GUI development with Rust and GTK 4

by Julian Hofer, with contributions from the community

GTK 4 is the newest version of a popular cross-platform widget toolkit written in C. Thanks to GObject-Introspection, GTK's API can be easily targeted by various programming languages. The API even describes the ownership of its parameters!

Managing ownership without giving up speed is one of Rust's greatest strengths, which makes it an excellent choice to develop GTK apps with. With this combination you don't have to worry about hitting bottlenecks mid-project anymore. Additionally, with Rust you will have nice things such as

- thread safety,
- memory safety,
- sensible dependency management as well as
- excellent third party libraries.

The `gtk-rs` project provides bindings to many GTK-related libraries which we will be using throughout this book.

Who this book is for

This book assumes that you know your way around Rust code. If this is not already the case, reading The Rust Programming Language is an enjoyable way to get you to that stage. If you have experience with another low-level language such as C or C++ you might find that reading A half hour to learn Rust gives you sufficient information as well.

Luckily, this — together with the wish to develop graphical applications — is all that is necessary to benefit from this book.

How to use this book

In general, this book assumes that you are reading it in sequence from front to back. However, if you are using it as a reference for a certain topic, you might find it useful to just jump into it.

There are two kinds of chapters in this book: concept chapters and project chapters. In concept chapters, you will learn about an aspect of GTK development. In project chapters, we will build small programs together, applying what you've learned so far.

The book strives to explain essential GTK concepts paired with practical examples. However, if a concept can be better conveyed with a less practical example, we took this path most of the time. If you are interested in contained and useful examples, we refer you to the corresponding section of `gtk4-rs`' repository.

Every valid code snippet in the book is part of a listing. Like the examples, the listings be found in the repository of `gtk4-rs`.

License

Installation

In order to develop a `gtk-rs` app, you basically need two things on your workstation:

- the Rust toolchain, and
- the GTK 4 library.

As so often the devil hides in the details, which is why we will list the installation instructions for each operating system in the following chapters.

Linux

You first have to install rustup. You can find the up-to-date instructions on rustup.rs.

Then install GTK 4 and the build essentials. To do this, execute the command belonging to the distribution you are using.

Fedora and derivatives:

```
sudo dnf install gtk4-devel gcc
```

Debian and derivatives:

```
sudo apt install libgtk-4-dev build-essential
```

Arch and derivatives:

```
sudo pacman -S gtk4 base-devel
```

macOS

First install rustup. You can find the up-to-date instructions on rustup.rs.

Then install homebrew.

Finally, install GTK 4 by executing the following in your terminal:

```
brew install gtk4
```

Windows

When preparing your Windows machine, you have to decide between either using the **MSVC toolchain** or the **GNU toolchain**. If in doubt, go for MSVC since that is the default on Windows. You will want to go for the GNU toolchain if you depend on libraries that can only be compiled with the GNU toolchain.

Install Rustup

Install the Rust toolchain via rustup.

Install GTK 4

▸ Build GTK 4 with gvsbuild and MSVC (recommended)
▸ Build GTK 4 manually with MSVC
▸ Install GTK 4 with MSYS2 and the GNU toolchain

Project Setup

Let's begin by installing all necessary tools. First, follow the instructions on the GTK website in order to install GTK 4. Then install Rust with rustup.

Now, create a new project and move into the newly created folder by executing:

```
cargo new my-gtk-app
cd my-gtk-app
```

Find out the GTK 4 version on your machine by running

```
pkg-config --modversion gtk4
```

Use this information to add the gtk4 crate to your dependencies in `Cargo.toml`. At the time of this writing the newest version is `4.12`.

```
cargo add gtk4 --rename gtk --features v4_12
```

By specifying this feature you opt-in to API that was added with minor releases of GTK 4.

Now, you can run your application by executing:

```
cargo run
```

Hello World!

Now that we've got a working installation, let's get right into it!

At the very least, we need to create a `gtk::Application` instance with an application id. For that we use the builder pattern which many `gtk-rs` objects support. Note that we also import the prelude to bring the necessary traits into scope.

Filename: listings/hello_world/1/main.rs

```
use gtk::prelude::*;
use gtk::{glib, Application};

const APP_ID: &str = "org.gtk_rs.HelloWorld1";

fn main() -> glib::ExitCode {
    // Create a new application
    let app = Application::builder().application_id(APP_ID).build();

    // Run the application
    app.run()
}
```

It builds fine, but nothing but a warning in our terminal appears.

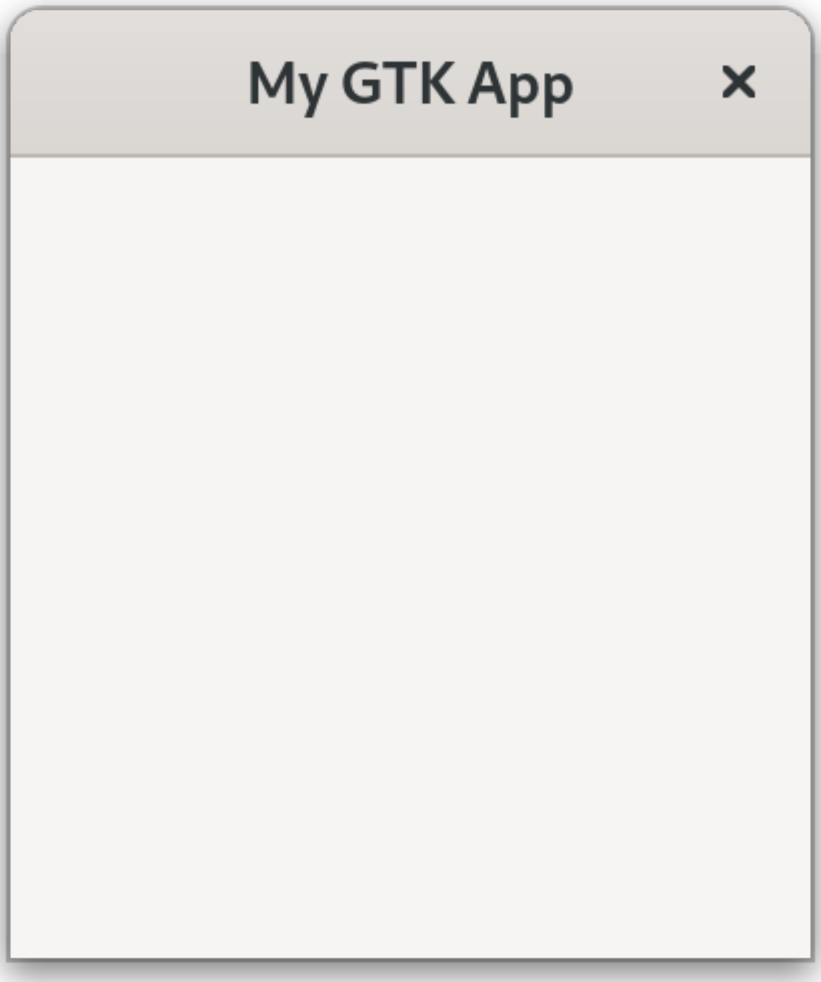
```
GLib-GIO-WARNING: Your application does not implement
g_application_activate()
and has no handlers connected to the 'activate' signal. It should do one of
these.
```

GTK tells us that something should be called in its `activate` step. So let's create a `gtk::ApplicationWindow` there.

Filename: listings/hello_world/2/main.rs

```rust
use gtk::prelude::*;
use gtk::{glib, Application, ApplicationWindow};

const APP_ID: &str = "org.gtk_rs.HelloWorld2";

fn main() -> glib::ExitCode {
    // Create a new application
    let app = Application::builder().application_id(APP_ID).build();

    // Connect to "activate" signal of `app`
    app.connect_activate(build_ui);

    // Run the application
    app.run()
}

fn build_ui(app: &Application) {
    // Create a window and set the title
    let window = ApplicationWindow::builder()
        .application(app)
        .title("My GTK App")
        .build();

    // Present window
    window.present();
}
```

That is better!



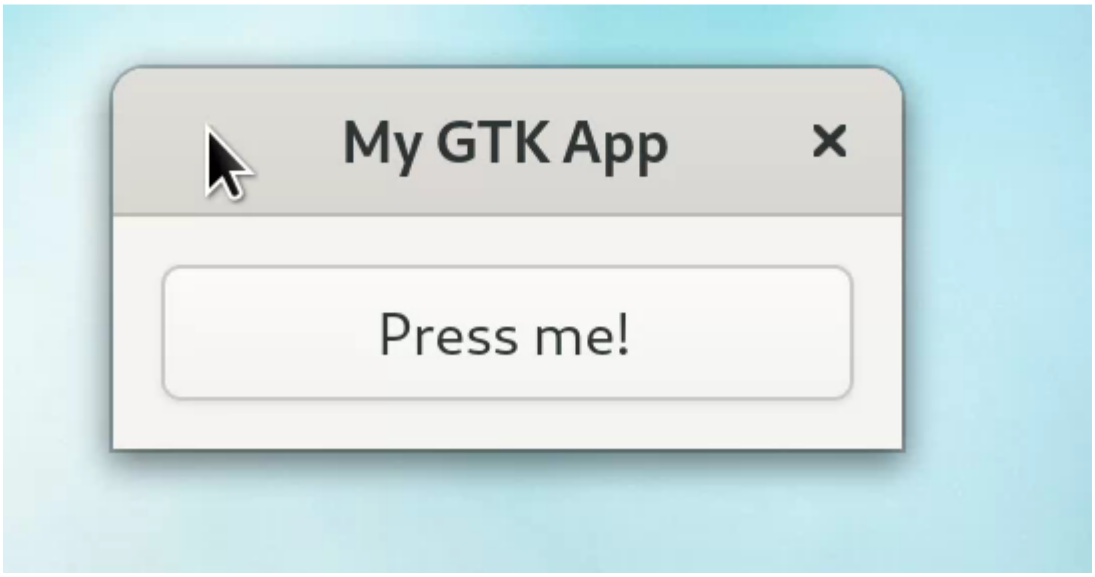
Normally we expect to be able to interact with the user interface. Also, the name of the chapter suggests that the phrase "Hello World!" will be involved.

Filename: listings/hello_world/3/main.rs

```
fn build_ui(app: &Application) {
    // Create a button with label and margins
    let button = Button::builder()
        .label("Press me!")
        .margin_top(12)
        .margin_bottom(12)
        .margin_start(12)
        .margin_end(12)
        .build();

    // Connect to "clicked" signal of `button`
    button.connect_clicked(|button| {
        // Set the label to "Hello World!" after the button has been clicked
on
        button.set_label("Hello World!");
    });

    // Create a window
    let window = ApplicationWindow::builder()
        .application(app)
        .title("My GTK App")
        .child(&button)
        .build();

    // Present window
    window.present();
}
```

If you look closely at the code snippet you will notice that it has a small eye symbol on its top right. After you press on it you can see the full code of the listing. We will use this throughout the book to hide details which are not important to bring the message across. Pay attention to this if you want to write apps by following the book step-by-step. Here, we've hidden that we brought `gtk::Button` into scope.

There is now a button and if we click on it, its label becomes "Hello World!".

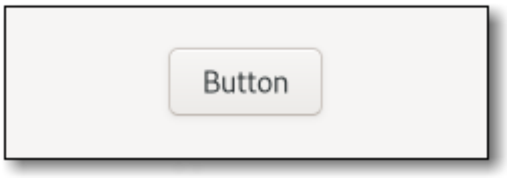


Wasn't that hard to create our first `gtk-rs` app, right? Let's now get a better understanding of what we did here.

Widgets

Widgets are the components that make up a GTK application. GTK offers many widgets and if those don't fit, you can even create custom ones. There are, for example, display widgets, buttons, containers and windows. One kind of widget might be able to contain other widgets, it might present information and it might react to interaction.

The Widget Gallery is useful to find out which widget fits your needs. Let's say we want to add a button to our app. We have quite a bit of choice here, but let's take the simplest one — a `Button`.



GTK is an object-oriented framework, so all widgets are part of an inheritance tree with `GObject` at the top. The inheritance tree of a `Button` looks like this:

```
GObject
└── Widget
    └── Button
```

The GTK documentation also tells us that `Button` implements the interfaces `GtkAccessible`, `GtkActionable`, `GtkBuildable`, `GtkConstraintTarget`.

Now let's compare that with the corresponding `Button` struct in `gtk-rs`. The gtk-rs documentation tells us which traits it implements. We find that these traits either have a corresponding base class or interface in the GTK docs. In the "Hello World" app we wanted to react to a button click. This behavior is specific to a button, so we expect to find a suitable method in the `ButtonExt` trait. And indeed, `ButtonExt` includes the method `connect_clicked`.

Filename: listings/hello_world/3/main.rs

```
    // Create a button with label and margins
    let button = Button::builder()
        .label("Press me!")
        .margin_top(12)
        .margin_bottom(12)
        .margin_start(12)
        .margin_end(12)
        .build();

    // Connect to "clicked" signal of `button`
    button.connect_clicked(|button| {
        // Set the label to "Hello World!" after the button has been clicked
on
        button.set_label("Hello World!");
    });
```

GObject Concepts

GTK is an object-oriented framework. It is written in C, which does not support object-orientation out of the box. That is why GTK relies on the GObject library to provide the object system.

We have already learned that `gtk-rs` maps GObject concepts, like inheritance and interfaces, to Rust traits. In this chapter we will learn:

- How memory of GObjects is managed
- How to create our own GObjects via subclassing
- How to deal with generic values
- How to use properties
- How to emit and receive signals

Memory Management

Memory management when writing a gtk-rs app can be a bit tricky. Let's have a look why that is the case and how to deal with that.

With our first example, we have window with a single button. Every button click should increment an integer `number` by one.

```rust
// DOES NOT COMPILE!
fn main() -> glib::ExitCode {
    // Create a new application
    let app = Application::builder().application_id(APP_ID).build();

    // Connect to "activate" signal of `app`
    app.connect_activate(build_ui);

    // Run the application
    app.run()
}

fn build_ui(application: &Application) {
    // Create two buttons
    let button_increase = Button::builder()
        .label("Increase")
        .margin_top(12)
        .margin_bottom(12)
        .margin_start(12)
        .margin_end(12)
        .build();

    // A mutable integer
    let mut number = 0;

    // Connect callbacks
    // When a button is clicked, `number` should be changed
    button_increase.connect_clicked(|_| number += 1);

    // Create a window
    let window = ApplicationWindow::builder()
        .application(application)
        .title("My GTK App")
        .child(&button_increase)
        .build();

    // Present the window
    window.present();
}
```

The Rust compiler refuses to compile this application while spitting out multiple error messages. Let's have a look at them one by one.

```
error[E0373]: closure may outlive the current function, but it borrows
`number`, which is owned by the current function
   |
32 |     button_increase.connect_clicked(|_| number += 1);
   |                                     ^^^ ------ `number` is borrowed here
   |                                     |
   |                                     may outlive borrowed value `number`
   |
note: function requires argument type to outlive `'static`
   |
32 |     button_increase.connect_clicked(|_| number += 1);
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
help: to force the closure to take ownership of `number` (and any other
referenced variables), use the `move` keyword
   |
32 |     button_increase.connect_clicked(move |_| number += 1);
   |
```

Our closure only borrows `number`. Signal handlers in GTK require `static'` lifetimes for their references, so we cannot borrow a variable that only lives for the scope of the function `build_ui`. The compiler also suggests how to fix this. By adding the `move` keyword in front of the closure, `number` will be moved into the closure.

```
    // DOES NOT COMPILE!
    // A mutable integer
    let mut number = 0;

    // Connect callbacks
    // When a button is clicked, `number` should be changed
    button_increase.connect_clicked(move |_| number += 1);
```

This still leaves the following error message:

```
error[E0594]: cannot assign to `number`, as it is a captured variable in a
`Fn` closure
   |
32 |     button_increase.connect_clicked(move |_| number += 1);
   |                                              ^^^^^^^^^^^ cannot assign
```

In order to understand that error message we have to understand the difference between the three closure traits `FnOnce`, `FnMut` and `Fn`. APIs that take closures implementing the `FnOnce` trait give the most freedom to the API consumer. The closure is called only once, so it can even consume its state. Signal handlers can be called multiple times, so they cannot accept `FnOnce`.

The more restrictive `FnMut` trait doesn't allow closures to consume their state, but they can still mutate it. Signal handlers can't allow this either, because they can be called from inside themselves. This would lead to multiple mutable references which the borrow checker doesn't appreciate at all.

This leaves `Fn`. State can be immutably borrowed, but then how can we modify `number`? We need a data type with interior mutability like `std::cell::Cell`.

> The `Cell` class is only suitable for objects that implement the `Copy` trait. For other objects, `RefCell` is the way to go. You can learn more about interior mutability in this section of the book *Rust Atomics and Locks*.

Filename: listings/g_object_memory_management/1/main.rs

```rust
fn build_ui(application: &Application) {
    // Create two buttons
    let button_increase = Button::builder()
        .label("Increase")
        .margin_top(12)
        .margin_bottom(12)
        .margin_start(12)
        .margin_end(12)
        .build();

    // A mutable integer
    let number = Cell::new(0);

    // Connect callbacks
    // When a button is clicked, `number` should be changed
    button_increase.connect_clicked(move |_| number.set(number.get() + 1));

    // Create a window
    let window = ApplicationWindow::builder()
        .application(application)
        .title("My GTK App")
        .child(&button_increase)
        .build();

    // Present the window
    window.present();
}
```

This now compiles as expected. Let's try a slightly more complicated example: two buttons which both modify the same `number`. For that, we need a way that both closures take ownership of the same value?

That is exactly what the `std::rc::Rc` type is there for. `Rc` counts the number of strong references created via `Clone::clone` and released via `Drop::drop`, and only deallocates the value when this number drops to zero. If we want to modify the content of our `Rc`, we can again use the `Cell` type.

Filename: listings/g_object_memory_management/2/main.rs

```
    // Reference-counted object with inner-mutability
    let number = Rc::new(Cell::new(0));

    // Connect callbacks, when a button is clicked `number` will be changed
    let number_copy = number.clone();
    button_increase.connect_clicked(move |_|
number_copy.set(number_copy.get() + 1));
    button_decrease.connect_clicked(move |_| number.set(number.get() - 1));
```

It is not very nice though to fill the scope with temporary variables like `number_copy`. We can improve that by using the `glib::clone!` macro.

Filename: listings/g_object_memory_management/3/main.rs

```
    button_increase.connect_clicked(clone!(@strong number => move |_| {
        number.set(number.get() + 1);
    }));
    button_decrease.connect_clicked(move |_| {
        number.set(number.get() - 1);
    });
```

Just like `Rc<Cell<T>>`, GObjects are reference-counted and mutable. Therefore, we can pass the buttons the same way to the closure as we did with `number`.

Filename: listings/g_object_memory_management/4/main.rs

```
    // Connect callbacks
    // When a button is clicked, `number` and label of the other button will
be changed
    button_increase.connect_clicked(clone!(@weak number, @strong
button_decrease =>
        move |_| {
            number.set(number.get() + 1);
            button_decrease.set_label(&number.get().to_string());
    }));
    button_decrease.connect_clicked(clone!(@strong button_increase =>
        move |_| {
            number.set(number.get() - 1);
            button_increase.set_label(&number.get().to_string());
    }));
```

If we now click on one button, the other button's label gets changed.

But whoops! Did we forget about one annoyance of reference-counted systems? Yes we did: reference cycles. `button_increase` holds a strong reference to `button_decrease` and vice-versa. A strong reference keeps the referenced value from being deallocated. If this chain leads to a circle, none of the values in this cycle ever get deallocated. With weak references we can break this cycle, because they don't keep their value alive but instead provide a way to retrieve a strong reference if the value is still alive. Since we want our apps to free unneeded memory, we should use weak references for the buttons instead.

Filename: listings/g_object_memory_management/5/main.rs

```
    // Connect callbacks
    // When a button is clicked, `number` and label of the other button will
 be changed
    button_increase.connect_clicked(clone!(@weak number, @weak
 button_decrease =>
        move |_| {
            number.set(number.get() + 1);
            button_decrease.set_label(&number.get().to_string());
    }));
    button_decrease.connect_clicked(clone!(@weak button_increase =>
        move |_| {
            number.set(number.get() - 1);
            button_increase.set_label(&number.get().to_string());
    }));
```

The reference cycle is broken. Every time the button is clicked, `glib::clone` tries to upgrade the weak reference. If we now for example click on one button and the other button is not there anymore, the callback will be skipped. Per default, it immediately returns from the closure with `()` as return value. In case the closure expects a different return value `@default-return` can be specified.

Notice that we move `number` in the second closure. If we had moved weak references in both closures, nothing would have kept `number` alive and the closure would have never been called. Thinking about this, `button_increase` and `button_decrease` are also dropped at the end of the scope of `build_ui`. Who then keeps the buttons alive?

Filename: listings/g_object_memory_management/5/main.rs

```
    // Add buttons to `gtk_box`
    let gtk_box = gtk::Box::builder()
        .orientation(Orientation::Vertical)
        .build();
    gtk_box.append(&button_increase);
    gtk_box.append(&button_decrease);
```

When we append the buttons to the `gtk_box`, `gtk_box` keeps a strong reference to them.

Filename: listings/g_object_memory_management/5/main.rs

```
    // Create a window
    let window = ApplicationWindow::builder()
        .application(app)
        .title("My GTK App")
        .child(&gtk_box)
        .build();
```

When we set `gtk_box` as child of `window`, `window` keeps a strong reference to it. Until we close the `window` it keeps `gtk_box` and with it the buttons alive. Since our application has only one window, closing it also means exiting the application.

As long as you use weak references whenever possible, you will find it perfectly doable to avoid memory cycles within your application. Without memory cycles, you can rely on GTK to properly manage the memory of GObjects you pass to it.

Subclassing

GObjects rely heavily on inheritance. Therefore, it makes sense that if we want to create a custom GObject, this is done via subclassing. Let's see how this works by replacing the button in our "Hello World!" app with a custom one. First, we need to create an implementation struct that holds the state and overrides the virtual methods.

Filename: listings/g_object_subclassing/1/custom_button/imp.rs

```
use gtk::glib;
use gtk::subclass::prelude::*;

// Object holding the state
#[derive(Default)]
pub struct CustomButton;

// The central trait for subclassing a GObject
#[glib::object_subclass]
impl ObjectSubclass for CustomButton {
    const NAME: &'static str = "MyGtkAppCustomButton";
    type Type = super::CustomButton;
    type ParentType = gtk::Button;
}

// Trait shared by all GObjects
impl ObjectImpl for CustomButton {}

// Trait shared by all widgets
impl WidgetImpl for CustomButton {}

// Trait shared by all buttons
impl ButtonImpl for CustomButton {}
```

The description of the subclassing is in `ObjectSubclass`.

- `NAME` should consist of crate-name and object-name in order to avoid name collisions. Use UpperCamelCase here.
- `Type` refers to the actual GObject that will be created afterwards.
- `ParentType` is the GObject we inherit of.

After that, we would have the option to override the virtual methods of our ancestors. Since we only want to have a plain button for now, we override nothing. We still have to add the empty `impl` though. Next, we describe the public interface of our custom GObject.

Filename: listings/g_object_subclassing/1/custom_button/mod.rs

```
mod imp;

use glib::Object;
use gtk::glib;

glib::wrapper! {
    pub struct CustomButton(ObjectSubclass<imp::CustomButton>)
        @extends gtk::Button, gtk::Widget,
        @implements gtk::Accessible, gtk::Actionable, gtk::Buildable,
gtk::ConstraintTarget;
}

impl CustomButton {
    pub fn new() -> Self {
        Object::builder().build()
    }

    pub fn with_label(label: &str) -> Self {
        Object::builder().property("label", label).build()
    }
}
```

`glib::wrapper!` implements the same traits that our `ParentType` implements. Theoretically that would mean that the `ParentType` is also the only thing we have to specify here. Unfortunately, nobody has yet found a good way to do that. Which is why, as of today, subclassing of GObjects in Rust requires to mention all ancestors and interfaces apart from `GObject` and `GInitiallyUnowned`. For `gtk::Button`, we can look up the ancestors and interfaces in the corresponding doc page of GTK4.

After these steps, nothing is stopping us from replacing `gtk::Button` with our `CustomButton`.

Filename: listings/g_object_subclassing/1/main.rs

```
mod custom_button;

use custom_button::CustomButton;
use gtk::prelude::*;
use gtk::{glib, Application, ApplicationWindow};

const APP_ID: &str = "org.gtk_rs.GObjectSubclassing1";

fn main() -> glib::ExitCode {
    // Create a new application
    let app = Application::builder().application_id(APP_ID).build();

    // Connect to "activate" signal of `app`
    app.connect_activate(build_ui);

    // Run the application
    app.run()
}

fn build_ui(app: &Application) {
    // Create a button
    let button = CustomButton::with_label("Press me!");
    button.set_margin_top(12);
    button.set_margin_bottom(12);
    button.set_margin_start(12);
    button.set_margin_end(12);

    // Connect to "clicked" signal of `button`
    button.connect_clicked(move |button| {
        // Set the label to "Hello World!" after the button has been clicked
on
        button.set_label("Hello World!");
    });

    // Create a window
    let window = ApplicationWindow::builder()
        .application(app)
        .title("My GTK App")
        .child(&button)
        .build();

    // Present window
    window.present();
}
```

Describing objects with two structs is a peculiarity coming from how GObjects are defined in C. `imp::CustomButton` handles the state of the GObject and the overridden virtual methods. `CustomButton` determines the exposed methods from the implemented traits and added methods.

Adding Functionality

We are able to use `CustomButton` as a drop-in replacement for `gtk::Button`. This is cool, but also not very tempting to do in a real application. For the gain of zero benefits, it did involve quite a bit of boilerplate after all.

So let's make it a bit more interesting! `gtk::Button` does not hold much state, but we can let `CustomButton` hold a number.

Filename: listings/g_object_subclassing/2/custom_button/imp.rs

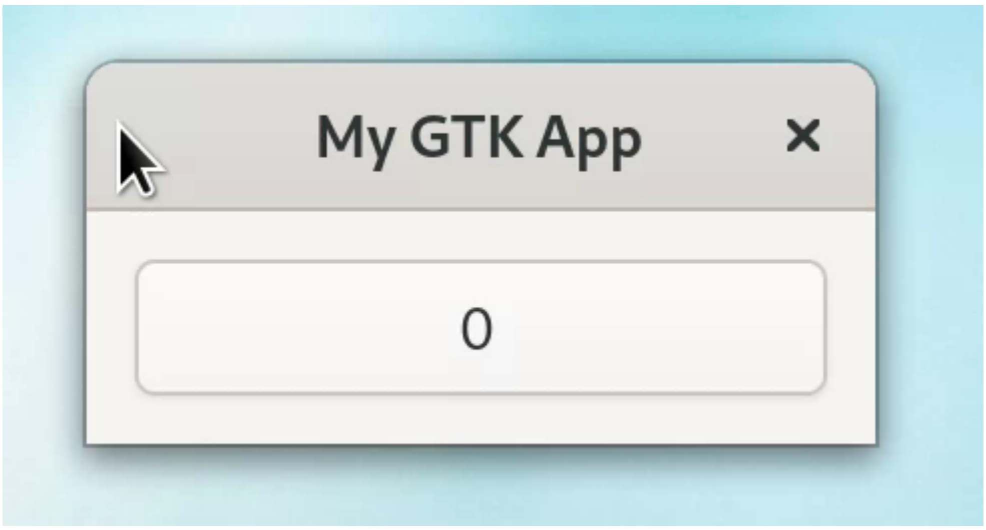
```
use std::cell::Cell;

use gtk::glib;
use gtk::prelude::*;
use gtk::subclass::prelude::*;

// Object holding the state
#[derive(Default)]
pub struct CustomButton {
    number: Cell<i32>,
}

// The central trait for subclassing a GObject
#[glib::object_subclass]
impl ObjectSubclass for CustomButton {
    const NAME: &'static str = "MyGtkAppCustomButton";
    type Type = super::CustomButton;
    type ParentType = gtk::Button;
}

// Trait shared by all GObjects
impl ObjectImpl for CustomButton {
    fn constructed(&self) {
        self.parent_constructed();
        self.obj().set_label(&self.number.get().to_string());
    }
}

// Trait shared by all widgets
impl WidgetImpl for CustomButton {}

// Trait shared by all buttons
impl ButtonImpl for CustomButton {
    fn clicked(&self) {
        self.number.set(self.number.get() + 1);
        self.obj().set_label(&self.number.get().to_string())
    }
}
```

We override `constructed` in `ObjectImpl` so that the label of the button initializes with `number`. We also override `clicked` in `ButtonImpl` so that every click increases `number` and updates the label.

Filename: listings/g_object_subclassing/2/main.rs

```
fn build_ui(app: &Application) {
    // Create a button
    let button = CustomButton::new();
    button.set_margin_top(12);
    button.set_margin_bottom(12);
    button.set_margin_start(12);
    button.set_margin_end(12);

    // Create a window
    let window = ApplicationWindow::builder()
        .application(app)
        .title("My GTK App")
        .child(&button)
        .build();

    // Present window
    window.present();
}
```

In `build_ui` we stop calling `connect_clicked`, and that was it. After a rebuild, the app now features our `CustomButton` with the label "0". Every time we click on the button, the number displayed by the label increases by 1.



So, when do we want to inherit from GObject?

- We want to use a certain widget, but with added state and overridden virtual functions.
- We want to pass a Rust object to a function, but the function expects a GObject.
- We want to add properties or signals to an object.

Generic Values

Some GObject-related functions rely on generic values for their arguments or return parameters. Since GObject introspection works through a C interface, these functions cannot rely on any powerful Rust concepts. In these cases `glib::Value` or `glib::Variant` are used.

Value

Let's start with `Value`. Conceptually, a `Value` is similar to a Rust `enum` defined like this:

```
enum Value <T> {
    bool(bool),
    i8(i8),
    i32(i32),
    u32(u32),
    i64(i64),
    u64(u64),
    f32(f32),
    f64(f64),
    // boxed types
    String(Option<String>),
    Object(Option<dyn IsA<glib::Object>>),
}
```

For example, this is how you would use a `Value` representing an `i32`.

Filename: listings/g_object_values/1/main.rs

```
    // Store `i32` as `Value`
    let integer_value = 10.to_value();

    // Retrieve `i32` from `Value`
    let integer = integer_value
        .get::<i32>()
        .expect("The value needs to be of type `i32`.");

    // Check if the retrieved value is correct
    assert_eq!(integer, 10);
```

Also note that in the `enum` above boxed types such as `String` or `glib::Object` are wrapped in an `Option`. This comes from C, where every boxed type can potentially be `None` (or `NULL` in C terms). You can still access it the same way as with the `i32` above. `get` will then not only return `Err` if you specified the wrong type, but also if the `Value` represents `None`.

Filename: listings/g_object_values/1/main.rs

```
    // Store string as `Value`
    let string_value = "Hello!".to_value();

    // Retrieve `String` from `Value`
    let string = string_value
        .get::<String>()
        .expect("The value needs to be of type `String`.");

    // Check if the retrieved value is correct
    assert_eq!(string, "Hello!".to_string());
```

If you want to differentiate between specifying the wrong type and a `Value` representing `None`, just call `get::<Option<T>>` instead.

Filename: listings/g_object_values/1/main.rs

```
    // Store `Option<String>` as `Value`
    let string_some_value = "Hello!".to_value();
    let string_none_value = None::<String>.to_value();

    // Retrieve `String` from `Value`
    let string_some = string_some_value
        .get::<Option<String>>()
        .expect("The value needs to be of type `Option<String>`.");
    let string_none = string_none_value
        .get::<Option<String>>()
        .expect("The value needs to be of type `Option<String>`.");

    // Check if the retrieved value is correct
    assert_eq!(string_some, Some("Hello!".to_string()));
    assert_eq!(string_none, None);
```

We will use `Value` when we deal with properties and signals later on.

Variant

A `Variant` is used whenever data needs to be serialized, for example for sending it to another process or over the network, or for storing it on disk. Although `GVariant` supports arbitrarily complex types, the Rust bindings are currently limited to `bool`, `u8`, `i16`, `u16`, `i32`, `u32`, `i64`, `u64`, `f64`, `&str` / `String`, and `VariantDict`. Containers of the above types are possible as well, such as `HashMap`, `Vec`, `Option`, tuples up to 16 elements, and `Variant`. Variants can even be derived from Rust structs as long as its members can be represented by variants.

In the most simple case, converting Rust types to `Variant` and vice-versa is very similar to the way it worked with `Value`.

Filename: listings/g_object_values/2/main.rs

```
    // Store `i32` as `Variant`
    let integer_variant = 10.to_variant();

    // Retrieve `i32` from `Variant`
    let integer = integer_variant
        .get::<i32>()
        .expect("The variant needs to be of type `i32`.");

    // Check if the retrieved value is correct
    assert_eq!(integer, 10);
```

However, a `Variant` is also able to represent containers such as `HashMap` or `Vec`. The following snippet shows how to convert between `Vec` and `Variant`. More examples can be found in the docs.

Filename: listings/g_object_values/2/main.rs

```
    let variant = vec!["Hello", "there!"].to_variant();
    assert_eq!(variant.n_children(), 2);
    let vec = &variant
        .get::<Vec<String>>()
        .expect("The variant needs to be of type `String`.");
    assert_eq!(vec[0], "Hello");
```

We will use `Variant` when saving settings using `gio::Settings` or activating actions via `gio::Action`.

Properties

Properties provide a public API for accessing state of GObjects.

Let's see how this is done by experimenting with the `Switch` widget. One of its properties is called active. According to the GTK docs, it can be read and be written to. That is why `gtk-rs` provides corresponding `is_active` and `set_active` methods.

Filename: listings/g_object_properties/1/main.rs

```
    // Create the switch
    let switch = Switch::new();

    // Set and then immediately obtain active property
    switch.set_active(true);
    let switch_active = switch.is_active();

    // This prints: "The active property of switch is true"
    println!("The active property of switch is {}", switch_active);
```

Properties can not only be accessed via getters & setters, they can also be bound to each other. Let's see how that would look like for two `Switch` instances.

Filename: listings/g_object_properties/2/main.rs

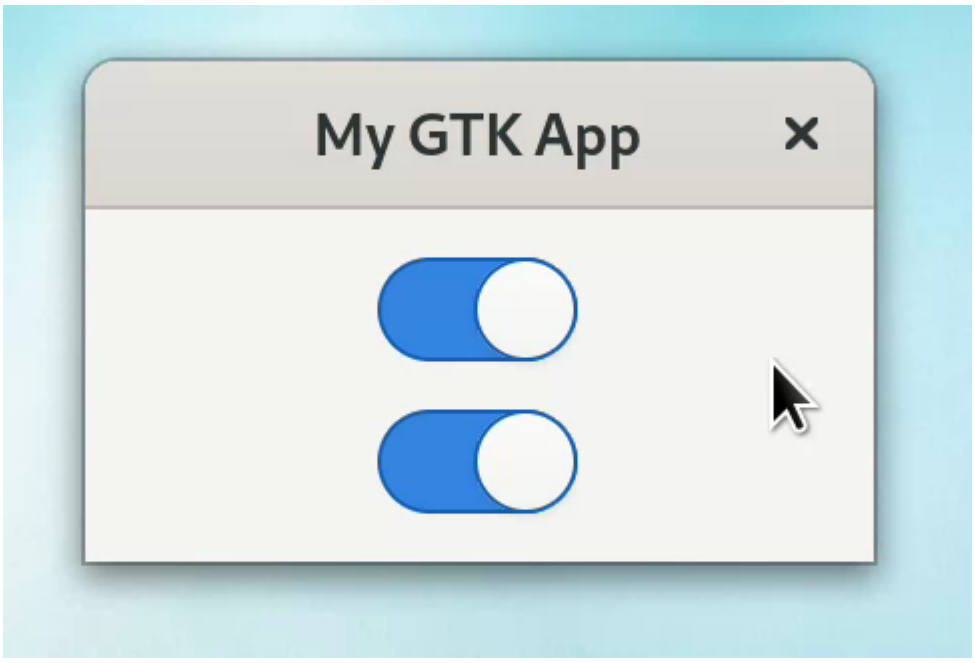
```
    // Create the switches
    let switch_1 = Switch::new();
    let switch_2 = Switch::new();
```

In our case, we want to bind the "active" property of `switch_1` to the "active" property of `switch_2`. We also want the binding to be bidirectional, so we specify by calling the `bidirectional` method.

Filename: listings/g_object_properties/2/main.rs

```
    switch_1
        .bind_property("active", &switch_2, "active")
        .bidirectional()
        .build();
```

Now when we click on one of the two switches, the other one is toggled as well.



Adding Properties to Custom GObjects

We can also add properties to custom GObjects. We can demonstrate that by binding the `number` of our `CustomButton` to a property. Most of the work is done by the `glib::Properties` derive macro. We tell it that the wrapper type is `super::CustomButton`. We also annotate `number`, so that macro knows that it should create a property "number" that is readable and writable. It also generates wrapper methods which we are going to use later in this chapter.

Filename: listings/g_object_properties/3/custom_button/imp.rs

```
// Object holding the state
#[derive(Properties, Default)]
#[properties(wrapper_type = super::CustomButton)]
pub struct CustomButton {
    #[property(get, set)]
    number: Cell<i32>,
}
```

The `glib::derived_properties` macro generates boilerplate that is the same for every GObject that generates its properties with the `Property` macro. In `constructed` we use our new property "number" by binding the "label" property to it. `bind_property` converts the integer value of "number" to the string of "label" on its own. Now we don't have to adapt the label in the "clicked" callback anymore.

Filename: listings/g_object_properties/3/custom_button/imp.rs

```
// Trait shared by all GObjects
#[glib::derived_properties]
impl ObjectImpl for CustomButton {
    fn constructed(&self) {
        self.parent_constructed();

        // Bind label to number
        // `SYNC_CREATE` ensures that the label will be immediately set
        let obj = self.obj();
        obj.bind_property("number", obj.as_ref(), "label")
            .sync_create()
            .build();
    }
}
```

We also have to adapt the `clicked` method. Before we modified `number` directly, now we can use the generated wrapper methods `number` and `set_number` . This way the "notify" signal will be emitted, which is necessary for the bindings to work as expected.

```
// Trait shared by all buttons
impl ButtonImpl for CustomButton {
    fn clicked(&self) {
        let incremented_number = self.obj().number() + 1;
        self.obj().set_number(incremented_number);
    }
}
```

Let's see what we can do with this by creating two custom buttons.

Filename: listings/g_object_properties/3/main.rs

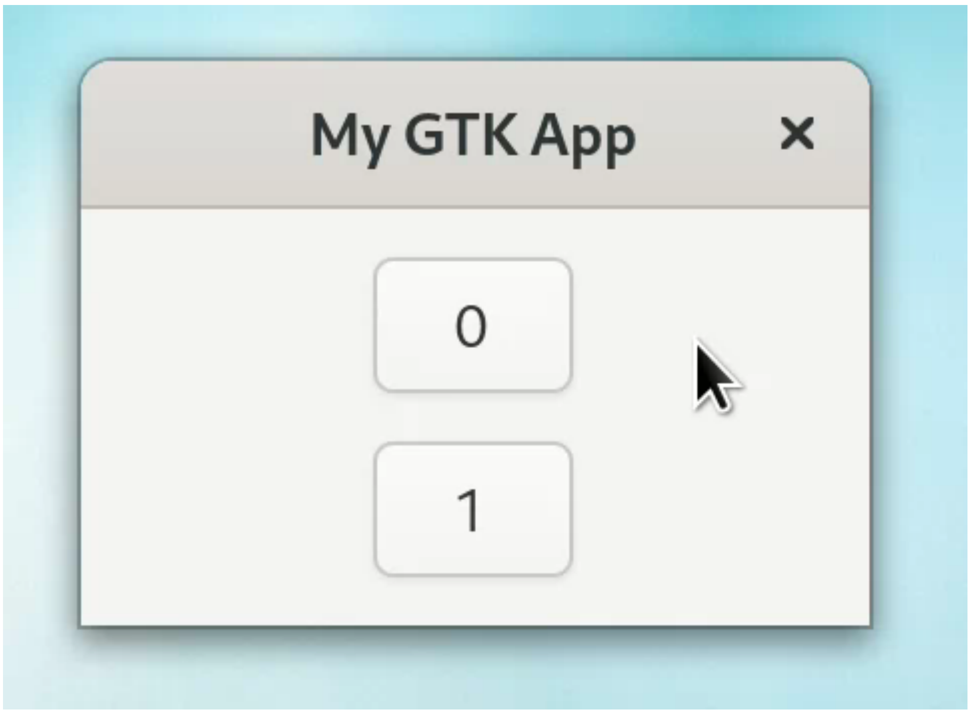
```
    // Create the buttons
    let button_1 = CustomButton::new();
    let button_2 = CustomButton::new();
```

We have already seen that bound properties don't necessarily have to be of the same type. By leveraging `transform_to` and `transform_from` , we can assure that `button_2` always displays a number which is 1 higher than the number of `button_1` .

Filename: listings/g_object_properties/3/main.rs

```
    // Assure that "number" of `button_2` is always 1 higher than "number" of
`button_1`
    button_1
        .bind_property("number", &button_2, "number")
        // How to transform "number" from `button_1` to "number" of
`button_2`
        .transform_to(|_, number: i32| {
            let incremented_number = number + 1;
            Some(incremented_number.to_value())
        })
        // How to transform "number" from `button_2` to "number" of
`button_1`
        .transform_from(|_, number: i32| {
            let decremented_number = number - 1;
            Some(decremented_number.to_value())
        })
        .bidirectional()
        .sync_create()
        .build();
```

Now if we click on one button, the "number" and "label" properties of the other button change as well.



Another nice feature of properties is, that you can connect a callback to the event, when a property gets changed. For example like this:

Filename: listings/g_object_properties/3/main.rs

```
    // The closure will be called
    // whenever the property "number" of `button_1` gets changed
    button_1.connect_number_notify(|button| {
        println!("The current number of `button_1` is {}.", button.number());
    });
```

Now, whenever the "number" property gets changed, the closure gets executed and prints the current value of "number" to standard output.

Introducing properties to your custom GObjects is useful if you want to

- bind state of (different) GObjects
- notify consumers whenever a property value changes

Note that it has a (computational) cost to send a signal each time the value changes. If you only want to expose internal state, adding getter and setter methods is the better option.

Signals

GObject signals are a system for registering callbacks for specific events. For example, if we press on a button, the "clicked" signal will be emitted. The signal then takes care that all the registered callbacks will be executed.

`gtk-rs` provides convenience methods for registering callbacks. In our "Hello World" example we connected the "clicked" signal to a closure which sets the label of the button to "Hello World" as soon as it gets called.

Filename: listings/hello_world/3/main.rs

```
    // Connect to "clicked" signal of `button`
    button.connect_clicked(|button| {
        // Set the label to "Hello World!" after the button has been clicked
 on
        button.set_label("Hello World!");
    });
```

If we wanted to, we could have connected to it with the generic `connect_closure` method and the `glib::closure_local!` macro.

Filename: listings/g_object_signals/1/main.rs

```
    // Connect to "clicked" signal of `button`
    button.connect_closure(
        "clicked",
        false,
        closure_local!(move |button: Button| {
            // Set the label to "Hello World!" after the button has been
 clicked on
            button.set_label("Hello World!");
        }),
    );
```

The advantage of `connect_closure` is that it also works with custom signals.

> If you need to clone reference counted objects into your closure you don't have to wrap it within another `clone!` macro. `closure_local!` accepts the same syntax for creating strong/weak references, plus a watch feature that automatically disconnects the closure once the watched object is dropped.

Adding Signals to Custom GObjects

Let's see how we can create our own signals. Again we do that by extending our `CustomButton`. First we override the `signals` method in `ObjectImpl`. In order to do that, we need to lazily initialize a static item `SIGNALS`. `std::sync::OnceLock` ensures that `SIGNALS` will only be initialized once.

Filename: listings/g_object_signals/2/custom_button/imp.rs

```
// Trait shared by all GObjects
#[glib::derived_properties]
impl ObjectImpl for CustomButton {
    fn signals() -> &'static [Signal] {
        static SIGNALS: OnceLock<Vec<Signal>> = OnceLock::new();
        SIGNALS.get_or_init(|| {
            vec![Signal::builder("max-number-reached")
                .param_types([i32::static_type()])
                .build()]
        })
    }
```

The `signals` method is responsible for defining a set of signals. In our case, we only create a single signal named "max-number-reached". When naming our signal, we make sure to do that in kebab-case. When emitted, it sends a single `i32` value.

We want the signal to be emitted, whenever `number` reaches `MAX_NUMBER`. Together with the signal we send the value `number` currently holds. After we did that, we set `number` back to 0.

Filename: listings/g_object_signals/2/custom_button/imp.rs

```
static MAX_NUMBER: i32 = 8;

// Trait shared by all buttons
impl ButtonImpl for CustomButton {
    fn clicked(&self) {
        let incremented_number = self.obj().number() + 1;
        let obj = self.obj();
        // If `number` reached `MAX_NUMBER`,
        // emit "max-number-reached" signal and set `number` back to 0
        if incremented_number == MAX_NUMBER {
            obj.emit_by_name::<()>("max-number-reached",
&[&incremented_number]);
            obj.set_number(0);
        } else {
            obj.set_number(incremented_number);
        }
    }
}
```

If we now press on the button, the number of its label increases until it reaches `MAX_NUMBER`. Then it emits the "max-number-reached" signal which we can nicely connect to. Whenever we now receive the "max-number-reached" signal, the accompanying number is printed to standard output.

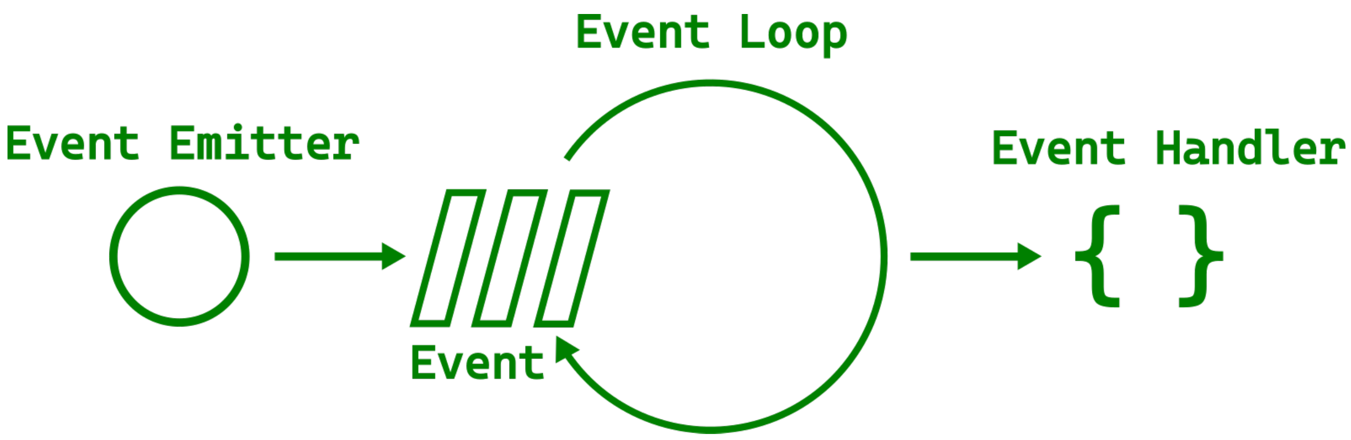
Filename: listings/g_object_signals/2/main.rs

```
    button.connect_closure(
        "max-number-reached",
        false,
        closure_local!(move |_button: CustomButton, number: i32| {
            println!("The maximum number {} has been reached", number);
        }),
    );
```

You now know how to connect to every kind of signal and how to create your own. Custom signals are especially useful, if you want to notify consumers of your GObject that a certain event occurred.

The Main Event Loop

We now got comfortable using callbacks, but how do they actually work? All of this happens asynchronously, so there must be something managing the events and scheduling the responses. Unsurprisingly, this is called the main event loop.



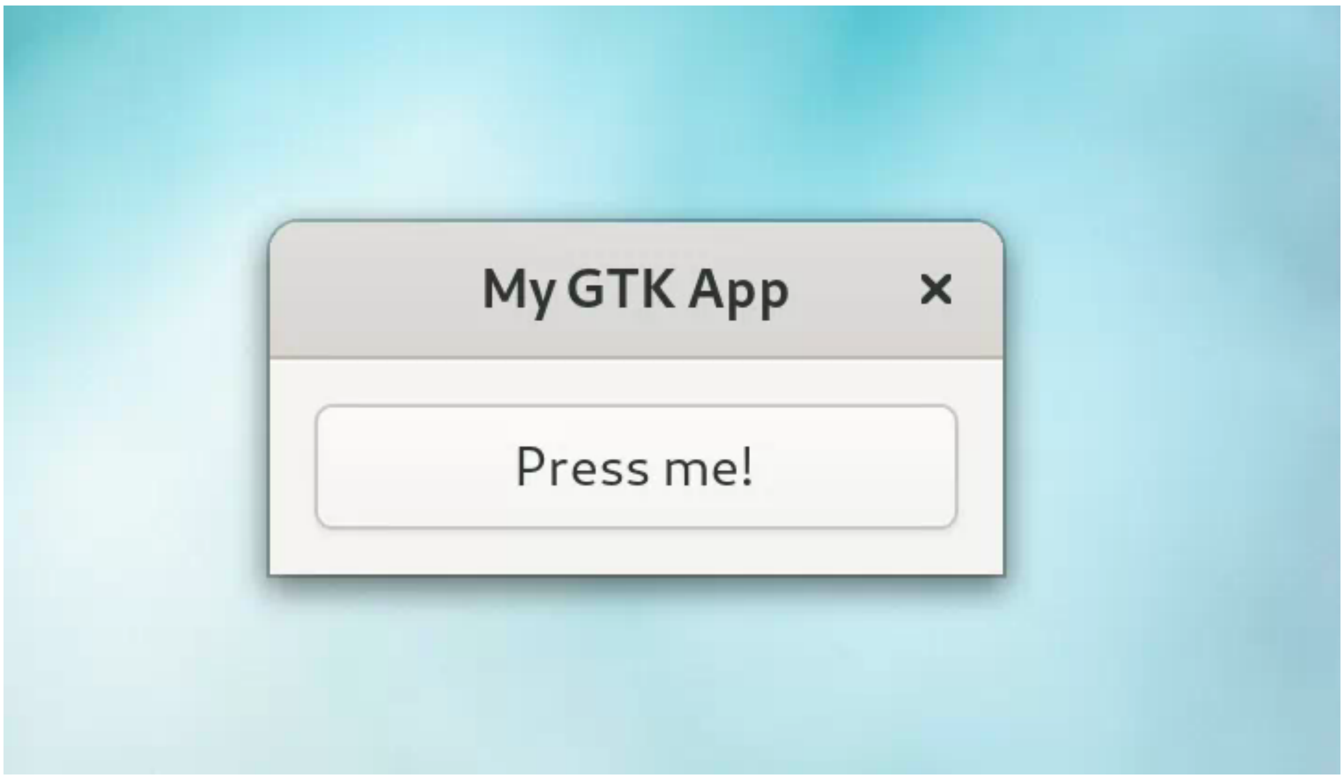
The main loop manages all kinds of events — from mouse clicks and keyboard presses to file events. It does all of that within the same thread. Quickly iterating between all tasks gives the illusion of parallelism. That is why you can move the window at the same time as a progress bar is growing.

However, you surely saw GUIs that became unresponsive, at least for a few seconds. That happens when a single task takes too long. The following example uses `std::thread::sleep` to represent a long-running task.

Filename: listings/main_event_loop/1/main.rs

```rust
use std::thread;
use std::time::Duration;

use gtk::prelude::*;
use gtk::{self, glib, Application, ApplicationWindow, Button};

const APP_ID: &str = "org.gtk_rs.MainEventLoop1";

fn main() -> glib::ExitCode {
    // Create a new application
    let app = Application::builder().application_id(APP_ID).build();

    // Connect to "activate" signal of `app`
    app.connect_activate(build_ui);

    // Run the application
    app.run()
}

fn build_ui(app: &Application) {
    // Create a button
    let button = Button::builder()
        .label("Press me!")
        .margin_top(12)
        .margin_bottom(12)
        .margin_start(12)
        .margin_end(12)
        .build();

    // Connect to "clicked" signal of `button`
    button.connect_clicked(move |_| {
        // GUI is blocked for 5 seconds after the button is pressed
        let five_seconds = Duration::from_secs(5);
        thread::sleep(five_seconds);
    });

    // Create a window
    let window = ApplicationWindow::builder()
        .application(app)
        .title("My GTK App")
        .child(&button)
        .build();

    // Present window
    window.present();
}
```

After we press the button, the GUI is completely frozen for five seconds. We can't even move the window. The `sleep` call is an artificial example, but frequently, we want to run a slightly longer operation in one go.



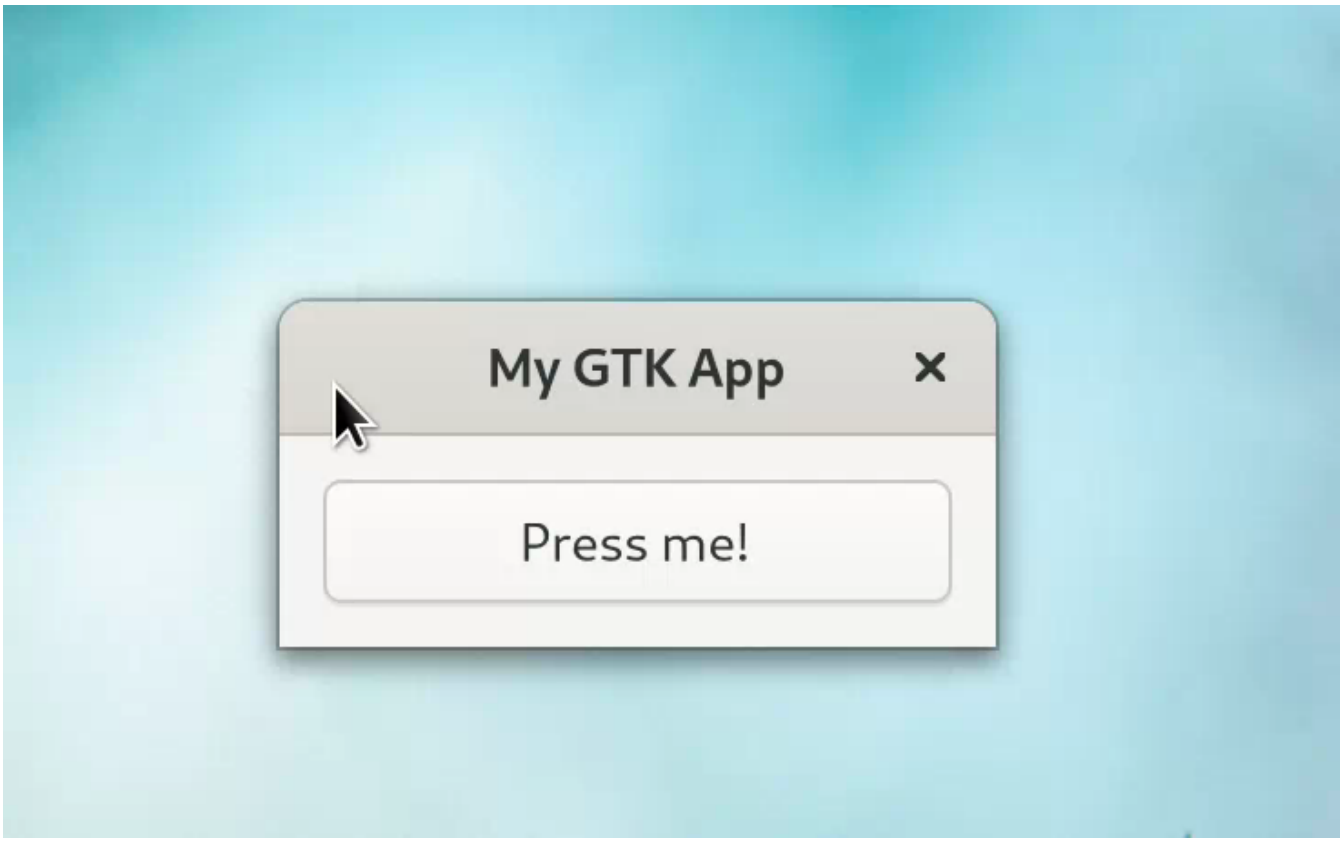
How to Avoid Blocking the Main Loop

In order to avoid blocking the main loop, we can spawn a new task with `gio::spawn_blocking` and let the operation run on the thread pool.

Filename: listings/main_event_loop/2/main.rs

```
    // Connect to "clicked" signal of `button`
    button.connect_clicked(move |_| {
        // The long running operation runs now in a separate thread
        gio::spawn_blocking(move || {
            let five_seconds = Duration::from_secs(5);
            thread::sleep(five_seconds);
        });
    });
```

Now the GUI doesn't freeze when we press the button. However, nothing stops us from spawning as many tasks as we want at the same time. This is not necessarily what we want.



If you come from another language than Rust, you might be uncomfortable with the thought of running tasks in separate threads before even looking at other options. Luckily, Rust's safety guarantees allow you to stop worrying about the nasty bugs that concurrency tends to bring.

Channels

Typically, we want to keep track of the work in the task. In our case, we don't want the user to spawn additional tasks while an existing one is still running. In order to exchange information with the task we can create a channel with the crate `async-channel`. Let's add it by executing the following in the terminal:

```
cargo add async-channel
```

We want to send a `bool` to inform, whether we want the button to react to clicks or not. Since we send in a separate thread, we can use `send_blocking`. But what about receiving? Every time we get a message, we want to set the sensitivity of the button according to the `bool` we've received. However, we don't want to block the main loop while waiting for a message to receive. That is the whole point of the exercise after all!

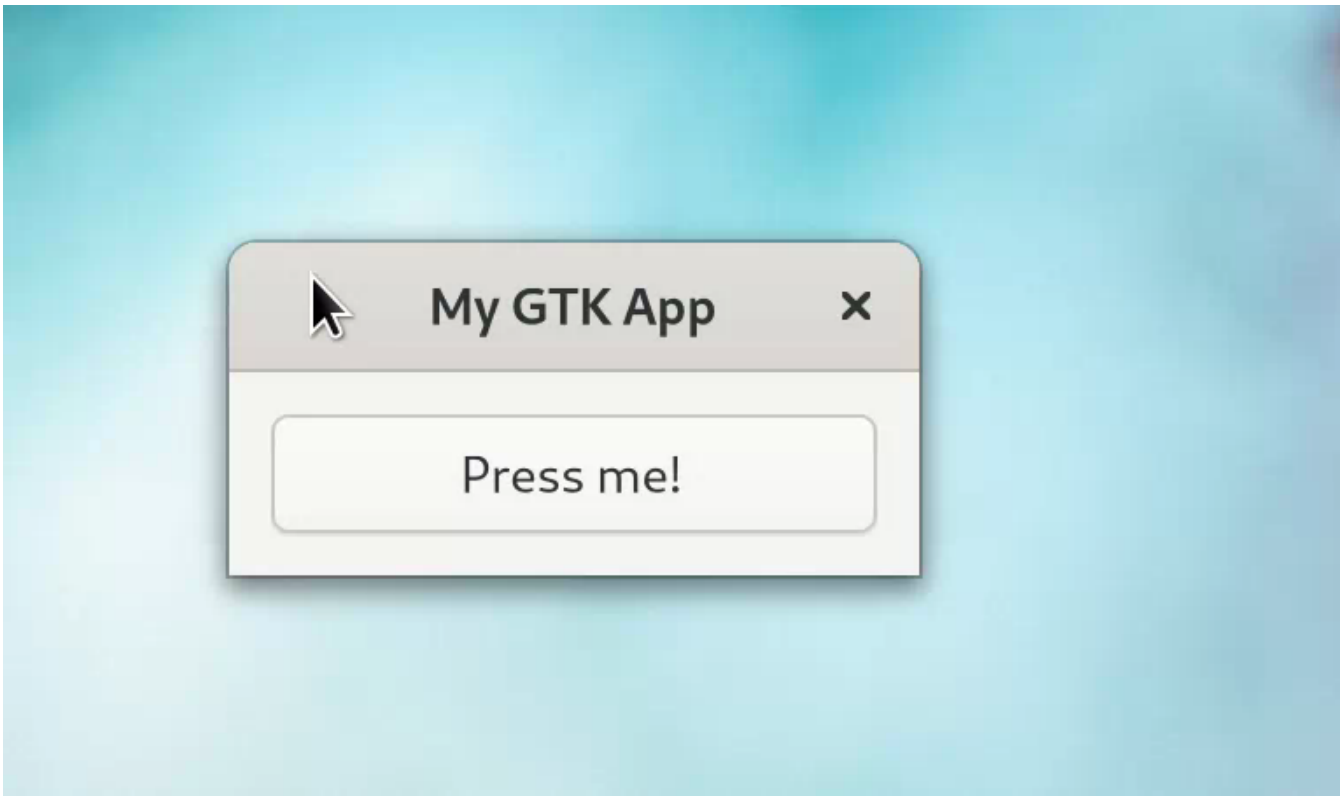
We solve that problem by waiting for messages in an `async` block. This `async` block is spawned on the `glib` main loop with `spawn_future_local`

> See also `spawn_future` for spawning async blocks on the main loop from outside the main thread.

Filename: listings/main_event_loop/3/main.rs

```
    // Create channel that can hold at most 1 message at a time
    let (sender, receiver) = async_channel::bounded(1);
    // Connect to "clicked" signal of `button`
    button.connect_clicked(move |_| {
        let sender = sender.clone();
        // The long running operation runs now in a separate thread
        gio::spawn_blocking(move || {
            // Deactivate the button until the operation is done
            sender
                .send_blocking(false)
                .expect("The channel needs to be open.");
            let five_seconds = Duration::from_secs(5);
            thread::sleep(five_seconds);
            // Activate the button again
            sender
                .send_blocking(true)
                .expect("The channel needs to be open.");
        });
    });

    // The main loop executes the asynchronous block
    glib::spawn_future_local(clone!(@weak button => async move {
        while let Ok(enable_button) = receiver.recv().await {
            button.set_sensitive(enable_button);
        }
    }));
```

As you can see, spawning a task still doesn't freeze our user interface. However, now we can't spawn multiple tasks at the same time since the button becomes insensitive after the first task has been spawned. After the task is finished, the button becomes sensitive again.



What if the task is asynchronous by nature? Let's try `glib::timeout_future_seconds` as representation for our task instead of `std::thread::sleep`. It returns a `std::future::Future`, which means we can `await` on it within an `async` context. The converted code looks and behaves very similar to the multithreaded code.

Filename: listings/main_event_loop/4/main.rs

```
    // Create channel that can hold at most 1 message at a time
    let (sender, receiver) = async_channel::bounded(1);
    // Connect to "clicked" signal of `button`
    button.connect_clicked(move |_| {
        glib::spawn_future_local(clone!(@strong sender => async move {
            // Deactivate the button until the operation is done
            sender.send(false).await.expect("The channel needs to be open.");
            glib::timeout_future_seconds(5).await;
            // Activate the button again
            sender.send(true).await.expect("The channel needs to be open.");
        }));
    });

    // The main loop executes the asynchronous block
    glib::spawn_future_local(clone!(@weak button => async move {
        while let Ok(enable_button) = receiver.recv().await {
            button.set_sensitive(enable_button);
        }
    }));
```

Since we are single-threaded again, we can even get rid of the channel while achieving the same result.

Filename: listings/main_event_loop/5/main.rs

```
    // Connect to "clicked" signal of `button`
    button.connect_clicked(move |button| {
        glib::spawn_future_local(clone!(@weak button => async move {
            // Deactivate the button until the operation is done
            button.set_sensitive(false);
            glib::timeout_future_seconds(5).await;
            // Activate the button again
            button.set_sensitive(true);
        }));
    });
```

But why did we not do the same thing with our multithreaded example?

```
    // DOES NOT COMPILE!
    // Connect to "clicked" signal of `button`
    button.connect_clicked(move |button| {
        button.clone();
        // The long running operation runs now in a separate thread
        gio::spawn_blocking(move || {
            // Deactivate the button until the operation is done
            button.set_sensitive(false);
            let five_seconds = Duration::from_secs(5);
            thread::sleep(five_seconds);
            // Activate the button again
            button.set_sensitive(true);
        });
    });
```

Simply because we would get this error message:

```
error[E0277]: `NonNull<GObject>` cannot be shared between threads safely

help: within `gtk4::Button`, the trait `Sync` is not implemented for
`NonNull<GObject>`
```

After reference cycles we found the second disadvantage of GTK GObjects: They are not thread safe.

Embed blocking calls in an `async` context

We've seen in the previous snippets that spawning an `async` block or `async` future on the `glib` main loop can lead to more concise code than running tasks on separate threads. Let's focus on a few more aspects that are interesting to know when running `async` functions with gtk-rs apps.

For a start, blocking functions can be embedded within an `async` context. In the following listing, we want to execute a synchronous function that returns a boolean and takes ten seconds to run. In order to integrate it in our `async` block, we run the function in a separate thread via `spawn_blocking`. We can then get the return value of the function by calling `await` on the return value of `spawn_blocking`.

Filename: listings/main_event_loop/6/main.rs

```rust
    // Connect to "clicked" signal of `button`
    button.connect_clicked(move |button| {
        // The main loop executes the asynchronous block
        glib::spawn_future_local(clone!(@weak button => async move {
            // Deactivate the button until the operation is done
            button.set_sensitive(false);
            let enable_button = gio::spawn_blocking(move || {
                let five_seconds = Duration::from_secs(5);
                thread::sleep(five_seconds);
                true
            })
            .await
            .expect("Task needs to finish successfully.");
            // Set sensitivity of button to `enable_button`
            button.set_sensitive(enable_button);
        }));
    });
```

Run `async` functions from external crates

Asynchronous functions from the `glib` ecosystem can always be spawned on the `glib` main loop. Typically, crates depending on `async-std` or `smol` work as well. Let us take `ashpd` for example which allows sandboxed applications to interact with the desktop. Per default it depends on `async-std`. We can add it to our dependencies by running the following command.

```
cargo add ashpd --features gtk4
```

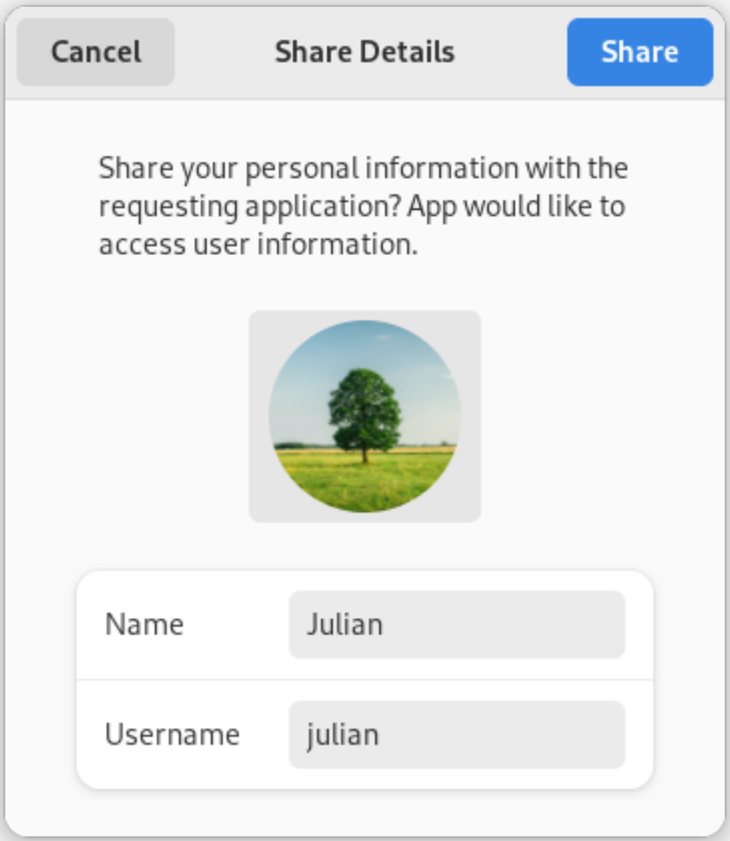
You need to use a Linux desktop environment in order to run the following example locally. This example is using `ashpd::desktop::account::UserInformation` to access user information. We are getting a `gtk::Native` object from our button, create a `ashpd::WindowIdentifier` and pass it to the user information request.

> We need to pass the `WindowIdentifier` to make the dialog modal. This means that it will be on top of the window and freezes the rest of the application from user input.

Filename: listings/main_event_loop/7/main.rs

```
    // Connect to "clicked" signal of `button`
    button.connect_clicked(move |button| {
        // The main loop executes the asynchronous block
        glib::spawn_future_local(clone!(@weak button => async move {
            // Get native of button for window identifier
            let native = button.native().expect("Need to be able to get
native.");
            // Get window identifier so that the dialog will be modal to the
main window
            let identifier = WindowIdentifier::from_native(&native).await;
            let request = UserInformation::request()
                .reason("App would like to access user information.")
                .identifier(identifier)
                .send()
                .await;

            if let Ok(response) = request.and_then(|r| r.response()) {
                println!("User name: {}", response.name());
            } else {
                println!("Could not access user information.")
            }
        }));
    });
```

After pressing the button, a dialog should open that shows the information that will be shared. If you decide to share it, you user name will be printed on the console.



Share your personal information with the requesting application? App would like to access user information.

Name Julian

Username julian

Tokio

`tokio` is Rust's most popular asynchronous platform. Therefore, many high-quality crates are part of its ecosystem. The web client `reqwest` belongs to this group. Let's add it by executing the following command

```
cargo add reqwest@0.11 --features rustls-tls --no-default-features
```

As soon as the button is pressed, we want to send a `GET` request to www.gtk-rs.org. The response should then be sent to the main thread via a channel.

Filename: listings/main_event_loop/8/main.rs

```
    let (sender, receiver) = async_channel::bounded(1);
    // Connect to "clicked" signal of `button`
    button.connect_clicked(move |_| {
        // The main loop executes the asynchronous block
        glib::spawn_future_local(clone!(@strong sender => async move {
            let response = reqwest::get("https://www.gtk-rs.org").await;
            sender.send(response).await.expect("The channel needs to be
open.");
        }));
    });

    // The main loop executes the asynchronous block
    glib::spawn_future_local(async move {
        while let Ok(response) = receiver.recv().await {
            if let Ok(response) = response {
                println!("Status: {}", response.status());
            } else {
                println!("Could not make a `GET` request.");
            }
        }
    });
```

This compiles fine and even seems to run. However, nothing happens when we press the button. Inspecting the console gives the following error message:

```
thread 'main' panicked at
'there is no reactor running, must be called from the context of a Tokio 1.x
runtime'
```

At the time of writing, `reqwest` doesn't document this requirement. Unfortunately, that is also the case for other libraries depending on `tokio`. Let's bite the bullet and add `tokio`:

```
cargo add tokio@1 --features rt-multi-thread
```

Since we already run the `glib` main loop on our main thread, we don't want to run the `tokio` runtime there. For this reason, we **avoid** using the `#[tokio::main]` macro or using a top-level `block_on` call. Doing this will block one of the runtime's threads with the GLib main loop, which is a waste of resources and a potential source of strange bugs.

Instead, we bind `tokio::runtime::Runtime` to a static variable.

```
// DOES NOT COMPILE!
static RUNTIME: Runtime =
    Runtime::new().expect("Setting up tokio runtime needs to succeed.");
```

Unfortunately, this doesn't compile. As usual, Rust's error messages are really helpful.

```
cannot call non-const fn `tokio::runtime::Runtime::new` in statics
calls in statics are limited to constant functions, tuple structs and tuple
variants
consider wrapping this expression in `Lazy::new(|| ...)` from the `once_cell`
crate
```

We could follow the advice directly, but the standard library also provides solutions for that. With `std::sync::OnceLock` we can initialize the static with the const function `OnceLock::new()` and initialize it the first time our function `runtime` is called.

Filename: listings/main_event_loop/9/main.rs

```
fn runtime() -> &'static Runtime {
    static RUNTIME: OnceLock<Runtime> = OnceLock::new();
    RUNTIME.get_or_init(|| {
        Runtime::new().expect("Setting up tokio runtime needs to succeed.")
    })
}
```

In the button callback we can now spawn the `reqwest` `async` block with `tokio` rather than with `glib`.

Filename: listings/main_event_loop/9/main.rs

```
    let (sender, receiver) = async_channel::bounded(1);
    // Connect to "clicked" signal of `button`
    button.connect_clicked(move |_| {
        runtime().spawn(clone!(@strong sender => async move {
            let response = reqwest::get("https://www.gtk-rs.org").await;
            sender.send(response).await.expect("The channel needs to be
open.");
        }));
    });

    // The main loop executes the asynchronous block
    glib::spawn_future_local(async move {
        while let Ok(response) = receiver.recv().await {
            if let Ok(response) = response {
                println!("Status: {}", response.status());
            } else {
                println!("Could not make a `GET` request.");
            }
        }
    });
```

If we now press the button, we should find the following message in our console:

```
Status: 200 OK
```

We will not need `tokio`, `reqwest` or `ashpd` in the following chapters, so let's remove them again by executing:

```
cargo remove tokio reqwest ashpd
```

How to find out whether you can spawn an `async` task on the `glib` main loop? `glib` should be able to spawn the task when the called functions come from libraries that either:

- come from the `glib` ecosystem,
- don't depend on a runtime but only on the `futures` family of crates (`futures-io` , `futures-core` etc),
- depend on the `async-std` or `smol` runtimes, or
- have cargo features that let them depend on `async-std` / `smol` instead of `tokio` .

Conclusion

You don't want to block the main thread long enough that it is noticeable by the user. But when should you spawn an `async` task, instead of spawning a task in a separate thread? Let's go again through the different scenarios.

If the task spends its time calculating rather than waiting for a web response, it is CPU-bound. That means you have to run the task in a separate thread and let it send results back via a channel.

If your task is IO bound, the answer depends on the crates at your disposal and the type of work to be done.

- Light I/O work with functions from crates using `glib` , `smol` , `async-std` or the `futures` trait family can be spawned on the main loop. This way, you can often avoid synchronization via channels.
- Heavy I/O work might still benefit from running in a separate thread / an async executor to avoid saturating the main loop. If you are unsure, benchmarking is advised.

If the best crate for the job relies on `tokio` , you will have to spawn it with the tokio runtime and communicate via channels.

Settings

We have now learned multiple ways to handle states. However, every time we close the application all of it is gone. Let's learn how to use `gio::Settings` by storing the state of a `Switch` in it.

At the very beginning we have to create a `GSchema` xml file in order to describe the kind of data our application plans to store in the settings.

Filename: listings/settings/1/org.gtk_rs.Settings1.gschema.xml

```
<?xml version="1.0" encoding="utf-8"?>
<schemalist>
  <schema id="org.gtk_rs.Settings1" path="/org/gtk_rs/Settings1/">
    <key name="is-switch-enabled" type="b">
      <default>false</default>
      <summary>Default switch state</summary>
    </key>
  </schema>
</schemalist>
```

Let's get through it step by step. The `id` is the same application id we used when we created our application.

Filename: listings/settings/1/main.rs

```
    // Create a new application
    let app = Application::builder().application_id(APP_ID).build();
```

The `path` must start and end with a forward slash character ('/') and must not contain two sequential slash characters. When creating a `path`, we advise to take the `id`, replace the '.' with '/' and add '/' at the front and end of it.

We only want to store a single key with the `name` "is-switch-enabled". This is a boolean value so its `type` is "b" (see GVariant Format Strings for the other options). We also set its default value to `false` (see GVariant Text Format for the full syntax). Finally, we add a summary.

Now we need to copy and compile the schema.

You can install the schema by executing the following commands on a Linux or macOS machine:

```
mkdir -p $HOME/.local/share/glib-2.0/schemas
cp org.gtk_rs.Settings1.gschema.xml $HOME/.local/share/glib-2.0/schemas/
glib-compile-schemas $HOME/.local/share/glib-2.0/schemas/
```

On Windows run:

```
mkdir C:/ProgramData/glib-2.0/schemas/
cp org.gtk_rs.Settings1.gschema.xml C:/ProgramData/glib-2.0/schemas/
glib-compile-schemas C:/ProgramData/glib-2.0/schemas/
```

We initialize the `Settings` object by specifying the application id.

Filename: listings/settings/1/main.rs

```
    // Initialize settings
    let settings = Settings::new(APP_ID);
```

Then we get the settings key and use it when we create our `Switch`.

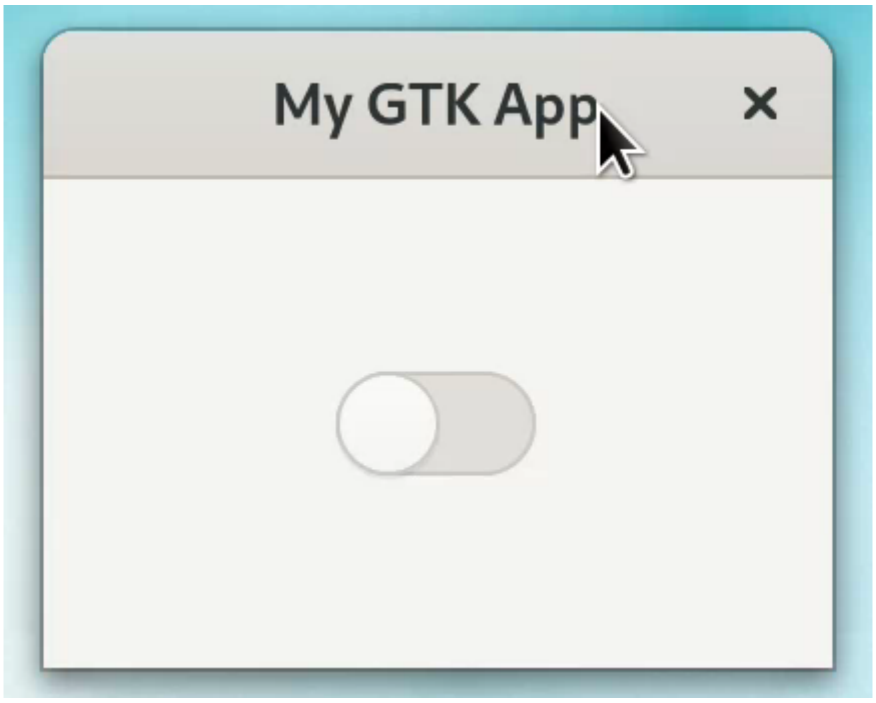
Filename: listings/settings/1/main.rs

```
    // Get the last switch state from the settings
    let is_switch_enabled = settings.boolean("is-switch-enabled");

    // Create a switch
    let switch = Switch::builder()
        .margin_top(48)
        .margin_bottom(48)
        .margin_start(48)
        .margin_end(48)
        .valign(Align::Center)
        .halign(Align::Center)
        .state(is_switch_enabled)
        .build();
```

Finally, we assure that the switch state is stored in the settings whenever we click on it.

Filename: listings/settings/1/main.rs

```
    switch.connect_state_set(move |_, is_enabled| {
        // Save changed switch state in the settings
        settings
            .set_boolean("is-switch-enabled", is_enabled)
            .expect("Could not set setting.");
        // Allow to invoke other event handlers
        glib::Propagation::Proceed
    });
```



The `Switch` now retains its state even after closing the application. But we can make this even better. The `Switch` has a property "active" and `Settings` allows us to bind properties to a specific setting. So let's do exactly that.

We can remove the `boolean` call before initializing the `Switch` as well as the `connect_state_set` call. We then bind the setting to the property by specifying the key, object and name of the property. Filename: listings/settings/2/main.rs

```
    settings
        .bind("is-switch-enabled", &switch, "active")
        .build();
```

Whenever you have a property which nicely correspond to a setting, you probably want to bind it to it. In other cases, interacting with the settings via the getter and setter methods tends to be the right choice.

Saving Window State

Quite often, we want the window state to persist between sessions. If the user resizes or maximizes the window, they might expect to find it in the same state the next time they open the app. GTK does not provide this functionality out of the box, but luckily it is not too hard to manually implement it. We basically want two integers (`height` & `width`) and a boolean (`is_maximized`) to persist. We already know how to do this by using `gio::Settings` .

Filename: listings/saving_window_state/1/org.gtk_rs.SavingWindowState1.gschema.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<schemalist>
  <schema id="org.gtk_rs.SavingWindowState1" path="/org/gtk_rs/
SavingWindowState1/">
    <key name="window-width" type="i">
      <default>-1</default>
      <summary>Default window width</summary>
    </key>
    <key name="window-height" type="i">
      <default>-1</default>
      <summary>Default window height</summary>
    </key>
    <key name="is-maximized" type="b">
      <default>false</default>
      <summary>Default window maximized behaviour</summary>
    </key>
  </schema>
</schemalist>
```

Since we don't care about intermediate state, we only load the window state when the window is constructed and save it when we close the window. That can be done by creating a custom window. First, we create one and add convenience methods for accessing settings as well as the window state.

Filename: listings/saving_window_state/1/custom_window/mod.rs

```
glib::wrapper! {
    pub struct Window(ObjectSubclass<imp::Window>)
        @extends gtk::ApplicationWindow, gtk::Window, gtk::Widget,
        @implements gio::ActionGroup, gio::ActionMap, gtk::Accessible,
gtk::Buildable,
                    gtk::ConstraintTarget, gtk::Native, gtk::Root,
gtk::ShortcutManager;
}

impl Window {
    pub fn new(app: &Application) -> Self {
        // Create new window
        Object::builder().property("application", app).build()
    }

    fn setup_settings(&self) {
        let settings = Settings::new(APP_ID);
        self.imp()
            .settings
            .set(settings)
            .expect("`settings` should not be set before calling
`setup_settings`.");
    }

    fn settings(&self) -> &Settings {
        self.imp()
            .settings
            .get()
            .expect("`settings` should be set in `setup_settings`.")
    }

    pub fn save_window_size(&self) -> Result<(), glib::BoolError> {
        // Get the size of the window
        let size = self.default_size();

        // Set the window state in `settings`
        self.settings().set_int("window-width", size.0)?;
        self.settings().set_int("window-height", size.1)?;
        self.settings()
            .set_boolean("is-maximized", self.is_maximized())?;

        Ok(())
    }

    fn load_window_size(&self) {
        // Get the window state from `settings`
        let width = self.settings().int("window-width");
        let height = self.settings().int("window-height");
        let is_maximized = self.settings().boolean("is-maximized");

        // Set the size of the window
        self.set_default_size(width, height);

        // If the window was maximized when it was closed, maximize it again
        if is_maximized {
            self.maximize();
        }
    }
}
```

> We set the property "application" by passing it to `glib::Object::new`. You can even set multiple properties that way. When creating new GObjects, this is nicer than calling the setter methods manually.

The implementation struct holds the `settings`. You can see that we embed `Settings` in `std::cell::OnceCell`. This is a nice alternative to `RefCell<Option<T>>` when you know that you will initialize the value only once.

We also override the `constructed` and `close_request` methods, where we load or save the window state.

Filename: listings/saving_window_state/1/custom_window/imp.rs

```
#[derive(Default)]
pub struct Window {
    pub settings: OnceCell<Settings>,
}

#[glib::object_subclass]
impl ObjectSubclass for Window {
    const NAME: &'static str = "MyGtkAppWindow";
    type Type = super::Window;
    type ParentType = ApplicationWindow;
}
impl ObjectImpl for Window {
    fn constructed(&self) {
        self.parent_constructed();
        // Load latest window state
        let obj = self.obj();
        obj.setup_settings();
        obj.load_window_size();
    }
}
impl WidgetImpl for Window {}
impl WindowImpl for Window {
    // Save window state right before the window will be closed
    fn close_request(&self) -> glib::Propagation {
        // Save window size
        self.obj()
            .save_window_size()
            .expect("Failed to save window state");
        // Allow to invoke other event handlers
        glib::Propagation::Proceed
    }
}
impl ApplicationWindowImpl for Window {}
```

That is it! Now our window retains its state between app sessions.

List Widgets

Sometimes you want to display a list of elements in a certain arrangement. `gtk::ListBox` and `gtk::FlowBox` are two container widgets which allow you to do this. `ListBox` describes a vertical list and `FlowBox` describes a grid.

Let's explore this concept by adding labels to a `ListBox`. Each label will display an integer starting from 0 and ranging up to 100.

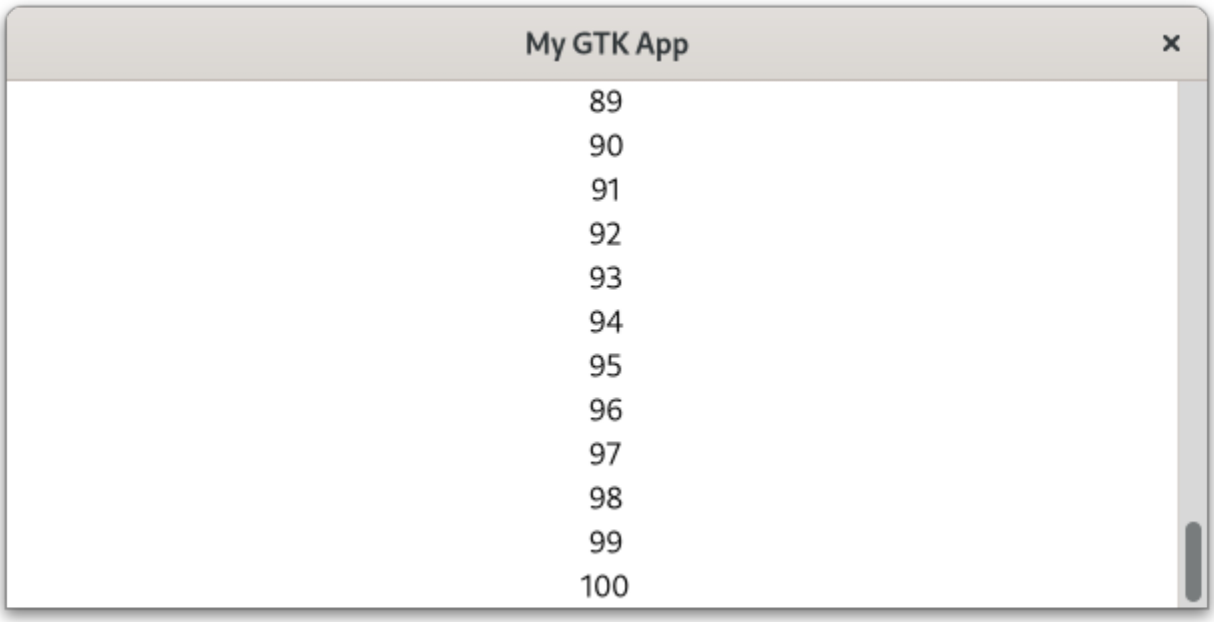
Filename: listings/list_widgets/1/main.rs

```
    // Create a `ListBox` and add labels with integers from 0 to 100
    let list_box = ListBox::new();
    for number in 0..=100 {
        let label = Label::new(Some(&number.to_string()));
        list_box.append(&label);
    }
```

We cannot display so many widgets at once. Therefore, we add `ListBox` to a `gtk::ScrolledWindow`. Now we can scroll through our elements.

Filename: listings/list_widgets/1/main.rs

```
    let scrolled_window = ScrolledWindow::builder()
        .hscrollbar_policy(PolicyType::Never) // Disable horizontal scrolling
        .min_content_width(360)
        .child(&list_box)
        .build();

    // Create a window
    let window = ApplicationWindow::builder()
        .application(app)
        .title("My GTK App")
        .default_width(600)
        .default_height(300)
        .child(&scrolled_window)
        .build();

    // Present window
    window.present();
```



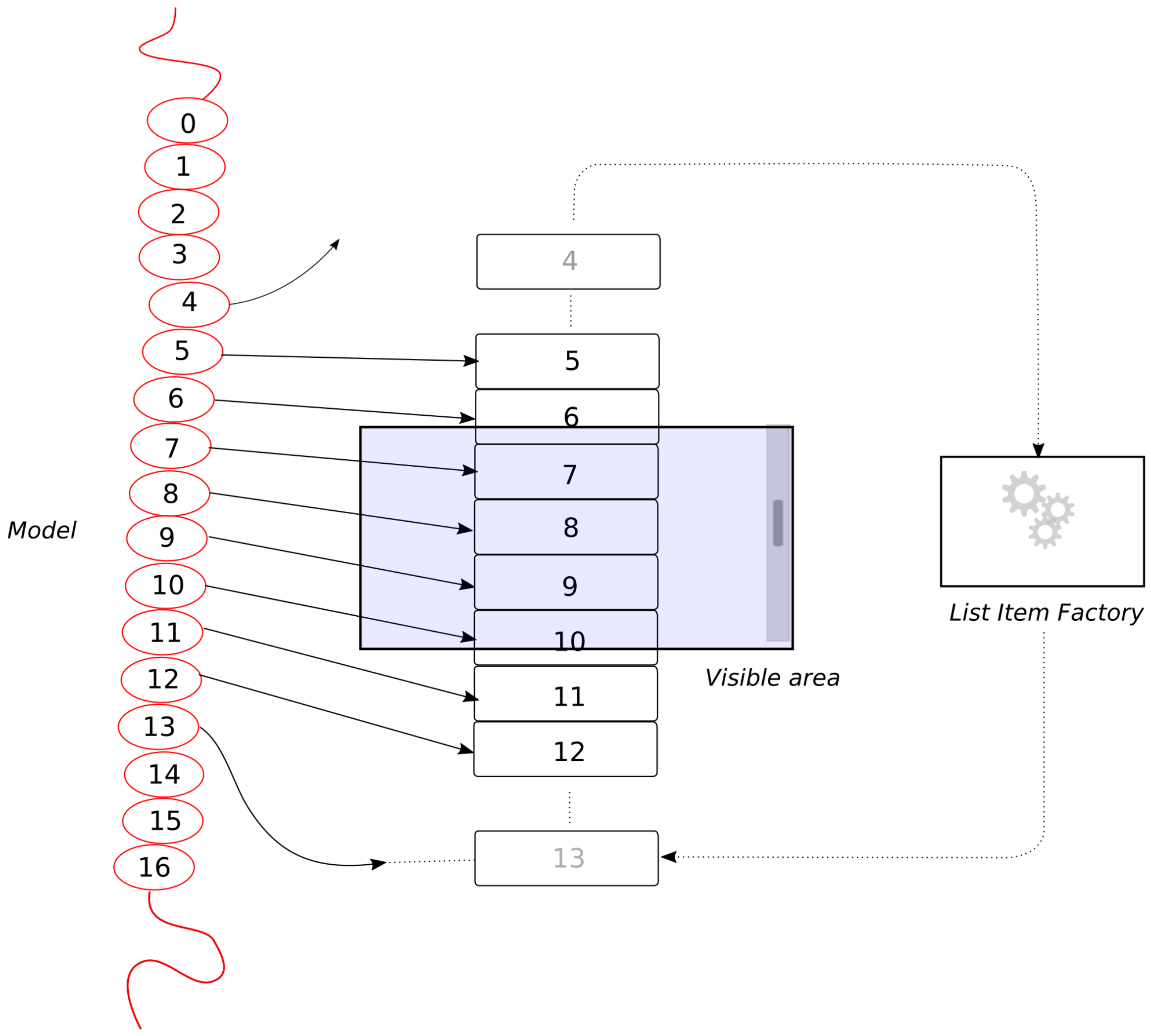
Views

That was easy enough. However, we currently create one widget per element. Since each widget takes up a bit of resources, many of them can lead to slow and unresponsive user interfaces. Depending on the widget type even thousands of elements might not be a problem. But how could we possibly deal with the infinite amount of posts in a social media timeline?

We use scalable lists instead!

- The **model** holds our data, filters it and describes its order.
- The **list item factory** defines how the data transforms into widgets.
- The **view** specifies how the widgets are then arranged.

What makes this concept scalable is that GTK only has to create slightly more widgets than we can currently look at. As we scroll through our elements, the widgets which become invisible will be reused. The following figure demonstrates how this works in practice.



100 000 elements is something `ListBox` will struggle with, so let's use this to demonstrate scalable lists.

We start by defining and filling up our model. The model is an instance of `gio::ListStore`. The main limitation here is that `gio::ListStore` only accepts GObjects. So let's create a custom GObject `IntegerObject` that which is initialized with a number.

Filename: listings/list_widgets/2/integer_object/mod.rs

```
glib::wrapper! {
    pub struct IntegerObject(ObjectSubclass<imp::IntegerObject>);
}

impl IntegerObject {
    pub fn new(number: i32) -> Self {
        Object::builder().property("number", number).build()
    }
}
```

This number represents the internal state of `IntegerObject`.

Filename: listings/list_widgets/2/integer_object/imp.rs

```
// Object holding the state
#[derive(Properties, Default)]
#[properties(wrapper_type = super::IntegerObject)]
pub struct IntegerObject {
    #[property(get, set)]
    number: Cell<i32>,
}
```

We now fill the model with integers from 0 to 100 000. Please note that models only takes care of the data. Neither `Label` nor any other widget is mentioned here.

Filename: listings/list_widgets/2/main.rs

```
    // Create a `Vec<IntegerObject>` with numbers from 0 to 100_000
    let vector: Vec<IntegerObject> = (0.
.=100_000).map(IntegerObject::new).collect();

    // Create new model
    let model = gio::ListStore::new::<IntegerObject>();

    // Add the vector to the model
    model.extend_from_slice(&vector);
```

The `ListItemFactory` takes care of the widgets as well as their relationship to the model. Here, we use the `SignalListItemFactory` which emits a signal for every relevant step in the life of a `ListItem`. The "setup" signal will be emitted when new widgets have to be created. We connect to it to create a `Label` for every requested widget.

Filename: listings/list_widgets/2/main.rs

```
    let factory = SignalListItemFactory::new();
    factory.connect_setup(move |_, list_item| {
        let label = Label::new(None);
        list_item
            .downcast_ref::<ListItem>()
            .expect("Needs to be ListItem")
            .set_child(Some(&label));
    });
```

In the "bind" step we bind the data in our model to the individual list items.

Filename: listings/list_widgets/2/main.rs

```
    factory.connect_bind(move |_, list_item| {
        // Get `IntegerObject` from `ListItem`
        let integer_object = list_item
            .downcast_ref::<ListItem>()
            .expect("Needs to be ListItem")
            .item()
            .and_downcast::<IntegerObject>()
            .expect("The item has to be an `IntegerObject`.");

        // Get `Label` from `ListItem`
        let label = list_item
            .downcast_ref::<ListItem>()
            .expect("Needs to be ListItem")
            .child()
            .and_downcast::<Label>()
            .expect("The child has to be a `Label`.");

        // Set "label" to "number"
        label.set_label(&integer_object.number().to_string());
    });
```

We only want single items to be selectable, so we choose `SingleSelection`. The other options would have been `MultiSelection` or `NoSelection`. Then we pass the model and the factory to the `ListView`.

Filename: listings/list_widgets/2/main.rs

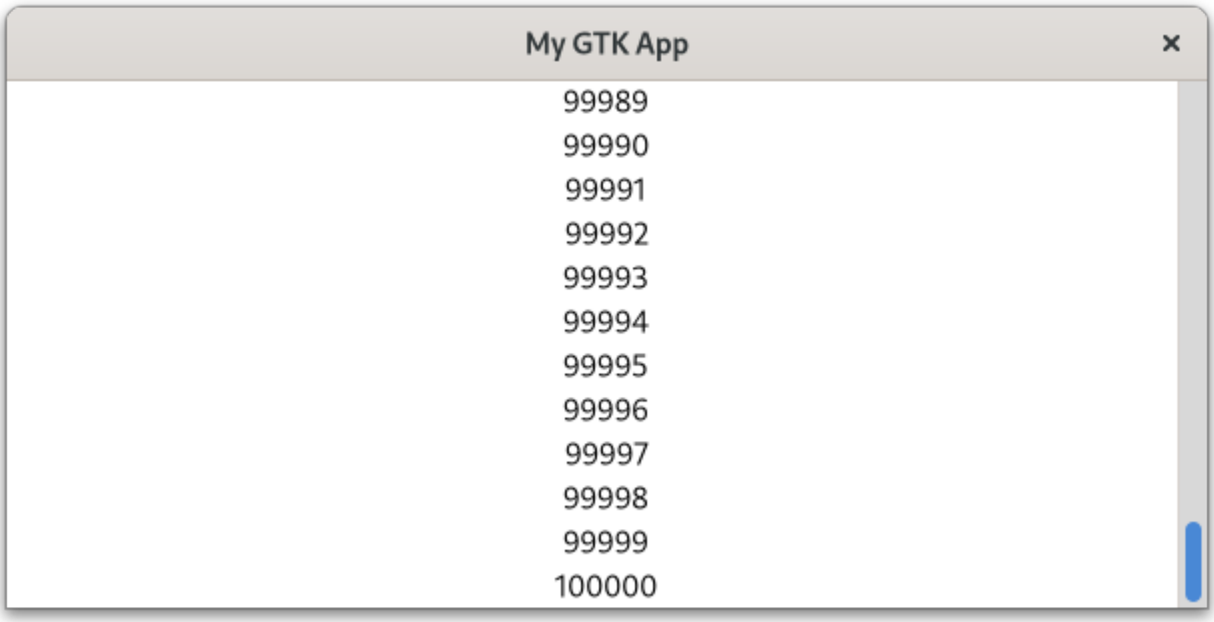
```
    let selection_model = SingleSelection::new(Some(model));
    let list_view = ListView::new(Some(selection_model), Some(factory));
```

Every `ListView` has to be a direct child of a `ScrolledWindow`, so we are adding it to one.

Filename: listings/list_widgets/2/main.rs

```
    let scrolled_window = ScrolledWindow::builder()
        .hscrollbar_policy(PolicyType::Never) // Disable horizontal scrolling
        .min_content_width(360)
        .child(&list_view)
        .build();

    // Create a window
    let window = ApplicationWindow::builder()
        .application(app)
        .title("My GTK App")
        .default_width(600)
        .default_height(300)
        .child(&scrolled_window)
        .build();

    // Present window
    window.present();
```

We can now easily scroll through our long list of integers.



99989
99990
99991
99992
99993
99994
99995
99996
99997
99998
99999
100000

Let's see what else we can do. We might want to increase the number every time we activate its row. For that we first add the method `increase_number` to our `IntegerObject`.

Filename: listings/list_widgets/3/integer_object/mod.rs

```rust
impl IntegerObject {
    pub fn new(number: i32) -> Self {
        Object::builder().property("number", number).build()
    }

    pub fn increase_number(self) {
        self.set_number(self.number() + 1);
    }
}
```

In order to interact with our `ListView`, we connect to its "activate" signal.

Filename: listings/list_widgets/3/main.rs

```
    list_view.connect_activate(move |list_view, position| {
        // Get `IntegerObject` from model
        let model = list_view.model().expect("The model has to exist.");
        let integer_object = model
            .item(position)
            .and_downcast::<IntegerObject>()
            .expect("The item has to be an `IntegerObject`.");

        // Increase "number" of `IntegerObject`
        integer_object.increase_number();
    });
```

Now every time we activate an element, for example by double-clicking on it, the corresponding "number" property of the `IntegerObject` in the model will be increased by 1. However, just because the `IntegerObject` has been modified the corresponding `Label` does not immediately change. One naive approach would be to bind the properties in the "bind" step of the `SignalListItemFactory`.

Filename: listings/list_widgets/3/main.rs

```
    factory.connect_bind(move |_, list_item| {
        // Get `IntegerObject` from `ListItem`
        let integer_object = list_item
            .downcast_ref::<ListItem>()
            .expect("Needs to be ListItem")
            .item()
            .and_downcast::<IntegerObject>()
            .expect("The item has to be an `IntegerObject`.");

        // Get `Label` from `ListItem`
        let label = list_item
            .downcast_ref::<ListItem>()
            .expect("Needs to be ListItem")
            .child()
            .and_downcast::<Label>()
            .expect("The child has to be a `Label`.");

        // Bind "label" to "number"
        integer_object
            .bind_property("number", &label, "label")
            .sync_create()
            .build();
    });
```

At first glance, that seems to work. However, as you scroll around and activate a few list elements, you will notice that sometimes multiple numbers change even though you only activated a single one. This relates to how the view works internally. Not every model item belongs to a single widget, but the widgets get recycled instead as you scroll through the view. That also means that in our case, multiple numbers will be bound to the same widget.

Expressions

Situations like these are so common that GTK offers an alternative to property binding: expressions. As a first step it allows us to remove the "bind" step. Let's see how the "setup" step now works.

Filename: listings/list_widgets/4/main.rs

```
    factory.connect_setup(move |_, list_item| {
        // Create label
        let label = Label::new(None);
        let list_item = list_item
            .downcast_ref::<ListItem>()
            .expect("Needs to be ListItem");
        list_item.set_child(Some(&label));

        // Bind `list_item->item->number` to `label->label`
        list_item
            .property_expression("item")
            .chain_property::<IntegerObject>("number")
            .bind(&label, "label", Widget::NONE);
    });
```

An expression provides a way to describe references to values. One interesting part here is that these references can be several steps away. This allowed us in the snippet above to bind the property "number" of the property "item" of `list_item` to the property "label" of `label`.

It is also worth noting that at the "setup" stage there is no way of knowing which list item belongs to which label, simply because this changes as we scroll through the list. Here, another power of expressions becomes evident. Expressions allow us to describe relationships between objects or properties that might not even exist yet. We just had to tell it to change the label whenever the number that belongs to it changes. That way, we also don't face the problem that multiple labels are bound to the same number. When we now activate a label, only the corresponding number visibly changes.

Let's extend our app a bit more. We can, for example, filter our model to only allow even numbers. We do that by passing it to a `gtk::FilterListModel` together with a `gtk::CustomFilter`

Filename: listings/list_widgets/5/main.rs

```
    let filter = CustomFilter::new(move |obj| {
        // Get `IntegerObject` from `glib::Object`
        let integer_object = obj
            .downcast_ref::<IntegerObject>()
            .expect("The object needs to be of type `IntegerObject`.");

        // Only allow even numbers
        integer_object.number() % 2 == 0
    });
    let filter_model = FilterListModel::new(Some(model),
Some(filter.clone()));
```

Additionally, we can reverse the order of our model. Now we pass the filtered model to `gtk::SortListModel` together with `gtk::CustomSorter`.

Filename: listings/list_widgets/5/main.rs

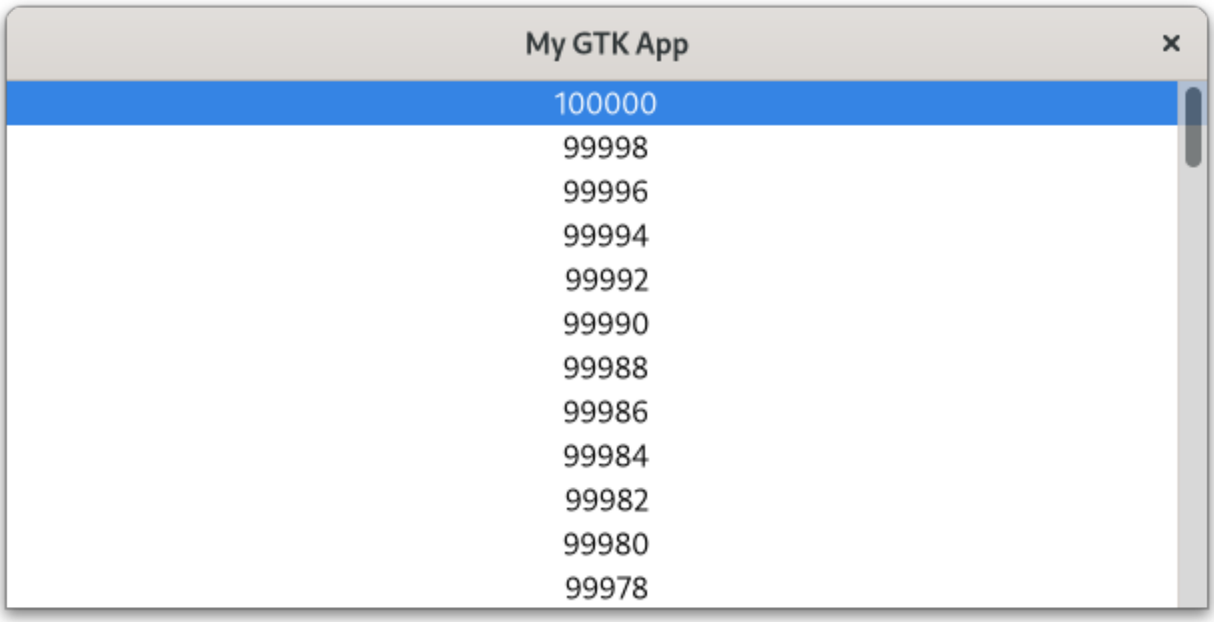
```
    let sorter = CustomSorter::new(move |obj1, obj2| {
        // Get `IntegerObject` from `glib::Object`
        let integer_object_1 = obj1
            .downcast_ref::<IntegerObject>()
            .expect("The object needs to be of type `IntegerObject`.");
        let integer_object_2 = obj2
            .downcast_ref::<IntegerObject>()
            .expect("The object needs to be of type `IntegerObject`.");

        // Get property "number" from `IntegerObject`
        let number_1 = integer_object_1.number();
        let number_2 = integer_object_2.number();

        // Reverse sorting order -> large numbers come first
        number_2.cmp(&number_1).into()
    });
    let sort_model = SortListModel::new(Some(filter_model),
Some(sorter.clone()));
```

To ensure that our filter and sorter get updated when we modify the numbers, we call the `changed` method on them.

Filename: listings/list_widgets/5/main.rs

```
    list_view.connect_activate(move |list_view, position| {
        // Get `IntegerObject` from model
        let model = list_view.model().expect("The model has to exist.");
        let integer_object = model
            .item(position)
            .and_downcast::<IntegerObject>()
            .expect("The item has to be an `IntegerObject`.");

        // Increase "number" of `IntegerObject`
        integer_object.increase_number();

        // Notify that the filter and sorter have been changed
        filter.changed(FilterChange::Different);
        sorter.changed(SorterChange::Different);
    });
```

After our changes, the application looks like this:



String List

Often, all you want is to display a list of strings. However, if you either need to filter and sort your displayed data or have too many elements to be displayed by `ListBox`, you will still want to use a view. GTK provides a convenient model for this use case: `gtk::StringList`.

Let's see with a small example how to use this API. Filter and sorter is controlled by the factory, so nothing changes here. This is why we will skip this topic here.

First, we add a bunch of strings to our model.

Filename: listings/list_widgets/6/main.rs

```
    // Create a `StringList` with number from 0 to 100_000
    // `StringList` implements FromIterator<String>
    let model: StringList = (0..=100_000).map(|number|
number.to_string()).collect();
```

Note that we can create a `StringList` directly from an iterator over strings. This means we don't have to create a custom GObject for our model anymore.

As usual, we connect the label to the list item via an expression. Here we can use `StringObject`, which exposes its content via the property "string".

Filename: listings/list_widgets/6/main.rs

```
    factory.connect_setup(move |_, list_item| {
        // Create label
        let label = Label::new(None);
        let list_item = list_item
            .downcast_ref::<ListItem>()
            .expect("Needs to be ListItem");
        list_item.set_child(Some(&label));

        // Bind `list_item->item->string` to `label->label`
        list_item
            .property_expression("item")
            .chain_property::<StringObject>("string")
            .bind(&label, "label", Widget::NONE);
    });
```

Conclusion

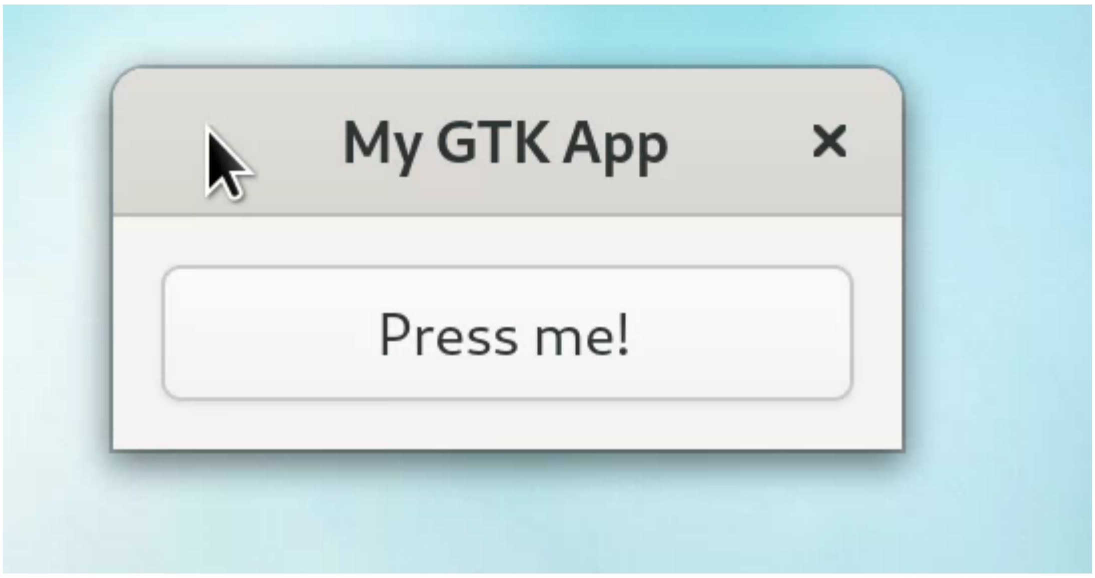
We now know how to display a list of data. Small amount of elements can be handled by `ListBox` or `FlowBox`. These widgets are easy to use and can, if necessary, be bound to a model such as `gio::ListStore`. Their data can then be modified, sorted and filtered more easily. However, if we need the widgets to be scalable, we still need to use `ListView`, `ColumnView` or `GridView` instead.

Composite Templates

Until now, whenever we constructed pre-defined widgets we relied on the builder pattern. As a reminder, that is how we used it to build our trusty "Hello World!" app.

Filename: listings/hello_world/3/main.rs

```
use gtk::prelude::*;
use gtk::{glib, Application, ApplicationWindow, Button};
const APP_ID: &str = "org.gtk_rs.HelloWorld3";

fn main() -> glib::ExitCode {
    // Create a new application
    let app = Application::builder().application_id(APP_ID).build();

    // Connect to "activate" signal of `app`
    app.connect_activate(build_ui);

    // Run the application
    app.run()
}

fn build_ui(app: &Application) {
    // Create a button with label and margins
    let button = Button::builder()
        .label("Press me!")
        .margin_top(12)
        .margin_bottom(12)
        .margin_start(12)
        .margin_end(12)
        .build();

    // Connect to "clicked" signal of `button`
    button.connect_clicked(|button| {
        // Set the label to "Hello World!" after the button has been clicked
on
        button.set_label("Hello World!");
    });

    // Create a window
    let window = ApplicationWindow::builder()
        .application(app)
        .title("My GTK App")
        .child(&button)
        .build();

    // Present window
    window.present();
}
```



Creating widgets directly from code is fine, but it makes it harder to separate the logic from the user interface. This is why most toolkits allow to describe the user interface with a markup language and GTK is no exception here. For example the following `xml` file describes the window widget of the "Hello World!" app.

Filename: listings/composite_templates/1/resources/window.ui

```xml
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <template class="MyGtkAppWindow" parent="GtkApplicationWindow">
    <property name="title">My GTK App</property>
    <child>
      <object class="GtkButton" id="button">
        <property name="label">Press me!</property>
        <property name="margin-top">12</property>
        <property name="margin-bottom">12</property>
        <property name="margin-start">12</property>
        <property name="margin-end">12</property>
      </object>
    </child>
  </template>
</interface>
```

The most outer tag always has to be the `<interface>`. Then you start listing the elements you want to describe. In order to define a composite template, we specify the name `MyGtkAppWindow` of the custom widget we want to create and the parent `gtk::ApplicationWindow` it derives of. These `xml` files are independent of the programming language, which is why the classes have the original names. Luckily, they all convert like this: `gtk::ApplicationWindow` → `GtkApplicationWindow`. Then we can specify properties which are listed here for `ApplicationWindow`. Since `ApplicationWindow` can contain other widgets we use the `<child>` tag to add a `gtk::Button`. We want to be able to refer to the button later on so we also set its `id`.

Resources

In order to embed the template file into our application we take advantage of `gio::Resource`. The files to embed are again described by an `xml` file. For our template file we also add the `compressed` and `preprocess` attribute in order to reduce the final size of the resources.

Filename: listings/composite_templates/1/resources/resources.gresource.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<gresources>
  <gresource prefix="/org/gtk_rs/example/">
    <file compressed="true" preprocess="xml-stripblanks">window.ui</file>
  </gresource>
</gresources>
```

Now we have to compile the resources and link it to our application. One way to do this is to execute `glib_build_tools::compile_resources` within a cargo build script.

First, we have to add `glib-build-tools` as build dependency in `Cargo.toml` by executing:

```
cargo add glib-build-tools --build
```

Then, we create a `build.rs` at the root of our package with the following content. This will compile the resources whenever we trigger a build with cargo and then statically link our executable to them.

Filename: listings/build.rs

```
fn main() {
    glib_build_tools::compile_resources(
        &["composite_templates/1/resources"],
        "composite_templates/1/resources/resources.gresource.xml",
        "composite_templates_1.gresource",
    );
}
```

Finally, we register and include the resources by calling the macro `gio::resources_register_include!`. In your own apps take care to register the resources before creating the `gtk::Application`.

Filename: listings/composite_templates/1/main.rs

```
mod window;

use gtk::prelude::*;
use gtk::{gio, glib, Application};
use window::Window;

const APP_ID: &str = "org.gtk_rs.CompositeTemplates1";

fn main() -> glib::ExitCode {
    // Register and include resources
    gio::resources_register_include!("composite_templates_1.gresource")
        .expect("Failed to register resources.");

    // Create a new application
    let app = Application::builder().application_id(APP_ID).build();

    // Connect to "activate" signal of `app`
    app.connect_activate(build_ui);

    // Run the application
    app.run()
}
fn build_ui(app: &Application) {
    // Create new window and present it
    let window = Window::new(app);
    window.present();
}
```

Within our code we create a custom widget inheriting from `gtk::ApplicationWindow` to make use of our template. Filename: listings/composite_templates/1/window/mod.rs

```
mod imp;

use glib::Object;
use gtk::{gio, glib, Application};

glib::wrapper! {
    pub struct Window(ObjectSubclass<imp::Window>)
        @extends gtk::ApplicationWindow, gtk::Window, gtk::Widget,
        @implements gio::ActionGroup, gio::ActionMap, gtk::Accessible,
gtk::Buildable,
                    gtk::ConstraintTarget, gtk::Native, gtk::Root,
gtk::ShortcutManager;
}

impl Window {
    pub fn new(app: &Application) -> Self {
        // Create new window
        Object::builder().property("application", app).build()
    }
}
```

In the implementation struct, we then add the derive macro `gtk::CompositeTemplate`.

We also specify that the template information comes from a resource of prefix `/org/gtk-rs/example` containing a file `window.ui`.

One very convenient feature of templates is the template child. You use it by adding a struct member with the same name as one `id` attribute in the template. `TemplateChild` then stores a reference to the widget for later use. This will be useful later, when we want to add a callback to our button.

Filename: listings/composite_templates/1/window/imp.rs

```
// Object holding the state
#[derive(CompositeTemplate, Default)]
#[template(resource = "/org/gtk_rs/example/window.ui")]
pub struct Window {
    #[template_child]
    pub button: TemplateChild<Button>,
}
```

Within the `ObjectSubclass` trait, we make sure that `NAME` corresponds to `class` in the template and `ParentType` corresponds to `parent` in the template. We also bind and initialize the template in `class_init` and `instance_init`.

Filename: listings/composite_templates/1/window/imp.rs

```
// The central trait for subclassing a GObject
#[glib::object_subclass]
impl ObjectSubclass for Window {
    // `NAME` needs to match `class` attribute of template
    const NAME: &'static str = "MyGtkAppWindow";
    type Type = super::Window;
    type ParentType = gtk::ApplicationWindow;

    fn class_init(klass: &mut Self::Class) {
        klass.bind_template();
    }

    fn instance_init(obj: &InitializingObject<Self>) {
        obj.init_template();
    }
}
```

Finally, we connect the callback to the "clicked" signal of `button` within `constructed`. The button is easily available thanks to the stored reference in `self`.

Filename: listings/composite_templates/1/window/imp.rs

```
// Trait shared by all GObjects
impl ObjectImpl for Window {
    fn constructed(&self) {
        // Call "constructed" on parent
        self.parent_constructed();

        // Connect to "clicked" signal of `button`
        self.button.connect_clicked(move |button| {
            // Set the label to "Hello World!" after the button has been
clicked on
            button.set_label("Hello World!");
        });
    }
}
```

Custom Widgets

We can also instantiate custom widgets within a template file. First we define `CustomButton` that inherits from `gtk::Button`. As usual, we define the implementation struct within `imp.rs`. Note the `NAME` we define here, we will need it later to refer to it in the template.

Filename: listings/composite_templates/2/custom_button/imp.rs

```
// Object holding the state
#[derive(Default)]
pub struct CustomButton;

// The central trait for subclassing a GObject
#[glib::object_subclass]
impl ObjectSubclass for CustomButton {
    const NAME: &'static str = "MyGtkAppCustomButton";
    type Type = super::CustomButton;
    type ParentType = gtk::Button;
}
```

We also define the public struct in `mod.rs`.

Filename: listings/composite_templates/2/custom_button/mod.rs

```
mod imp;

glib::wrapper! {
    pub struct CustomButton(ObjectSubclass<imp::CustomButton>)
        @extends gtk::Button, gtk::Widget,
        @implements gtk::Accessible, gtk::Actionable,
                    gtk::Buildable, gtk::ConstraintTarget;
}
```

Since we want to refer to a `CustomButton` now we also have to change the type of the template child to it.

Filename: listings/composite_templates/2/window/imp.rs

```
// Object holding the state
#[derive(CompositeTemplate, Default)]
#[template(resource = "/org/gtk_rs/example/window.ui")]
pub struct Window {
    #[template_child]
    pub button: TemplateChild<CustomButton>,
}
```

Finally, we can replace `GtkButton` with `MyGtkAppCustomButton` within our composite template. Since the custom button is a direct subclass of `gtk::Button` without any modifications, the behavior of our app stays the same.

Filename: listings/composite_templates/2/resources/window.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <template class="MyGtkAppWindow" parent="GtkApplicationWindow">
    <property name="title">My GTK App</property>
    <child>
      <object class="MyGtkAppCustomButton" id="button">
        <property name="label">Press me!</property>
        <property name="margin-top">12</property>
        <property name="margin-bottom">12</property>
        <property name="margin-start">12</property>
        <property name="margin-end">12</property>
      </object>
    </child>
  </template>
</interface>
```

Template Callbacks

We can even specify the handlers of signals within composite templates. This can be done with a `<signal>` tag containing the name of the signal and the handler in our Rust code.

Filename: listings/composite_templates/3/resources/window.ui

```xml
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <template class="MyGtkAppWindow" parent="GtkApplicationWindow">
    <property name="title">My GTK App</property>
    <child>
      <object class="MyGtkAppCustomButton" id="button">
        <signal name="clicked" handler="handle_button_clicked"/>
        <property name="label">Press me!</property>
        <property name="margin-top">12</property>
        <property name="margin-bottom">12</property>
        <property name="margin-start">12</property>
        <property name="margin-end">12</property>
      </object>
    </child>
  </template>
</interface>
```

Then we define the `handle_button_clicked` with the `template_callbacks` macro applied to it. We can determine the function signature by having a look at the `connect_*` method of the signal we want to handle. In our case that would be `connect_clicked`. It takes a function of type `Fn(&Self)`. `Self` refers to our button. This means that `handle_button_clicked` has a single parameter of type `&CustomButton`.

Filename: listings/composite_templates/3/window/imp.rs

```rust
#[gtk::template_callbacks]
impl Window {
    #[template_callback]
    fn handle_button_clicked(button: &CustomButton) {
        // Set the label to "Hello World!" after the button has been clicked
on
        button.set_label("Hello World!");
    }
}
```

Then we have to bind the template callbacks with `bind_template_callbacks`. We also need to remove the `button.connect_clicked` callback implemented in `window/imp.rs`.

Filename: listings/composite_templates/3/window/imp.rs

```
// The central trait for subclassing a GObject
#[glib::object_subclass]
impl ObjectSubclass for Window {
    // `NAME` needs to match `class` attribute of template
    const NAME: &'static str = "MyGtkAppWindow";
    type Type = super::Window;
    type ParentType = gtk::ApplicationWindow;

    fn class_init(klass: &mut Self::Class) {
        klass.bind_template();
        klass.bind_template_callbacks();
    }

    fn instance_init(obj: &InitializingObject<Self>) {
        obj.init_template();
    }
}
```

We can also access the state of our widget. Let's say we want to manipulate a `number` stored in `imp::Window`.

Filename: listings/composite_templates/4/window/imp.rs

```
// Object holding the state
#[derive(CompositeTemplate, Default)]
#[template(resource = "/org/gtk_rs/example/window.ui")]
pub struct Window {
    #[template_child]
    pub button: TemplateChild<CustomButton>,
    pub number: Cell<i32>,
}
```

In order to access the widget's state we have to add `swapped="true"` to the `signal` tag.

Filename: listings/composite_templates/4/resources/window.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <template class="MyGtkAppWindow" parent="GtkApplicationWindow">
    <property name="title">My GTK App</property>
    <child>
      <object class="MyGtkAppCustomButton" id="button">
        <signal name="clicked" handler="handle_button_clicked"
swapped="true"/>
        <property name="label">Press me!</property>
        <property name="margin-top">12</property>
        <property name="margin-bottom">12</property>
        <property name="margin-start">12</property>
        <property name="margin-end">12</property>
      </object>
    </child>
  </template>
</interface>
```

Now we can add `&self` as first parameter to `handle_button_clicked`. This lets us access the state of the window and therefore manipulate `number`.

Filename: listings/composite_templates/4/window/imp.rs

```
#[gtk::template_callbacks]
impl Window {
    #[template_callback]
    fn handle_button_clicked(&self, button: &CustomButton) {
        let number_increased = self.number.get() + 1;
        self.number.set(number_increased);
        button.set_label(&number_increased.to_string())
    }
}
```

Registering Types

Now that we use template callbacks we don't access the template child anymore. Let's remove it.

Filename: listings/composite_templates/5/window/imp.rs

```
// Object holding the state
#[derive(CompositeTemplate, Default)]
#[template(resource = "/org/gtk_rs/example/window.ui")]
pub struct Window {
    pub number: Cell<i32>,
}
```

However, when we now run it GTK doesn't see `MyGtkAppCustomButton` as valid object type anymore. So what happened here?

```
Gtk-CRITICAL **: Error building template class 'MyGtkAppWindow' for an
instance of
                  type 'MyGtkAppWindow': Invalid object type
'MyGtkAppCustomButton'
```

Turns out adding a template child not only gives a convenient reference to a widget within the template. It also ensures that the widget type is registered. Luckily we can also do that by ourselves.

Filename: listings/composite_templates/6/window/imp.rs

```rust
// The central trait for subclassing a GObject
#[glib::object_subclass]
impl ObjectSubclass for Window {
    // `NAME` needs to match `class` attribute of template
    const NAME: &'static str = "MyGtkAppWindow";
    type Type = super::Window;
    type ParentType = gtk::ApplicationWindow;

    fn class_init(klass: &mut Self::Class) {
        // Register `CustomButton`
        CustomButton::ensure_type();

        klass.bind_template();
        klass.bind_template_callbacks();
    }

    fn instance_init(obj: &InitializingObject<Self>) {
        obj.init_template();
    }
}
```

We call the `ensure_type` method within `class_init` and voilà: our app works again.

Conclusion

Thanks to custom widgets we can

- keep state and part of it as properties,
- add signals as well as
- override behavior.

Thanks to composite templates we can

- describe complex user interfaces concisely,
- easily access widgets within the template as well as

- specify handler functions for signals.

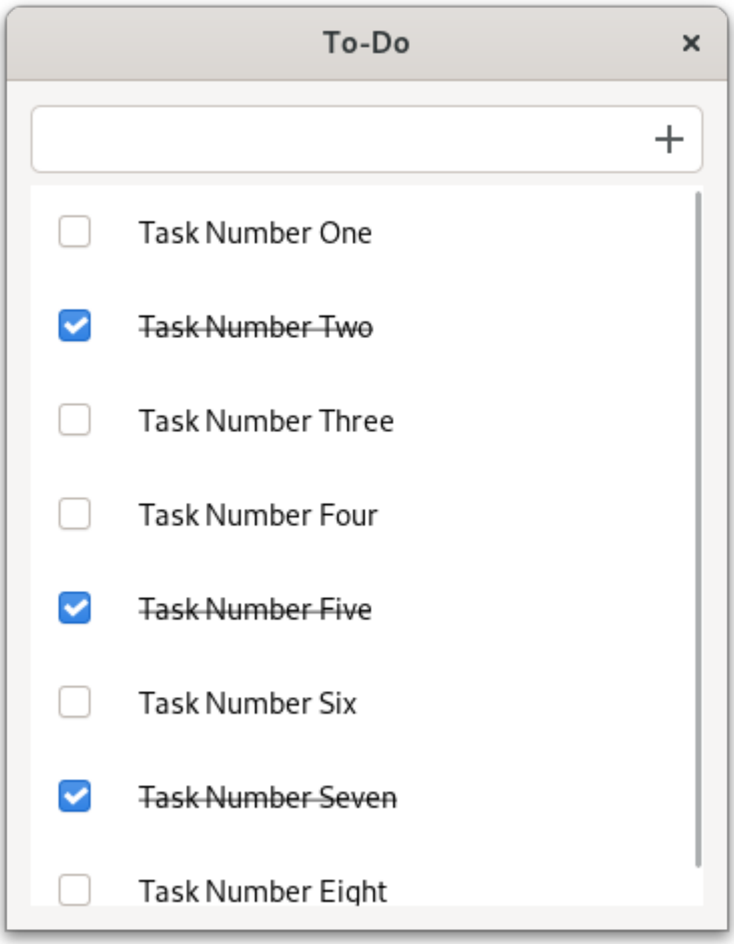
The API involved here is extensive so especially at the beginning you will want to check out the documentation. The basic syntax of the `ui` files is explained within `Builder`, syntax specific to widgets within `Widget`. If a certain widget accepts additional element, then they are typically explained in the docs of the widget.

In the following chapter, we will see how composite templates help us to create slightly bigger apps such as a To-Do app.

Building a Simple To-Do App

After we have learned so many concepts, it is finally time to put them into practice. We are going to build a To-Do app!

For now, we would already be satisfied with a minimal version. An entry to input new tasks and a list view to display them will suffice. Something like this:



Window

This mockup can be described by the following composite template.

Filename: listings/todo/1/resources/window.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <template class="TodoWindow" parent="GtkApplicationWindow">
    <property name="width-request">360</property>
    <property name="title" translatable="yes">To-Do</property>
    <child>
      <object class="GtkBox">
        <property name="orientation">vertical</property>
        <property name="margin-top">12</property>
        <property name="margin-bottom">12</property>
        <property name="margin-start">12</property>
        <property name="margin-end">12</property>
        <property name="spacing">6</property>
        <child>
          <object class="GtkEntry" id="entry">
            <property name="placeholder-text" translatable="yes">Enter a
Task…</property>
            <property name="secondary-icon-name">list-add-symbolic</property>
          </object>
        </child>
        <child>
          <object class="GtkScrolledWindow">
            <property name="hscrollbar-policy">never</property>
            <property name="min-content-height">360</property>
            <property name="vexpand">true</property>
            <child>
              <object class="GtkListView" id="tasks_list">
                <property name="valign">start</property>
              </object>
            </child>
          </object>
        </child>
      </object>
    </child>
  </template>
</interface>
```

In order to use the composite template, we create a custom widget. The `parent` is `gtk::ApplicationWindow`, so we inherit from it. As usual, we have to list all ancestors and interfaces apart from `GObject` and `GInitiallyUnowned`.

Filename: listings/todo/1/window/mod.rs

```
glib::wrapper! {
    pub struct Window(ObjectSubclass<imp::Window>)
        @extends gtk::ApplicationWindow, gtk::Window, gtk::Widget,
        @implements gio::ActionGroup, gio::ActionMap, gtk::Accessible,
gtk::Buildable,
                    gtk::ConstraintTarget, gtk::Native, gtk::Root,
gtk::ShortcutManager;
}
```

Then we initialize the composite template for `imp::Window`. We store references to the entry, the list view as well as the list model. This will come in handy when we later add methods to our window. After that, we add the typical boilerplate for initializing composite templates. We only have to assure that the `class` attribute of the template in `window.ui` matches `NAME`.

Filename: listings/todo/1/window/imp.rs

```rust
// Object holding the state
#[derive(CompositeTemplate, Default)]
#[template(resource = "/org/gtk_rs/Todo1/window.ui")]
pub struct Window {
    #[template_child]
    pub entry: TemplateChild<Entry>,
    #[template_child]
    pub tasks_list: TemplateChild<ListView>,
    pub tasks: RefCell<Option<gio::ListStore>>,
}

// The central trait for subclassing a GObject
#[glib::object_subclass]
impl ObjectSubclass for Window {
    // `NAME` needs to match `class` attribute of template
    const NAME: &'static str = "TodoWindow";
    type Type = super::Window;
    type ParentType = gtk::ApplicationWindow;

    fn class_init(klass: &mut Self::Class) {
        klass.bind_template();
    }

    fn instance_init(obj: &InitializingObject<Self>) {
        obj.init_template();
    }
}
```

`main.rs` also does not hold any surprises for us.

Filename: listings/todo/1/main.rs

```rust
fn main() -> glib::ExitCode {
    // Register and include resources
    gio::resources_register_include!("todo_1.gresource")
        .expect("Failed to register resources.");

    // Create a new application
    let app = Application::builder()
        .application_id("org.gtk_rs.Todo1")
        .build();

    // Connect to "activate" signal of `app`
    app.connect_activate(build_ui);

    // Run the application
    app.run()
}

fn build_ui(app: &Application) {
    // Create a new custom window and present it
    let window = Window::new(app);
    window.present();
}
```

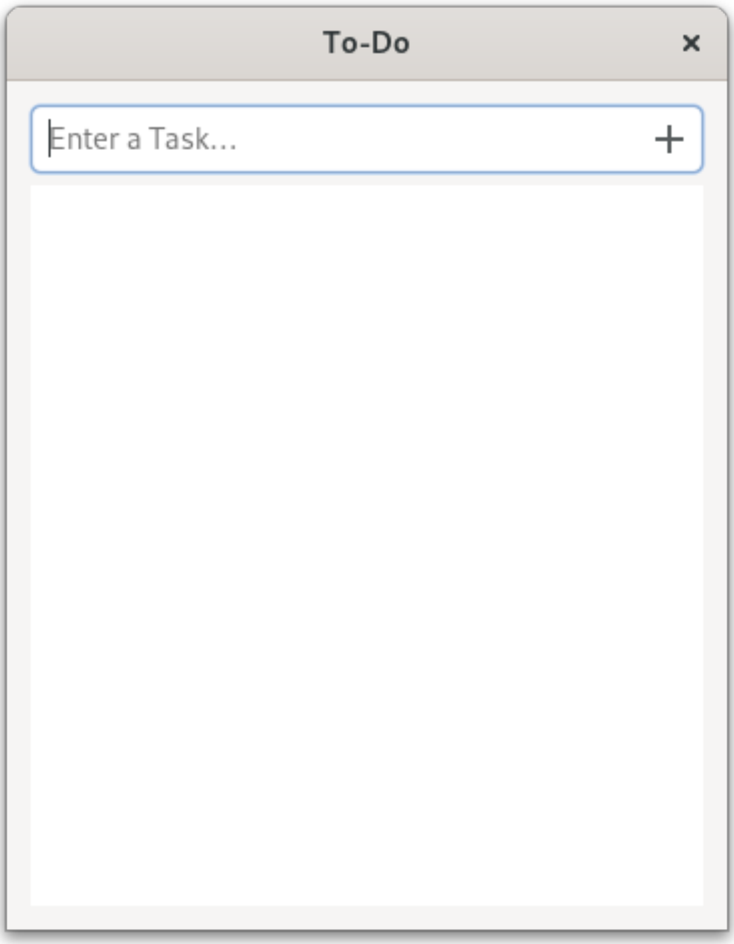
Finally, we specify our resources. Here, they already include `task_row.ui` which we will handle later in this chapter.

Filename: listings/todo/1/resources/resources.gresource.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<gresources>
  <gresource prefix="/org/gtk_rs/Todo1/">
    <file compressed="true" preprocess="xml-stripblanks">task_row.ui</file>
    <file compressed="true" preprocess="xml-stripblanks">window.ui</file>
  </gresource>
</gresources>
```

Task Object

So far so good. The main user interface is done, but the entry does not react to input yet. Also, where would the input go? We haven't even set up the list model yet. Let's do that!



As discussed in the list widgets chapter, we start out by creating a custom GObject. This object will store the state of the task consisting of:

- a boolean describing whether the task is completed or not, and
- a string holding the task name.

Filename: listings/todo/1/task_object/mod.rs

```rust
glib::wrapper! {
    pub struct TaskObject(ObjectSubclass<imp::TaskObject>);
}

impl TaskObject {
    pub fn new(completed: bool, content: String) -> Self {
        Object::builder()
            .property("completed", completed)
            .property("content", content)
            .build()
    }
}
```

Unlike the lists chapter, the state is stored in a struct rather than in individual members of `imp::TaskObject`. This will be very convenient when saving the state in one of the following chapters.

Filename: listings/todo/1/task_object/mod.rs

```
#[derive(Default)]
pub struct TaskData {
    pub completed: bool,
    pub content: String,
}
```

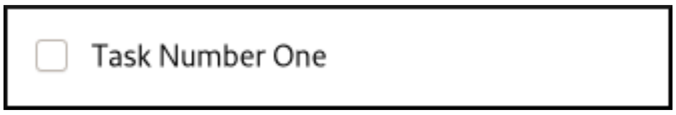
We are going to expose `completed` and `content` as properties. Since the data is now inside a struct rather than individual member variables we have to add more annotations. For each property we additionally specify the name, the type and which member variable of `TaskData` we want to access.

Filename: listings/todo/1/task_object/imp.rs

```
// Object holding the state
#[derive(Properties, Default)]
#[properties(wrapper_type = super::TaskObject)]
pub struct TaskObject {
    #[property(name = "completed", get, set, type = bool, member =
completed)]
    #[property(name = "content", get, set, type = String, member = content)]
    pub data: RefCell<TaskData>,
}

// The central trait for subclassing a GObject
#[glib::object_subclass]
impl ObjectSubclass for TaskObject {
    const NAME: &'static str = "TodoTaskObject";
    type Type = super::TaskObject;
}

// Trait shared by all GObjects
#[glib::derived_properties]
impl ObjectImpl for TaskObject {}
```

Task Row

Let's move on to the individual tasks. The row of a task should look like this:



Again, we describe the mockup with a composite template.

Filename: listings/todo/1/resources/task_row.ui

```xml
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <template class="TodoTaskRow" parent="GtkBox">
    <child>
      <object class="GtkCheckButton" id="completed_button">
        <property name="margin-top">12</property>
        <property name="margin-bottom">12</property>
        <property name="margin-start">12</property>
        <property name="margin-end">12</property>
      </object>
    </child>
    <child>
      <object class="GtkLabel" id="content_label">
        <property name="margin-top">12</property>
        <property name="margin-bottom">12</property>
        <property name="margin-start">12</property>
        <property name="margin-end">12</property>
      </object>
    </child>
  </template>
</interface>
```

In the code, we derive `TaskRow` from `gtk:Box` :

Filename: listings/todo/1/task_row/mod.rs

```rust
glib::wrapper! {
    pub struct TaskRow(ObjectSubclass<imp::TaskRow>)
    @extends gtk::Box, gtk::Widget,
    @implements gtk::Accessible, gtk::Buildable, gtk::ConstraintTarget,
gtk::Orientable;
}
```

In `imp::TaskRow` , we hold references to `completed_button` and `content_label` . We also store a mutable vector of bindings. Why we need that will become clear as soon as we get to bind the state of `TaskObject` to the corresponding `TaskRow` .

Filename: listings/todo/1/task_row/imp.rs

```
// Object holding the state
#[derive(Default, CompositeTemplate)]
#[template(resource = "/org/gtk_rs/Todo1/task_row.ui")]
pub struct TaskRow {
    #[template_child]
    pub completed_button: TemplateChild<CheckButton>,
    #[template_child]
    pub content_label: TemplateChild<Label>,
    // Vector holding the bindings to properties of `TaskObject`
    pub bindings: RefCell<Vec<Binding>>,
}

// The central trait for subclassing a GObject
#[glib::object_subclass]
impl ObjectSubclass for TaskRow {
    // `NAME` needs to match `class` attribute of template
    const NAME: &'static str = "TodoTaskRow";
    type Type = super::TaskRow;
    type ParentType = gtk::Box;

    fn class_init(klass: &mut Self::Class) {
        klass.bind_template();
    }

    fn instance_init(obj: &glib::subclass::InitializingObject<Self>) {
        obj.init_template();
    }
}
```

Now we can bring everything together. We override the `imp::Window::constructed` in order to set up window contents at the time of its construction.

Filename: listings/todo/1/window/imp.rs

```
// Trait shared by all GObjects
impl ObjectImpl for Window {
    fn constructed(&self) {
        // Call "constructed" on parent
        self.parent_constructed();

        // Setup
        let obj = self.obj();
        obj.setup_tasks();
        obj.setup_callbacks();
        obj.setup_factory();
    }
}
```

Since we need to access the list model quite often, we add the convenience method `Window::model` for that. In `Window::setup_tasks` we create a new model. Then we store a reference to the model in `imp::Window` as well as in `gtk::ListView`.

Filename: listings/todo/1/window/mod.rs

```
    fn tasks(&self) -> gio::ListStore {
        // Get state
        self.imp()
            .tasks
            .borrow()
            .clone()
            .expect("Could not get current tasks.")
    }

    fn setup_tasks(&self) {
        // Create new model
        let model = gio::ListStore::new::<TaskObject>();

        // Get state and set model
        self.imp().tasks.replace(Some(model));

        // Wrap model with selection and pass it to the list view
        let selection_model = NoSelection::new(Some(self.tasks()));
        self.imp().tasks_list.set_model(Some(&selection_model));
    }
```

We also create a method `new_task` which takes the content of the entry, clears the entry and uses the content to create a new task.

Filename: listings/todo/1/window/mod.rs

```
    fn new_task(&self) {
        // Get content from entry and clear it
        let buffer = self.imp().entry.buffer();
        let content = buffer.text().to_string();
        if content.is_empty() {
            return;
        }
        buffer.set_text("");

        // Add new task to model
        let task = TaskObject::new(false, content);
        self.tasks().append(&task);
    }
```

In `Window::setup_callbacks` we connect to the "activate" signal of the entry. This signal is triggered when we press the enter key in the entry. Then a new `TaskObject` with the content will be created and appended to the model. Finally, the entry will be cleared.

Filename: listings/todo/1/window/mod.rs

```
    fn setup_callbacks(&self) {
        // Setup callback for activation of the entry
        self.imp()
            .entry
            .connect_activate(clone!(@weak self as window => move |_| {
                window.new_task();
            }));

        // Setup callback for clicking (and the releasing) the icon of the
entry
        self.imp().entry.connect_icon_release(
            clone!(@weak self as window => move |_,_| {
                window.new_task();
            }),
        );
    }
```

The list elements for the `gtk::ListView` are produced by a factory. Before we move on to the implementation, let's take a step back and think about which behavior we expect here. `content_label` of `TaskRow` should follow `content` of `TaskObject`. We also want `completed_button` of `TaskRow` follow `completed` of `TaskObject`. This could be achieved with expressions similar to what we did in the lists chapter.

However, if we toggle the state of `completed_button` of `TaskRow`, `completed` of `TaskObject` should change too. Unfortunately, expressions cannot handle bidirectional relationships. This means we have to use property bindings. We will need to unbind them manually when they are no longer needed.

We will create empty `TaskRow` objects in the "setup" step in `Window::setup_factory` and deal with binding in the "bind" and "unbind" steps.

Filename: listings/todo/1/window/mod.rs

```
    fn setup_factory(&self) {
        // Create a new factory
        let factory = SignalListItemFactory::new();

        // Create an empty `TaskRow` during setup
        factory.connect_setup(move |_, list_item| {
            // Create `TaskRow`
            let task_row = TaskRow::new();
            list_item
                .downcast_ref::<ListItem>()
                .expect("Needs to be ListItem")
                .set_child(Some(&task_row));
        });

        // Tell factory how to bind `TaskRow` to a `TaskObject`
        factory.connect_bind(move |_, list_item| {
            // Get `TaskObject` from `ListItem`
            let task_object = list_item
                .downcast_ref::<ListItem>()
                .expect("Needs to be ListItem")
                .item()
                .and_downcast::<TaskObject>()
                .expect("The item has to be an `TaskObject`.");

            // Get `TaskRow` from `ListItem`
            let task_row = list_item
                .downcast_ref::<ListItem>()
                .expect("Needs to be ListItem")
                .child()
                .and_downcast::<TaskRow>()
                .expect("The child has to be a `TaskRow`.");

            task_row.bind(&task_object);
        });

        // Tell factory how to unbind `TaskRow` from `TaskObject`
        factory.connect_unbind(move |_, list_item| {
            // Get `TaskRow` from `ListItem`
            let task_row = list_item
                .downcast_ref::<ListItem>()
                .expect("Needs to be ListItem")
                .child()
                .and_downcast::<TaskRow>()
                .expect("The child has to be a `TaskRow`.");

            task_row.unbind();
        });

        // Set the factory of the list view
        self.imp().tasks_list.set_factory(Some(&factory));
    }
```

Binding properties in `TaskRow::bind` works just like in former chapters. The only difference is that we store the bindings in a vector. This is necessary because a `TaskRow` will be reused as you scroll through the list. That means that over time a `TaskRow` will need to bound to a new `TaskObject` and has to be unbound from the old one. Unbinding will only work if it can access the stored `glib::Binding`.

Filename: listings/todo/1/task_row/mod.rs

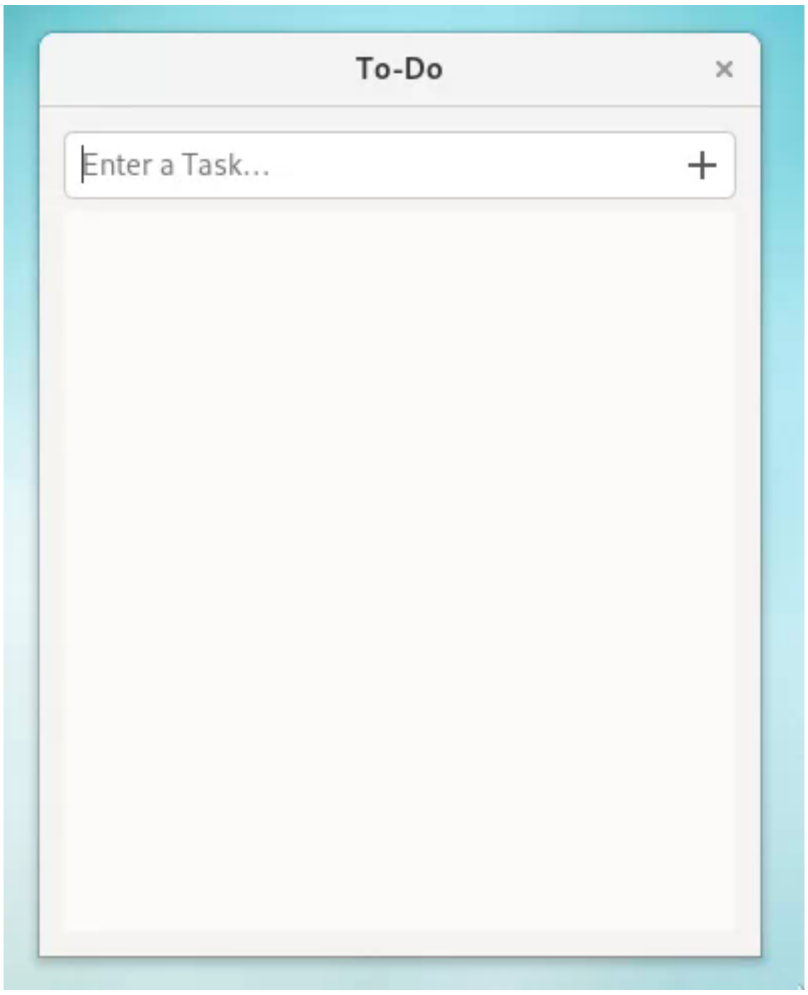
```
    pub fn bind(&self, task_object: &TaskObject) {
        // Get state
        let completed_button = self.imp().completed_button.get();
        let content_label = self.imp().content_label.get();
        let mut bindings = self.imp().bindings.borrow_mut();

        // Bind `task_object.completed` to `task_row.completed_button.active`
        let completed_button_binding = task_object
            .bind_property("completed", &completed_button, "active")
            .bidirectional()
            .sync_create()
            .build();
        // Save binding
        bindings.push(completed_button_binding);

        // Bind `task_object.content` to `task_row.content_label.label`
        let content_label_binding = task_object
            .bind_property("content", &content_label, "label")
            .sync_create()
            .build();
        // Save binding
        bindings.push(content_label_binding);

        // Bind `task_object.completed` to
`task_row.content_label.attributes`
        let content_label_binding = task_object
            .bind_property("completed", &content_label, "attributes")
            .sync_create()
            .transform_to(|_, active| {
                let attribute_list = AttrList::new();
                if active {
                    // If "active" is true, content of the label will be
strikethrough
                    let attribute = AttrInt::new_strikethrough(true);
                    attribute_list.insert(attribute);
                }
                Some(attribute_list.to_value())
            })
            .build();
        // Save binding
        bindings.push(content_label_binding);
    }
```

`TaskRow::unbind` takes care of the cleanup. It iterates through the vector and unbinds each binding. In the end, it clears the vector.

Filename: listings/todo/1/task_row/mod.rs

```
    pub fn unbind(&self) {
        // Unbind all stored bindings
        for binding in self.imp().bindings.borrow_mut().drain(..) {
            binding.unbind();
        }
    }
```

That was it, we created a basic To-Do app! We will extend it with additional functionality in the following chapters.



Actions

By now, we've already learned many ways to glue our widgets together. We can send messages through channels, emit signals, share reference-counted state and bind properties. Now, we will complete our set by learning about actions.

An action is a piece of functionality bound to a certain GObject. Let's check out the simplest case where we activate an action without a parameter.

Filename: listings/actions/1/main.rs

```rust
fn build_ui(app: &Application) {
    // Create a window and set the title
    let window = ApplicationWindow::builder()
        .application(app)
        .title("My GTK App")
        .width_request(360)
        .build();

    // Add action "close" to `window` taking no parameter
    let action_close = ActionEntry::builder("close")
        .activate(|window: &ApplicationWindow, _, _| {
            window.close();
        })
        .build();
    window.add_action_entries([action_close]);

    // Present window
    window.present();
}
```

First, we created a new `gio::ActionEntry` which is named "close" and takes no parameter. We also connected a callback which closes the window when the action is activated. Finally, we add the action entry to the window via `add_action_entries` .

Filename: listings/actions/1/main.rs

```
const APP_ID: &str = "org.gtk_rs.Actions1";

fn main() -> glib::ExitCode {
    // Create a new application
    let app = Application::builder().application_id(APP_ID).build();

    // Connect to "activate" signal of `app`
    app.connect_activate(build_ui);

    // Set keyboard accelerator to trigger "win.close".
    app.set_accels_for_action("win.close", &["<Ctrl>W"]);

    // Run the application
    app.run()
}
```

One of the most popular reasons to use actions are keyboard accelerators, so we added one here. With `set_accels_for_action` one can assign one or more accelerators to a certain action. Check the documentation of `accelerator_parse` in order to learn more about its syntax.

Before we move on to other aspects of actions, let's appreciate a few things that are curious here. The "win" part of "win.close" is the group of the action. But how does GTK know that "win" is the action group of our window? The answer is that it is so common to add actions to windows and applications that there are already two predefined groups available:

- "app" for actions global to the application, and
- "win" for actions tied to an application window.

We can add an action group to any widget via the method `insert_action_group`. Let's add our action to the action group "custom-group" and add the group then to our window. The action entry isn't specific to our window anymore, the first parameter of the "activate" callback is of type `SimpleActionGroup` instead of `ApplicationWindow`. This means we have to clone `window` into the closure.

Filename: listings/actions/2/main.rs

```rust
fn build_ui(app: &Application) {
    // Create a window and set the title
    let window = ApplicationWindow::builder()
        .application(app)
        .title("My GTK App")
        .width_request(360)
        .build();

    // Add action "close" to `window` taking no parameter
    let action_close = ActionEntry::builder("close")
        .activate(clone!(@weak window => move |_, _, _| {
            window.close();
        }))
        .build();

    // Create a new action group and add actions to it
    let actions = SimpleActionGroup::new();
    actions.add_action_entries([action_close]);
    window.insert_action_group("custom-group", Some(&actions));

    // Present window
    window.present();
}
```

If we bind the accelerator to "custom-group.close", it works just as before.

Filename: listings/actions/2/main.rs

```rust
    // Set keyboard accelerator to trigger "custom-group.close".
    app.set_accels_for_action("custom-group.close", &["<Ctrl>W"]);
```

Also, if we had multiple instances of the same windows, we would expect that only the currently focused window will be closed when activating "win.close". And indeed, the "win.close" will be dispatched to the currently focused window. However, that also means that we actually define one action per window instance. If we want to have a single globally accessible action instead, we call `add_action_entries` on our application instead.

> Adding "win.close" was useful as a simple example. However, in the future we will use the pre-defined "window.close" action which does exactly the same thing.

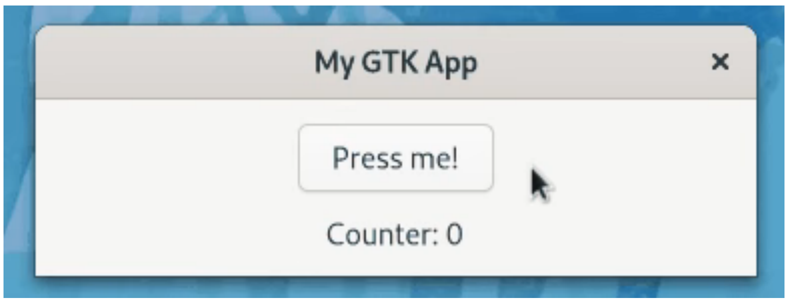
Parameter and State

An action, like most functions, can take a parameter. However, unlike most functions it can also be stateful. Let's see how this works.

Filename: listings/actions/3/main.rs

```
fn build_ui(app: &Application) {
    let original_state = 0;
    let label = Label::builder()
        .label(format!("Counter: {original_state}"))
        .build();

    // Create a button with label
    let button = Button::builder().label("Press me!").build();

    // Connect to "clicked" signal of `button`
    button.connect_clicked(move |button| {
        // Activate "win.count" and pass "1" as parameter
        let parameter = 1;
        button
            .activate_action("win.count", Some(&parameter.to_variant()))
            .expect("The action does not exist.");
    });

    // Create a `gtk::Box` and add `button` and `label` to it
    let gtk_box = gtk::Box::builder()
        .orientation(Orientation::Vertical)
        .margin_top(12)
        .margin_bottom(12)
        .margin_start(12)
        .margin_end(12)
        .spacing(12)
        .halign(Align::Center)
        .build();
    gtk_box.append(&button);
    gtk_box.append(&label);

    // Create a window, set the title and add `gtk_box` to it
    let window = ApplicationWindow::builder()
        .application(app)
        .title("My GTK App")
        .width_request(360)
        .child(&gtk_box)
        .build();

    // Add action "count" to `window` taking an integer as parameter
    let action_count = ActionEntry::builder("count")
        .parameter_type(Some(&i32::static_variant_type()))
        .state(original_state.to_variant())
        .activate(move |_, action, parameter| {
            // Get state
            let mut state = action
                .state()
                .expect("Could not get state.")
                .get::<i32>()
                .expect("The variant needs to be of type `i32`.");

            // Get parameter
            let parameter = parameter
                .expect("Could not get parameter.")
                .get::<i32>()
                .expect("The variant needs to be of type `i32`.");
            // Increase state by parameter and store state
            state += parameter;
            action.set_state(&state.to_variant());

            // Update label with new state
            label.set_label(&format!("Counter: {state}"));
        })
        .build();
    window.add_action_entries([action_count]);

    // Present window
    window.present();
}
```

Here, we created a "win.count" action that increases its state by the given parameter every time it is activated. It also takes care of updating the `label` with the current state. The button activates the action with each click while passing "1" as parameter. This is how our app works:



Actionable

Connecting actions to the "clicked" signal of buttons is a typical use case, which is why all buttons implement the `Actionable` interface. This way, the action can be specified by setting the "action-name" property. If the action accepts a parameter, it can be set via the "action-target" property. With `ButtonBuilder`, we can set everything up by calling its methods.

Filename: listings/actions/4/main.rs

```
    // Create a button with label and action
    let button = Button::builder()
        .label("Press me!")
        .action_name("win.count")
        .action_target(&1.to_variant())
        .build();
```

Actionable widgets are also easily accessible through the interface builder. As usual, we build up the window via a composite template. Within the template we can then set the "action-name" and "action-target" properties.

Filename: listings/actions/5/resources/window.ui

```xml
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <template class="MyGtkAppWindow" parent="GtkApplicationWindow">
    <property name="title">My GTK App</property>
    <child>
      <object class="GtkBox" id="gtk_box">
        <property name="orientation">vertical</property>
        <property name="margin-top">12</property>
        <property name="margin-bottom">12</property>
        <property name="margin-start">12</property>
        <property name="margin-end">12</property>
        <property name="spacing">12</property>
        <property name="halign">center</property>
        <child>
          <object class="GtkButton" id="button">
            <property name="label">Press me!</property>
            <property name="action-name">win.count</property>
            <property name="action-target">1</property>
          </object>
        </child>
        <child>
          <object class="GtkLabel" id="label">
            <property name="label">Counter: 0</property>
          </object>
        </child>
      </object>
    </child>
  </template>
</interface>
```

We will connect the actions and add them to the window in the `Window::setup_actions` method.

Filename: listings/actions/5/window/mod.rs

```rust
impl Window {
    pub fn new(app: &Application) -> Self {
        // Create new window
        Object::builder().property("application", app).build()
    }

    fn setup_actions(&self) {
        // Add stateful action "count" to `window` taking an integer as
parameter
        let original_state = 0;
        let action_count = ActionEntry::builder("count")
            .parameter_type(Some(&i32::static_variant_type()))
            .state(original_state.to_variant())
            .activate(move |window: &Self, action, parameter| {
                // Get state
                let mut state = action
                    .state()
                    .expect("Could not get state.")
                    .get::<i32>()
                    .expect("The variant needs to be of type `i32`.");

                // Get parameter
                let parameter = parameter
                    .expect("Could not get parameter.")
                    .get::<i32>()
                    .expect("The variant needs to be of type `i32`.");

                // Increase state by parameter and store state
                state += parameter;
                action.set_state(&state.to_variant());

                // Update label with new state
                window.imp().label.set_label(&format!("Counter: {state}"));
            })
            .build();
        self.add_action_entries([action_count]);
    }
}
```

Finally, `setup_actions` will be called within `constructed`.

Filename: listings/actions/5/window/imp.rs

```rust
// Trait shared by all GObjects
impl ObjectImpl for Window {
    fn constructed(&self) {
        // Call "constructed" on parent
        self.parent_constructed();

        // Add actions
        self.obj().setup_actions();
    }
}
```

This app behaves the same as our previous example, but it will make it simpler for us to add a menu in the following section.

Menus

If you want to create a menu, you have to use actions, and you will want to use the interface builder. Typically, a menu entry has an action fitting one of these three descriptions:

- no parameter and no state, or
- no parameter and boolean state, or
- string parameter and string state.

Let's modify our small app to demonstrate these cases. First, we extend `setup_actions`. For the action without parameter or state, we can use the pre-defined "window.close" action. Therefore, we don't have to add anything here.

With the action "button-frame", we manipulate the "has-frame" property of `button`. Here, the convention is that actions with no parameter and boolean state should behave like toggle actions. This means that the caller can expect the boolean state to toggle after activating the action. Luckily for us, that is the default behavior for `gio::PropertyAction` with a boolean property.

Filename: listings/actions/6/window/mod.rs

```
        // Add property action "button-frame" to `window`
        let button = self.imp().button.get();
        let action_button_frame =
            PropertyAction::new("button-frame", &button, "has-frame");
        self.add_action(&action_button_frame);
```

A `PropertyAction` is useful when you need an action that manipulates the property of a GObject. The property then acts as the state of the action. As mentioned above, if the property is a boolean the action has no parameter and toggles the property on activation. In all other cases, the action has a parameter of the same type as the property. When activating the action, the property gets set to the same value as the parameter of the action.

Finally, we add "win.orientation", an action with string parameter and string state. This action can be used to change the orientation of `gtk_box`. Here the convention is that the state should be set to the given parameter. We don't need the action state to implement orientation switching, however it is useful for making the menu display the current orientation.

Filename: listings/actions/6/window/mod.rs

```
        // Add stateful action "orientation" to `window` taking a string as
parameter
        let action_orientation = ActionEntry::builder("orientation")
            .parameter_type(Some(&String::static_variant_type()))
            .state("Vertical".to_variant())
            .activate(move |window: &Self, action, parameter| {
                // Get parameter
                let parameter = parameter
                    .expect("Could not get parameter.")
                    .get::<String>()
                    .expect("The value needs to be of type `String`.");

                let orientation = match parameter.as_str() {
                    "Horizontal" => Orientation::Horizontal,
                    "Vertical" => Orientation::Vertical,
                    _ => unreachable!(),
                };

                // Set orientation and save state
                window.imp().gtk_box.set_orientation(orientation);
                action.set_state(&parameter.to_variant());
            })
            .build();
        self.add_action_entries([action_count, action_orientation]);
```

Even though `gio::Menu` can also be created with the bindings, the most convenient way is to use the interface builder for that. We do that by adding the menu in front of the template.

Filename: listings/actions/6/resources/window.ui

```
 <?xml version="1.0" encoding="UTF-8"?>
 <interface>
+  <menu id="main-menu">
+    <item>
+      <attribute name="label" translatable="yes">_Close window</attribute>
+      <attribute name="action">window.close</attribute>
+    </item>
+    <item>
+      <attribute name="label" translatable="yes">_Toggle button frame</
attribute>
+      <attribute name="action">win.button-frame</attribute>
+    </item>
+    <section>
+      <attribute name="label" translatable="yes">Orientation</attribute>
+      <item>
+        <attribute name="label" translatable="yes">_Horizontal</attribute>
+        <attribute name="action">win.orientation</attribute>
+        <attribute name="target">Horizontal</attribute>
+      </item>
+      <item>
+        <attribute name="label" translatable="yes">_Vertical</attribute>
+        <attribute name="action">win.orientation</attribute>
+        <attribute name="target">Vertical</attribute>
+      </item>
+    </section>
+  </menu>
   <template class="MyGtkAppWindow" parent="GtkApplicationWindow">
     <property name="title">My GTK App</property>
+    <property name="width-request">360</property>
+    <child type="titlebar">
+      <object class="GtkHeaderBar">
+        <child type ="end">
+          <object class="GtkMenuButton">
+            <property name="icon-name">open-menu-symbolic</property>
+            <property name="menu-model">main-menu</property>
+          </object>
+        </child>
+      </object>
+    </child>
     <child>
       <object class="GtkBox" id="gtk_box">
         <property name="orientation">vertical</property>
```

Since we connect the menu to the `gtk::MenuButton` via the menu-model property, the `Menu` is expected to be a `gtk::PopoverMenu`. The documentation for `PopoverMenu` also explains its `xml` syntax for the interface builder.

Also note how we specified the target:

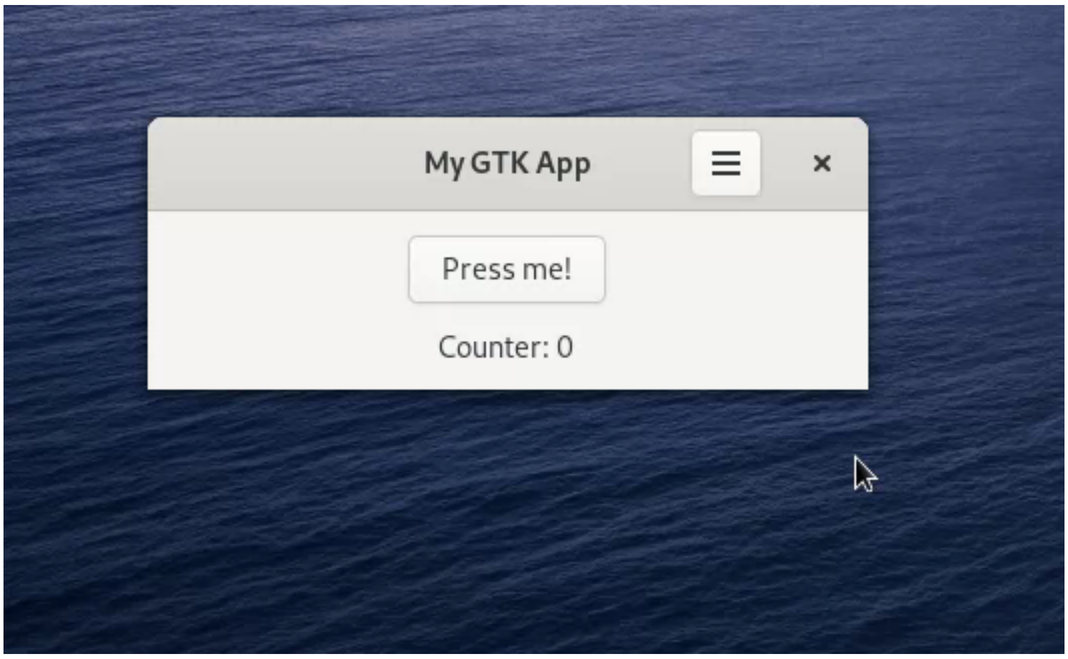
```
<attribute name="target">Horizontal</attribute>
```

String is the default type of the target which is why we did not have to specify a type. With targets of other types you need to manually specify the correct GVariant format string.

For example, an `i32` variable with value "5" would correspond to this:

```xml
<attribute name="target" type="i">5</attribute>
```

This is how the app looks in action:



We changed the icon of the `MenuButton` by setting its property "icon-name" to "open-menu-symbolic". You can find more icons with the Icon Library. They can be embedded with `gio::Resource` and then be referenced within the composite templates (or other places).

Settings

The menu entries nicely display the state of our stateful actions, but after the app is closed, all changes to that state are lost. As usual, we solve this problem with `gio::Settings`. First we create a schema with settings corresponding to the stateful actions we created before.

Filename: listings/actions/7/org.gtk_rs.Actions7.gschema.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<schemalist>
  <schema id="org.gtk_rs.Actions7" path="/org/gtk_rs/Actions7/">
    <key name="button-frame" type="b">
      <default>true</default>
      <summary>Whether the button has a frame</summary>
    </key>
    <key name="orientation" type="s">
      <choices>
        <choice value='Horizontal'/>
        <choice value='Vertical'/>
      </choices>
      <default>'Vertical'</default>
      <summary>Orientation of GtkBox</summary>
    </key>
  </schema>
</schemalist>
```

Again, we install the schema as described in the settings chapter. Then we add the settings to `imp::Window`. Since `gio::Settings` does not implement `Default`, we wrap it in a `std::cell::OnceCell`.

Filename: listings/actions/7/window/imp.rs

```rust
// Object holding the state
#[derive(CompositeTemplate, Default)]
#[template(resource = "/org/gtk_rs/example/window.ui")]
pub struct Window {
    #[template_child]
    pub gtk_box: TemplateChild<gtk::Box>,
    #[template_child]
    pub button: TemplateChild<Button>,
    #[template_child]
    pub label: TemplateChild<Label>,
    pub settings: OnceCell<Settings>,
}
```

Now we create functions to make it easier to access settings.

Filename: listings/actions/7/window/mod.rs

```
    fn setup_settings(&self) {
        let settings = Settings::new(APP_ID);
        self.imp()
            .settings
            .set(settings)
            .expect("`settings` should not be set before calling
`setup_settings`.");
    }

    fn settings(&self) -> &Settings {
        self.imp()
            .settings
            .get()
            .expect("`settings` should be set in `setup_settings`.")
    }
```

Creating stateful actions from setting entries is so common that `Settings` provides a method for that exact purpose. We create actions with the `create_action` method and then add them to the action group of our window.

Filename: listings/actions/7/window/mod.rs

```
        // Create action from key "button-frame" and add to action group
"win"
        let action_button_frame = self.settings().create_action("button-
frame");
        self.add_action(&action_button_frame);

        // Create action from key "orientation" and add to action group "win"
        let action_orientation =
self.settings().create_action("orientation");
        self.add_action(&action_orientation);
```

Since actions from `create_action` follow the aforementioned conventions, we can keep further changes to a minimum. The action "win.button-frame" toggles its state with each activation and the state of the "win.orientation" action follows the given parameter.

We still have to specify what should happen when the actions are activated though. For the stateful actions, instead of adding callbacks to their "activate" signals, we bind the settings to properties we want to manipulate.

Filename: listings/actions/7/window/mod.rs

```
    fn bind_settings(&self) {
        // Bind setting "button-frame" to "has-frame" property of `button`
        let button = self.imp().button.get();
        self.settings()
            .bind("button-frame", &button, "has-frame")
            .build();

        // Bind setting "orientation" to "orientation" property of `button`
        let gtk_box = self.imp().gtk_box.get();
        self.settings()
            .bind("orientation", &gtk_box, "orientation")
            .mapping(|variant, _| {
                let orientation = variant
                    .get::<String>()
                    .expect("The variant needs to be of type `String`.");

                let orientation = match orientation.as_str() {
                    "Horizontal" => Orientation::Horizontal,
                    "Vertical" => Orientation::Vertical,
                    _ => unreachable!(),
                };

                Some(orientation.to_value())
            })
            .build();
    }
```

Finally, we make sure that `bind_settings` is called within `constructed`.

Filename: listings/actions/7/window/imp.rs

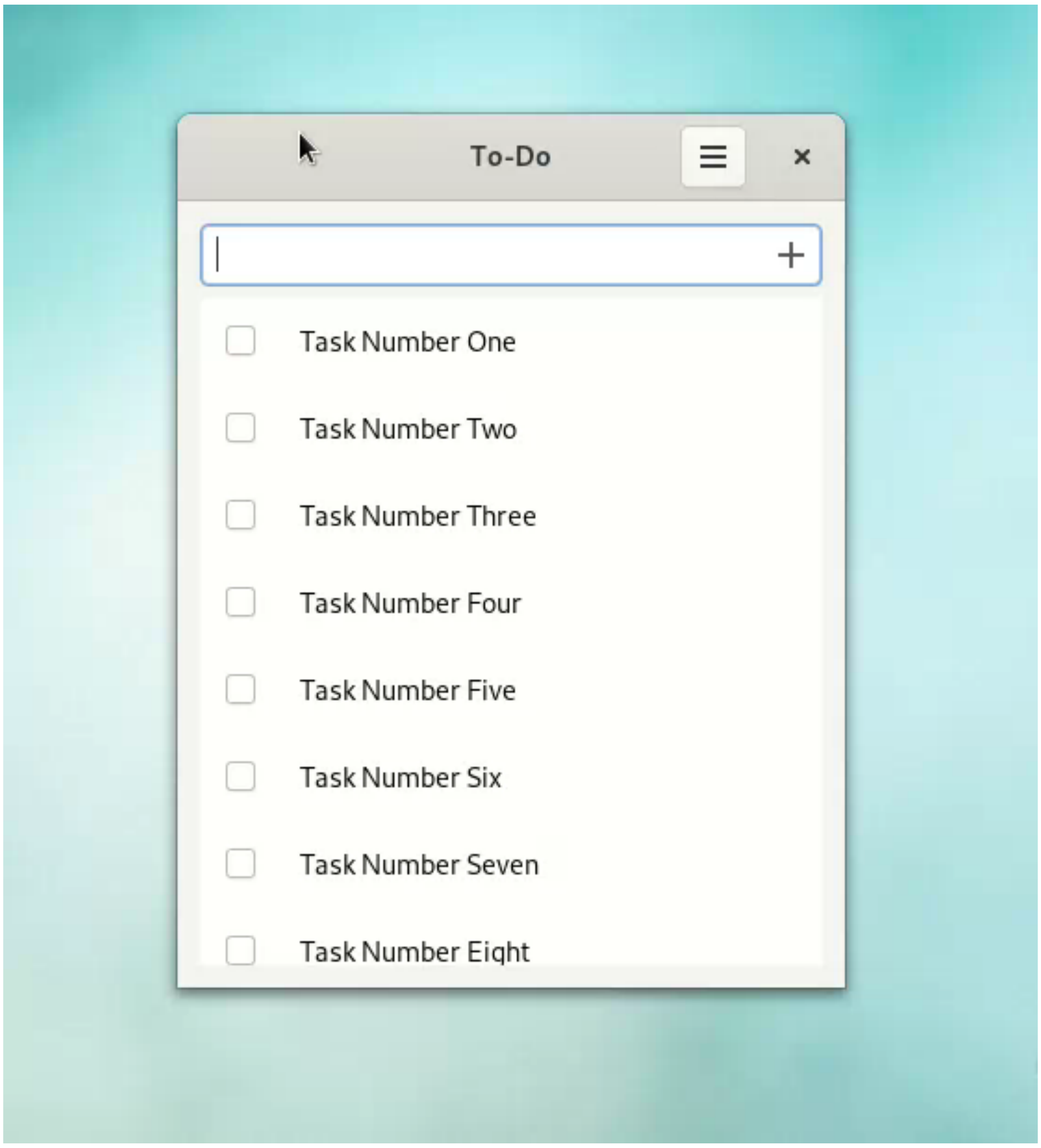
```
// Trait shared by all GObjects
impl ObjectImpl for Window {
    fn constructed(&self) {
        // Call "constructed" on parent
        self.parent_constructed();

        // Setup
        let obj = self.obj();
        obj.setup_settings();
        obj.setup_actions();
        obj.bind_settings();
    }
}
```

Actions are extremely powerful, and we are only scratching the surface here. If you want to learn more about them, the GNOME developer documentation is a good place to start.

Manipulating State of To-Do App

Filtering Tasks

Now it is time to continue working on our To-Do app. One nice feature to add would be filtering of the tasks. What a chance to use our newly gained knowledge of actions! Using actions, we can access the filter via the menu as well as via keyboard shortcuts. This is how we want this to work in the end:



Note that the screencast also shows a button with label "Clear" which will remove all done tasks. This will come in handy when we later make the app preserve the tasks between sessions.

Let's start by adding a menu and a header bar to `window.ui`. After reading the actions chapter the added code should feel familiar.

Filename: listings/todo/2/resources/window.ui

```
  <?xml version="1.0" encoding="UTF-8"?>
  <interface>
+   <menu id="main-menu">
+     <submenu>
+       <attribute name="label" translatable="yes">_Filter</attribute>
+       <item>
+         <attribute name="label" translatable="yes">_All</attribute>
+         <attribute name="action">win.filter</attribute>
+         <attribute name="target">All</attribute>
+       </item>
+       <item>
+         <attribute name="label" translatable="yes">_Open</attribute>
+         <attribute name="action">win.filter</attribute>
+         <attribute name="target">Open</attribute>
+       </item>
+       <item>
+         <attribute name="label" translatable="yes">_Done</attribute>
+         <attribute name="action">win.filter</attribute>
+         <attribute name="target">Done</attribute>
+       </item>
+     </submenu>
+     <item>
+       <attribute name="label" translatable="yes">_Remove Done Tasks</
attribute>
+       <attribute name="action">win.remove-done-tasks</attribute>
+     </item>
+     <item>
+       <attribute name="label" translatable="yes">_Keyboard Shortcuts</
attribute>
+       <attribute name="action">win.show-help-overlay</attribute>
+     </item>
+   </menu>
    <template class="TodoWindow" parent="GtkApplicationWindow">
      <property name="width-request">360</property>
      <property name="title" translatable="yes">To-Do</property>
+     <child type="titlebar">
+       <object class="GtkHeaderBar">
+         <child type="end">
+           <object class="GtkMenuButton" id="menu_button">
+             <property name="icon-name">open-menu-symbolic</property>
+             <property name="menu-model">main-menu</property>
+           </object>
+         </child>
+       </object>
+     </child>
```

Then, we create a settings schema. Again, the "filter" setting correspond to the stateful actions called by the menu.

Filename: listings/todo/2/org.gtk_rs.Todo2.gschema.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<schemalist>
  <schema id="org.gtk_rs.Todo2" path="/org/gtk_rs/Todo2/">
    <key name="filter" type="s">
      <choices>
        <choice value='All'/>
        <choice value='Open'/>
        <choice value='Done'/>
      </choices>
      <default>'All'</default>
      <summary>Filter of the tasks</summary>
    </key>
  </schema>
</schemalist>
```

We install the schema as described in the settings chapter Then we add a reference to `settings` to `imp::Window`.

Filename: listings/todo/2/window/imp.rs

```rust
// Object holding the state
#[derive(CompositeTemplate, Default)]
#[template(resource = "/org/gtk_rs/Todo2/window.ui")]
pub struct Window {
    #[template_child]
    pub entry: TemplateChild<Entry>,
    #[template_child]
    pub tasks_list: TemplateChild<ListView>,
    pub tasks: RefCell<Option<gio::ListStore>>,
    pub settings: OnceCell<Settings>,
}
```

Again, we create functions to make it easier to access settings.

Filename: listings/todo/2/window/mod.rs

```rust
    fn setup_settings(&self) {
        let settings = Settings::new(APP_ID);
        self.imp()
            .settings
            .set(settings)
            .expect("`settings` should not be set before calling
`setup_settings`.");
    }

    fn settings(&self) -> &Settings {
        self.imp()
            .settings
            .get()
            .expect("`settings` should be set in `setup_settings`.")
    }
```

We also add the methods `is_completed`, `task_data` and `from_task_data` to `TaskObject`. We will make use of them in the following snippets.

Filename: listings/todo/2/task_object/mod.rs

```rust
impl TaskObject {
    pub fn new(completed: bool, content: String) -> Self {
        Object::builder()
            .property("completed", completed)
            .property("content", content)
            .build()
    }

    pub fn is_completed(&self) -> bool {
        self.imp().data.borrow().completed
    }

    pub fn task_data(&self) -> TaskData {
        self.imp().data.borrow().clone()
    }

    pub fn from_task_data(task_data: TaskData) -> Self {
        Self::new(task_data.completed, task_data.content)
    }
}
```

Similar to the previous chapter, we let `settings` create the action. Then we add the newly created action "filter" to our window.

Filename: listings/todo/2/window/mod.rs

```rust
    fn setup_actions(&self) {
        // Create action from key "filter" and add to action group "win"
        let action_filter = self.settings().create_action("filter");
        self.add_action(&action_filter);
    }
```

We also add an action which allows us to remove done tasks. This time we use another method called `install_action`. This method has a couple of limitation. It can only be used when subclassing widgets, and it doesn't support stateful actions. On the flipside, its usage is concise and it has a corresponding sister-method `install_action_async` which we will use in one of the future chapters.

Filename: listings/todo/2/window/imp.rs

```rust
// The central trait for subclassing a GObject
#[glib::object_subclass]
impl ObjectSubclass for Window {
    // `NAME` needs to match `class` attribute of template
    const NAME: &'static str = "TodoWindow";
    type Type = super::Window;
    type ParentType = gtk::ApplicationWindow;

    fn class_init(klass: &mut Self::Class) {
        klass.bind_template();

        // Create action to remove done tasks and add to action group "win"
        klass.install_action("win.remove-done-tasks", None, |window, _, _| {
            window.remove_done_tasks();
        });
    }

    fn instance_init(obj: &InitializingObject<Self>) {
        obj.init_template();
    }
}
```

This is the implementation of `remove_done_tasks`. We iterate through the `gio::ListStore` and remove all completed task objects.

Filename: listings/todo/2/window/mod.rs

```rust
    fn remove_done_tasks(&self) {
        let tasks = self.tasks();
        let mut position = 0;
        while let Some(item) = tasks.item(position) {
            // Get `TaskObject` from `glib::Object`
            let task_object = item
                .downcast_ref::<TaskObject>()
                .expect("The object needs to be of type `TaskObject`.");

            if task_object.is_completed() {
                tasks.remove(position);
            } else {
                position += 1;
            }
        }
    }
```

After activating the action "win.filter", the corresponding setting will be changed. So we need a method which translates this setting into a filter that the `gtk::FilterListModel` understands. The possible states are "All", "Open" and "Done". We return `Some(filter)` for "Open" and "Done". If the state is "All" nothing has to be filtered out, so we return `None`.

Filename: listings/todo/2/window/mod.rs

```
    fn filter(&self) -> Option<CustomFilter> {
        // Get filter_state from settings
        let filter_state: String = self.settings().get("filter");

        // Create custom filters
        let filter_open = CustomFilter::new(|obj| {
            // Get `TaskObject` from `glib::Object`
            let task_object = obj
                .downcast_ref::<TaskObject>()
                .expect("The object needs to be of type `TaskObject`.");

            // Only allow completed tasks
            !task_object.is_completed()
        });
        let filter_done = CustomFilter::new(|obj| {
            // Get `TaskObject` from `glib::Object`
            let task_object = obj
                .downcast_ref::<TaskObject>()
                .expect("The object needs to be of type `TaskObject`.");

            // Only allow done tasks
            task_object.is_completed()
        });

        // Return the correct filter
        match filter_state.as_str() {
            "All" => None,
            "Open" => Some(filter_open),
            "Done" => Some(filter_done),
            _ => unreachable!(),
        }
    }
```

Now, we can set up the model. We initialize `filter_model` with the state from the settings by calling the method `filter`. Whenever the state of the key "filter" changes, we call the method `filter` again to get the updated `filter_model`.

Filename: listings/todo/2/window/mod.rs

```
    fn setup_tasks(&self) {
        // Create new model
        let model = gio::ListStore::new::<TaskObject>();

        // Get state and set model
        self.imp().tasks.replace(Some(model));

        // Wrap model with filter and selection and pass it to the list view
        let filter_model = FilterListModel::new(Some(self.tasks()),
self.filter());
        let selection_model = NoSelection::new(Some(filter_model.clone()));
        self.imp().tasks_list.set_model(Some(&selection_model));

        // Filter model whenever the value of the key "filter" changes
        self.settings().connect_changed(
            Some("filter"),
            clone!(@weak self as window, @weak filter_model => move |_, _| {
                filter_model.set_filter(window.filter().as_ref());
            }),
        );
    }
```

Then, we bind the shortcuts to their actions with `set_accels_for_action`. Here as well, a detailed action name is used. Since this has to be done at the application level, `setup_shortcuts` takes a `gtk::Application` as parameter.

Filename: listings/todo/2/main.rs

```
fn main() -> glib::ExitCode {
    // Register and include resources
    gio::resources_register_include!("todo_2.gresource")
        .expect("Failed to register resources.");

    // Create a new application
    let app = Application::builder().application_id(APP_ID).build();

    // Connect to signals
    app.connect_startup(setup_shortcuts);
    app.connect_activate(build_ui);

    // Run the application
    app.run()
}

fn setup_shortcuts(app: &Application) {
    app.set_accels_for_action("win.filter('All')", &["<Ctrl>a"]);
    app.set_accels_for_action("win.filter('Open')", &["<Ctrl>o"]);
    app.set_accels_for_action("win.filter('Done')", &["<Ctrl>d"]);
}
```

Now that we created all these nice shortcuts we will want our users to find them. We do that by creating a shortcut window. Again we use an `ui` file to describe it, but here we don't want to use it as template for our custom widget. Instead we instantiate a widget of the existing class `gtk::ShortcutsWindow` with it.

Filename: listings/todo/2/resources/shortcuts.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <object class="GtkShortcutsWindow" id="help_overlay">
    <property name="modal">True</property>
    <child>
      <object class="GtkShortcutsSection">
        <property name="section-name">shortcuts</property>
        <property name="max-height">10</property>
        <child>
          <object class="GtkShortcutsGroup">
            <property name="title" translatable="yes" context="shortcut
window">General</property>
            <child>
              <object class="GtkShortcutsShortcut">
                <property name="title" translatable="yes" context="shortcut
window">Show shortcuts</property>
                <property name="action-name">win.show-help-overlay</property>
              </object>
            </child>
            <child>
              <object class="GtkShortcutsShortcut">
                <property name="title" translatable="yes" context="shortcut
window">Filter to show all tasks</property>
                <property name="action-name">win.filter('All')</property>
              </object>
            </child>
            <child>
              <object class="GtkShortcutsShortcut">
                <property name="title" translatable="yes" context="shortcut
window">Filter to show only open tasks</property>
                <property name="action-name">win.filter('Open')</property>
              </object>
            </child>
            <child>
              <object class="GtkShortcutsShortcut">
                <property name="title" translatable="yes" context="shortcut
window">Filter to show only completed tasks</property>
                <property name="action-name">win.filter('Done')</property>
              </object>
            </child>
          </object>
        </child>
      </object>
    </child>
  </object>
</interface>
```

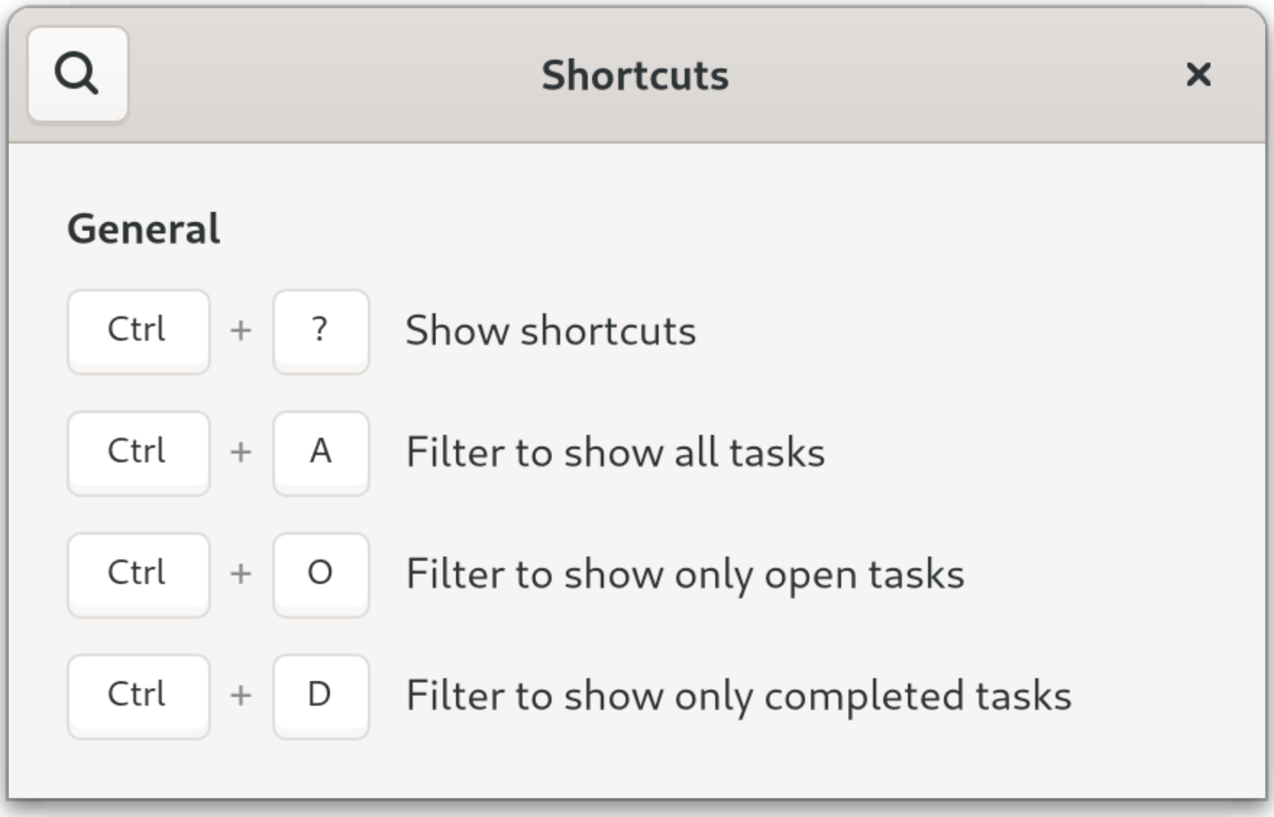
The entries can be organized with `gtk::ShortcutsSection` and `gtk::ShortcutsGroup`. If we specify the action name, we also don't have to repeat the keyboard accelerator. `gtk::ShortcutsShortcut` looks it up on its own.

Note the way we set `action-name` for `ShortcutsShortcut`. Instead of using a separate property for the target, it takes a *detailed* action name. Detailed names look like this: `action_group.action_name(target)`. Formatting of the target depends on its type and is documented here. In particular, strings have to be enclosed single quotes as you can see in this example.

Finally, we have to add the `shortcuts.ui` to our resources. Note that we give it the alias `gtk/help-overlay.ui`. We do that to take advantage of a convenience functionality documented here. It will look for a resource at `gtk/help-overlay.ui` which defines a `ShortcutsWindow` with id `help_overlay`. If it can find one it will create a action `win.show-help-overlay` which will show the window and associate the shortcut `Ctrl` + `?` with this action.

Filename: listings/todo/2/resources/resources.gresource.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<gresources>
  <gresource prefix="/org/gtk_rs/Todo2/">
    <file compressed="true" preprocess="xml-stripblanks" alias="gtk/help-
overlay.ui">shortcuts.ui</file>
    <file compressed="true" preprocess="xml-stripblanks">task_row.ui</file>
    <file compressed="true" preprocess="xml-stripblanks">window.ui</file>
  </gresource>
</gresources>
```



Saving and Restoring Tasks

Since we use `Settings`, our filter state will persist between sessions. However, the tasks themselves will not. Let us implement that. We could store our tasks in `Settings`, but it would be inconvenient. When it comes to serializing and deserializing nothing beats the crate `serde`. Combined with `serde_json` we can save our tasks as serialized json files.

First, we extend our `Cargo.toml` with the `serde` and `serde_json` crate.

```
cargo add serde --features derive
cargo add serde_json
```

Filename: listings/Cargo.toml

```
[dependencies]
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
```

Serde is a framework for serializing and deserializing Rust data structures. The `derive` feature allows us to make our structures (de-)serializable with a single line of code. We also use the `rc` feature so that Serde can deal with `std::rc::Rc` objects. This is why we stored the data of `TodoObject` in a distinct `TodoData` structure. Doing so allows us to derive `Serialize` and `Deserialize` for `TodoData`.

Filename: listings/todo/2/task_object/mod.rs

```
#[derive(Default, Clone, Serialize, Deserialize)]
pub struct TaskData {
    pub completed: bool,
    pub content: String,
}
```

We plan to store our data as a file, so we create a utility function to provide a suitable file path for us. We use `glib::user_config_dir` to get the path to the config directory and create a new subdirectory for our app. Then we return the file path.

Filename: listings/todo/2/utils.rs

```
pub fn data_path() -> PathBuf {
    let mut path = glib::user_data_dir();
    path.push(APP_ID);
    std::fs::create_dir_all(&path).expect("Could not create directory.");
    path.push("data.json");
    path
}
```

We override the `close_request` virtual function to save the tasks when the window is closed. To do so, we first iterate through all entries and store them in a `Vec`. Then we serialize the `Vec` and store the data as a json file.

Filename: listings/todo/2/window/imp.rs

```
// Trait shared by all windows
impl WindowImpl for Window {
    fn close_request(&self) -> glib::Propagation {
        // Store task data in vector
        let backup_data: Vec<TaskData> = self
            .obj()
            .tasks()
            .iter::<TaskObject>()
            .filter_map(Result::ok)
            .map(|task_object| task_object.task_data())
            .collect();

        // Save state to file
        let file = File::create(data_path()).expect("Could not create json
file.");
        serde_json::to_writer(file, &backup_data)
            .expect("Could not write data to json file");

        // Pass close request on to the parent
        self.parent_close_request()
    }
}
```

Let's it have a look into what a `Vec<TaskData>` will be serialized. Note that `serde_json::to_writer` saves the data into a more concise, but also less readable way. To create the equivalent but nicely formatted json below you can just replace `to_writer` with `serde_json::to_writer_pretty`.

Filename: data.json

```
[
  {
    "completed": true,
    "content": "Task Number Two"
  },
  {
    "completed": false,
    "content": "Task Number Five"
  },
  {
    "completed": true,
    "content": "Task Number Six"
  },
  {
    "completed": false,
    "content": "Task Number Seven"
  },
  {
    "completed": false,
    "content": "Task Number Eight"
  }
]
```

When we start the app, we will want to restore the saved data. Let us add a `restore_data` method for that. We make sure to handle the case where there is no data file there yet. It might be the first time that we started the app and therefore there is no former session to restore.

Filename: listings/todo/2/window/mod.rs

```
    fn restore_data(&self) {
        if let Ok(file) = File::open(data_path()) {
            // Deserialize data from file to vector
            let backup_data: Vec<TaskData> =
serde_json::from_reader(file).expect(
                "It should be possible to read `backup_data` from the json
file.",
            );

            // Convert `Vec<TaskData>` to `Vec<TaskObject>`
            let task_objects: Vec<TaskObject> = backup_data
                .into_iter()
                .map(TaskObject::from_task_data)
                .collect();

            // Insert restored objects into model
            self.tasks().extend_from_slice(&task_objects);
        }
    }
```

Finally, we make sure that everything is set up in `constructed`.

Filename: listings/todo/2/window/imp.rs

```
// Trait shared by all GObjects
impl ObjectImpl for Window {
    fn constructed(&self) {
        // Call "constructed" on parent
        self.parent_constructed();

        // Setup
        let obj = self.obj();
        obj.setup_settings();
        obj.setup_tasks();
        obj.restore_data();
        obj.setup_callbacks();
        obj.setup_factory();
        obj.setup_actions();
    }
}
```

Our To-Do app suddenly became much more useful. Not only can we filter tasks, we also retain our tasks between sessions.

CSS

When you want to modify the style of your website, you use CSS. Similarly, GTK supports its own variant of CSS in order to style your app.

> We will not explain every piece of syntax used in this chapter. If you are new to CSS or just need a refresher, have a look at the MDN Web Docs.

Let's say we have a button and we want to set its font color to magenta. Every type of widget has a corresponding CSS node. In the case of `gtk::Button`, this node is called `button`. Therefore, we create a `style.css` file with the following content:

Filename: listings/css/1/style.css

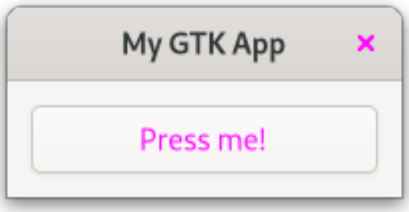
```css
button {
  color: magenta;
}
```

Next, we need to load the CSS file in the startup step of the application. As usual, the widgets are created during the "activate" step.

Filename: listings/css/1/main.rs

```rust
const APP_ID: &str = "org.gtk_rs.Css1";

fn main() -> glib::ExitCode {
    // Create a new application
    let app = Application::builder().application_id(APP_ID).build();

    // Connect to signals
    app.connect_startup(|_| load_css());
    app.connect_activate(build_ui);

    // Run the application
    app.run()
}

fn load_css() {
    // Load the CSS file and add it to the provider
    let provider = CssProvider::new();
    provider.load_from_string(include_str!("style.css"));

    // Add the provider to the default screen
    gtk::style_context_add_provider_for_display(
        &Display::default().expect("Could not connect to a display."),
        &provider,
        gtk::STYLE_PROVIDER_PRIORITY_APPLICATION,
    );
}

fn build_ui(app: &Application) {
    // Create button
    let button = Button::builder()
        .label("Press me!")
        .margin_top(12)
        .margin_bottom(12)
        .margin_start(12)
        .margin_end(12)
        .build();

    // Create a new window and present it
    let window = ApplicationWindow::builder()
        .application(app)
        .title("My GTK App")
        .child(&button)
        .build();
    window.present();
}
```

When we now run the app, we notice that our button *and* the "close" button are magenta. Probably not what we wanted, but that is what our CSS snippet does. We did not specify for which button the rule should apply, so it was applied to both of them.



The `GtkInspector` comes in quite handy (not only) when playing with CSS. Make sure that the window of your app is focused and press `Ctrl` + `Shift` + `D`. A window will pop up which lets you browse and even manipulate the state of your app. Open the CSS view and override the button color with the following snippet.

```
button {
 color: blue;
}
```

With the pause button you can toggle whether your CSS code is active or not.

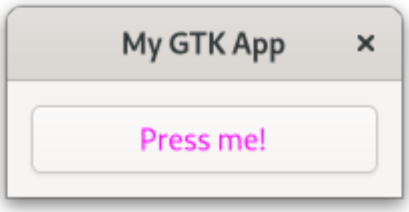
Style Classes Applied by GTK

Class selectors are one way to choose which specific elements a CSS rule applies to. GTK adds style classes to many of its widgets, often depending on their content. A `gtk::Button`, for example, will get the `text-button` style class when its content is a label. That is why we create a new CSS rule which only applies to `button` nodes with the style class `text_button`.

Filename: listings/css/2/style.css

```
button.text-button {
  color: magenta;
}
```

Now only the font of our button becomes magenta.



Adding Your Own Style Class

With `add_css_class` we can also add our own style classes to widgets. One use-case for this is when you want a rule to apply to a hand-picked set of widgets. For example if we have two buttons, but want only one of them to have magenta font. Relying on one of the style classes which GTK adds will not help since both will get the same ones. Which is why we add the style class `button-1` to the first one.

Filename: listings/css/3/main.rs

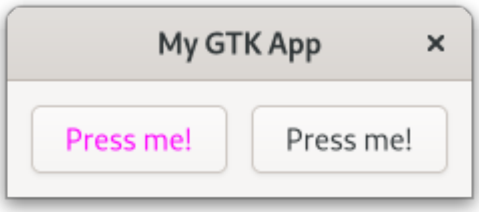
```
    // Create buttons
    let button_1 = Button::with_label("Press me!");
    let button_2 = Button::with_label("Press me!");

    button_1.add_css_class("button-1");
```

Then, we create a CSS rule that applies to `button` nodes with the style class `button-1`.

Filename: listings/css/3/style.css

```
button.button-1 {
  color: magenta;
}
```

We can see that this way only the first button gets colored magenta.



Specifying Name of a Widget

If you want that your rule only applies to a single widget, matching with style classes can be fine. Ideally however, you would give the widget a name and match with that name instead. This way your intentions are more clear, compared to matching with style classes that can apply to multiple widgets.

Again, we have two buttons but want to color only one of them magenta. We set the name of the first one with `set_widget_name`.

Filename: listings/css/4/main.rs

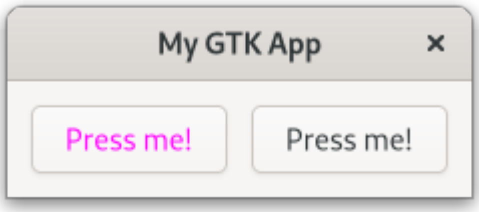
```
    // Create buttons
    let button_1 = Button::with_label("Press me!");
    let button_2 = Button::with_label("Press me!");

    button_1.set_widget_name("button-1");
```

Then, create a CSS rule that applies to `button` nodes with the name `button-1`. The name is specified after the `#` symbol.

Filename: listings/css/4/style.css

```
button#button-1 {
  color: magenta;
}
```

Again, the style rule only applies to the first button.

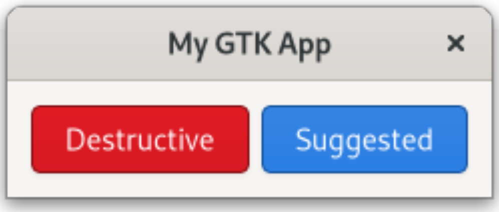


CSS Rules Provided by GTK

Certain styles are common enough that GTK provides CSS rules for them. For example, if you want to indicate that your button leads to a destructive or suggested action you don't have to provide your own CSS rules. All you have to do is to add "destructive-action" or "suggested-action" style class to your button. Most widgets will document these rules in their documentation under CSS nodes.

Filename: listings/css/5/main.rs

```
    // Create buttons
    let button_1 = Button::with_label("Destructive");
    let button_2 = Button::with_label("Suggested");

    button_1.add_css_class("destructive-action");
    button_2.add_css_class("suggested-action");
```



Interface Builder

We can also add style classes with the interface builder. Just add the `<style>` element to your widget. The `<style>` element is documented together with `gtk::Widget`. Adding again destructive and suggested buttons, would then look like this:

Filename: listings/css/6/window/window.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <template class="MyGtkAppWindow" parent="GtkApplicationWindow">
    <property name="title" translatable="yes">My GTK App</property>
    <child>
      <object class="GtkBox">
        <property name="margin-top">12</property>
        <property name="margin-bottom">12</property>
        <property name="margin-start">12</property>
        <property name="margin-end">12</property>
        <property name="spacing">6</property>
        <child>
          <object class="GtkButton">
            <property name="label">Destructive</property>
            <style>
              <class name="destructive-action"/>
            </style>
          </object>
        </child>
        <child>
          <object class="GtkButton">
            <property name="label">Suggested</property>
            <style>
              <class name="suggested-action"/>
            </style>
          </object>
        </child>
      </object>
    </child>
  </template>
</interface>
```

Pseudo-classes

Sometimes you want your CSS rules to apply under even more precise conditions than style classes allow. That is where pseudo-classes come in. Let's use a single button with name `button-1` to demonstrate this concept.

Filename: listings/css/7/window/window.ui

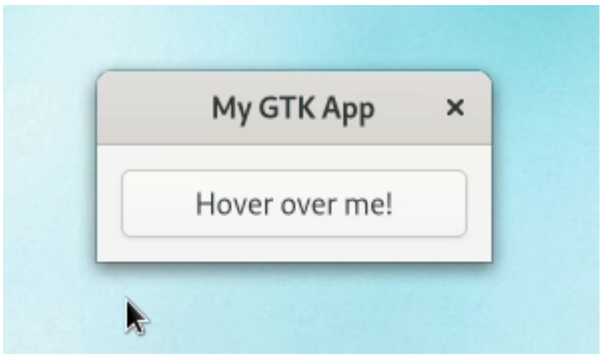
```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <template class="MyGtkAppWindow" parent="GtkApplicationWindow">
    <property name="title" translatable="yes">My GTK App</property>
    <child>
      <object class="GtkButton">
        <property name="label">Hover over me!</property>
        <property name="margin-top">12</property>
        <property name="margin-bottom">12</property>
        <property name="margin-start">12</property>
        <property name="margin-end">12</property>
        <property name="name">button-1</property>
      </object>
    </child>
  </template>
</interface>
```

By adding the pseudo-class `hover`, we say that we want this rule to only apply to a `button` node with name `button-1` when hovering over it with the mouse pointer.

Filename: listings/css/7/style.css

```
button#button-1:hover {
  color: magenta;
  background: yellow;
}
```

If we now hover over the button, we see that over the span of one second its background turns yellow and its font turns magenta. After we removed the cursor, the button returns to its original state.



Nodes

In the previous examples, a widget always corresponded to a single CSS node. This is not always the case. For example, `gtk::MenuButton` has multiple CSS nodes. Let's see how that works.

First, we create a single `MenuButton`.

Filename: listings/css/8/window/window.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <template class="MyGtkAppWindow" parent="GtkApplicationWindow">
    <property name="title" translatable="yes">My GTK App</property>
    <child>
      <object class="GtkMenuButton">
        <property name="margin-top">12</property>
        <property name="margin-bottom">12</property>
        <property name="margin-start">12</property>
        <property name="margin-end">12</property>
      </object>
    </child>
  </template>
</interface>
```

You can make a `MenuButton` show an icon or a label. If you choose to do neither of those, as we currently do, it shows an image displaying an arrow.

An inheritance tree of CSS nodes displays this situation:

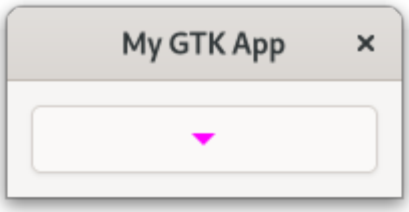
```
menubutton
╰── button.toggle
    ╰── <content>
         ╰── [arrow]
```

We see that the `menubutton` node has children, which themselves have children and attached style classes. Now we know that we have to add a CSS rule that applies to the `arrow` node, which is a descendant of `menubutton`.

Filename: listings/css/8/style.css

```
menubutton arrow {
  color: magenta;
}
```

Indeed, we get a `MenuButton` with a magenta arrow.



Set CSS Name and Use Exported Colors

We already learned how to give an instance of a widget a name with pseudo-classes. But what if we have a custom widget and we want to reference all instances of it? Let's see how to deal with this situation by messing with our To-Do app once more.

The class `TaskRow` inherits from `gtk::Box`, so we could just match for the node `box`. However, in that case we would also match with other instance of `gtk::Box`. What we will want to do instead is to give `TaskRow` its own CSS name. When calling `set_css_name`, we change the name of the CSS node of the widget class. In our case, the widget `TaskRow` then corresponds to the node `task-row`.

Filename: listings/todo/3/task_row/imp.rs

```rust
// The central trait for subclassing a GObject
#[glib::object_subclass]
impl ObjectSubclass for TaskRow {
    // `NAME` needs to match `class` attribute of template
    const NAME: &'static str = "TodoTaskRow";
    type Type = super::TaskRow;
    type ParentType = gtk::Box;

    fn class_init(klass: &mut Self::Class) {
        klass.bind_template();
        klass.set_css_name("task-row");
    }

    fn instance_init(obj: &glib::subclass::InitializingObject<Self>) {
        obj.init_template();
    }
}
```

What to do with the new node name now? Let's change the background color once more but this time with a twist. We are going to use the named color `success_color`.

Filename: listings/todo/3/resources/style.css

```css
task-row {
  background-color: @success_color;
}
```

The `Default` stylesheet of GTK provides pre-defined colors for various use-cases. As of this writing, these exported colors can only be found in its source code.

There we find the color `success_color`, which in real scenarios should be used to indicate success. We can then access the pre-defined color by adding an `@` in front of its name.

We also have to add `style.css` to `resources.gresource.xml`.

Filename: listings/todo/3/resources/resources.gresource.xml

```
 <?xml version="1.0" encoding="UTF-8"?>
 <gresources>
   <gresource prefix="/org/gtk_rs/Todo3/">
     <file compressed="true" preprocess="xml-stripblanks" alias="gtk/help-
overlay.ui">shortcuts.ui</file>
     <file compressed="true" preprocess="xml-stripblanks">task_row.ui</file>
     <file compressed="true" preprocess="xml-stripblanks">window.ui</file>
+    <file compressed="true">style.css</file>
   </gresource>
 </gresources>
```

Additionally, we call `load_css()` in `connect_startup`.

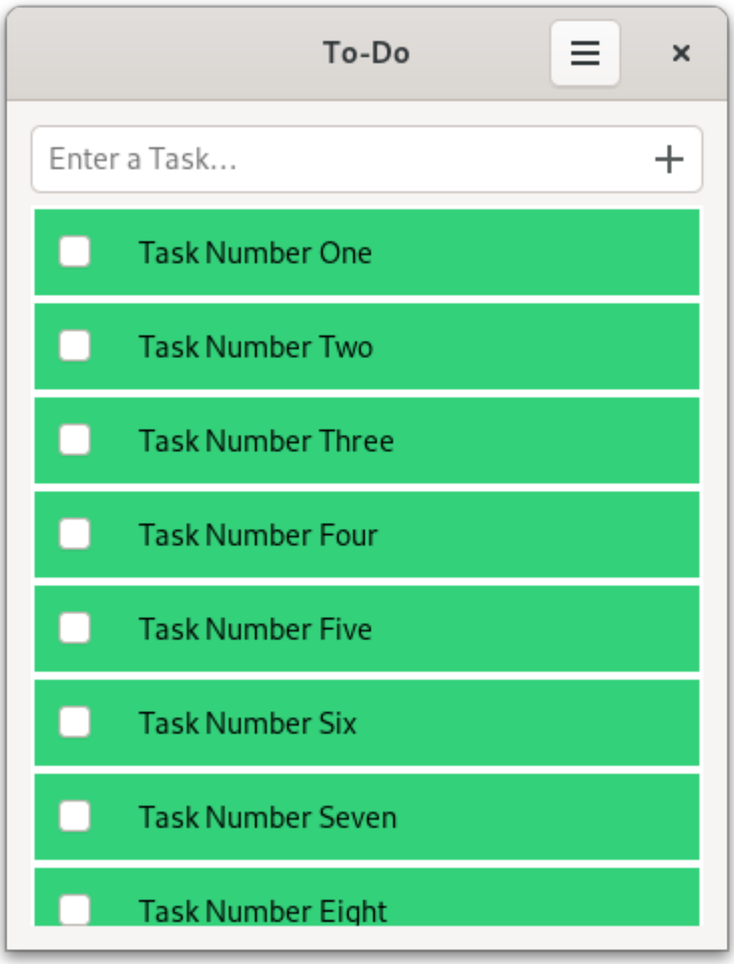
Filename: listings/todo/3/main.rs

```
    // Connect to signals
    app.connect_startup(|app| {
        setup_shortcuts(app);
        load_css()
    });
```

`load_css()` is very similar to the one shown at the beginning of the chapter. However, this time we load styles using `load_from_resource()`.

```
fn load_css() {
    // Load the CSS file and add it to the provider
    let provider = CssProvider::new();
    provider.load_from_resource("/org/gtk_rs/Todo3/style.css");

    // Add the provider to the default screen
    gtk::style_context_add_provider_for_display(
        &Display::default().expect("Could not connect to a display."),
        &provider,
        gtk::STYLE_PROVIDER_PRIORITY_APPLICATION,
    );
}
```

And that is how the task rows look like after the change. Probably better to revert this immediately again.

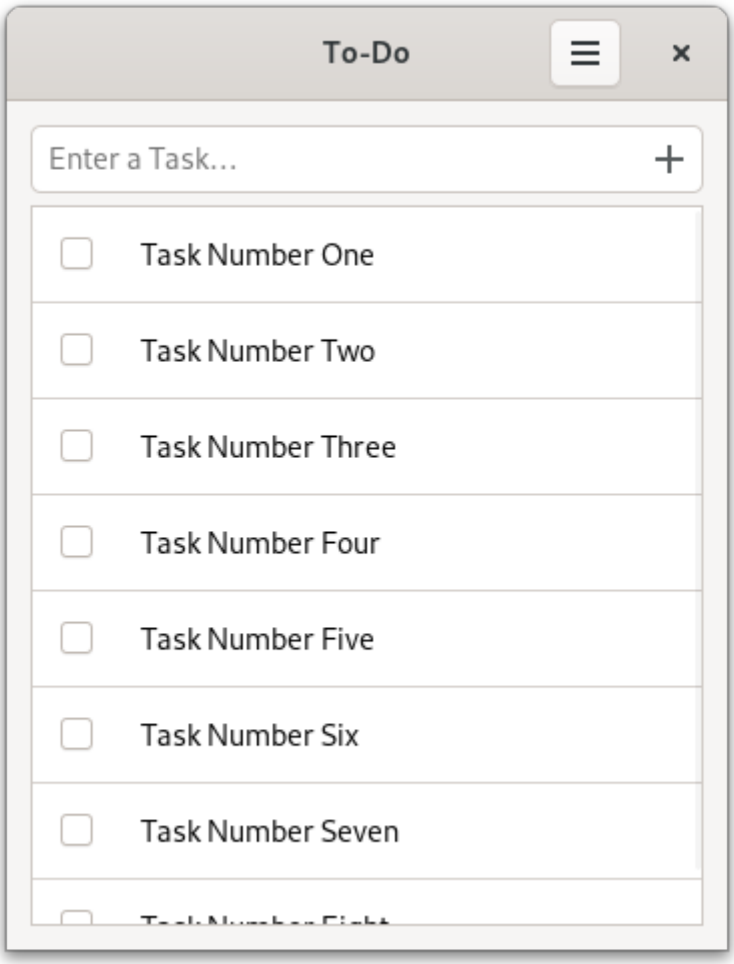


Adapt Todo App

Luckily, finding an actual use for CSS in our To-Do app isn't too hard. Until now the different tasks weren't nicely separated. We can change that by adding the `frame` and the `separators` style class to our `tasks_list`.

Filename: listings/todo/4/resources/window.ui

```
  <object class="GtkListView" id="tasks_list">
    <property name="valign">start</property>
+   <style>
+     <class name="frame"/>
+     <class name="separators"/>
+   </style>
  </object>
```



Conclusion

There are surely enough ways to define CSS rules. Let's briefly recap the syntax we learned. The following rule matches the node `arrow`, which is a descendant of the node `button` with the name `button-1` and the style classes `toggle` and `text-button`. The rule then actually applies, when we also hover over `arrow`.

```
button#button-1.toggle.text-button arrow:hover {
  color: magenta;
}
```

When the rule applies, the `color` parameter will be set to magenta. You can find the full list of supported parameters in GTK's documentation.

Libadwaita

If you target a certain platform with your GUI, you will want to follow the platform's Human Interface Guidelines (HIG). With a GTK application, chances are the platform is either elementary OS or GNOME. In this chapter we will discuss how to follow GNOME's HIG with libadwaita.

Libadwaita is a library augmenting GTK 4 which:

- provides widgets to better follow GNOME's HIG
- provides widgets to let applications change their layout based on the available space
- integrates the Adwaita stylesheet
- allows runtime recoloring with named colors
- adds API to support the cross-desktop dark style preference

In order to use the Rust bindings, add the libadwaita crate as dependency by executing:

```
cargo add libadwaita --rename adw --features v1_4
```

The versions of the `gtk4` and `libadwaita` crates need to be synced. Just remember that when you update one of them to the newest version to update the other one as well.

Installation of the library itself works similar to GTK. Just follow the installation instruction that is suitable for your distribution.

Linux

Fedora and derivatives:

```
sudo dnf install libadwaita-devel
```

Debian and derivatives:

```
sudo apt install libadwaita-1-dev
```

Arch and derivatives:

```
sudo pacman -S libadwaita
```

macOS

```
brew install libadwaita
```

Windows

If using gvsbuild

If you used `gvsbuild` to build GTK 4:

```
gvsbuild build libadwaita librsvg
```

If building manually with MSVC:

From the Windows start menu, search for `x64 Native Tools Command Prompt for VS 2019`. That will open a terminal configured to use MSVC x64 tools. From there, run the following commands:

```
cd /
git clone --branch libadwaita-1-3 https://gitlab.gnome.org/GNOME/
libadwaita.git --depth 1
cd libadwaita
meson setup builddir -Dprefix=C:/gnome -Dintrospection=disabled -Dvapi=false
meson install -C builddir
```

Work around missing icons

This workaround is needed for GTK < 4.10 due to this issue.

gvsbuild

From a command prompt:

```
xcopy /s /i C:\gtk-build\gtk\x64\release\share\icons\hicolor\scalable\apps C:
\gtk-build\gtk\x64\release\share\icons\hicolor\scalable\actions
gtk4-update-icon-cache.exe -t -f C:\gtk-
build\gtk\x64\release\share\icons\hicolor
```

Manually with MSVC

```
xcopy /s /i C:\gnome\share\icons\hicolor\scalable\apps C:
\gnome\share\icons\hicolor\scalable\actions
gtk4-update-icon-cache.exe -t -f C:\gnome\share\icons\hicolor
```

Let To-Do App use Libadwaita

Within this chapter we will adapt our To-Do app so that it follows GNOME's HIG. Let's start by installing Libadwaita and adding the `libadwaita` crate to our dependencies as explained in the previous chapter.

The simplest way to take advantage of Libadwaita is by replacing `gtk::Application` with `adw::Application`.

Filename: listings/todo/5/main.rs

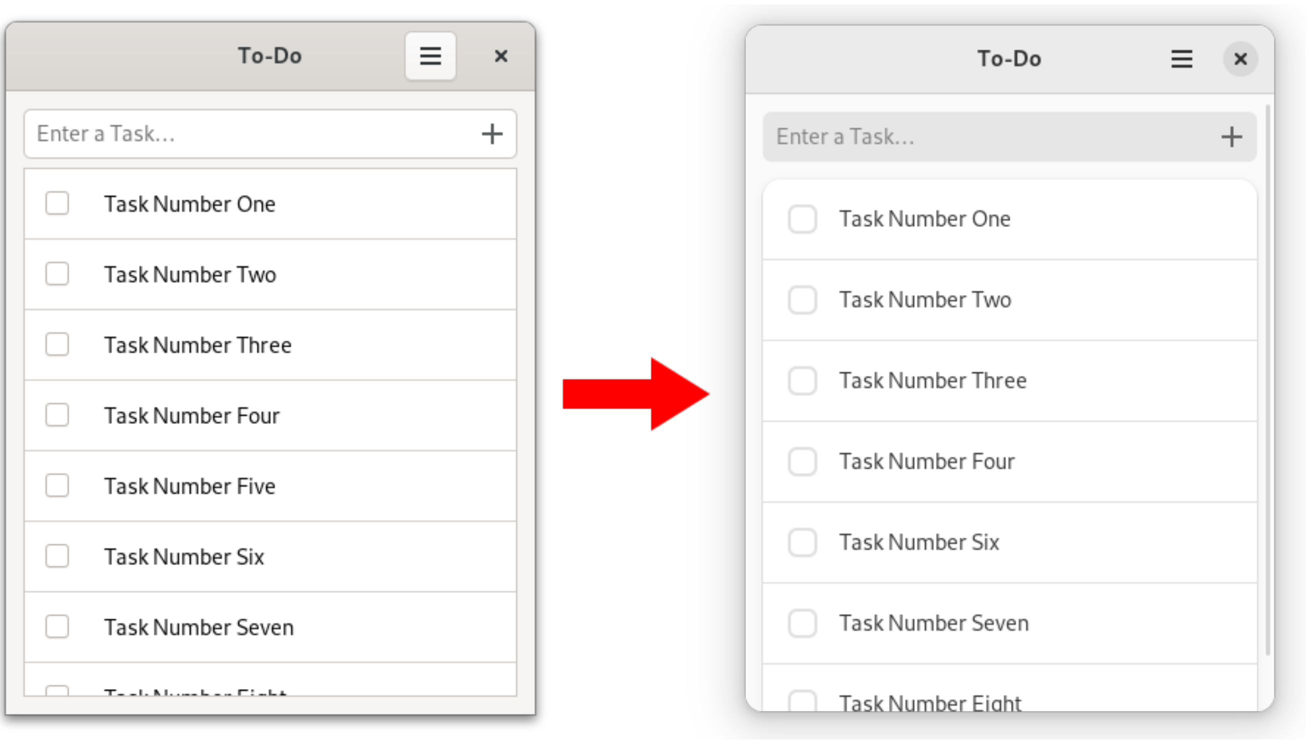
```
fn main() -> glib::ExitCode {
    gio::resources_register_include!("todo_5.gresource")
        .expect("Failed to register resources.");

    // Create a new application
    //          👇 changed
    let app = adw::Application::builder().application_id(APP_ID).build();

    // Connect to signals
    app.connect_startup(setup_shortcuts);
    app.connect_activate(build_ui);

    // Run the application
    app.run()
}

//                         👇 changed
fn setup_shortcuts(app: &adw::Application) {
    app.set_accels_for_action("win.filter('All')", &["<Ctrl>a"]);
    app.set_accels_for_action("win.filter('Open')", &["<Ctrl>o"]);
    app.set_accels_for_action("win.filter('Done')", &["<Ctrl>d"]);
}

//                  👇 changed
fn build_ui(app: &adw::Application) {
    // Create a new custom window and present it
    let window = Window::new(app);
    window.present();
}
```

Filename: listings/todo/5/window/mod.rs

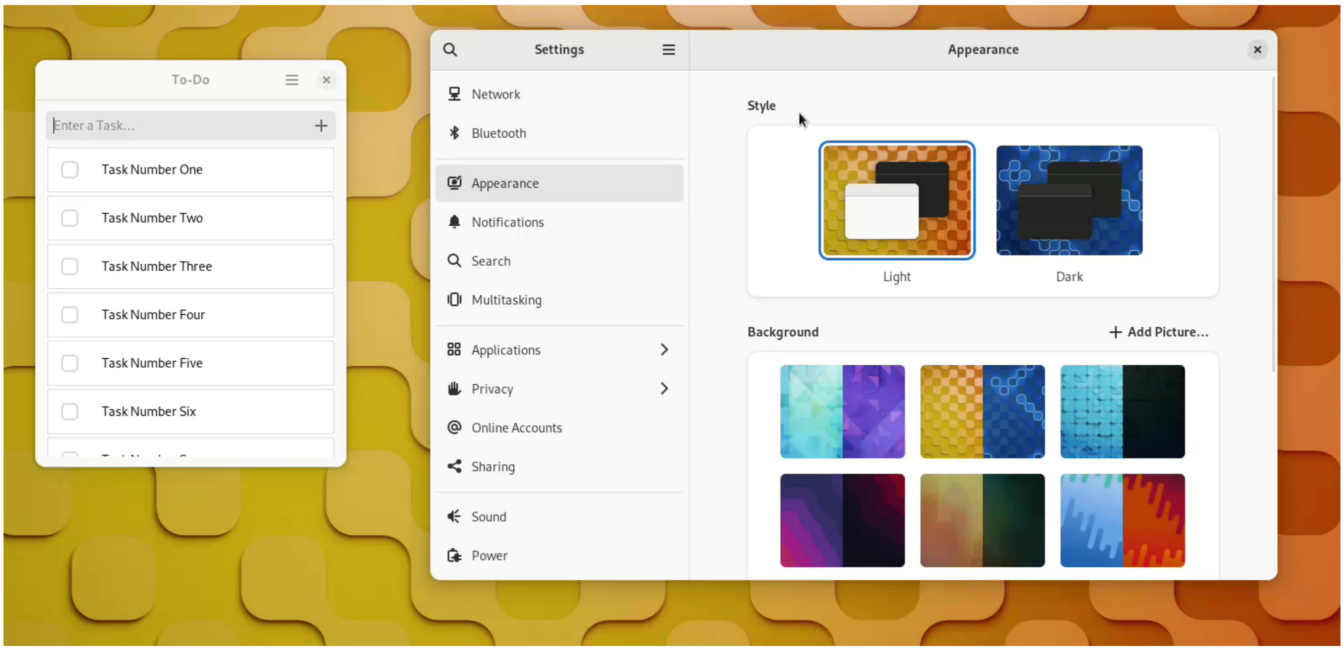
```
    pub fn new(app: &adw::Application) -> Self {
        // Create new window
        Object::builder().property("application", app).build()
    }
```

`adw::Application` calls `adw::init` internally and makes sure that translations, types, stylesheets, and icons are set up properly for Libadwaita. It also loads stylesheets automatically from resources as long as they are named correctly.

Looking at our To-Do app we can see that the looks of its widgets changed. This is because the `Default` stylesheet provided by GTK has been replaced with the `Adwaita` stylesheet provided by Libadwaita.



Also, our app now switches to the dark style together with the rest of the system.



Boxed lists

Of course Libadwaita is more than just a couple of stylesheets and a `StyleManager`. But before we get to the interesting stuff, we will make our lives easier for the future by replacing all occurrences of `gtk::prelude` and `gtk::subclass::prelude` with `adw::prelude` and `adw::subclass::prelude`. This works because the `adw` preludes, in addition to the Libadwaita-specific traits, re-export the corresponding `gtk` preludes.

Now we are going let our tasks follow the boxed lists pattern. The HIG does not require us to use this style and there's a good reason for that: it is incompatible with recycling lists. This means they cannot be used with list views and are therefore only appropriate for relatively small lists.

> Try to add tasks programmatically and see how many of them you have to add until the UI noticeably slows down. Determine for yourself if you think that is a reasonable number or if we should have rather stuck with list views.

We can use boxed lists by using `gtk::ListBox` instead of `gtk::ListView`. We will also add the `boxed-list` style class provided by Libadwaita.

Let's implement all these changes in the `window.ui` file. All of the changes are confined within the second child of the `ApplicationWindow`. To see the complete file, just click on the link after "Filename".

Filename: listings/todo/6/resources/window.ui

```
<child>
  <object class="GtkScrolledWindow">
    <property name="hscrollbar-policy">never</property>
    <property name="min-content-height">420</property>
    <property name="vexpand">True</property>
    <property name="child">
      <object class="AdwClamp">
        <property name="child">
          <object class="GtkBox">
            <property name="orientation">vertical</property>
            <property name="spacing">18</property>
            <property name="margin-top">24</property>
            <property name="margin-bottom">24</property>
            <property name="margin-start">12</property>
            <property name="margin-end">12</property>
            <child>
              <object class="GtkEntry" id="entry">
                <property name="placeholder-text" translatable="yes">Enter a
Task…</property>
                <property name="secondary-icon-name">list-add-symbolic</
property>
              </object>
            </child>
            <child>
              <object class="GtkListBox" id="tasks_list">
                <property name="visible">False</property>
                <property name="selection-mode">none</property>
                <style>
                  <class name="boxed-list" />
                </style>
              </object>
            </child>
          </object>
        </property>
      </object>
    </property>
  </object>
</child>
```

In order to follow the boxed list pattern, we switched to `gtk::ListBox`, set its property "selection-mode" to "none" and added the `boxed-list` style class.

Let's continue with `window/imp.rs`. The member variable `tasks_list` now describes a `ListBox` rather than a `ListView`.

Filename: listings/todo/6/window/imp.rs

```
// Object holding the state
#[derive(CompositeTemplate, Default)]
#[template(resource = "/org/gtk_rs/Todo6/window.ui")]
pub struct Window {
    #[template_child]
    pub entry: TemplateChild<Entry>,
    #[template_child]
    pub tasks_list: TemplateChild<ListBox>,
    pub tasks: RefCell<Option<gio::ListStore>>,
    pub settings: OnceCell<Settings>,
}

// The central trait for subclassing a GObject
#[glib::object_subclass]
impl ObjectSubclass for Window {
    // `NAME` needs to match `class` attribute of template
    const NAME: &'static str = "TodoWindow";
    type Type = super::Window;
    type ParentType = gtk::ApplicationWindow;

    fn class_init(klass: &mut Self::Class) {
        klass.bind_template();

        // Create action to remove done tasks and add to action group "win"
        klass.install_action("win.remove-done-tasks", None, |window, _, _| {
            window.remove_done_tasks();
        });
    }

    fn instance_init(obj: &InitializingObject<Self>) {
        obj.init_template();
    }
}
```

We now move on to `window/mod.rs` . `ListBox` supports models just fine, but without any widget recycling we don't need factories anymore. `setup_factory` can therefore be safely deleted. To setup the `ListBox` , we call `bind_model` in `setup_tasks` . There we specify the model, as well as a closure describing how to transform the given GObject into a widget the list box can display.

Filename: listings/todo/6/window/mod.rs

```
        // Wrap model with filter and selection and pass it to the list box
        let filter_model = FilterListModel::new(Some(self.tasks()),
self.filter());
        let selection_model = NoSelection::new(Some(filter_model.clone()));
        self.imp().tasks_list.bind_model(
            Some(&selection_model),
            clone!(@weak self as window => @default-panic, move |obj| {
                let task_object = obj.downcast_ref().expect("The object
should be of type `TaskObject`.");
                let row = window.create_task_row(task_object);
                row.upcast()
            }),
        );
```

We still have to specify the `create_task_row` method. Here, we create an `adw::ActionRow` with a `gtk::CheckButton` as activatable widget. Without recycling, a GObject will always belong to the same widget. That means we can just bind their properties without having to worry about unbinding them later on.

Filename: listings/todo/6/window/mod.rs

```
    fn create_task_row(&self, task_object: &TaskObject) -> ActionRow {
        // Create check button
        let check_button = CheckButton::builder()
            .valign(Align::Center)
            .can_focus(false)
            .build();

        // Create row
        let row = ActionRow::builder()
            .activatable_widget(&check_button)
            .build();
        row.add_prefix(&check_button);

        // Bind properties
        task_object
            .bind_property("completed", &check_button, "active")
            .bidirectional()
            .sync_create()
            .build();
        task_object
            .bind_property("content", &row, "title")
            .sync_create()
            .build();

        // Return row
        row
    }
```

When using boxed lists, you also have to take care to hide the `ListBox` when there is no task present.

Filename: listings/todo/6/window/mod.rs

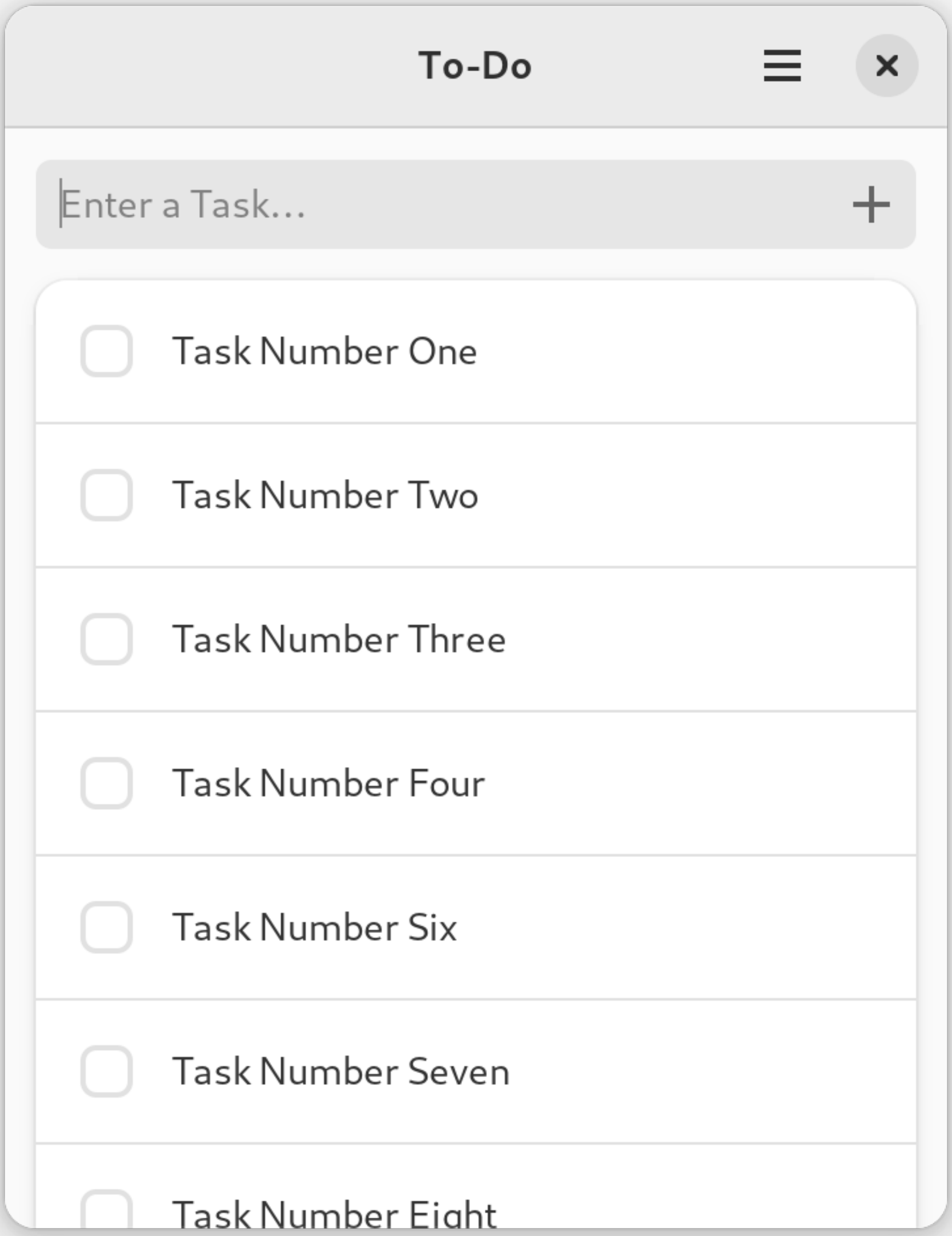
```
        // Assure that the task list is only visible when it is supposed to
        self.set_task_list_visible(&self.tasks());
        self.tasks().connect_items_changed(
            clone!(@weak self as window => move |tasks, _, _, _| {
                window.set_task_list_visible(tasks);
            }),
        );
```

Finally, we define the `set_task_list_visible` method.

Filename: listings/todo/6/window/mod.rs

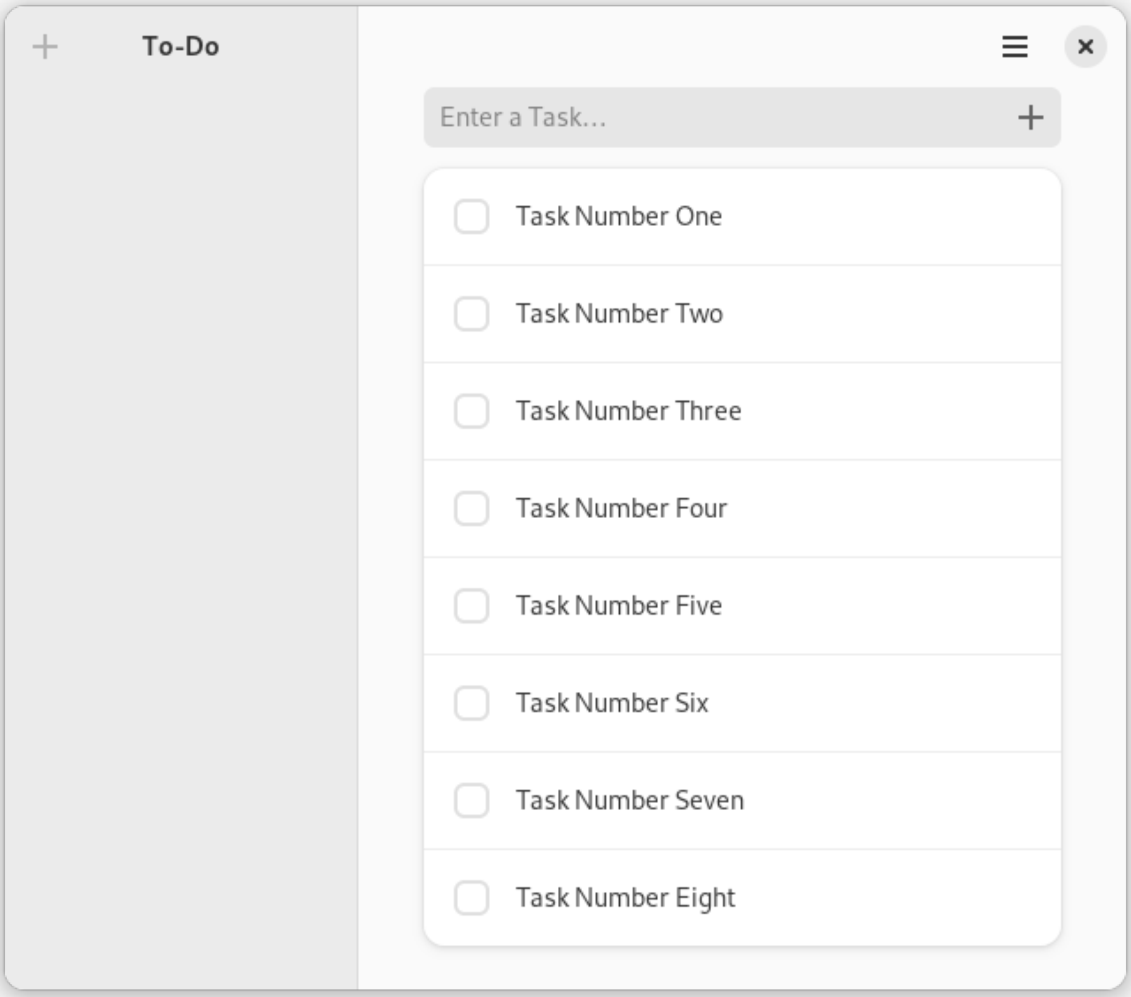
```
    /// Assure that `tasks_list` is only visible
    /// if the number of tasks is greater than 0
    fn set_task_list_visible(&self, tasks: &gio::ListStore) {
        self.imp().tasks_list.set_visible(tasks.n_items() > 0);
    }
```

This is how the boxed list style looks like in our app.


To-Do
Enter a Task...
Task Number One
Task Number Two
Task Number Three
Task Number Four
Task Number Six
Task Number Seven
Task Number Eight

Adding Collections

Using Libadwaita on its own was already a big leap forward when it came to the look and feel of the To-Do app. Let us go one step further by adding a way to group tasks into collections. These collections will get their own sidebar on the left of the app. We will start by adding an empty sidebar without any functionality.

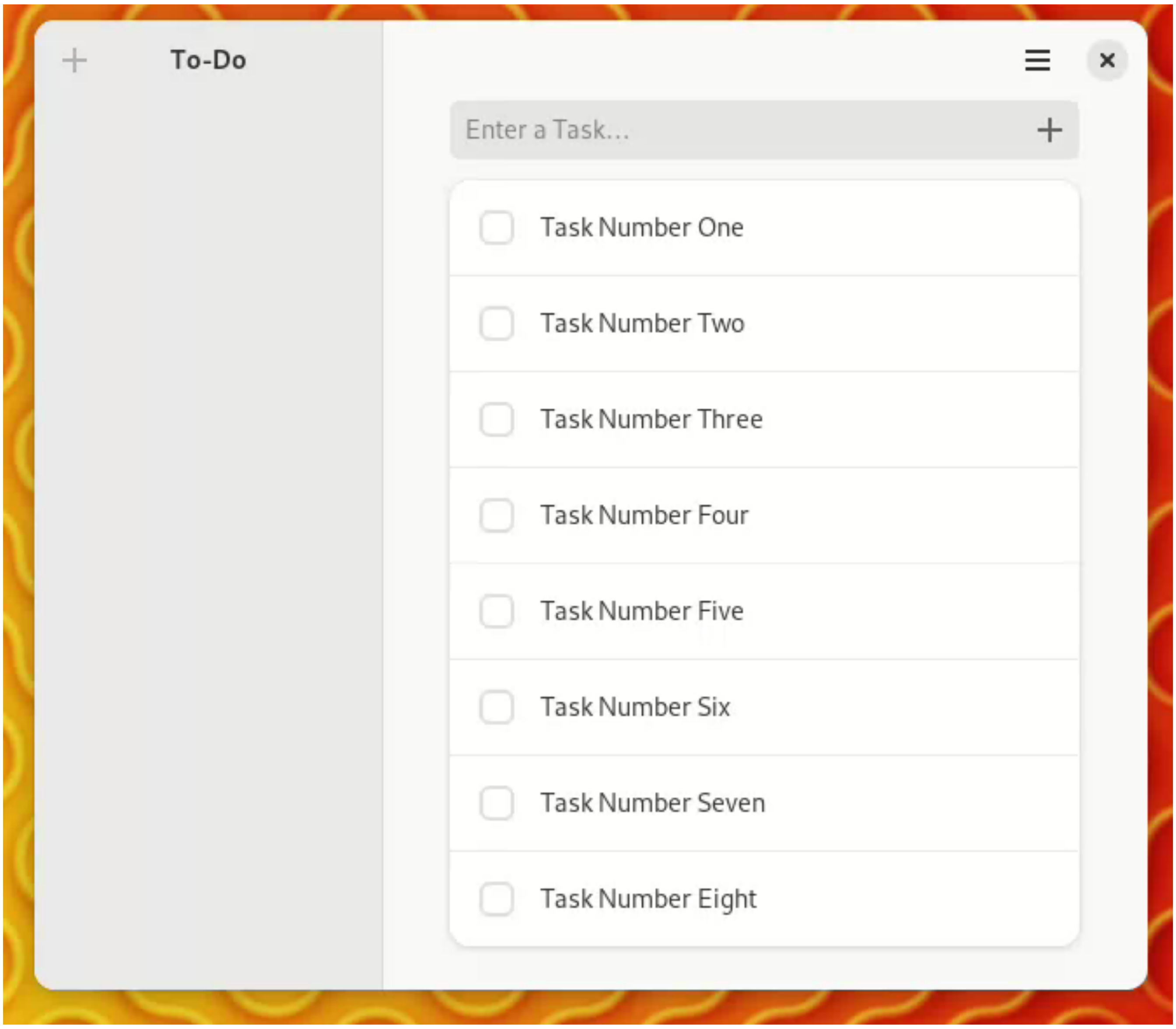


There are a couple of steps we have to go through to get to this state. First, we have to replace `gtk::ApplicationWindow` with `adw::ApplicationWindow`. The main difference between those two is that `adw::ApplicationWindow` has no titlebar area. That comes in handy when we build up our interface with `adw::NavigationSplitView`. In the screenshot above, the `NavigationSplitView` adds a sidebar for the collection view to the left, while the task view occupies the space on the right. When using `adw::ApplicationWindow` the collection view and task view have their own `adw::HeaderBar` and the separator spans over the whole window.

Filename: listings/todo/7/resources/window.ui

```xml
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <menu id="main-menu">
    <!--Menu implementation-->
  </menu>
  <template class="TodoWindow" parent="AdwApplicationWindow">
    <property name="title" translatable="yes">To-Do</property>
    <property name="width-request">360</property>
    <property name="height-request">200</property>
    <child>
      <object class="AdwBreakpoint">
        <condition>max-width: 500sp</condition>
        <setter object="split_view" property="collapsed">True</setter>
      </object>
    </child>
    <property name="content">
      <object class="AdwNavigationSplitView" id="split_view">
        <property name="min-sidebar-width">200</property>
        <property name="sidebar">
          <object class="AdwNavigationPage">
            <!--Collection view implementation-->
          </object>
        </property>
        <property name="content">
          <object class="AdwNavigationPage">
            <!--Task view implementation-->
          </object>
        </property>
      </object>
    </property>
  </template>
</interface>
```

`NavigationSplitView` also helps with making your app adaptive/ As soon as the requested size is too small to fit all children at the same time, the splitview collapses, and starts behaving like a `gtk::Stack`. This means that it only displays one of its children at a time. The adaptive behavior of the leaflet allows the To-Do app to work on smaller screen sizes (like e.g. phones) even with the added collection view.



We add the necessary UI elements for the collection view, such as a header bar with a button to add a new collection, as well as the list box `collections_list` to display the collections later on. We also add the style navigations-sidebar to `collections_list`.

Filename: listings/todo/7/resources/window.ui

```xml
<object class="AdwNavigationPage">
  <property name="title" bind-source="TodoWindow"
    bind-property="title" bind-flags="sync-create" />
  <property name="child">
    <object class="AdwToolbarView">
      <child type="top">
        <object class="AdwHeaderBar">
          <child type="start">
            <object class="GtkToggleButton">
              <property name="icon-name">list-add-symbolic</property>
              <property name="tooltip-text" translatable="yes">New
Collection</property>
              <property name="action-name">win.new-collection</property>
            </object>
          </child>
        </object>
      </child>
      <property name="content">
        <object class="GtkScrolledWindow">
          <property name="child">
            <object class="GtkListBox" id="collections_list">
              <style>
                <class name="navigation-sidebar" />
              </style>
            </object>
          </property>
        </object>
      </property>
    </object>
```

We also add a header bar to the task view.

Filename: listings/todo/7/resources/window.ui

```
<object class="AdwNavigationPage">
  <property name="title" translatable="yes">Tasks</property>
  <property name="child">
    <object class="AdwToolbarView">
      <child type="top">
        <object class="AdwHeaderBar">
          <property name="show-title">False</property>
          <child type="end">
            <object class="GtkMenuButton">
              <property name="icon-name">open-menu-symbolic</property>
              <property name="menu-model">main-menu</property>
              <property name="tooltip-text" translatable="yes">Main Menu</
property>
            </object>
          </child>
        </object>
      </child>
      <property name="content">
        <object class="GtkScrolledWindow">
          <property name="child">
            <object class="AdwClamp">
              <property name="maximum-size">400</property>
              <property name="tightening-threshold">300</property>
              <property name="child">
                <object class="GtkBox">
                  <property name="orientation">vertical</property>
                  <property name="margin-start">12</property>
                  <property name="margin-end">12</property>
                  <property name="spacing">12</property>
                  <child>
                    <object class="GtkEntry" id="entry">
                      <property name="placeholder-text"
translatable="yes">Enter a Task…</property>
                      <property name="secondary-icon-name">list-add-
symbolic</property>
                    </object>
                  </child>
                  <child>
                    <object class="GtkListBox" id="tasks_list">
                      <property name="visible">False</property>
                      <property name="selection-mode">none</property>
                      <style>
                        <class name="boxed-list" />
                      </style>
                    </object>
                  </child>
                </object>
              </property>
            </object>
          </property>
        </object>
      </property>
    </object>
  </property>
</object>
```

We also have to adapt the window implementation. For example, the parent type of our window is now `adw::ApplicationWindow` instead of `gtk::ApplicationWindow`.

Filename: listings/todo/7/window/imp.rs

```rust
// The central trait for subclassing a GObject
#[glib::object_subclass]
impl ObjectSubclass for Window {
    // `NAME` needs to match `class` attribute of template
    const NAME: &'static str = "TodoWindow";
    type Type = super::Window;
    //                👇 changed
    type ParentType = adw::ApplicationWindow;

    fn class_init(klass: &mut Self::Class) {
        klass.bind_template();

        // Create action to remove done tasks and add to action group "win"
        klass.install_action("win.remove-done-tasks", None, |window, _, _| {
            window.remove_done_tasks();
        });
    }

    fn instance_init(obj: &InitializingObject<Self>) {
        obj.init_template();
    }
}
```

That also means that we have to implement the trait `AdwApplicationWindowImpl`.

Filename: listings/todo/7/window/imp.rs

```rust
// Trait shared by all adwaita application windows
impl AdwApplicationWindowImpl for Window {}
```

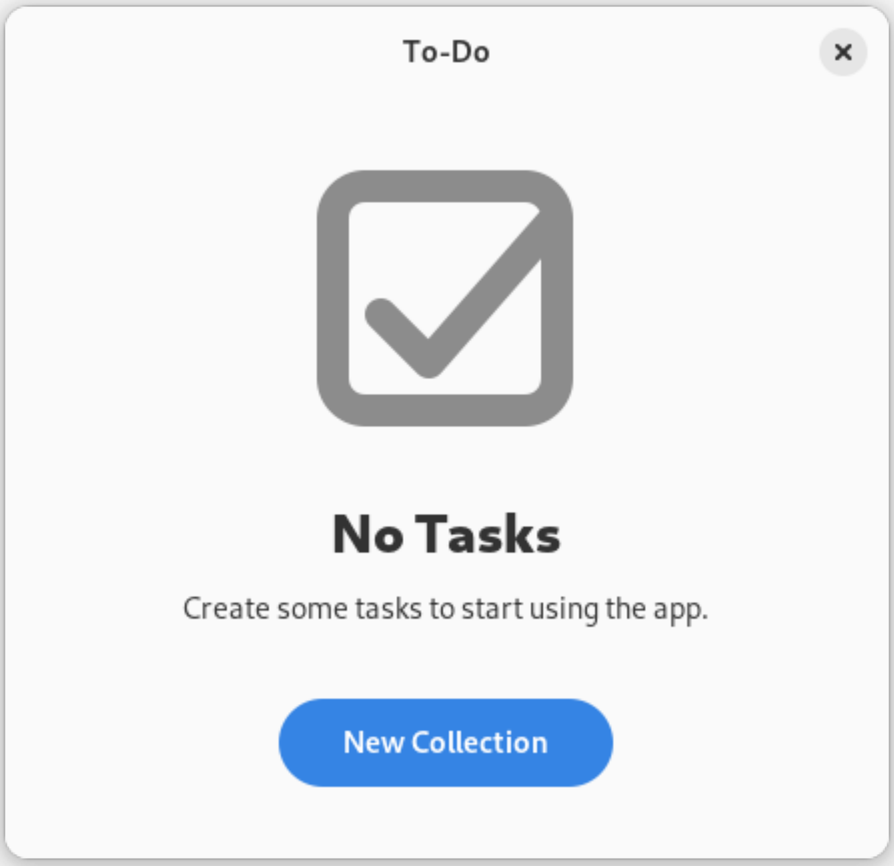
Finally, we add `adw::ApplicationWindow` to the ancestors of `Window` in `mod.rs`.

Filename: listings/todo/7/window/mod.rs

```rust
glib::wrapper! {
    pub struct Window(ObjectSubclass<imp::Window>)
        //        👇 changed
        @extends adw::ApplicationWindow, gtk::ApplicationWindow, gtk::Window,
gtk::Widget,
        @implements gio::ActionGroup, gio::ActionMap, gtk::Accessible,
gtk::Buildable,
                    gtk::ConstraintTarget, gtk::Native, gtk::Root,
gtk::ShortcutManager;
}
```

Placeholder Page

Even before we start to populate the collection view, we ought to think about a different challenge: the empty state of our To-Do app. Before, the empty state without a single task was quite okay. It was clear that you had to add tasks in the entry bar. However, now the situation is different. Users will have to add a collection first, and we have to make that clear. The GNOME HIG suggests to use a placeholder page for that. In our case, this placeholder page will be presented to the user if they open the app without any collections present.



We now wrap our UI in a `gtk::Stack`. One stack page describes the placeholder page, the other describes the main page. We will later wire up the logic to display the correct stack page in the Rust code.

Filename: listings/todo/8/resources/window.ui

```xml
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <menu id="main-menu">
    <!--Menu implementation-->
  </menu>
  <template class="TodoWindow" parent="AdwApplicationWindow">
    <property name="title" translatable="yes">To-Do</property>
    <property name="width-request">360</property>
    <property name="height-request">200</property>
    <child>
      <object class="AdwBreakpoint">
        <condition>max-width: 500sp</condition>
        <setter object="split_view" property="collapsed">True</setter>
      </object>
    </child>
    <property name="content">
      <object class="GtkStack" id="stack">
        <property name="transition-type">crossfade</property>
        <child>
          <object class="GtkStackPage">
            <property name="name">placeholder</property>
            <property name="child">
              <object class="GtkBox">
                <!--Placeholder page implementation-->
              </object>
            </property>
          </object>
        </child>
        <child>
          <object class="GtkStackPage">
            <property name="name">main</property>
            <property name="child">
              <object class="AdwNavigationSplitView" id="split_view">
                <!--Main page implementation-->
              </object>
            </property>
          </object>
        </child>
      </object>
    </property>
  </template>
</interface>
```

In order to create the pageholder page as displayed before, we combine a flat header bar with `adw::StatusPage` .

Filename: listings/todo/8/resources/window.ui

```
<object class="GtkBox">
  <property name="orientation">vertical</property>
  <child>
    <object class="GtkHeaderBar">
      <style>
        <class name="flat" />
      </style>
    </object>
  </child>
  <child>
    <object class="GtkWindowHandle">
      <property name="vexpand">True</property>
      <property name="child">
        <object class="AdwStatusPage">
          <property name="icon-name">checkbox-checked-symbolic</property>
          <property name="title" translatable="yes">No Tasks</property>
          <property name="description" translatable="yes">Create some tasks
to start using the app.</property>
          <property name="child">
            <object class="GtkButton">
              <property name="label" translatable="yes">_New Collection</
property>
              <property name="use-underline">True</property>
              <property name="halign">center</property>
              <property name="action-name">win.new-collection</property>
              <style>
                <class name="pill" />
                <class name="suggested-action" />
              </style>
            </object>
          </property>
        </object>
      </property>
    </object>
  </child>
</object>
```

Collections

We still need a way to store our collections. Just like we have already created `TaskObject`, we will now introduce `CollectionObject`. It will have the members `title` and `tasks`, both of which will be exposed as properties. As usual, the full implementation can be seen by clicking at the eye symbol at the top right of the snippet.

Filename: listings/todo/8/collection_object/imp.rs

```
// Object holding the state
#[derive(Properties, Default)]
#[properties(wrapper_type = super::CollectionObject)]
pub struct CollectionObject {
    #[property(get, set)]
    pub title: RefCell<String>,
    #[property(get, set)]
    pub tasks: OnceCell<gio::ListStore>,
}

// The central trait for subclassing a GObject
#[glib::object_subclass]
impl ObjectSubclass for CollectionObject {
    const NAME: &'static str = "TodoCollectionObject";
    type Type = super::CollectionObject;
}
```

We also add the struct `CollectionData` to aid in serialization and deserialization.

Filename: listings/todo/8/collection_object/mod.rs

```
#[derive(Default, Clone, Serialize, Deserialize)]
pub struct CollectionData {
    pub title: String,
    pub tasks_data: Vec<TaskData>,
}
```

Finally, we add methods to `CollectionObject` in order to

- construct it with `new`,
- easily access the tasks `ListStore` with `tasks` and
- convert to and from `CollectionData` with `to_collection_data` and `from_collection_data`.

Filename: listings/todo/8/collection_object/mod.rs

```rust
impl CollectionObject {
    pub fn new(title: &str, tasks: gio::ListStore) -> Self {
        Object::builder()
            .property("title", title)
            .property("tasks", tasks)
            .build()
    }

    pub fn to_collection_data(&self) -> CollectionData {
        let title = self.imp().title.borrow().clone();
        let tasks_data = self
            .tasks()
            .iter::<TaskObject>()
            .filter_map(Result::ok)
            .map(|task_object| task_object.task_data())
            .collect();
        CollectionData { title, tasks_data }
    }

    pub fn from_collection_data(collection_data: CollectionData) -> Self {
        let title = collection_data.title;
        let tasks_to_extend: Vec<TaskObject> = collection_data
            .tasks_data
            .into_iter()
            .map(TaskObject::from_task_data)
            .collect();

        let tasks = gio::ListStore::new::<TaskObject>();
        tasks.extend_from_slice(&tasks_to_extend);

        Self::new(&title, tasks)
    }
}
```

Window

In order to hook up the new logic, we have to add more state to `imp::Window`. There are additional widgets that we access via the `template_child` macro. Additionally, we reference the `collections` list store, the `current_collection` as well as the `current_filter_model`. We also store `tasks_changed_handler_id`. Its purpose will become clear in later snippets.

Filename: listings/todo/8/window/imp.rs

```
// Object holding the state
#[derive(CompositeTemplate, Default)]
#[template(resource = "/org/gtk_rs/Todo8/window.ui")]
pub struct Window {
    pub settings: OnceCell<Settings>,
    #[template_child]
    pub entry: TemplateChild<Entry>,
    #[template_child]
    pub tasks_list: TemplateChild<ListBox>,
    // 👇 all members below are new
    #[template_child]
    pub collections_list: TemplateChild<ListBox>,
    #[template_child]
    pub split_view: TemplateChild<NavigationSplitView>,
    #[template_child]
    pub stack: TemplateChild<Stack>,
    pub collections: OnceCell<gio::ListStore>,
    pub current_collection: RefCell<Option<CollectionObject>>,
    pub current_filter_model: RefCell<Option<FilterListModel>>,
    pub tasks_changed_handler_id: RefCell<Option<SignalHandlerId>>,
}
```

Further, we add a couple of helper methods which will come in handy later on.

Filename: listings/todo/8/window/mod.rs

```
    fn tasks(&self) -> gio::ListStore {
        self.current_collection().tasks()
    }

    fn current_collection(&self) -> CollectionObject {
        self.imp()
            .current_collection
            .borrow()
            .clone()
            .expect("`current_collection` should be set in
`set_current_collections`.")
    }

    fn collections(&self) -> gio::ListStore {
        self.imp()
            .collections
            .get()
            .expect("`collections` should be set in `setup_collections`.")
            .clone()
    }

    fn set_filter(&self) {
        self.imp()
            .current_filter_model
            .borrow()
            .clone()
            .expect("`current_filter_model` should be set in
`set_current_collection`.")
            .set_filter(self.filter().as_ref());
    }
```

As always, we want our data to be saved when we close the window. Since most of the implementation is in the method `CollectionObject::to_collection_data`, the implementation of `close_request` doesn't change much.

Filename: listings/todo/8/window/imp.rs

```
// Trait shared by all windows
impl WindowImpl for Window {
    fn close_request(&self) -> glib::Propagation {
        // Store task data in vector
        let backup_data: Vec<CollectionData> = self
            .obj()
            .collections()
            .iter::<CollectionObject>()
            .filter_map(|collection_object| collection_object.ok())
            .map(|collection_object| collection_object.to_collection_data())
            .collect();

        // Save state to file
        let file = File::create(data_path()).expect("Could not create json
file.");
        serde_json::to_writer(file, &backup_data)
            .expect("Could not write data to json file");

        // Pass close request on to the parent
        self.parent_close_request()
    }
}
```

`constructed` stays mostly the same as well. Instead of `setup_tasks` we now call `setup_collections`.

Filename: listings/todo/8/window/imp.rs

```
// Trait shared by all GObjects
impl ObjectImpl for Window {
    fn constructed(&self) {
        // Call "constructed" on parent
        self.parent_constructed();

        // Setup
        let obj = self.obj();
        obj.setup_settings();
        obj.setup_collections();
        obj.restore_data();
        obj.setup_callbacks();
        obj.setup_actions();
    }
}
```

`setup_collections` sets up the `collections` list store as well as assuring that changes in the model will be reflected in the `collections_list`. To do that it uses the method `create_collection_row`.

Filename: listings/todo/8/window/mod.rs

```
    fn setup_collections(&self) {
        let collections = gio::ListStore::new::<CollectionObject>();
        self.imp()
            .collections
            .set(collections.clone())
            .expect("Could not set collections");

        self.imp().collections_list.bind_model(
            Some(&collections),
            clone!(@weak self as window => @default-panic, move |obj| {
                let collection_object = obj
                    .downcast_ref()
                    .expect("The object should be of type
`CollectionObject`.");
                let row = window.create_collection_row(collection_object);
                row.upcast()
            }),
        )
    }
```

`create_collection_row` takes a `CollectionObject` and builds a `gtk::ListBoxRow` from its information.

Filename: listings/todo/8/window/mod.rs

```
    fn create_collection_row(
        &self,
        collection_object: &CollectionObject,
    ) -> ListBoxRow {
        let label = Label::builder()
            .ellipsize(pango::EllipsizeMode::End)
            .xalign(0.0)
            .build();

        collection_object
            .bind_property("title", &label, "label")
            .sync_create()
            .build();

        ListBoxRow::builder().child(&label).build()
    }
```

We also adapt `restore_data`. Again, the heavy lifting comes from `CollectionObject::from_collection_data`, so we don't have to change too much here. Since the rows of `collections_list` can be selected, we have to select one of them after restoring the data. We choose the first one and let the method `set_current_collection` do the rest.

Filename: listings/todo/8/window/mod.rs

```
    fn restore_data(&self) {
        if let Ok(file) = File::open(data_path()) {
            // Deserialize data from file to vector
            let backup_data: Vec<CollectionData> =
serde_json::from_reader(file)
                .expect(
                    "It should be possible to read `backup_data` from the
json file.",
                );

            // Convert `Vec<CollectionData>` to `Vec<CollectionObject>`
            let collections: Vec<CollectionObject> = backup_data
                .into_iter()
                .map(CollectionObject::from_collection_data)
                .collect();

            // Insert restored objects into model
            self.collections().extend_from_slice(&collections);

            // Set first collection as current
            if let Some(first_collection) = collections.first() {
                self.set_current_collection(first_collection.clone());
            }
        }
    }
```

`set_current_collection` assures that all elements accessing tasks refer to the task model of the current collection. We bind the `tasks_list` to the current collection and store the filter model. Whenever there are no tasks in our current collection we want to hide our tasks list. Otherwise, the list box will leave a bad-looking line behind. However, we don't want to accumulate signal handlers whenever we switch collections. This is why we store the `tasks_changed_handler_id` and disconnect the old handler as soon as we set a new collection. Finally, we select the collection row.

Filename: listings/todo/8/window/mod.rs

```
    fn set_current_collection(&self, collection: CollectionObject) {
        // Wrap model with filter and selection and pass it to the list box
        let tasks = collection.tasks();
        let filter_model = FilterListModel::new(Some(tasks.clone()),
self.filter());
        let selection_model = NoSelection::new(Some(filter_model.clone()));
        self.imp().tasks_list.bind_model(
            Some(&selection_model),
            clone!(@weak self as window => @default-panic, move |obj| {
                let task_object = obj
                    .downcast_ref()
                    .expect("The object should be of type `TaskObject`.");
                let row = window.create_task_row(task_object);
                row.upcast()
            }),
        );

        // Store filter model
        self.imp().current_filter_model.replace(Some(filter_model));

        // If present, disconnect old `tasks_changed` handler
        if let Some(handler_id) = self.imp().tasks_changed_handler_id.take()
{
            self.tasks().disconnect(handler_id);
        }

        // Assure that the task list is only visible when it is supposed to
        self.set_task_list_visible(&tasks);
        let tasks_changed_handler_id = tasks.connect_items_changed(
            clone!(@weak self as window => move |tasks, _, _, _| {
                window.set_task_list_visible(tasks);
            }),
        );
        self.imp()
            .tasks_changed_handler_id
            .replace(Some(tasks_changed_handler_id));

        // Set current tasks
        self.imp().current_collection.replace(Some(collection));

        self.select_collection_row();
    }
```

Previously, we used the method `set_task_list_visible`. It assures that `tasks_list` is only visible if the number of tasks is greater than 0.

Filename: listings/todo/8/window/mod.rs

```
    fn set_task_list_visible(&self, tasks: &gio::ListStore) {
        self.imp().tasks_list.set_visible(tasks.n_items() > 0);
    }
```

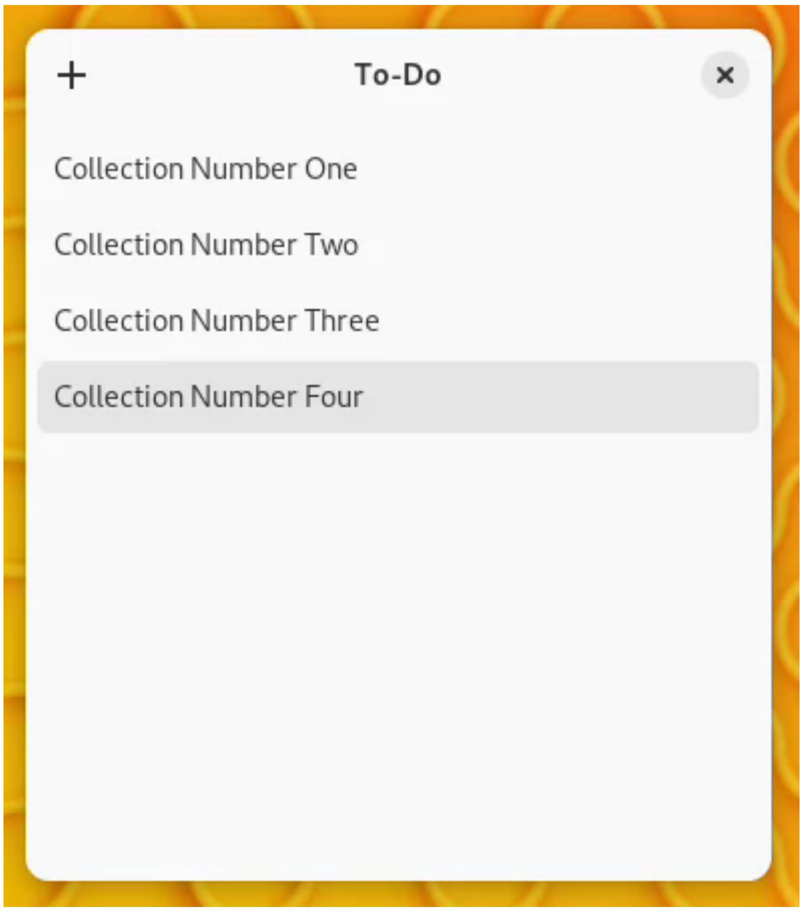
`select_collection_row` assures that the row for the current collection is selected in `collections_list`.

Filename: listings/todo/8/window/mod.rs

```
    fn select_collection_row(&self) {
        if let Some(index) =
self.collections().find(&self.current_collection()) {
            let row = self.imp().collections_list.row_at_index(index as i32);
            self.imp().collections_list.select_row(row.as_ref());
        }
    }
```

Message Dialog

There isn't yet a way to add a collection. Let's implement that functionality.



Collection Number One
Collection Number Two
Collection Number Three
Collection Number Four

The screencast above demonstrates the desired behavior. When we activate the button with the `+` symbol, a dialog appears. While the entry is empty, the "Create" button remains insensitive. As soon as we start typing, the button becomes sensitive. When we remove all typed letters and the entry becomes empty again, the "Create" button becomes insensitive and the entry gets the "error" style. After clicking the "Create" button, a new collection is created, and we navigate to its task view.

To implement that behavior we will first add a "new-collection" action to `class_init` method. This action will be activated by a click on the `+` button as well as on the button in the placeholder page. We are using `install_action_async`. It is a convenient way to add asynchronous actions to subclassed widgets.

Filename: listings/todo/8/window/imp.rs

```
// The central trait for subclassing a GObject
#[glib::object_subclass]
impl ObjectSubclass for Window {
    // `NAME` needs to match `class` attribute of template
    const NAME: &'static str = "TodoWindow";
    type Type = super::Window;
    type ParentType = adw::ApplicationWindow;

    fn class_init(klass: &mut Self::Class) {
        klass.bind_template();

        // Create action to remove done tasks and add to action group "win"
        klass.install_action("win.remove-done-tasks", None, |window, _, _| {
            window.remove_done_tasks();
        });

        // Create async action to create new collection and add to action
group "win"
        klass.install_action_async(
            "win.new-collection",
            None,
            |window, _, _| async move {
                window.new_collection().await;
            },
        );
    }

    fn instance_init(obj: &InitializingObject<Self>) {
        obj.init_template();
    }
}
```

As soon as the "new-collection" action is activated, the `async new_collection` method is called. Here, we create the `adw::MessageDialog`, set up the buttons as well as add the entry to it. We add a callback to the entry to ensure that when the content changes, an empty content sets `dialog_button` as insensitive and adds an "error" CSS class to the entry. We then `await` on the user pressing a button on the dialog. If they click "Cancel", we simply return. However, if they click "Create", we want a new collection to be created and set as current collection. Afterwards we navigate forward on our leaflet, which means we navigate to the task view.

Filename: listings/todo/8/window/mod.rs

```
    async fn new_collection(&self) {
        // Create entry
        let entry = Entry::builder()
            .placeholder_text("Name")
            .activates_default(true)
            .build();

        let cancel_response = "cancel";
        let create_response = "create";

        // Create new dialog
        let dialog = MessageDialog::builder()
            .heading("New Collection")
            .transient_for(self)
            .modal(true)
            .destroy_with_parent(true)
            .close_response(cancel_response)
            .default_response(create_response)
            .extra_child(&entry)
            .build();
        dialog
            .add_responses(&[(cancel_response, "Cancel"), (create_response,
"Create")]);
        // Make the dialog button insensitive initially
        dialog.set_response_enabled(create_response, false);
        dialog.set_response_appearance(create_response,
ResponseAppearance::Suggested);

        // Set entry's css class to "error", when there is no text in it
        entry.connect_changed(clone!(@weak dialog => move |entry| {
            let text = entry.text();
            let empty = text.is_empty();

            dialog.set_response_enabled(create_response, !empty);

            if empty {
                entry.add_css_class("error");
            } else {
                entry.remove_css_class("error");
            }
        }));

        let response = dialog.choose_future().await;

        // Return if the user chose `cancel_response`
        if response == cancel_response {
            return;
        }

        // Create a new list store
        let tasks = gio::ListStore::new::<TaskObject>();

        // Create a new collection object from the title the user provided
        let title = entry.text().to_string();
        let collection = CollectionObject::new(&title, tasks);
        // Add new collection object and set current tasks
        self.collections().append(&collection);
        self.set_current_collection(collection);

        // Show the content
        self.imp().split_view.set_show_content(true);
    }
```

We also add more callbacks to `setup_callbacks`. Importantly, we want to filter our current task model whenever the value of the "filter" setting changes. Whenever the items of our collections change we also want to set the stack. This makes sure that our placeholder page is shown if there are no collections. Finally, we assure that when we click on a row of `collections_list`, `current_collection` is set to the selected collection and the split view shows the task view.

Filename: listings/todo/8/window/mod.rs

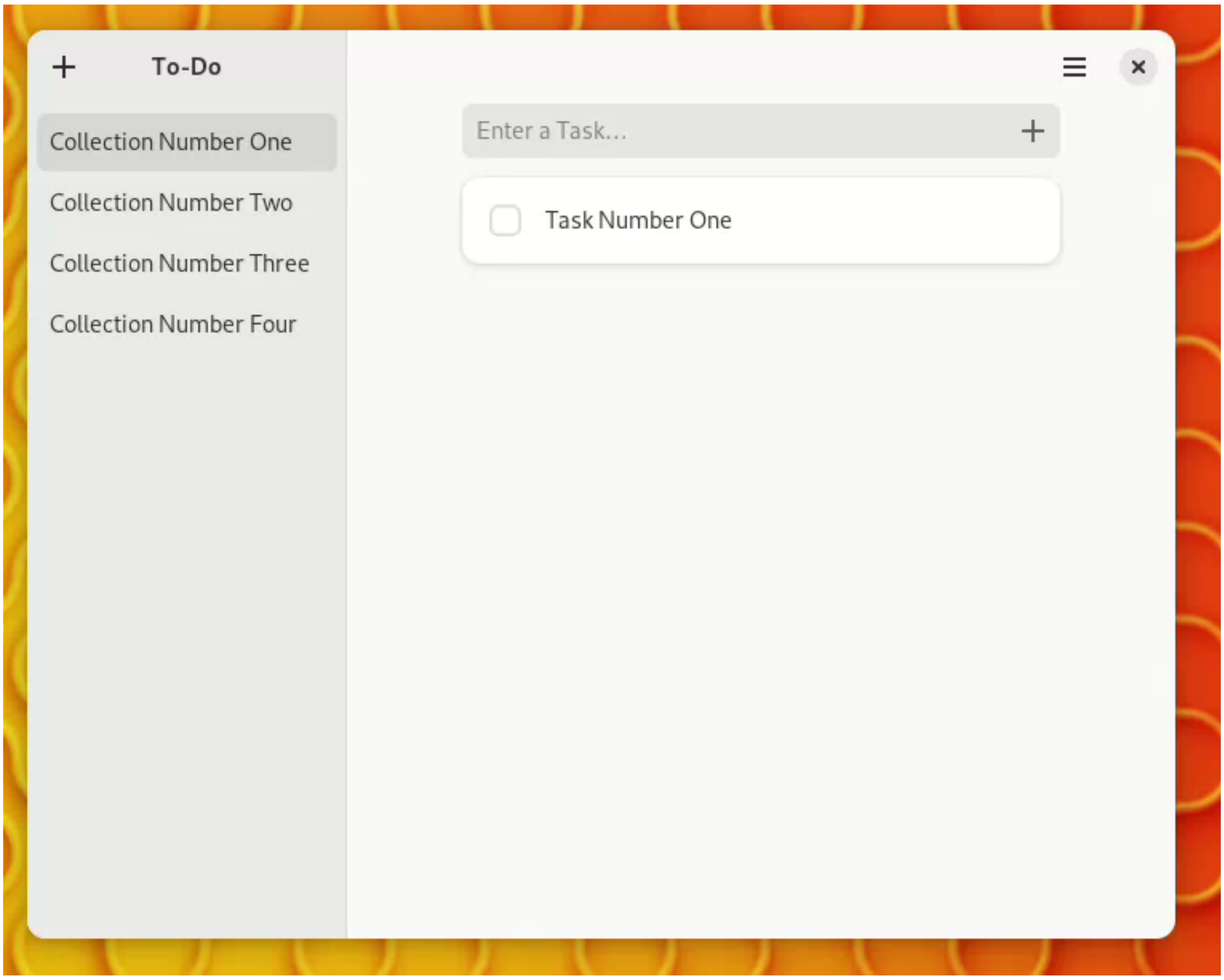
```
        // Filter model whenever the value of the key "filter" changes
        self.settings().connect_changed(
            Some("filter"),
            clone!(@weak self as window => move |_, _| {
                window.set_filter();
            }),
        );

        // Setup callback when items of collections change
        self.set_stack();
        self.collections().connect_items_changed(
            clone!(@weak self as window => move |_, _, _, _| {
                window.set_stack();
            }),
        );

        // Setup callback for activating a row of collections list
        self.imp().collections_list.connect_row_activated(
            clone!(@weak self as window => move |_, row| {
                let index = row.index();
                let selected_collection = window.collections()
                    .item(index as u32)
                    .expect("There needs to be an object at this position.")
                    .downcast::<CollectionObject>()
                    .expect("The object needs to be a `CollectionObject`.");
                window.set_current_collection(selected_collection);
                window.imp().split_view.set_show_content(true);
            }),
        );
```

Before, we called the method `set_stack`. This method ensure when there is at least one collection, the "main" page is shown, and the "placeholder" page otherwise.

Filename: listings/todo/8/window/mod.rs

```
    fn set_stack(&self) {
        if self.collections().n_items() > 0 {
            self.imp().stack.set_visible_child_name("main");
        } else {
            self.imp().stack.set_visible_child_name("placeholder");
        }
    }
```

And that was it! Now we can enjoy the final result.



You might have noticed that there is not yet a way to remove a collection. Try to implement this missing piece of functionality in your local version of the To-Do app. Which edge cases do you have to consider?
