
Builder Documentation

Release 3.25.2-nightly

Christian Hergert, et al.

Jun 19, 2017

1	Contents	3
1.1	Installation	3
1.1.1	via Flatpak	3
1.1.1.1	Command Line	3
1.1.2	Local Flatpak Builds	4
1.1.3	via JHBuild	4
1.1.3.1	Command Line	4
1.1.4	via Release Tarball	5
1.1.5	Troubleshooting	5
1.2	Exploring the Interface	5
1.2.1	Project Greeter	6
1.2.2	Workbench Window	6
1.2.3	Header Bar	6
1.2.4	Switching Perspectives	7
1.2.5	Showing and Hiding Panels	7
1.2.6	Build your Project	8
1.2.7	Preferences	10
1.2.8	Command Bar	11
1.3	Projects	12
1.3.1	Creating and Importing Projects	12
1.3.1.1	Creating a new Project	12
1.3.1.2	Cloning an Existing Project	13
1.3.2	Building your Project	13
1.3.3	Debugging your Project	14
1.3.4	Profiling your Project	14
1.3.5	Sharing your Project	14
1.4	Plugins	14
1.4.1	Creating Your First Plugin	15
1.4.1.1	Loading our Plugin	15
1.4.2	Extending the Workbench	16
1.4.2.1	The Basics	16
1.4.2.2	Registering Workbench Actions	16
1.4.2.3	Adding Widgets to the Header Bar	17
1.4.2.4	Adding Widgets to the Workbench	18
1.4.2.5	Registering Perspectives	18
1.4.2.6	Registering Panels	18

1.4.3	Extending the Greeter	19
1.4.4	Extending the Editor	19
1.4.4.1	Managing Buffers	19
1.4.4.2	Syntax Highlighting	19
1.4.4.3	Diagnostics and Fix-Its	20
1.4.4.4	Autocompletion	20
1.4.4.5	Snippets	20
1.4.4.6	File Settings and Indentation	20
1.4.5	Symbols and Semantic Analysis	20
1.4.5.1	Go To Definition	20
1.4.5.2	Extending the Symbol Tree	20
1.4.5.3	Renaming Symbols	20
1.4.6	Extending the Build Pipeline	20
1.4.6.1	Implementing a Build System	20
1.4.6.2	Extending the Build Pipeline	21
1.4.7	Processes and Containers	21
1.4.7.1	Application Runtimes and Containers	21
1.4.7.2	Subprocesses and Psuedo Terminals	22
1.4.8	Extending the Device Manager	25
1.4.9	Extending the Run Manager	25
1.4.10	Registering Keybindings	25
1.4.11	Integrating Language Servers	25
1.4.12	Extending Project Search	25
1.4.13	Extending Application Menus	25
1.4.14	Registering Application Preferences	25
1.4.15	Creating and Performing Transfers	25
1.4.16	Managing Worker Processes	25
1.4.17	Integrating Version Control	25
1.5	How-To Guides	25
1.5.1	Contents	25
1.5.1.1	Changing Indentation	25
1.5.1.2	Search and Replace	26
1.6	Troubleshooting	26
1.6.1	Verbose Output	26
1.6.2	Support Log	26
1.6.3	Counters	26
1.6.4	Test Builder Nightly	27
1.6.5	File a Bug	27
1.7	Contributing	27
1.7.1	Planning and Project Management	27
1.7.1.1	Responsibilities	27
1.7.2	Writing Documentation	27
1.7.2.1	Submitting Patches	28
1.7.2.2	Creating a Patch	28
1.7.2.3	Submitting a Patch	28
1.7.2.4	GNOME git Best Practices	29
1.7.3	Contributing Code	29
1.7.3.1	Where to Contribute?	29
1.7.4	IRC	29
1.7.5	File A Bug	29
1.7.6	Find A Bug To Work On	29
1.7.7	Building From Source	30
1.8	Credits	30

Welcome to the Builder project!

We're excited to have you here! The Builder project started out of a class teaching people to program for the GNOME platform. In the process, we realized that we need to improve our tooling so we started creating Builder! We hope you love using Builder to create great software for GNOME!

Installation

The preferred installation method for Builder is *via Flatpak*. This provides a bandwidth efficient and safe to use installation method that can be easily kept up to date. It is also the engine behind Builder's powerful SDK!

via Flatpak

If you have a recent Linux distribution, such as Fedora 25, simply download the [Stable Flatpak](#) and click **Install** when [Software](#) opens. If [Software](#) does not automatically open, try opening the [Stable flatpakref](#) by double clicking it in your file browser.

If you want to track Builder development, you might want the [Nightly](#) channel instead of [Stable](#).

Note: To build flatpak-based applications, ensure that the `flatpak-builder` program is installed. On Fedora, this is the `flatpak-builder` package.

Command Line

You can also use the command line to install Builder:

Stable

```
$ flatpak install --user --from https://git.gnome.org/browse/gnome-apps-nightly/plain/  
→gnome-builder.flatpakref?h=stable  
$ flatpak run org.gnome.Builder
```

Nightly

```
$ flatpak install --user --from https://git.gnome.org/browse/gnome-apps-nightly/plain/  
↳gnome-builder.flatpakref  
$ flatpak run org.gnome.Builder
```

Note: Nightly builds are built with tracing enabled. The tracing is fairly lightweight, but it includes a great deal of more debugging information.

Local Flatpak Builds

You can also build Builder as a flatpak yourself to test local changes. First, make a repo for your local builds:

```
$ mkdir ~/my-flatpak-builds  
$ flatpak remote-add --user --no-gpg-verify my-flatpak-builds ~/my-flatpak-builds
```

Now, in Builder's source directory, use `flatpak-builder` to build a Builder flatpak and install it

```
$ git clone https://git.gnome.org/browse/gnome-builder/  
$ cd gnome-builder  
$ mkdir app  
$ flatpak-builder --ccache --repo=$HOME/my-flatpak-builds app org.gnome.Builder.json  
$ flatpak install --user my-flatpak-builds org.gnome.Builder
```

Note: After following these steps once you can omit adding the remote or creating the app directory. You'll also need to add the `--force-clean` option to `flatpak-builder` and use `flatpak update` rather than `flatpak install`.

via JHBuild

If you plan on contributing to the GNOME desktop and application suite, you may want to install Builder via [JHBuild](#). See the [Newcomers Tutorial](#) for more information on joining the community and installing [JHBuild](#).

We are aggressively moving towards using Flatpak for contributing to Builder, but we aren't quite there yet.

Command Line

Note: Please review the [GNOME Newcomers Tutorial](#) on how to build a GNOME application before proceeding.

```
# Make sure you have the following packages installed before starting  
  
# On Fedora  
$ sudo dnf install clang-devel llvm-devel libssh2-devel  
  
# On Ubuntu  
$ sudo apt-get install clang-3.9 libclang-3.9-dev llvm-3.9-dev libssh2-1-dev
```



```
$ git clone git://git.gnome.org/jhbuild.git
$ cd jhbuild
$ ./autogen.sh --simple-install
$ make
$ make install
$ jhbuild sysdeps --install gnome-builder
$ jhbuild build gnome-builder
$ jhbuild run gnome-builder
```

Warning: While it may be tempting to install jhbuild using your Linux distribution’s package manager, it will lack an updated description of the GNOME modules and is therefore insufficient. Always install jhbuild from git.

via Release Tarball

We do not recommend installing from release tarballs unless you are a Linux distribution. Builder has a complex set of dependencies which heavily target the current release of GNOME. Keeping up with these requires updating much of the GNOME desktop.

Please install via Flatpak, which does not have this restriction.

We use Meson (and thereby Ninja) to build Builder.

```
$ meson . build
$ ninja -C build install
```

Troubleshooting

If you are having trouble running Builder, start Builder with verbose output. This will log more information about the running system. The `gnome-builder` program can take multiple arguments of `-v` to increase verbosity. For example, if running from flatpak:

```
$ flatpak run org.gnome.Builder -vvvv
```

If you’re running from a system installed package of Builder, the binary name is `gnome-builder`.

```
$ gnome-builder -vvvv
```

If your issue persists, please consider [filing a bug](#).

Exploring the Interface

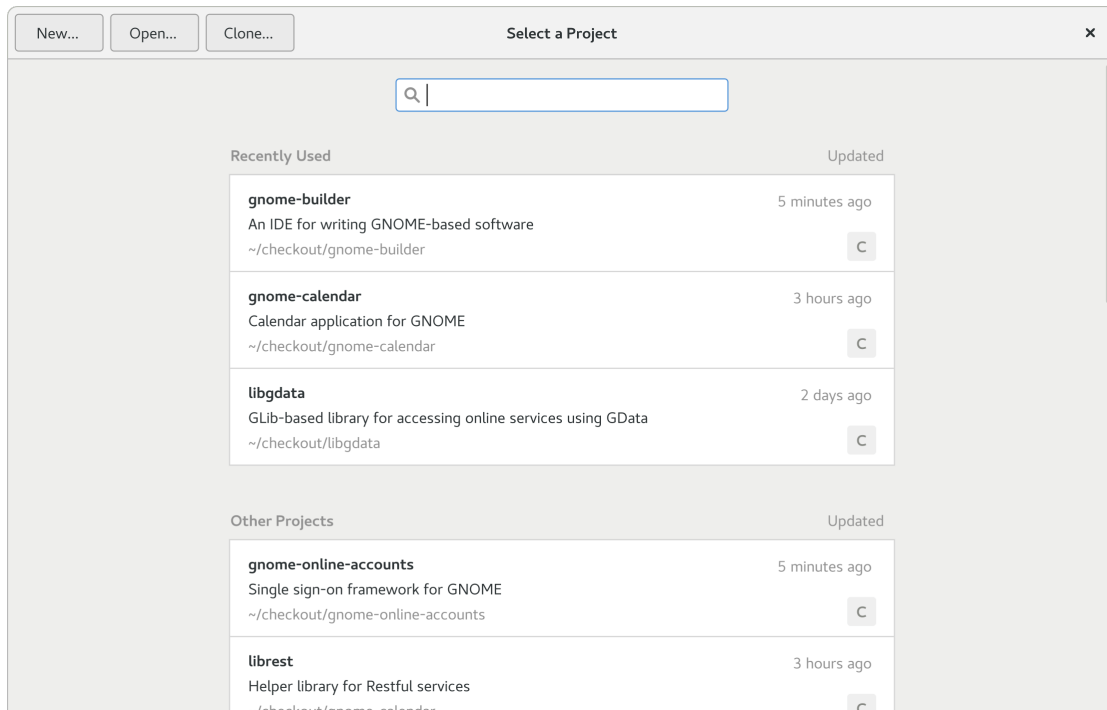
The following sections will help you get to know Builder.

- [Project Greeter](#)
- [Workbench Window](#)
- [Header Bar](#)
- [Switching Perspectives](#)
- [Showing and Hiding Panels](#)

- *Build your Project*
- *Preferences*
- *Command Bar*

Project Greeter

When you start Builder, you will be asked to select a project to be opened:



The window displays projects that were discovered on your system. By default, the `~/Projects` directory will be scanned for projects when Builder starts. Projects you have previously opened will be shown at the top.

Selecting a project row opens the project or pressing “Enter” will open the last project that was open. You can also start typing to search the projects followed by “Enter” to open.

If you’d like to remove a previously opened project from the list, activate *Selection mode*. Press the “Select” button in the top right corner to the left of the close application button and then select the row you would like to remove. Select the row(s) you’d like to remove and then click “Remove” in the lower left corner of the window.

Workbench Window

The application window containing your project is called the “**Workbench Window**”. The Workbench is split up into two main areas. At the top is the *Header Bar* and below is the current “**Perspective**”.

Builder has many perspectives, including the Editor, Build Preferences, Application Preferences, and the Profiler.

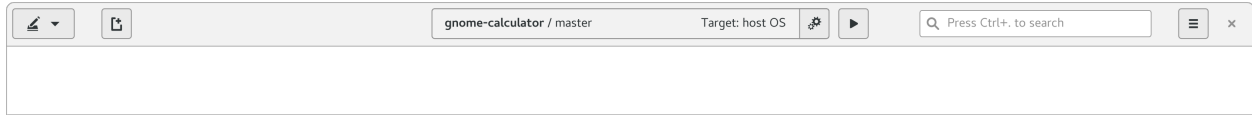
Header Bar

The header bar is shown below. This contains a button in the top left for *Switching Perspectives*. In the center is the “OmniBar” which can be used to *Build your Project*.

To the right of the OmniBar is the *Run* button. Clicking the arrow next to *Run* allows you to change how Builder will run your application. You can run normally, with a profiler, or with Valgrind.

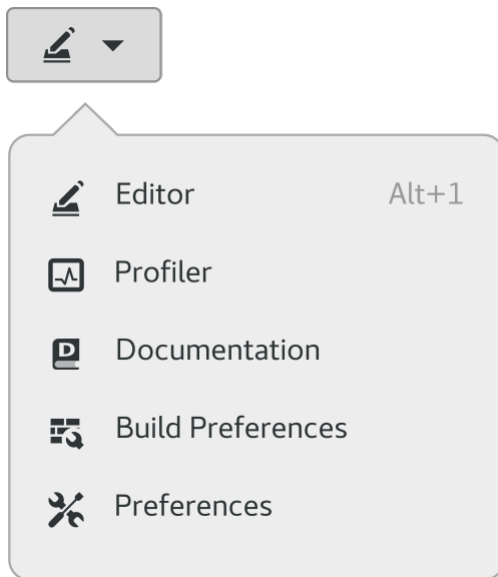
On the right is the search box. Type a few characters from the file you would like to open and it will fuzzy search your project tree. Use “Enter” to complete the request and open the file.

To the right of the search box is the workbench menu. You can find more options here such as *Showing and Hiding Panels*.



Switching Perspectives

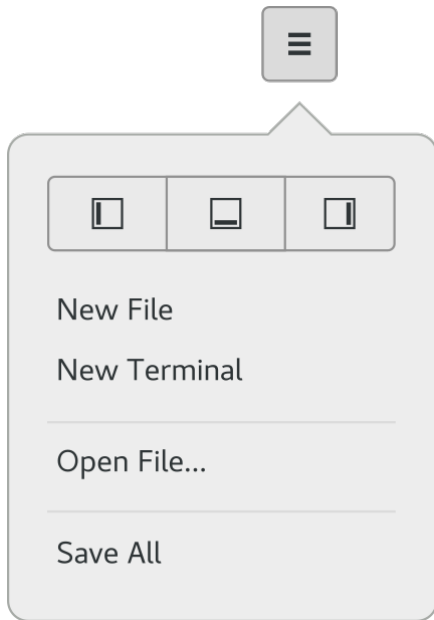
To switch perspectives, click the perspective selector button in the top left of the workbench window. Perspectives that support a keyboard accelerator will display the appropriate accelerator next to name of the perspective.



Select the row to change perspectives.

Showing and Hiding Panels

Sometimes panels get in the way of focusing on code. You can move them out of the way using the workbench menu in the top-right.



Additionally, you can use the “left-visible”, “right-visible”, “bottom-visible” commands from the *Command Bar* to toggle their visibility.

Build your Project


To build your project, use the OmniBar in the center of the header bar. To the right of the OmniBar is a button for starting a build as shown in the image below.

Build

gnome-calculator / master  Build failed 

Project ~/Projects/gnome-builder
Branch master

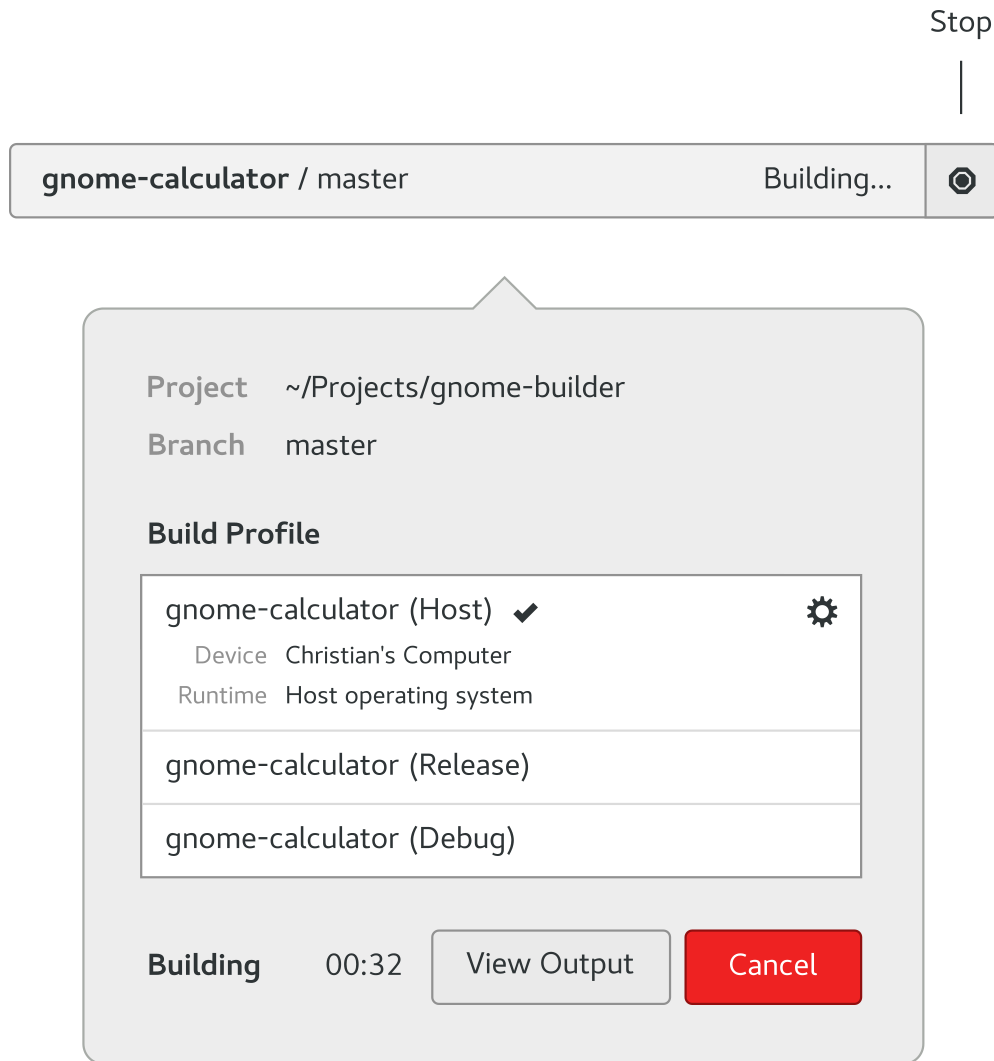
Build Profile

gnome-calculator (Host) ✓ 
Device Christian's Computer
Runtime Host operating system
gnome-calculator (Release)
gnome-calculator (Debug)

Last build Failed [View Output](#)
 Friday June 3rd, 11:49

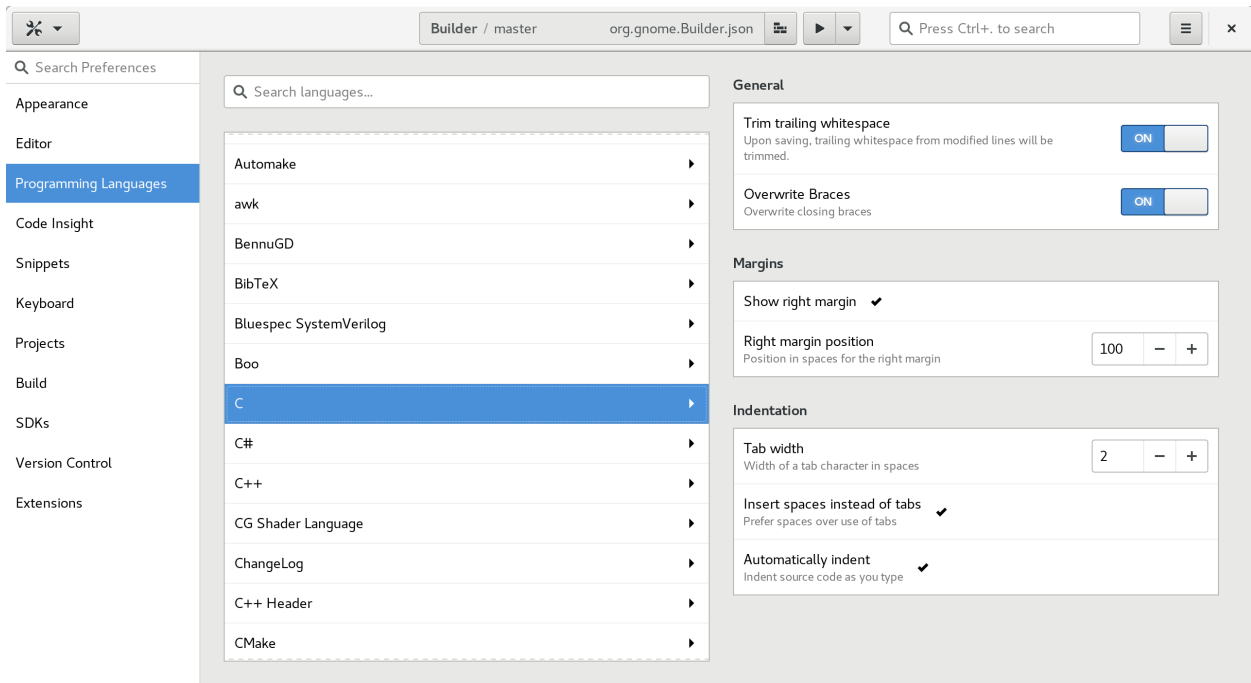
You can also use the “build”, “rebuild”, or “clean” commands from the command bar.

While the project is building, the build button will change to a cancel button. Clicking the cancel button will abort the current build.



Preferences

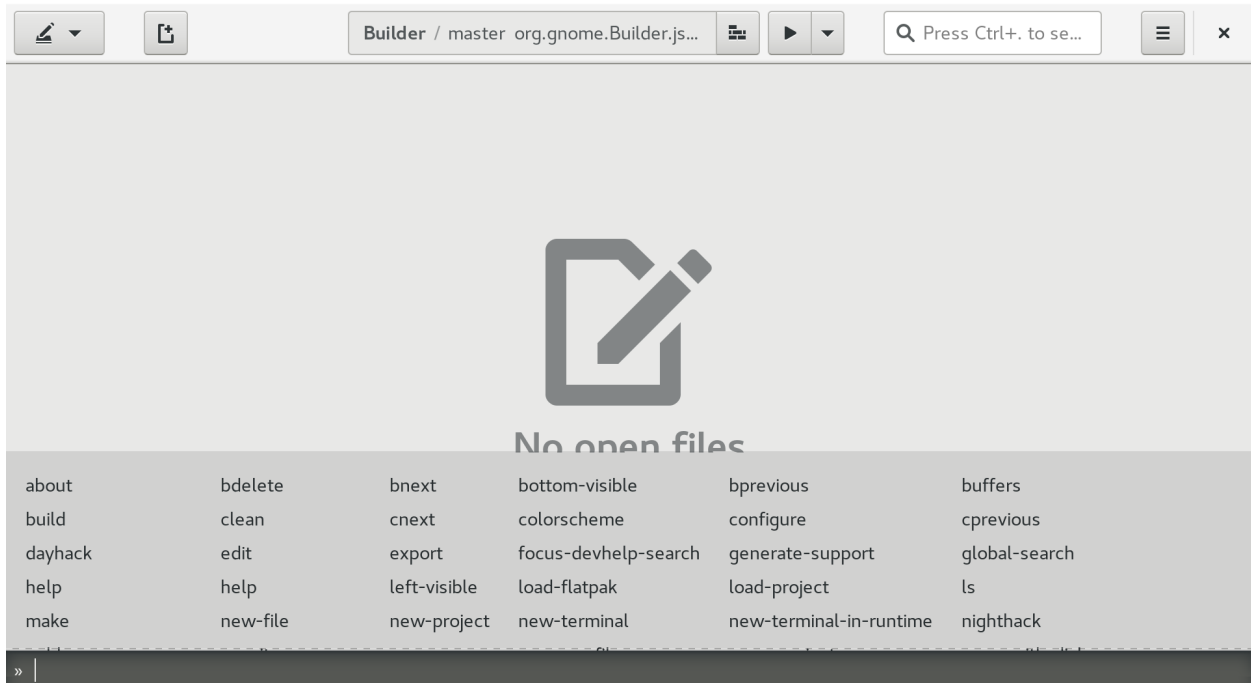
The preferences perspective allows you to change settings for Builder and its plugins. You can search for preferences using the keyword search in the top left of the preferences perspective.



Command Bar

The command bar provides a command-line-interface into Builder. You can type various actions to activate them. If Vim-mode is enabled, you can also activate some Vim-inspired commands here.

The command bar includes tab completion as shown below.



Projects

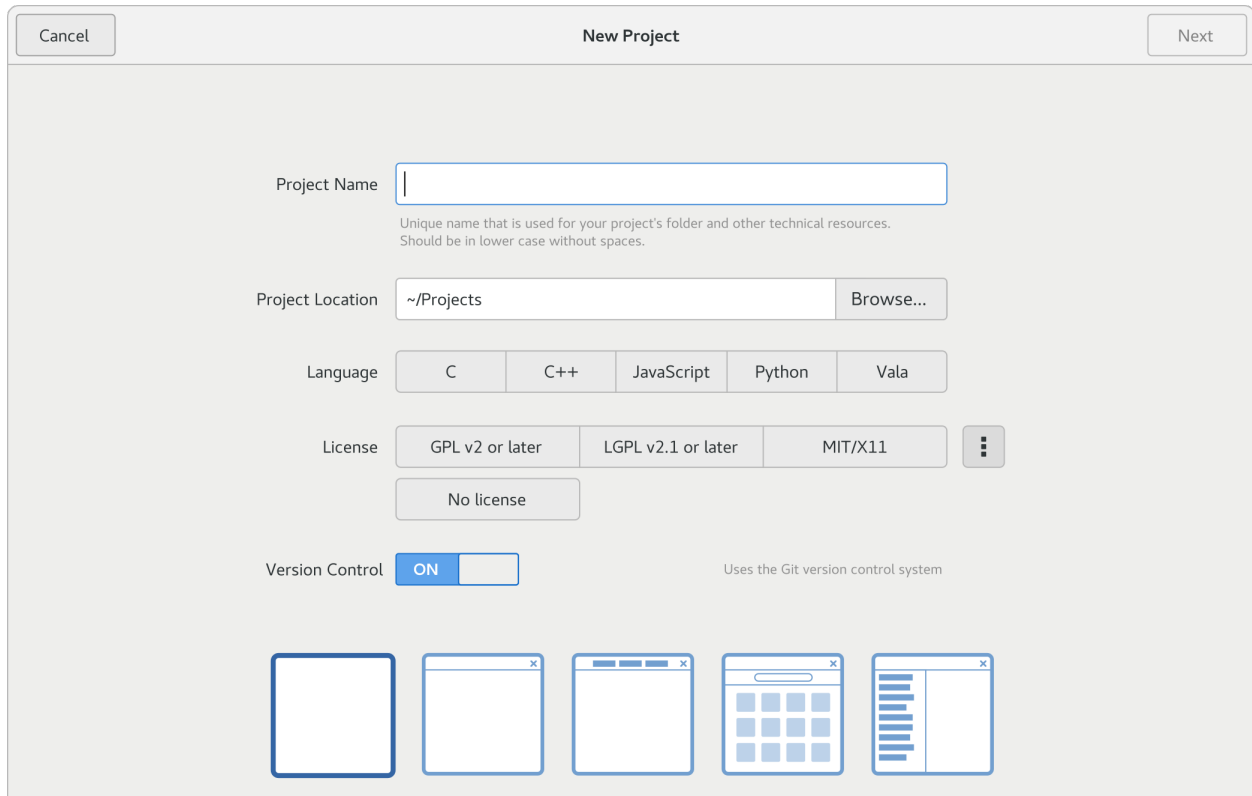
Whether you want to create a new project or import an existing project, Builder has features that can help you. Take a look below to learn how to create or import a project, build, and profile your application. As we add new features you can learn about them here.

Creating and Importing Projects

Builder supports creating new projects and importing existing projects. When importing a project, you can either open it from your local computer or clone it from a remote git repository.

Creating a new Project

To create a new project, select “New” from the project greeter. The project creation guide will be displayed.



Give your project a meaningful name, as this is not easily changeable later. The project name should not include spaces and if the project needs multiple words, use a hyphen “-” to separate the words.

Choose the language you would like to use for the project. Depending on the language, different templates are available.

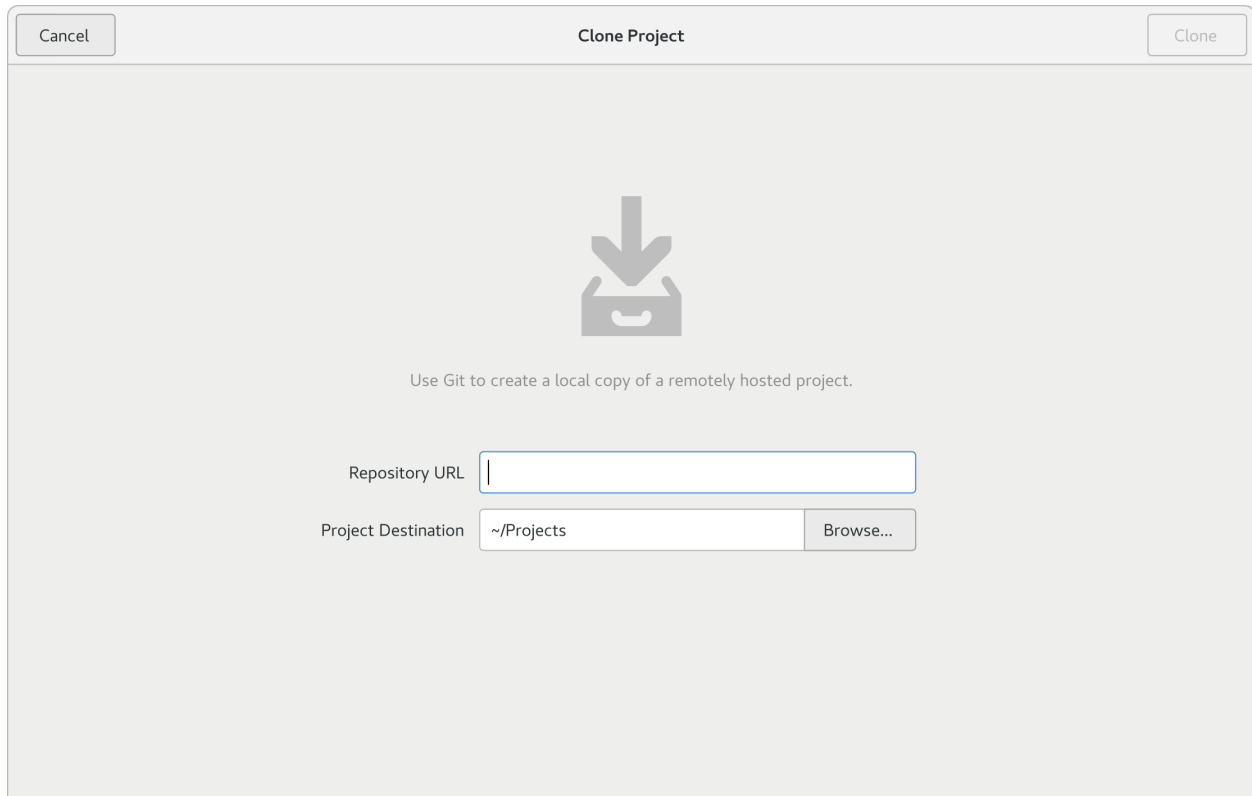
Choosing a license helps promote sharing of your application. Builder is licensed as **GPLv3 or newer** and we suggest using **GPLv3** when writing new applications for GNOME.

If you do not want git-based version control, turn off the switch to disable git support.

Lastly, select a suitable template for your application. Some patterns are available to speed up the bootstrapping of your project.

Cloning an Existing Project

To clone an existing project, you will need the URL of your **git repository**. For example, to clone the Builder project, you could specify: `git://git.gnome.org/gnome-builder.git`



After entering the URL, press the “Clone” button in the upper-right corner of the window and wait for the operation to complete. Once the project has been cloned, you will be shown the workbench window.

Note: If the remote repository requires authorization a dialog will be displayed for you to input your credentials.

Building your Project

There are multiple ways to activate a build for your project:


- Press the **Build Button** on the right of the **OmniBar** as shown in the figure below
- Press the **Control** and **F7** keys together
- Activate the command bar at the bottom of the Builder window by pressing **Control** and **Enter** and typing “build” followed by the **Enter** key
- Click the **OmniBar** and press the **Build** button in the lower-left corner of the dialog window

Build

gnome-calculator / master  Build failed 

Project ~/Projects/gnome-builder
Branch master

Build Profile

gnome-calculator (Host) ✓ 
Device Christian's Computer
Runtime Host operating system
gnome-calculator (Release)
gnome-calculator (Debug)

Last build Failed [View Output](#)
Friday June 3rd, 11:49

Debugging your Project

Warning: Builder does not currently support debugging, but it is expected in version 3.26.

Profiling your Project

Builder integrates with the Sysprof profiler to provide a robust sampling profiler.

Sharing your Project

Plugins

The following provides examples of various ways you can extend Builder. All examples are provided in the Python 3 language for succinctness. You can also implement plugins in C or Vala.

Creating Your First Plugin

Plugins consist of two things. First, a meta-data file describing the plugin which includes things like a name, the author, and where to find the plugin. Second, the plugin code which can take the form of a shared library or python module.

Builder supports writing plugins in C, C++, Vala, or Python. We will be using Python for our examples in this tutorial because it is both succinct and easy to get started with.

First, we will look at our plugin meta-data file. The file should have the file-suffix of ".plugin" and it's format is familiar. It starts with a line containing "[Plugin]" indicating this is plugin meta-data. Then it is followed by a series of "Key=Value" key-pairs.

```
# my_plugin.plugin
[Plugin]
Name=My Plugin
Loader=python3
Module=my_plugin
Author=Angela Avery
```

Now we can create a simple plugin that will print "hello" when Builder starts and "goodbye" when Builder exits.

```
# my_plugin.py

import gi

from gi.repository import GObject
from gi.repository import Ide

class MyAppAddin(GObject.Object, Ide.ApplicationAddin):

    def do_load(self, application):
        print("hello")

    def do_unload(self, application):
        print("goodbye")
```

In the python file above, we define a new extension called `MyAppAddin`. It inherits from `GObject.Object` (which is our base object) and implements the interface `Ide.ApplicationAddin`. We won't get too much into objects and interfaces here, but the plugin manager uses this information to determine when and how to load our extension.

The `Ide.ApplicationAddin` requires that two methods are implemented. The first is called `do_load` and is executed when the extension should load. And the second is called `do_unload` and is executed when the plugin should cleanup after itself. Each of the two functions take a parameter called `application` which is an `Ide.Application` instance.

Loading our Plugin

Now place the two files in `~/.local/share/gnome-builder/plugins` as `my_plugin.plugin` and `my_plugin.py`. If we run Builder from the command line, we should see the output from our plugin!

```
[angela@localhost ~] gnome-builder
hello
```

Now if we close the window, we should see that our plugin was unloaded.

```
[angela@localhost ~] gnome-builder
hello
goodbye
```

Next, continue on to learn about other interfaces you can implement in Builder to extend it's features!

Extending the Workbench

The Basics

The basic mechanics of extending the workbench requires first creating an `Ide.WorkbenchAddin`. Your subclass will be created for each instance of the `Ide.Workbench`. This conveniently allows you to track the state needed for your plugin for each workbench.

Listing 1.1: A Basic `WorkbenchAddin` to demonstrate scaffolding

```
import gi

from gi.repository import GObject
from gi.repository import Ide

class BasicWorkbenchAddin(GObject.Object, Ide.WorkbenchAddin):

    def do_load(self, workbench: Ide.Workbench):
        pass

    def do_unload(self, workbench: Ide.Workbench):
        pass
```

You will notice that at the top we import the packages we'll be using. Here we use the `GObject` and `Ide` packages from `GObject Introspection`.

We then create a class which inherits from `GObject.Object` and implements the `Ide.WorkbenchAddin` interface. The `Ide.WorkbenchAddin` interface has two virtual methods to override, `Ide.WorkbenchAddin.load()` and `Ide.WorkbenchAddin.unload()`.

Note: `PyGObject` uses `do_` prefix to indicate we are overriding a virtual method.

The `load` virtual method is called to allow the plugin to initialize itself. This method is called when the workbench is setup or your plugin is loaded.

When the `unload` virtual method is called the plugin should clean up after itself to leave Builder and the workbench in a consistent state. This method is called when the workbench is destroyed or your plugin is unloaded.

Registering Workbench Actions

Using `Gio.Action` is a convenient way to attach actions to the workbench that contain state. For example, maybe for use by a button that should be insensitive when it cannot be used. Additionally, actions registered on the workbench can be activated using the command bar plugin.

Listing 1.2: Registering an action on the workbench

```
import gi

from gi.repository import GObject
from gi.repository import Gio
from gi.repository import Ide

class MyWorkbenchAddin(GObject.Object, Ide.WorkbenchAddin):

    def do_load(self, workbench):
        action = Gio.SimpleAction.new('hello', None)
        action.connect('activate', self.hello_activate)
        workbench.add_action(action)

    def do_unload(self, workbench):
        workbench.remove_action('hello')

    def hello_activate(self, action, param):
        print('Hello activated!')
```

This adds a new action named `hello` to the workbench. It can be connected to widgets by using the `win.hello` action-name. Additionally, you can call the action with `hello` from the command bar.

To toggle whether or not the action can be activated, set the `Gio.SimpleAction:enabled` property.

Adding Widgets to the Header Bar

You might want to add a button to the workbench header bar. To do this, use an `Ide.WorkbenchAddin` and fetch the header bar using `Ide.Workbench.get_headerbar()`. You can attach your widget to either the left or the right side of the `Ide.OmniBar` in the center of the header bar. Additionally, by specifying a `Gtk.PackType`, you can align the button within the left or right of the header bar.

We suggest using `Gio.SimpleAction` to attach an action to the workbench and then activating the action using the `Gtk.Button:action-name` property.

Listing 1.3: Adding a button to the workbench header bar

```
import gi

from gi.repository import GObject
from gi.repository import Ide

class MyWorkbenchAddin(GObject.Object, Ide.WorkbenchAddin):

    def do_load(self, workbench):
        headerbar = workbench.get_headerbar()

        # Add button to top-center-left
        self.button = Gtk.Button(label='Click', action_name='win.hello', visible=True)
        headerbar.insert_left(self.button, Gtk.PackType.PACK_END, 0)

    def do_unload(self, workbench):
        # remove the button we added
        self.button.destroy()
        self.button = None
```

Adding Widgets to the Workbench

```
# my_plugin.py

import gi

from gi.repository import GObject
from gi.repository import Ide

class MyWorkbenchAddin(GObject.Object, Ide.WorkbenchAddin):

    def do_load(self, workbench):
        pass

    def do_unload(self, workbench):
        pass
```

Registering Perspectives

```
# my_plugin.py

import gi

from gi.repository import GObject
from gi.repository import Ide

class MyWorkbenchAddin(GObject.Object, Ide.WorkbenchAddin):

    def do_load(self, workbench):
        pass

    def do_unload(self, workbench):
        pass
```

Registering Panels

```
# my_plugin.py

import gi

from gi.repository import GObject
from gi.repository import Ide

class MyWorkbenchAddin(GObject.Object, Ide.WorkbenchAddin):

    def do_load(self, workbench):
        pass

    def do_unload(self, workbench):
        pass
```

Extending the Greeter

Extending the Editor

Managing Buffers

Syntax Highlighting

Regex-based Highlighting

Syntax highlighting in Builder is performed by the GtkSourceView project. By providing an XML description of the syntax, GtkSourceView can automatically highlight the language of your choice. Thankfully, GtkSourceView already supports a large number of languages so the chances you need to add a new language is low. However, if you do, we suggest that you work with GtkSourceView to ensure that all applications, such as Gedit, benefit from your work.

Chances are you can find existing language syntax files on your system in `/usr/share/gtksourceview-3.0/language-specs/`. These language-spec files serve as a great example of how to make your own. If it is not there, chances are there is already a `.lang` file created but it has not yet been merged upstream.

Bundling Language Specs

Should you need to bundle your own language-spec, consider using `GResources` to embed the language-spec within your plugin. Then append the directory path of your language-specs to the `GtkSource.LanguageManager` so it knows where to locate them.

```
from gi.repository import GtkSource

manager = GtkSource.LanguageManager.get_default()
paths = manager.get_search_path()
paths.append('resources:///org/gnome/builder/plugins/my-plugin/language-specs/')
manager.set_search_path(paths)
```

Symantic Highlighting

If the language you are using provides an AST you may want to highlight additional information not easily discernable by a regex-based highlighter. To simplify this, Builder provides the `Ide.HighlightEngine` and `Ide.Highlighter` abstractions.

The `Ide.HighlightEngine` provides background updating of the document so that your `Ide.Highlighter` implementation can focus on highlighting without dealing with performance impacts.

Out of simplicity, most `Ide.Highlighter` implementations in Builder today use a simple word index and highlight based on the word. However, this is not required if you prefer to do something more technical such as matching ranges to the AST.

Diagnostics and Fix-Its

Autocompletion

Snippets

File Settings and Indentation

Symbols and Semantic Analysis

Go To Definition

Extending the Symbol Tree

Renaming Symbols

Extending the Build Pipeline

Implementing a Build System

Builder has support for many build systems such as autotools, meson, cmake, etc. The build system knows how to find build targets (binaries or scripts that are installed) for the runner, knows how to find build flags used by the clang service, and it can define where the build directory is. It also has an associated `Ide.BuildPipelineAddin` (see the next section) that specifies how to do operations like build, rebuild, clean, etc.

Listing 1.4: An outline for a Buildsystem

```
import gi

from gi.repository import Gio, Ide

class BasicBuildSystem(Ide.Object, Ide.BuildSystem, Gio.AsyncInitable):

    def do_init_async(self, priority, cancel, callback, data=None):
        task = Gio.Task.new(self, cancel, callback)
        task.set_priority(priority)
        # do something, like check if a build file exists
        task.return_boolean(True)

    def do_init_finish(self, result):
        return result.propagate_boolean()

    def do_get_priority(self):
        return 0 # Choose a priority based on other build systems' priority

    def do_get_build_flags_async(self, ifile, cancellable, callback, data=None):
        task = Gio.Task.new(self, cancellable, callback)
        task.ifile = ifile
        task.build_flags = []
        # get the build flags
        task.return_boolean(True)

    def do_get_build_flags_finish(self, result):
        if result.propagate_boolean():
            return result.build_flags
```



```

def do_get_build_targets_async(self, cancellable, callback, data=None):
    task = Gio.Task.new(self, cancellable, callback)
    task.build_targets = []
    # get the build targets
    task.return_boolean(True)

def do_get_build_targets_finish(self, result):
    if result.propagate_boolean():
        return result.build_targets

```

How does Builder know which build system to use for a project? Each has an associated “project file” (configure.ac for autotools) that has to exist in the source directory for the build system to be used. If a project has multiple project files, the priorities of each are used to decide which to use. You can see where the priority is defined in the code above. The project file is defined in the `.plugin` file with these lines (in the case of the make plugin):

Listing 1.5: A snippet from a `.plugin` file

```

X-Project-File-Filter-Pattern=Makefile
X-Project-File-Filter-Name=Makefile Project

```

When a project has the right file, the build system will be initialized by `IdeContext` during its own initialization process.

Extending the Build Pipeline

Processes and Containers

Builder needs to support a wide variety of ways to spawn processes. Whether that is inside a build environment, container, on the host, terminals, or even a remote system. The following sections describe some of the scenarios and how to best perform the task at hand.

Application Runtimes and Containers

A core abstraction in the design of builder is `Ide.Runtime`. This provides a way to setup and execute processes within a given environment. That environment could be your host operating system, a container, a build environment, or even a remote system.

For example, if we want to run the `make` command for your project and that project is targeting the GNOME Sdk we need to first enter the SDK environment. The `Flatpak` plugin provides an `Ide.Runtime` implementation to do this so that before your subprocess is lunched, the runtime is setup and initialized for execution with an alternate mount namespace, network namespace, and more.

How to get a runtime

If you need to run a process within the build environment you will want to access the runtime for the current build configuration. The current build configuration can be accessed from the `Ide.ConfigurationManager` object.

```

config_manager = context.get_configuration_manager()
config = config_manager.get_current()
runtime = config.get_runtime()

```

Note: It is possible that the configured runtime does not yet exist, so remember to check for None.

Creating a Subprocess

To create a subprocess in the runtime, use the `Ide.Subprocess.create_launcher()` method and then spawn a process using that launcher.

```
try:
    launcher = runtime.create_launcher()
    launcher.push_argv('which')
    launcher.push_argv('make')
    subprocess = launcher.spawn(None)
    _, stdout, stderr = subprocess.communicate_utf8(None, None)
except Exception as ex:
    print("Failed to create launcher: " + repr(ex))
    return
```

Subprocesses and Psuedo Terminals

Creating Subprocesses

Builder provides a powerful abstraction for creating subprocesses. `Ide.Subprocess` allows you to setup and modify how a processes should be launched without the burden of how to launch the subprocess. This means that Builder can use different strategies based on the host system, subprocess requirements, and plugins that may need to modify the program arguments.

When Builder is not running in a sandbox, it can generally execute subprocesses the normal way using fork and exec. However, if Builder is sandboxed, it may need to run the subprocess on the host rather than inside the sandbox. To ensure your subprocess is run on the host, use `Ide.Subprocess.set_run_on_host()`.

Note: You can only run programs on the host that are already installed. Use `which program-name` to determine if the process is available on the host.

If you are integrating an external tool, such as Valgrind, you might need to inject arguments into the argument array. For the Valgrind case, `Ide.Subprocess.prepend_argv()` of "valgrind" would be appropriate.

Some runtime plugins may need to modify the argument array even further. For example, the Flatpak plugin will require that all commands start with `flatpak build ...` so that the commands are never run on the host system, but instead inside the runtime. Plugins that require this should inject their additional arguments from the `Ide.SubprocessLauncher.spawn()` virtual-method so that plugins do not get confused about the placement of arguments.

```
from gi.repository import GLib
from gi.repository import Ide

# You may want access to stdin/stdout/stderr. If so, ensure you specify
# the appropriate Gio.SubprocessFlags for your subprocess.
launcher = Ide.SubprocessLauncher.new(Gio.SubprocessFlags.STDOUT_PIPE |
                                       Gio.SubprocessFlags.STDERR_PIPE |
                                       Gio.SubprocessFlags.STDIN_PIPE)
```

```

# If you need to specify where to launch the process. The default
# is the home directory.
launcher.set_cwd(os.path.join(GLib.get_home_dir(), 'Projects'))

# Push some arguments onto argv
launcher.push_argv('which')
launcher.push_argv('ls')

# Set some environment variables
launcher.setenv('LANG', 'C', True)

# Spawn the process. If you pass in a Gio.Cancellable, you can kill the
# subprocess by calling Gio.Cancellable.cancel().
subprocess = launcher.spawn(None)

# We need to wait for the child to complete. If you want to read the
# output of the subprocess, see Ide.Subprocess.communicate_utf8().
# wait_check() will ensure the return value is zero. If you do not
# care about the return value, just use wait().
try:
    subprocess.wait_check(None)
except Exception as ex:
    print(repr(ex))

# May Ide.Subprocess API have async variants. Consider using them to
# avoid needlessly blocking threads.

```

Supervising Subprocesses

There are times where you might want to respawn a process in case it exits prematurely. Builder provides the `Ide.SubprocessSupervisor` abstraction to simplify this for you.

The `Ide.SubprocessSupervisor` has a simple API. Just attach your `Ide.SubprocessLauncher` using `Ide.SubprocessSupervisor.set_launcher()` and call `Ide.SubprocessSupervisor.start()`.

If the subprocess begins flapping (exiting immediately after spawning) some delay will be added to slow things down.

To stop the subprocess, use `Ide.SubprocessSupervisor.stop()`.

If you need access to the subprocess, you can access it either via the `Ide.SubprocessSupervisor.get_subprocess()` method or by connecting to the `Ide.SubprocessSupervisor::spawned()` signal.

```

def on_subprocess_spawned(supervisor, subprocess):
    print("Spawned process " + subprocess.get_identifier())

launcher = create_launcher()

supervisor = Ide.SubprocessSupervisor()
supervisor.set_launcher(launcher)
supervisor.connect('spawned', on_subprocess_spawned)
supervisor.start()

```

Pseudo Terminals

Pseudo terminals are tricky business. In general, if you need access to a PTY, use the VTE library like Builder's terminal plugin. For an example of how to setup the PTY, we use a flow like this.

```
// This code does little to no error checking.
// Your code should be more careful.

// First create our PTY master
VtePty *pty = vte_terminal_pty_new_sync (terminal,
                                         VTE_PTY_DEFAULT | VTE_PTY_NO_LASTLOG | VTE_
↳PTY_NO_UTMP | VTE_PTY_NO_WTMP,
                                         NULL, &error);

// Now go through the PTY slave setup
int master_fd = vte_pty_get_fd (pty);

assert (grantpt (master_fd) != 0);
assert (unlockpt (master_fd) != 0);

// Get the path to the PTY slave
char name[PATH_MAX];
assert (ptsname_r (master_fd, name, sizeof name - 1) != 0);
name [sizeof name - 1] = '\0';

// Open the PTY slave
int slave_fd = open (name, O_RDWR | O_CLOEXEC);

// Now, when spawning a process, you can set stdin/stdout/stderr to the FD
// of the slave. We use dup() because the callee takes ownership.
ide_subprocess_launcher_take_stdin_fd (launcher, dup (slave_fd));
ide_subprocess_launcher_take_stdout_fd (launcher, dup (slave_fd));
ide_subprocess_launcher_take_stderr_fd (launcher, dup (slave_fd));
close (slave_fd);
```

When launching the subprocess with Builder, it will detect that stdin, stdout, or stderr are pseudo terminals and perform the proper `ioctl()` setup for you. This allows for the PTY to cross the sandbox boundary to the host, ensuring that you may have a host-based shell with a PTY from within the sandbox.

Extending the Device Manager

Extending the Run Manager

Registering Keybindings

Integrating Language Servers

Extending Project Search

Extending Application Menus

Registering Application Preferences

Creating and Performing Transfers

Managing Worker Processes

Integrating Version Control

How-To Guides

These quick and to the point “How To” articles are meant to help you answer common questions. If you have an addition to the list, please let us know!

Contents

Changing Indentation

To change the indentation rules you have two options. Either globally for your system, or for just the current project.

Project-Wide

If you would like to change settings for just your project, use a `.editorconfig` file. You can add a `.editorconfig` file to the root of your project in the `editorconfig` format.

It looks something like:

```
root = true

[*]
charset = utf-8
end_of_line = lf
```

Globally

If you would like to change the indentation rules for your user, and thereby all projects which do not contain an `.editorconfig` file, use the application preferences. You can access the preferences through the perspective selector or using the `Command+`, keyboard shortcut.

First select “Programming Languages” from the sidebar on the left. Then select the programming language from the list of options. On the right you will now see a list of preferences that may be tweaked for that language. Change the indentation level to your desired preference.

Search and Replace

Search and replace can be used to replace all instances of a keyword with another form of text. To bring up the “Search and Replace” tool, use `Ctrl+h` while focused in the editor.

Note: If you are using an alternate keyboard shortcut theme, your shortcut might be different.

Enter the search text in the first text entry. Enter the replacement text in the second text entry.

Tip: The replacement text may contain “regex backreferences” such as `\1` and others. See the [g_regex_replace](#) documentation for more information.

Select “Replace” to replace the next match or “Replace All” to replace all matches.

Troubleshooting

If you are having trouble with Builder you can help us help you by trying to do some basic troubleshooting. Here are some steps you can go through to try to discover what is going wrong.

Verbose Output

You can increase the log verbosity of Builder by adding up to four `-v` when launching from the command line.

```
# If running from flatpak
flatpak run org.gnome.Builder -vvvv

# If using distribution packages
gnome-builder -vvvv
```

Support Log

Builder has support to generate a support log which can provide us with details. From the application menu, select “Generate Support Log”. It will place a log file in your home directory.

Counters

Builder has internal counters which can be useful to debug problems. Use the command bar (activated by `Control+Enter`) and type “counters” followed by `Enter`. This will bring up a new window containing the current values of the counters.

If Builder has locked up, you can access the counters from outside of Builder. The command line tool `dazzle-list-counters`, can be used to access the counters.

```
dazzle-list-counters `pidof gnome-builder`
```

Note: When running Builder from Flatpak, we do not currently expose the counters to the host. Use flatpak enter \$PID /bin/bash to enter the mount namespace and then run dazzle-list-counters.

Test Builder Nightly

If you are running the stable branch or an older distribution package, please consider trying our Nightly release to see if the bug has already been fixed. Doing this before reporting bugs helps reduce the amount of bug traffic we need to look at. We'll usually ask you to try Nightly anyway before continuing the troubleshooting process.

See *installing from Flatpak* for installation notes.

File a Bug

We can help you troubleshoot! File a bug if you're stuck and we can help you help us.

See the [Builder Bugzilla](#) for creating a bug report.

Contributing

If you're interested in contributing to Builder and GNOME at large, we would love for you to join us! Only with people like you can GNOME exist. We love seeing people that use GNOME transform into people that create GNOME.

Planning and Project Management

Many of us that work on the Builder code-base are great at writing code. But we are not so great at managing schedules, planning feature priorities, and coordinating with other projects. Helping us do this in Builder will make you a shepard of geeks.

Responsibilities

- Helping us stay on schedule with GNOME releases and releasing tarballs on time.
- Ensure that we plan the feature before writing it.
- Helping to plan features on realistic timelines.
- Planning a roadmap that makes sense. Some features do not matter unless other features are implemented first.
- Help us find new contributors and ensure that our community is healthy towards new recruits.

Writing Documentation

We are using sphinx to write our new documentation.

In `conf.py` you'll see that we use the theme from readthedocs.io. That means you need to install that theme as well as sphinx to build the documentnation.

Listing 1.6: Install dependencies for building documentation (Fedora 25)

```
sudo dnf install python3-sphinx python3-sphinx_rtd_theme
```

Listing 1.7: Now build the documentation with sphinx

```
[user@host gnome-builder/]$ cd doc
[user@host doc/]$ sphinx-build . _build
[user@host doc/]$ xdg-open _build/index.html
```

The first command builds the documentation. Pay attention to warnings which will be shown in red. Some of them may be useful to help you track down issues quickly.

To open the documentation with your web browser, use `xdg-open _build/index.html`.

Submitting Patches

We will accept patches for documentation no matter how you get them to us. However, you will save us a lot of time if you can:

- Create a patch with git.
- Create a new bug on [Builder Bugzilla](#) and attach the patch.

Creating a Patch

First off, if you have not configured git to include your full name and email, type the following in a terminal:

```
$ git config --global user.name 'My Full Name'
$ git config --global user.email 'example@example.com'
```

After you have modified the documentation to your liking, prepare the files to be committed to git. The add a short commit message and commit the files.

```
[user@host doc/]$ git add path/to/file.rst
[user@host doc/]$ git commit -m 'doc: update example documentation'
```

Now we can export the patch to be uploaded to [Builder Bugzilla](#)

```
[user@host doc/]$ git format-patch HEAD^
```

At this point you'll see a file similar to `0001-doc-update-example-documentation.patch` in the current directory. We want to upload this patch to [Builder Bugzilla](#).

Submitting a Patch

Now that we have our patch file, we need to create a new bug. Head over to [Builder Bugzilla](#) and fill out the bug details.

Just give a bit of information about what you documented, and then find the “Add an Attachment” section near the bottom. Upload the patch file you exported with `git format-patch HEAD^` above.

Click “Submit Bug” and we'll take care of the rest!

GNOME git Best Practices

To learn more about using git with GNOME, including how to set up git, submitting patches, and good commit messages, visit the [git workflow](#) GNOME wiki page.

Contributing Code

Where to Contribute?

Builder wants to become a powerful tool to enable GNOME developers to build great software. To do this we need your help.

Do you have knowledge in a particular area of software development? You can use that knowledge to help Builder expand it's area of expertise.

Generally, code contributions fall into one of two categories: *Application Plumbing* or *Plugins*.

Application Plumbing

If you like working on application plumbing, which is the infrastructure that makes implementing plugins simple, then you want to look at [libide](#). Many of the core features of Builder are implemented here. That includes the application window, plugin interfaces and core machinery of Builder.

Plugins

Plugins are how we integrate features into Builder for a specific problem. For example, the [git](#) plugin is the glue between the version control abstraction in [libide](#) and [git](#).

There are many [existing plugins](#) already. You might want to contribute to an existing one that does not yet serve your needs well. Or maybe you want to *create a new plugin* that integrates a feature missing from Builder.

IRC

One great way to get started is to join us on IRC where you can chat with others who work on the Builder project. The Builder developer team can be found on the Builder IRC channel.

File A Bug

If you think you've found a bug in Builder, head over to [Builder Bug Tracker](#) and file a report. We will get back to you on any further details we need to track down the issue as soon as we can. Filing bugs helps us improve the software for everyone and we really appreciate your time.

For various spam reasons our bugtracker requires that you create an account. We hope this minor inconvenience is not too much trouble.

Find A Bug To Work On

If you wish to start contributing code to Builder, simply pick a bug from this list of [newcomer bugs](#).

Building From Source

Learn out how to install *via JHBuild* in our installation documentation.

Currently, JHBuild is how we recommend contributing to Builder. We do expect this to change very soon in that you'll be able to easily contribute to Builder from our flatpak-version of Builder.

Credits

Builder was started by Christian Hergert in 2014 to improve the quality of software for the GNOME ecosystem. Many contributors have helped create and improve Builder to the state you find it today.

Artwork by

Allan Day Hylke Bons Jakub Steiner
--

Code and documentation by

Akshaya Kakkilaya albfan Alex285 Alexander Larsson Alexandre Franke Allan Day Andreas Brauchli Andreas Henriksson anoop chandu Anoop Chandu Antoine Jacoutot Anwar Sadath Ben Iofel Boris Egorov burningTyger Carlos Soriano chandu Christian Hergert Cosimo Cecchi Daiki Ueno Damien Lespiau Daniel Boles Daniel Espinosa David King Debarshi Dimitrios Christidis Dimitris Zenios Dor Askayo Ekta Nandwani Elad Alfassa Erick Pérez Castellanos Evgeny Shulgin Fabiano Fidêncio Fangwen Yu Felix Schwarz Fernando Fernandez Florian

Florian Bäuerle
Florian Müllner
Garrett Regier
Gautier Pelloux-Prayer
Gennady Kovalev
Georges Basile Stavracas Neto
Georg Vienna
Giovanni Campagna
Hashem Nasarat
heroin
Ignacio Casal Quinteiro
Igor Gnatenko
Jakub Steiner
Jasper St. Pierre
Joaquim Rocha
Johan Svensson
Jonathon Jongsma
Jürg Billeter
Kris Thomsen
kritarth
Krzyszimir Nowak
Lars Uebernickel
Lionel Landwerlin
Lucie Charvat
Lucie Dvorakova
Marcin Kolny
Marek Černocký
Marinus Schraal
Mario Sanchez Prada
Martin Blanchard
Mathieu Bridon
Mathieu Duponchelle
Matthew Leeds
Matthias Clasen
Megh Parikh
Michael Biebl
Michael Catanzaro
Mohammed Sadiq
Mohan R
namanyadav12
Paolo Borelli
Patrick Griffis
Peter Sonntag
Philip Chimento
Philip Withnall
Piotr Drag
Raunaq Abhyankar
Ray Strode
Roberto Majadas
Sebastien Lafargue
Sébastien Lafargue
Simon Schampijer
Sourav
Thibault Saunier
Timm Bäder
Ting-Wei Lan
Tobias Schönberg
Trinh Anh Ngoc

```
Umang Jain  
Wolf Vollprecht  
Yannick Inizan  
Yosef Or Boczko  
Zhang Cheng  
zilla@hmt.im
```