNAME": {
      "template": "cmake"
    }
  }
}
```

If the library does not contain a CMake configuration, you may use the *qmake* or *custom* template instead. If you choose *custom*, you need to add a *build* command.

You should define a custom app build directory via the `dir` param as shown above. Otherwise a *clickable clean* (which is part of the default *clickable* command) would delete the library's build directory, too.

Optionally, configure the compilation by adding `build_args`, which may look like this:

```
{
  "template": "cmake",
  "dir": "build/app",
  "libraries": {
    "LIBNAME": {
      "template": "cmake",
      "build_args": [
        "-DBUILD_EXAMPLES=OFF",
        "-DBUILD_DOCS=OFF",
        "-DBUILD_TESTS=OFF",
        "-DBUILD_SHARED_LIBS=OFF"
      ]
    }
  }
}
```

Build arguments are usually project specific. Therefore, study the library's build instructions and also look for `option` settings in its `CMakeLists.txt` (if CMake is used). The default build directory is `build/LIBNAME/ARCHITECTURE`. Therefore builds for different architectures can exist in parallel.

To actually build the library run `clickable build-libs`. You may want to build it for the desktop, too, via `clickable build-libs --arch amd64`. Don't forget to mention this step in your README!

See how Teleports clickable.json uses the libraries feature to build its dependency tdlib.

Pre-built

A pre-built library is usually only available as a shared object, that needs to be linked dynamically. Furthermore, it may contain components that you don't need, resulting in a bloated app. It may even miss something, that you could achieve by compiling it yourself. Sometimes, a library is available in the Ubuntu Repositories, but is not installable for the architecture you need (likely `armhf`). In this case you have to compile the library as described above.

If the library is available in the Ubuntu Repositories, you can add it to the dependencies list, like this:

```
{
  "template": "cmake",
  "dependencies_target": [
    "libsomething-dev"
  ]
}
```

Clickable will install the specified package automatically for the target architecture inside the build container. An example can be found in Guitar Tools' clickable.json.

If the library is not in the Ubuntu Repositories, but in a PPA, you can add the PPA to the clickable.json, too. For example:

```
{
  "template": "cmake",
  "dependencies_ppa": [
    "ppa:someone/libsomething"
  ],
  "dependencies_target": [
    "libsomething-dev"
  ]
}
```

Otherwise you may need to add a script to download the pre-built library.

Usage

First, you may need to specify the include directory where the compiler can find the headers. Second, you need to link the library itself against your app's binary, except it is a header library, where all the source code is located in header files.

In case the library contains an appropriate CMake configuration file, you may use the find_package command. The additional lines on your CMakeLists.txt then may look like:

```
execute_process(
    COMMAND dpkg-architecture -qDEB_HOST_MULTIARCH
    OUTPUT_VARIABLE ARCH_TRIPLET
    OUTPUT_STRIP_TRAILING_WHITESPACE
)
set(SOMELIBRARY_DIR "${CMAKE_SOURCE_DIR}/build/somelib/${ARCH_TRIPLET}")
find_package(SOMELIBRARY REQUIRED)
include_directories(${SOMELIBRARY_INCLUDE_DIRS})
target_link_libraries(mytarget ${SOMELIBRARY_LIBS})
```

The command `dpkg-architecture -qDEB_HOST_MULTIARCH` is used to query the target architecture, which is part of the library build directory path, if you compiled the library with Clickable.

We define the variable `SOMELIBRARY_DIR` with the path to the libraries build directory, to help CMake find the configuration of the library named `SOMELIBRARY`. You may not need to do this, if you installed the library from the Ubuntu Repositories.

The `find_package` command defines the path to the include directory as `SOMELIBRARY_INCLUDE_DIRS` and the library's binaries as `SOMELIBRARY_LIBS`. We use those with the `include_directories` and `target_link_libraries` commands. See the Camera Scanner ImageProcessing CMakeLists.txt for a real world example.

Deployment

If you link a library statically with your app, you do not need to ship the library explicitly, as it is already inside your app binary. To do so, you usually need to compile the library yourself. Otherwise, continue with this section.

Find out which components you need to ship. Usually this is one or more `*.so` (shared objects) files, linked dynamically. To get the files into the click package, you need to add an `install` command to your build configuration. Add the following lines to your CMakeLists.txt:

```
execute_process(
    COMMAND dpkg-architecture -qDEB_HOST_MULTIARCH
    OUTPUT_VARIABLE ARCH_TRIPLET
    OUTPUT_STRIP_TRAILING_WHITESPACE
)
install(FILES /usr/lib/${ARCH_TRIPLET}/libSomething.so DESTINATION /lib/${ARCH_
↪TRIPLET})
```

This will copy the files into the `tmp` folder inside the build directory. This is where Clickable puts all the files that go into the click package. Again, the command `dpkg-architecture -qDEB_HOST_MULTIARCH` is used to query the target architecture, which is usually part of the file path.

7.5 Publishing

After you are done building your app, distribute it on the OpenStore with our *Publishing guides*.

7.5.1 App publishing guides

We are currently working on expanding our list of publishing guides. In the mean time you can get started over at the OpenStore's submission page.

If you are interested in helping create developer guides check out our GitLab Project.

7.6 Documentation

- QML API
- Cordova HTML5 API
- Clickable

7.6.1 More Documentation

Platform

Content Hub Each application can expose content outside its sandbox, giving the user precise control over what can be imported, exported or shared with the world and other apps.

Push notifications By using a push server and a companion client, instantly serve users with the latest information from their network and apps.

URL dispatcher Help users navigate between your apps and drive their journey with the URL dispatcher.

Online accounts Simplify user access to online services by integrating with the online accounts API. Accounts added by the user on the device are registered in a centralized hub, allowing other apps to re-use them.

Read the docs

AppArmor Policy Groups

This document contains a full list of Ubuntu Touch's available policy groups and a description of what they give your app permission to access.

Each entry follows this format

```
Title
-----

Description: Description from apparmor file

Usage: How common it is to use this policy (from apparmor file)

Optional longer description
```

Policy usage affects whether your app will be accepted by the OpenStore. Apps containing policies with common usage are generally accepted immediately, while reserved usage policies will need to be manually reviewed.

accounts

Description: Can use Online Accounts.

Usage: common

The accounts policy gives your app the permissions it needs to access the Online Accounts API.

audio

Description: Can play audio (allows playing remote content via media-hub)

Usage: common

The audio policy is needed for your app to play audio via pulseaudio or media-hub. The permission also gives it the ability to send album art to the thumbnailer service, which is then shown on the sound indicator.

bluetooth

Description: Use bluetooth (bluez5) as an administrator.

Usage: reserved

This policy grants unrestricted access to Bluetooth devices. It is provided for administration of bluetooth and as a stepping stone towards developing a safe bluetooth API all apps can access.

calendar

Description: Can access the calendar.

Usage: reserved

Calendar grants access to the Evolution dataserver's calendar and alarms APIs. It also grants access to sync-monitor.

This policy is reserved since it grants free access to all calendars on the device at any time. The legacy bug about this situation is LP #1227824 .

camera

Description: Can access the camera(s)

Usage: common

The camera policy grants access to device cameras.

connectivity

Description: Can access coarse network connectivity information

Usage: common

The connectivity policy allows apps to determine rough information about the device's connectivity. This includes whether the device is connected to the Internet and whether it is connected via a Wi-Fi or mobile data connection.

contacts

Description: Can access contacts.

Usage: reserved

The contacts policy allows apps to access the device user's contacts list. It is marked as reserved because it allows access to sync-monitor and unfettered access to the address book.

content_exchange

Description: Can request/import data from other applications

Usage: common

Using the content_exchange policy allows your app to be a consumer of content on content-hub.

content_exchange_source

Description: Can provide/export data to other applications

Usage: common

The content_exchange_source policy allows your app to provide content on content-hub.

debug

Description: Use special debugging tools. This should only be used in development and not for production packages. Note: use of this policy group provides significantly different confinement than normal and is not considered secure. You should never run untrusted programs using this policy group.

Usage: reserved

document_files

Description: Can read and write to document files. This policy group is reserved for certain applications, such as document viewers. Developers should typically use the content_exchange policy group and API to access document files instead.

Usage: reserved

This policy allows apps to read and write to the "Documents" folders in the user's home directory and external media.

document_files_read

Description: Can read all document files. This policy group is reserved for certain applications, such as document viewers. Developers should typically use the content_exchange policy group and API to access document files instead.

Usage: reserved

This policy allows apps to read the "Documents" folders in the user's home directory and external media.

history

Description: Can access the history-service. This policy group is reserved for vetted applications only in this version of the policy. A future version of the policy may move this out of reserved status.

Usage: reserved

keep-display-on

Description: Can request keeping the screen on

Usage: common

location

Description: Can access Location

Usage: common

Allows an app to request access to the device's current location.

microphone

Description: Can access the microphone

Usage: common

music_files

Description: Can read and write to music files. This policy group is reserved for certain applications, such as music players. Developers should typically use the content_exchange policy group and API to access music files instead.

Usage: reserved

The music_files policy group allows an app to read or write to the Music directories in the user's home folder or on external media.

music_files_read

Description: Can read all music files. This policy group is reserved for certain applications, such as music players. Developers should typically use the content_exchange policy group and API to access music files instead.

Usage: reserved

The music_files_read policy group allows an app to read the Music directories in the user's home folder or on external media.

networking

Description: Can access the network

Usage: common

The networking policy group allows an app to contact network devices and use the download manager.

picture_files

Description: Can read and write to picture files. This policy group is reserved for certain applications, such as gallery applications. Developers should typically use the content_exchange policy group and API to access picture files instead.

Usage: reserved

The picture_files policy group allows an app to read and write to the Pictures directories in the user's home folder or on external media.

picture_files_read

Description: Can read all picture files. This policy group is reserved for certain applications, such as gallery applications. Developers should typically use the content_exchange policy group and API to access picture files instead.

Usage: reserved

The picture_files_read policy group allows an app to read the Pictures directories in the user's home folder or on external media.

push-notification-client

Description: Can use push notifications as a client

Usage: common

sensors

Description: Can access the sensors

Usage: common

Allows apps to access device sensors

usermetrics

Description: Can use UserMetrics to update the InfoGraphic

Usage: common

Allows an app to write metrics to the UserMetrics service so they can be displayed on the InfoGraphic.

video

Description: Can play video (allows playing remote content via media-hub)

Usage: common

video_files

Description: Can read and write to video files. This policy group is reserved for certain applications, such as gallery applications. Developers should typically use the content_exchange policy group and API to access video files instead.

Usage: reserved

The video_files policy group allows an app to read and write to the Videos directories in the user's home folder or on external media.

video_files_read

Description: Can read all video files. This policy group is reserved for certain applications, such as gallery applications. Developers should typically use the content_exchange policy group and API to access video files instead.

Usage: reserved

The video_files_read policy group allows an app to read the Videos directories in the user's home folder or on external media.

webview

Description: Can use the UbuntuWebview

Usage: common

The webview policy group allows apps to embed a web browser view.

Click package

Every `click` application package must embed at least 3 files:

`manifest.json` file Contains application declarations such as application name, description, author, framework sdk target, and version.

Example `manifest.json` file:

```
{
    "name": "myapp.author",
    "title": "App Title",
    "version": "0.1"
    "description": "Description of the app",
    "framework": "ubuntu-sdk-16.04",
    "maintainer": "xxxx <xxx@xxxx>",
    "hooks": {
        "myapp": {
            "apparmor": "apparmor.json",
            "desktop": "app.desktop"
        }
    }
}
```

AppArmor profile policy file Contains which policy the app needs to work properly. See *Security and app isolation* below for more information on this file.

`.desktop` file The launcher file will tell UT how to launch the app, which name and icon to display on the home screen, and some other properties.

Example of `app.desktop`:

```
[Desktop Entry]
Name=Application title
Exec=qmlscene qml/Main.qml
Icon=assets/logo.svg
Terminal=false
Type=Application
X-Ubuntu-Touch=true
```

Non exhaustive list of properties:

- Name: Application title has shown in the dash
- Exec: Path to the executable file
- Icon: Path to the icon to display
- Terminal: `false` if will not run in terminal window
- Type: Specifies the type of the launcher file. The type can be Application, Link or Directory.
- X-Ubuntu-Touch: `true` to make the app visible
- X-Ubuntu-XMir-Enable: `true` if your app is built for X

Security and app isolation

All Ubuntu apps and scopes are confined respecting AppArmor access control mechanism (see Application Confinement) , meaning they only have access to their own resources and are isolated from other apps and parts of the system. The developer must declare which policy groups are needed for the app or scope to function properly with an apparmor `.json` file.

Example `apparmor.json` file:

```
{
    "policy_version": 16.04,
    "policy_groups": [
        "networking",
        "webview",
        "content_exchange"
    ]

}
```

For a full list of available policy groups, see *AppArmor Policy Groups*.

7.7 Sample apps

Learn more about app development by digging into our *Sample apps*.

7.7.1 Sample Apps

We are currently working on expanding our list of sample apps. In the mean time many apps on the OpenStore are open source and can be used as a reference.

If you are interested in helping create sample apps check out our GitLab Project.

7.8 Preinstalled apps

The *Preinstalled apps* page has information on developing the apps which are included with Ubuntu Touch.

CHAPTER 8

System software development

This section has various documents which will teach you how to work with the packages included with Ubuntu Touch. This includes the Ubuntu UI Toolkit, Unity8, and all of the other software that makes Ubuntu Touch what it is.

This section does not cover most of the *applications* preinstalled on Ubuntu Touch. See *Preinstalled apps* for more information on those.

8.1 System Software guides

These guides will give you general instructions on building and testing your own changes to Ubuntu Touch system software. They are not not an exhaustive reference on everything you will come across during development, but they are a great starting point.

Note: If you get stuck at any point while going through this documentation, please contact us for help via the UBports Forum or your preferred communication medium.

8.1.1 Making changes and testing locally

On this page you'll find information on how to build Ubuntu Touch system software for your device. Most of the software preinstalled on your Ubuntu Touch device is shipped in the device image in the form of a Debian package. This format is used by several Linux distributions, such as Debian, Ubuntu, and Linux Mint. Plenty of documentation on deb packages is available, so we won't be covering it here. Besides, in most cases you'll find yourself in need of modifying existing software rather than developing new packages from scratch. For this reason, this guide is mostly about recompiling an existing Ubuntu Touch package.

There are essentially two ways of developing Ubuntu Touch system software locally:

- *Cross-building with crossbuilder*
- *Building on the device itself*

We'll examine both methods, using address-book-app (the Contacts application) as an example.

We only recommend developing packages using a device with Ubuntu Touch installed from the devel channel. This ensures that you are testing your changes against the most current state of the Ubuntu Touch code.

Warning: Installing packages has a risk of damaging the software on your device, rendering it unusable. If this happens, you can *reinstall Ubuntu Touch*.

Cross-building with crossbuilder

Crossbuilder is a script which automates the setup and use of a crossbuild environment for Debian packages. It is suitable for developers with any device since the code compilation occurs on your desktop PC rather than the target device. This makes Crossbuilder the recommended way to develop non-trivial changes to Ubuntu Touch.

Note: Crossbuilder requires a Linux distribution with `lxd` installed and the unprivileged commandset available. In other words, you must be able to run the `lxc` command. If you are running Ubuntu on your host, Crossbuilder will set up `lxd` for you.

Start by installing Crossbuilder on your host:

```
cd ~
git clone https://github.com/ubports/crossbuilder.git
```

Crossbuilder is a shell script, so you don't need to build it. Instead, you will need to add its directory to your `PATH` environment variable, so that you can execute it from any directory:

```
echo "export PATH=$HOME/crossbuilder:$PATH" >> ~/.bashrc
# and add it to your own session:
export PATH="$HOME/crossbuilder:$PATH"
```

Now that Crossbuilder is installed, we can use it to set up LXD:

```
crossbuilder setup-lxd
```

If this is the first time you have used LXD, you might need to reboot your host once everything has completed.

After LXD has been set up, move to the directory where the source code of your project is located (for example, `cd ~/src/git/address-book-app`) and launch Crossbuilder:

```
crossbuilder
```

Crossbuilder will create the LXD container, download the development image, install all your package build dependencies, and perform the package build. It will also copy the packages over to your target device and install them if it is connected (see *Shell access via adb* to learn more about connecting your device). The first two steps (creating the LXD image and getting the dependencies) can take a few minutes, but will be executed only the first time you launch crossbuilder for a new package.

Now, whenever you change the source code in your git repository, the same changes will be available inside the container. The next time you type the `crossbuilder` command, only the changed files will be rebuilt.

Unit tests

By default crossbuilder does not run unit tests; that's both for speed reasons, and because the container created by crossbuilder is not meant to run native (target) executables: the development tools (qmake/cmake, make, gcc, etc.) are all run in the host architecture, with no emulation (again, for speed reasons). However, qemu emulation is available inside the container, so it should be possible to run unit tests. You can do that by getting a shell inside the container:

```
crossbuilder shell
```

Then find the unit tests and execute them. Be aware that the emulation is not perfect, so there's a very good chance that the tests will fail even when they'd otherwise succeed when run in a proper environment. For that reason, it's probably wiser not to worry about unit tests when working with crossbuilder, and run them only when not cross-compiling.

Building on the device itself

This is the fastest and simplest method to develop small changes and test them in nearly real-time. Depending on your device resources, however, it might not be possible to follow this path: if you don't have enough free space in your root filesystem you won't be able to install all the package build dependencies; you may also run out of RAM while compiling.

> **Warning:** This method is limited. Many devices do not have enough free image space to install the packages required to build components of Ubuntu Touch.

In this example, we'll build and install the address-book-app. All commands shown here must be run on your Ubuntu Touch device over a remote shell.

You can gain a shell on the device using *Shell access via adb* or *Shell access via ssh*. Remount the root filesystem read-write to begin:

```
sudo mount / -o remount,rw
```

Next, install all the packages needed to rebuild the component you want to modify (the Contacts app, in this example):

```
sudo apt update
sudo apt build-dep address-book-app
sudo apt install fakeroot
```

Additionally, you probably want to install `git` in order to get your app's source code on the device and later push your changes back into the repository:

```
sudo apt install git
```

Once you're finished, you can retrieve the source for an app (in our example, the address book) and move into its directory:

```
git clone https://github.com/ubports/address-book-app.git
cd address-book-app
```

Now, you are ready to build the package:

```
DEB_BUILD_OPTIONS="parallel=2 debug" dpkg-buildpackage -rfakeroot -b
```

The `dpkg-buildpackage` command will print out the names of generated packages. Install those packages with `dpkg`:

```
sudo dpkg -i ../<package>.deb [../<package2>.deb ...]
```

Note, however, that you might not need to install all the packages: generally, you can skip all packages whose names end with `-doc` or `dev`, since they don't contain code used by the device.

Next steps

Now that you've successfully made changes and tested them locally, you're ready to upload them to GitHub. Move on to the next page to learn about using the UBports CI to build and provide development packages!

8.1.2 Uploading and testing with `ubports-qa`

The UBports build service is capable of building Ubuntu Touch packages and deploying them to the UBports repository. This capability is offered to any developer who wishes to take advantage of it.

This guide assumes that you have a cursory understanding of using Git and making Pull Requests on GitHub.

To use the UBports build service, make sure you understand our *branch naming convention*. It is required that you follow the convention for deb-packages for CI to build your package correctly.

Fork the repository

The first step to make a change to any repository you don't have write access to is to fork it. Open your desired repository on GitHub and click the "Fork" button in the upper right corner. If offered, select an appropriate account to fork the repository to. Then, clone your fork to your computer.

Now you're ready to make changes!

Make and commit changes

Now that you have the package source downloaded, you can make your desired changes.

Before changing anything, make sure you have checked out the branch you want to work from (probably `xenial`, assuming you are making changes for the phone images). Then, create a new branch abiding by the *branch naming convention*.

After making your changes, commit them with a descriptive commit message stating what is wrong and why your changes fix that problem.

You have successfully created and committed your changes. Before pushing your changes, we'll want to make sure your device will install them.

Update the `debian/changelog` file

Generally, `apt` will not install a new package from any repository if it has a lower (or the same) version number as the package it replaces. Users may also want to see the changes that are included in a new version of a package. For that reason, we will need to update the package changelog to add a new version.

Note: This is not an exhaustive reference of the `debian/changelog` format. See deb-changelog(5) for more information.

Determine a new version number

To start, figure out what the current version numbering for the package is:

```
head debian/changelog
```

This will return a few lines, but the first is the most important to us:

```
morph-browser (0.24+ubports2) xenial; urgency=medium
```

The part inside the parentheses `(0.24+ubports2)` is our version number. It consists of several parts:

1. The `0.24` is the *upstream version number*, the version that the original project maintainers give to the release we are using. For most UBports projects, the repository you'll be working on is the original project code. This makes UBports the "upstream" of that project.

If you are making large changes to the repository and UBports is the upstream, you should increment the first part of the version number before the plus (+) and reset the distribution suffix. In our example above, you would make this new version number:

```
0.25+ubports0
```

If you are making changes only to the package build (files in the `debian/` folder), it is best to only increment the version suffix:

```
0.24+ubports3
```

Note: If you find a package which does not seem to follow the above versioning format, please contact us to ask how to proceed.

Write the changelog entry

Now it is time to write your changelog entry! Start with the following template:

```
PACKAGE-NAME (VERSION) DISTRIBUTION; urgency=medium

  * CHANGES

-- NAME <EMAIL>  DATETIME
```

If you open the `debian/changelog` file, you'll find that every entry follows this format. This helps everyone (including computers) read and understand the contents. This is used, for example, to name the output package correctly for every package version.

Let's assume I, John Doe, am making a packaging change to the `morph-browser` package for Ubuntu Touch. I'll replace the different all-caps placeholders above in the following way:

- PACKAGE-NAME is replaced with `morph-browser`
- VERSION is replaced with `0.24+ubports3` (which we determined above)
- DISTRIBUTION is replaced with `xenial`
- CHANGES is replaced with the changes I made in this release. This will include summarized information from my commit messages along with the bugs fixed by those changes. If I've fixed multiple bugs, I'll create multiple bullet points.

- NAME is replaced with my name, `John Doe`
- EMAIL is replaced with my e-mail, `john.doe@example.com`.

> **Note:** You should not use a "noreply" e-mail as your EMAIL for package changelog entries.

- DATETIME is replaced with the date and time I made this changelog entry in RFC2822/RFC5322 format. The easiest way to retrieve this is by running the command `date -R` in a terminal.

Note that no line in your changelog entry should exceed 80 characters in length.

With that, my new changelog entry follows:

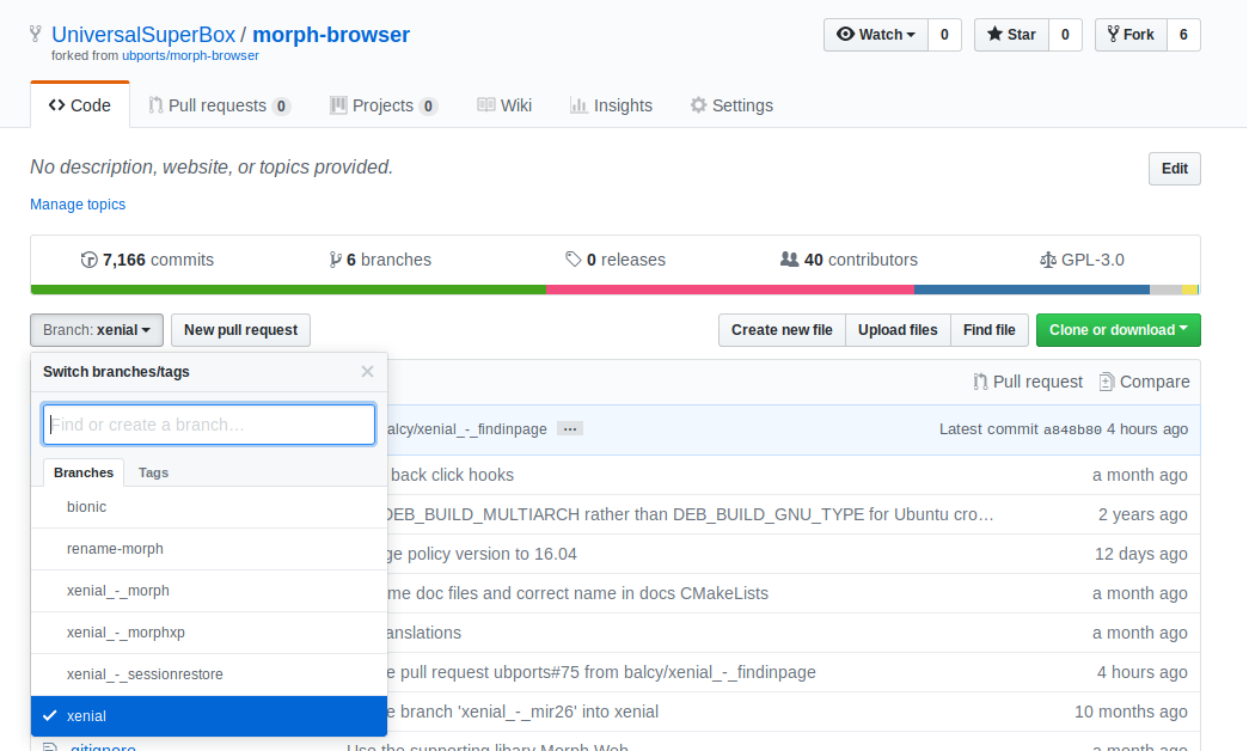
```
morph-browser (0.24+ubports3) xenial; urgency=medium

  * Add the new "Hello world" script to the package. Fixes
    https://github.com/ubports/morph-browser/issues/404.
  * Fix whitespace and formatting in the format.qml file

-- John Doe <john.doe@example.com>  Mon, 29 Oct 2018 12:53:08 -0500
```

Add your new changelog entry to the top of the `debian/changelog` file and commit it with the message "Update changelog". Push your changes. Now you're ready to make your Pull Request!

Create your pull request

A pull request asks UBports maintainers to review your code changes and add them to the official repository. We'll create one now.

Open your fork of the repository on GitHub. Navigate to the branch that you just pushed to using the "Branch" selector:



Once you've opened your desired branch, click the "New pull request" button to start your pull request. You'll be taken to a page where you can review your changes and create a pull request.

Give your pull request a descriptive title and description (include links to reference bugs or other material). Ensure that the "base" branch is the one you want your changes to be applied to (likely `xenial`), then click "Create pull request".

With your pull request created, we can move on to testing your changes using the UBports build service!

Test your changes

Once your pull request is built (a green check mark appears next to your last commit), you are ready to test your changes on your device.

Note: If a red "X" appears next to your last commit, your pull request has failed to build. Click the red "X" to view the build log. Until your build errors are resolved, your pull request cannot be installed or accepted.

We'll use `ubports-qa` to install your changes. Take note of your pull request's ID (noted as `#number` after the title of the pull request) and follow these steps to install your changes:

1. Ensure your device is running the newest version of Ubuntu Touch from the `devel` channel.
2. Get shell access to your device using *Shell access via adb* or *Shell access via ssh*.
3. Run `sudo ubports-qa install REPOSITORY PR`, replacing `REPOSITORY` with the name of the repository you have submitted a PR to (`morph-browser` for example) and PR with the number of your pull request (without the `#`).

`ubports-qa` will automatically add the repository containing your changed software and start the installation for you. All you will need to do is check the packages it asks you to install and say "yes" if they are correct.

If `ubports-qa` fails to install your packages, run it again with the `-v` flag (for example, `ubports-qa -v install ...`). If it still fails, submit the entire log (starting from the `$` before the `ubports-qa` command) to Ubuntu Pastebin and contact us for help.

Once `ubports-qa` is finished, test your changes to ensure they have fixed the original bug. Add the `ubports-qa` command to your pull request, then send the link to the pull request to other developers and testers so they may also test your changes.

When getting feedback from your testers, be sure to add the information to the pull request (or ask them to do it for you) so that everyone is updated on the status of your code.

Every time you make a change and push it to GitHub, it will trigger a new build. You can run `sudo ubports-qa update` to get the freshest changes every time this happens.

Celebrate!

If you and your testers are satisfied with the results of your pull request, it will be merged. Following the merge, the UBports build service will build your code and deploy it to Ubuntu Touch users worldwide.

Thank you for your contribution to Ubuntu Touch!

8.2 System Software reference

This section includes reference guides on how different parts of the Ubuntu Touch system interact to create the user experience.

There's nothing here yet, but maybe you'd like to add some reference material? Check out *our guide to contributing to documentation* to learn more.

CHAPTER 9

Halium porting

Note: If you are looking for information on installing Ubuntu Touch on a supported device, or if you would like to check if your device is supported, please see this page.

This section will introduce you to some of the specifics of porting Ubuntu Touch to an Android device by building a Halium image.

This process does not build Ubuntu Touch! A Halium image is installed along with a prebuilt Ubuntu Touch filesystem to create a running Ubuntu Touch system. If you already have an Ubuntu Touch device and would like to modify the software on it, you will be better served by *System software development*. If you would like to modify the Android compatibility image for the Nexus 5, Oneplus One, or Fairphone 2, *Legacy porting* is appropriate for you.

Before you begin, you'll want to head over to the Halium porting guide and get your `systemimage` built without errors. Once you're at the point of installing a rootfs, you can come back here.

Start at *Building halium-boot* if you'd like to install the UBports Ubuntu Touch 16.04 rootfs.

9.1 Building halium-boot

Halium-boot is a new proposed boot image in the Halium project, replacing hybris-boot. We will be building and using it for Ubuntu Touch.

9.1.1 Fix mounts

Halium-boot's `mount` is not aware of SELinux contexts. If your device's `fstab` file includes any contexts, the partition that they are on will fail to mount and your port will not work correctly.

The first step to this process is figuring out where your fstab actually is. For most, this is inside `BUILDDIR/device/MANUFACTUER/CODENAME/rootdir/etc` and it is named either `fstab.qcom` or `fstab.devicename`. Open the file for editing.

If the type of the 'data' or 'userdata' partition is `f2fs`, it is required to change it to `ext4`.

With the file open, remove all `context=` options from all block devices in the file. The option will start at the text `context=` and end at the comma following it.

For example, the line `ro,nosuid,nodev,context=u:object_r:firmware_file:s0,barrier=0` should become `ro,nosuid,nodev,barrier=0`

Save and exit.

9.1.2 Edit kernel config

Ubuntu Touch requires a slightly different kernel config than Halium, including enabling Apparmor. Luckily, we have a nice script for this purpose, `check-kernel-config`. It's included in the halium-boot repository. Simply run it on your config as follows:

```
./halium/halium-boot/check-kernel-config path/to/my/defconfig -w
```

You may have to do this twice. It will likely fix things both times. Then, run the script without the `-w` flag to see if there are any more errors. If there are, fix them manually. Once finished, run the script without the `-w` flag one more time to make sure everything is correct.

9.1.3 Build the image

Once halium-boot is in place, you can build it quite simply. You will also need to rebuild system.img due to our changes.

1. `cd` to your Halium BUILDDIR
2. `source build/envsetup.sh`
3. Run `breakfast` or `lunch`, whichever you use for your device
4. `mka halium-boot`
5. `mka systemimage`

9.1.4 Continue on

Now that you have halium-boot built, you can move on to *Installing Ubuntu Touch 16.04 images on Halium*.

9.2 Installing Ubuntu Touch 16.04 images on Halium

> **Warning:** These steps will wipe **all** of the data on your device. If there is anything that you would like to keep, ensure it is backed up and copied off of the device before continuing.

Now that you've *built halium-boot*, we're ready to install Ubuntu Touch on your device.

In order to install Ubuntu Touch, you will need a recovery with Busybox, such as TWRP, installed on your phone. You will also need to ensure the /data partition is formatted with ext4 and does not have any encryption on it.

9.2.1 Install halium-boot

We'll need to install the halium-boot image before installing an image. Reboot your phone into fastboot mode, then do the following from your Halium tree:

```
cout
fastboot flash boot halium-boot.img
```

9.2.2 Download the rootfs

Next we'll need to download the rootfs (root filesystem) that's appropriate for your device. Right now, we only have one available. Simply download `ubports-touch.rootfs-xenial-armhf.tar.gz` from our CI server. If you have a 64-bit ARM (aarch64) device, this same rootfs should work for you. If you have an x86 device, let us know. We do not have a rootfs available for these yet.

9.2.3 Install system.img and rootfs

Clone or download the halium-install repository. This repository contains tools that can be used to install a Halium system image and distribution rootfs. We'll use the `halium-install` script to install Ubuntu Touch on your device:

```
path/to/halium-install -p ut path/to/rootfs.tar.gz path/to/system.img
```

The script will copy and extract the files to their proper places, then allow you to set the phablet user's password.

9.2.4 Get SSH access

When your device boots, it will likely stay at the bootloader screen. However, you should also get a new network connection on the computer you have it plugged in to. We will use this to debug the system.

To confirm that your device has booted correctly, run `dmesg -w` and watch for "GNU/Linux device" in the output. If you instead get something similar to "Halium initrd Failed to boot", please get in contact with us so we can find out why.

Similar to the Halium reference rootfs, you should set your computer's IP on the newly connected RNDIS interface to `10.15.19.100` if you don't get one automatically. Then, run the following to access your device:

```
ssh phablet@10.15.19.82
```

The password will be the one that you set while running halium-install.

9.2.5 Common Problems

If you have any errors while performing these steps, check see if any of the following suggestions match what you are seeing. If you have installed successfully, skip down to *Continue on*.

Common installation problems

This page details problems commonly faced while following the *Installing Ubuntu Touch 16.04 images on Halium* page.

SSH hangs when trying to connect

The SSH connection may hang indefinitely when trying to connect. Attempts to stop the connection with Control-C may or may not return you to a shell prompt. If you run `ssh -vvvv phablet@10.15.19.82`, you only get the following output before the program stops:

```
debug1: Connecting to 10.15.19.82 [10.15.19.82] port 22.
debug1: Connection established.
[...]
debug1: Enabling compatibility mode for protocol 2.0
debug1: Local version string SSH[...]
```

This seems to be a common error on arm64 devices with kernel 3.10 when rsyslogd is enabled. If you have this error, please add your voice to ubports/ubuntu-touch#560 and then try the following workaround:

1. Reboot the device to recovery and connect with `adb shell`
2. Run the following commands to disable rsyslogd:

```
mkdir /a
mount /data/rootfs.img /a
echo manual |tee /a/etc/init/rsyslog.override
umount /a
sync
```

You may now reboot the device. You should be able to connect to SSH once it comes back online.

Device reboots after a minute

The device may reboot cleanly after about a minute of uptime. If you are logged in when the reboot occurs, you will see the following message:

```
Broadcast message from root@ubuntu-phablet
    (unknown) at 16:00 ...

The system is going down for reboot NOW!
```

This happens because `lightdm`, the Ubuntu Touch display manager, is crashing repeatedly. The system watchdog process sees this and reboots the device.

To fix this problem, log in before the reboot occurs and run the following command:

```
sudo stop lightdm
```

9.2.6 Continue on

Congratulations! Ubuntu Touch has now booted on your device. Move on to *Running Ubuntu Touch* to learn about more specific steps you will need to take for a complete port.

9.3 Running Ubuntu Touch

Now that you're logged in, there are a few more steps before Ubuntu Touch will be fully functional on your device.

9.3.1 Make / writable

Before we make any changes to the rootfs (which will be required for the next steps), you'll need to remount it with write permissions. To do that, run the following command:

```
sudo mount -o remount,rw /
```

9.3.2 Add udev rules

You must create some udev rules to allow Ubuntu Touch software to access your hardware. Run the following command, replacing [codename] with your device's codename:

```
sudo -i # And enter your password
cat /var/lib/lxc/android/rootfs/ueventd*.rc|grep ^/dev|sed -e 's/^\/dev\///'|awk '
↪{printf "ACTION==\"add\", KERNEL==\"%s\", OWNER=\"%s\", GROUP=\"%s\", MODE=\"%s\"\n
↪",$1,$3,$4,$2}' | sed -e 's/\r//' >/usr/lib/lxc-android-config/70-[codename].rules
```

Now, reboot the device. If all has gone well, you will eventually see the Ubuntu Touch spinner followed by Unity 8. Your lock password is the same as you set for SSH.

9.3.3 Display settings

When the device boots, you'll probably notice that everything is very small. There are two variables that set the content scaling for Unity 8 and Ubuntu Touch applications: `GRID_UNIT_PX` and `QTWEBKIT_DPR`.

There are also some other options available that may be useful for you depending on your device's form factor. These are discussed below.

All of these settings are guessed by Unity 8 if none are set. There are many cases, however, where the guess is wrong (for example, very high resolution phone displays will be identified as desktop computers). To manually set a value for these variables, simply create a file at `/etc/ubuntu-session.d/[codename].conf` specifying them. For example, this is the file for the Nexus 7 tablet:

```
$ cat /etc/ubuntu-touch-session.d/flo.conf
GRID_UNIT_PX=18
QTWEBKIT_DPR=2.0
NATIVE_ORIENTATION=landscape
FORM_FACTOR=tablet
```

Methods for deriving values for these variables are below.

Display scaling

`GRID_UNIT_PX` (Pixels per Grid Unit or Px/GU) is specific to each device. Its goal is to make the user interface of the system and its applications the same *perceived* size regardless of the device they are displayed on. It is primarily dependent on the pixel density of the device's screen and the distance to the screen the user is at. The latter value cannot be automatically detected and is based on heuristics. We assume that tablets and laptops are the same distance and that they are held at 1.235 times the distance phones tend to be held at.

`QTWEBKIT_DPR` sets the display scaling for the Oxide web engine, so changes to this value will affect the scale of the browser and webapps.

A reference device has been chosen from which we derive the values for all other devices. The reference device is a laptop with a 120ppi screen. However, there is no exact formula since these options are set for *perceived* size rather than *physical* size. Here are some values for other devices so you may derive the correct one for yours:

Device	Resolution	Display Size	PPI	Px/GU	QtWebKit DPR
'Normal' density laptop	N/A	N/A	96-150	8	1.0
ASUS Nexus 7	1280x800	7"	216	12	2.0
'High' density laptop	N/A	N/A	150-250	16	1.5
Samsung Galaxy Nexus	1280x720	4.65"	316	18	2.0
LG Nexus 4	1280x768	4.7"	320	18	2.0
Samsung Nexus 10	2560x1600	10.1"	299	20	2.0
Fairphone 2	1080x1920	5"	440	23	2.5
LG Nexus 5	1080x1920	4.95"	445	23	2.5

Experiment with a few values to find one that feels good when compared to the Ubuntu Touch experience on other devices. If you are unsure of which is the best, share some pictures (including some object for scale) along with the device specs with us.

Form factor

There are two other settings that may be of interest to you.

`FORM_FACTOR` specifies the device's form factor. This value is set as the device's Chassis, which you can find by running `hostnamectl`. The acceptable values are `handset`, `tablet`, `laptop` and `desktop`. Apps such as the gallery use this information to change their functionality. For more information on the Chassis, see the freedesktop.org hostnamed specification.

`NATIVE_ORIENTATION` sets the display orientation for the device's built-in screen. This value is used whenever autorotation isn't working correctly or when an app wishes to be locked to the device's native orientation. Acceptable values are `landscape`, which is normally used for tablets, laptops, and desktops; and `portrait`, which is usually used for phone handsets.

9.3.4 Common Problems

If you have any errors while performing these steps, check see if any of the following suggestions match what you are seeing. If you have completed these steps successfully, congratulations! You've reached the end of the porting guide for now. Try to check the functionality of your device by following the Smoke Testing information in *Quality Assurance*.

Common problems when running Ubuntu Touch

This page details problems commonly faced while following the *Running Ubuntu Touch* page.

Nothing shows on screen

If nothing is showing on screen even after adding udev rules to your port, it is likely that you have a problem with graphical applications crashing. See *Mir servers crash* for more information.

Mir servers crash

Mir servers crashing can be caused by many different problems with the port. To troubleshoot more, you can try the following:

Is the Android container running?

If the Android container is not running, many parts of Ubuntu Touch will not work. Run this command to check on the container's status:

```
sudo lxc-info -n android
```

If you get output similar to the following, the Android container is running and you can move on to the next troubleshooting step:

```
Name:           android
State:          RUNNING
PID:            1194
IP:             10.15.19.82
```

If you do not get `State: RUNNING`, the container is stopped. You can run `sudo start lxc-android-config` to attempt to start it. If the container does not start after that, you can run `sudo lxc-start -n android -F` to get a more detailed log of why it has failed.

Are all of the Android partitions mounted?

If some partitions used for Android drivers are not mounted, the container may start but not work correctly.

To check the mounted Android partitions, run `ls /android`. At least the following folders should be contained within:

```
data
system
firmware
persist
```

If any of these are missing, run `dmesg` to get the kernel log. Mounting Android partitions will start after the following line:

```
initrd: checking fstab [...] for additional mount points
```

Try to diagnose and fix any mounting errors that you find in the log for the partitions listed above.

Note: Some devices have a `vendor` partition that contains proprietary libraries and executables required to run Android. If your device has this partition, make sure that it is mounted in addition to the others listed above.

Getting more Mir logs

If the Android container is running and all of its partitions seem to be mounted, you will need to get a few more logs before enlisting community help.

First, stop the display manager if it is not already:

```
sudo stop lightdm
```

If you have Wi-Fi working (See the Halium docs for Wi-Fi), install the `libc6-dbg` package first:

```
sudo apt update
sudo apt install libc6-dbg
```

Then, run the following commands to get all of the needed logs:

```
sudo unity-system-compositor --debug-without-dm &> ~/usc.log
sudo gdb -ex 'set confirm off' -ex 'run' -ex 'bt full' -ex quit --args unity-system-
↪compositor --debug-without-dm &> ~/usc-gdb.log
sudo /system/bin/logcat -d &> ~/usc-logcat.log
```

Use `scp` or a similar program to copy the `usc.log`, `usc-gdb.log`, and `usc-logcat.log` files from phablet's home folder to your computer. Then, post the content of these files to paste.ubuntu.com, Pastebin, GitHub Gists, or a similar service so the people helping you can view them easily.

Programs hang before crashing

Sometimes processes will hang for a very long time and then abort or segfault. The reason for the hang is apport, which attempts to collect useful information about the crash before allowing the program to stop.

If you don't need apport's information and would rather have the programs crash faster while troubleshooting, issue the following commands:

```
sudo stop apport
sudo stop whoopsie
```

Enable /var/log/syslog

Normally the writing to the syslog is disabled. During porting it can be useful to enable this:

```
sudo touch /var/log/syslog
sudo chown syslog:syslog /var/log/syslog
sudo initctl stop rsyslog
sudo initctl start rsyslog
```

Now rsyslogd will write to the file and you can use it as usual. For example `less /var/log/syslog` or `tail -f /var/log/syslog`.

9.4 Getting community help

If you're having trouble with the porting steps after building your Halium systemimage, check out the "Common Problems" section of the page you're stuck on. If none of the suggestions are helpful, you can get help from the community via the following channels:

- Telegram: @ubports_porting
- IRC: #ubports-porting on freenode
- Forum: forums.ubports.com

CHAPTER 10

Legacy porting

This page documents the resources and process to build an Android compatibility image for the LG Nexus 5, OnePlus One, or Fairphone 2. This information should not be used to bring up any new devices, only to update these older images. All new ports should be created using Halium following the documentation starting at *Halium porting*.

This document is useful if you would like to:

- Fix an Android compatibility-related hardware issue (camera, sensors, radios)
- Experiment with the Linux kernel
- Experiment with the system-image upgrade process

This document is not useful if you would like to modify *Preinstalled apps* or *System software*. See the respective documentation for each.

This process does not build Ubuntu Touch! An Android compatibility image is installed along with a prebuilt Ubuntu Touch filesystem to create a running Ubuntu Touch system.

This document assumes you already have knowledge of building Android or Halium. It also assumes that your device has Ubuntu Touch installed.

10.1 Getting set up

`ubp-5.1` ports must be built using Ubuntu 16.04. A container or virtual machine based on 16.04 is recommended for this purpose.

Let's get started by installing some build dependencies:

```
sudo dpkg --add-architecture i386 && sudo apt update
sudo apt install schedtool gcc g++ g++-multilib zlib1g-dev:i386 \
     zip libxml2-utils bc python-launchpadlib phablet-tools
```

Create a directory for your `ubp-5.1` source:

```
mkdir ~/ubp-5.1
cd ~/ubp-5.1
```

Next, we'll initialize the repository:

```
repo init -u https://github.com/ubports/android -b ubp-5.1-allthefixings --depth=1
```

Note: The `allthefixings` branch is provided for convenience. It adds all of the current UT device ports to the tree at the expense of a bit more downloaded data. If you are download-sensitive, initialize using `-b ubp-5.1` and use the manifest in build-scripts to pick the repos you like.

Finally, we'll download the source:

```
repo sync -j10 -c
```

10.2 Set up and build

With the sources downloaded, we need to set up our environment and build the images. Make sure you're in your `ubp-5.1` directory to continue through these steps.

First, bring in the default Android build environment:

```
source build/envsetup.sh
```

Run `lunch` and pick the appropriate combo for your device. The name of the combination should start with `cm_`, followed by the device name and ending with `-userdebug`:

```
lunch
```

With that done, the build can be started:

```
mka
```

10.3 Install the new image

Now that the build is complete, we can flash it to the device. Note that all of these commands should be run from a terminal which has been set up with `source build/envsetup.sh` and `lunch` to ensure the needed tools are in your PATH.

We'll begin with the boot and recovery images. Boot your device into fastboot mode and run the following commands:

```
cout
fastboot flash boot boot.img
fastboot flash recovery recovery.img
```

Now boot your device to ensure your kernel build is sane. You may also want to boot into recovery to ensure it is working as well.

To install your new build of the system image, use the replace-android-system script. It can be run as follows with your device attached:

```
./replace-android-system system.img
```