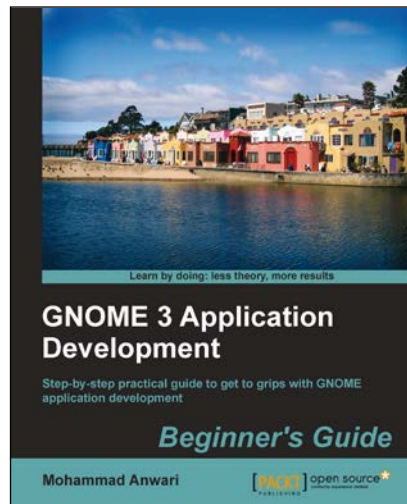


GNOME 3 Application Development Beginners' Guide

Mohammad Anwari



Chapter No. 4 "Using GNOME Core Libraries"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.4 "Using GNOME Core Libraries"

A synopsis of the book's content

Information on where to buy this book

About the Author

Mohammad Anwari is a software hacker from Indonesia with more than 13 years of experience in software development. He has been working with Linux-based systems, applications in GNOME, and Qt platforms. The projects he has worked on range from the development of constrained devices and desktop applications, to high traffic server systems and applications.

He worked for his own startup company during the dotcom era before moving to Finland to work for Nokia/MeeGo. Now he's back in Indonesia, regaining his entrepreneurship by establishing a new startup company that focuses on Node.js and Linux-based projects. In his free time, he serves as an executive director for BlankOn, one of the biggest open source projects in Indonesia.

In the past, he has published a couple of books on Linux in the Indonesian language.

This book would have been impossible to write without the great and continuous support from my family: Rini, Alif, and Abil.

For More Information:

www.packtpub.com/gnome-3-application-development-beginners-guide/book

GNOME 3 Application Development Beginner's Guide

This book is about developing GNOME 3 applications with the Vala and JavaScript programming languages. It guides you to build GTK+, Clutter, and HTML5 applications on the GNOME 3 platform. It covers GNOME 3 specific subsystems such as data access, multimedia, networking, and filesystem. It also covers good software engineering practices such as localization and testing.

What This Book Covers

Chapter 1, Installing GNOME 3 and SDK, discusses installing GNOME 3 and the Software Development Kit in some popular Linux distributions.

Chapter 2, Preparing Our Weapons, talks about the basic usage of the Integrated Development Environment and User Interface Designer used in this book: Anjuta and Glade. This chapter also touches on the development reference tool – Devhelp.

Chapter 3, Programming Languages, covers the basics of Vala and programming JavaScript with Seed. This chapter will be the foundation to understand the following chapters if you are yet not familiar with Vala and JavaScript.

Chapter 4, Using GNOME Core Libraries, guides you to exploit the commonly used features of the GNOME core libraries.

Chapter 5, Building Graphical User Interface Applications explains the steps of building GUI applications with GTK+ and Clutter.

Chapter 6, Creating Widgets, explains how to create GTK+ widgets from scratch. This chapter also talks about extending and customizing widgets.

Chapter 7, Having Fun with Multimedia, contains lots of information on GStreamer. It covers both playing multimedia stream and applying filters to the stream.

Chapter 8, Playing with Data, explains presenting data with the TreeView API family. While showing the data presentation, it also talks about getting data from Evolution Data Server.

For More Information:

www.packtpub.com/gnome-3-application-development-beginners-guide/book

Chapter 9, Deploying HTML5 Applications with GNOME, explains how to embed WebKit into a GTK+ application. It also talks about the JavaScriptCore library, which makes it possible to get the JavaScript running in WebKit to talk with the backend system written in Vala.

Chapter 10, Desktop Integration, discusses about creating applications that integrate nicely with the GNOME 3 desktop. It talks about D-Bus, session management, keyring, launcher, and notification services.

Chapter 11, Making Our Applications Go International, discusses about internationalization and localization in GNOME 3 applications. It also provides a proposal of the localization process as a bonus.

Chapter 12, Quality Made Easy, talks about performing unit testing and stubbing. This covers both testing GTK+ and non-GUI applications.

Chapter 13, Exciting Projects, offers two exciting projects to build a web browser and a Twitter client. It covers many aspects learned in the preceding chapters and uses them in these two projects.

For More Information:

www.packtpub.com/gnome-3-application-development-beginners-guide/book

4

Using GNOME Core Libraries

GNOME core libraries are a collection of foundation utility classes and functions. It covers many things from simple date-conversion functions to virtual filesystem access management. GNOME would not be as powerful as it is now without its core libraries. There are a lot of UI libraries out there that are not successful because of the lack of this kind of power. No wonder there are many libraries outside GNOME that also use GNOME core libraries to support their functionalities.

GNOME core libraries are composed from GLib and GIO, which are non-UI libraries for supporting our UI applications. These libraries connect our programs with files, networks, timers, and other important aspects in the operating system. Without this knowledge, we can probably make a beautiful program, but we would be incapable of interacting with the rest of the system.

In this chapter we shall learn about:

- ◆ The GLib main loop and basic functions
- ◆ The GObject signaling system and properties
- ◆ The GIO files, stream and networking
- ◆ The GSettings configuration system

Ok, let's get started.

For More Information:

www.packtpub.com/gnome-3-application-development-beginners-guide/book

Before we start

There are a few exercises in this chapter that need access to the Internet or the local network. Make sure you have a good connection before running the program. Another exercise requires access to removable hardware and mountable filesystems.

In this chapter, we will do something different regarding the Vala exercises. Because the nature of the discussions are independent of each other, each Vala exercise is done in its own project instead of continuously modifying a file in a single project. So, in each Vala exercise we will create a new project and work inside that project. The name of the project will be noted so you can easily compare your project with the source code that accompanies this book. Similar to the previous chapter, the project we create here is a Vala GTK+ (simple) project. In the project properties, we should not tick on the **GtkBuilder support for user interface** option and should pick **No license** in the **License** option.

In each exercise, the JavaScript code follows the Vala code and it is kept inside one file per exercise. The functionalities of the JavaScript code would be exactly the same. So you can opt to choose whether you want to use either the Vala or the JavaScript code, or both.

The GLib main loop

GLib provides a main event loop, which takes care of the events coming from various sources. With this event loop, we can catch these events and do the necessary processing.

Time for action – playing with the GLib main loop

Here, we will introduce ourselves to the GLib main loop.

1. Create a new Vala project called `core-mainloop` and use this code in the Main class:

```
using GLib;

public class Main : Object
{
    int counter = 0;

    bool printCounter() {
        stdout.printf("%d\n", counter++);
        return true;
    }

    public Main ()
    {
```

```

        Timeout.add(1000, printCounter);
    }

    static int main (string[] args)
    {
        Main main = new Main();
        var loop = new MainLoop();
        loop.run ();
        return 0;
    }
}

```

2. And this is the JavaScript code's counterpart; you can name the script as `core-mainloop.js`:

```

#!/usr/bin/env seed

GLib = imports.gi.GLib;
GObject = imports.gi.GObject;

Main = new GType({
    parent: GObject.Object.type,
    name: "Main",
    init: function() {
        var counter = 0;
        this.printCounter = function() {
            Seed.printf("%d", counter++);
            return true;
        };
        GLib.timeout_add(0, 1000, this.printCounter);
    }
});

var main = new Main();
var context = GLib.main_context_default();
var loop = new GLib.MainLoop.c_new(context);
loop.run();

```

3. Run it. Do you notice that the program prints the counter and stays running? You can do nothing except press the `Ctrl + C` key combination to kill it.

What just happened?

We have set up a GLib main loop with a single source of events, a timeout.

Initially, we set the `counter` variable to 0.

```
int counter = 0;
```

We prepare a function called `printCounter` to print the `counter` variable's value, and increase its value by one immediately after printing. Then we return `true` to indicate that we want the counter to continue.

```
bool printCounter() {
    stdout.printf("%d\n", counter++);
    return true;
}
```

In the constructor, we create a `Timeout` object with a 1000 ms interval pointing to our `printCounter` function. This means that `printCounter` will be called at every 1-second interval, and it will be repeatedly called as long as `printCounter` returns `true`.

```
public Main ()
{
    Timeout.add(1000, printCounter);
}
```

In the main function, we instantiate the `Main` class, create a `MainLoop` object, and call `run`. This will cause the program to stay running until we manually terminate it. When the loop is running, it can accept events submitted to it. The `Timeout` object that we created earlier produces such an event. Whenever the timer interval expires, it notifies the main loop, which in turn calls the `printCounter` function.

```
static int main (string[] args)
{
    Main main = new Main();
    var loop = new MainLoop();
    loop.run ();
    return 0;
}
```

Now, let's take a look at the JavaScript code. If you notice, the class structure is a bit different from what we learned in the previous chapter. Here we use Seed Runtime's construction of class.

```
GLib = imports.gi.GLib;
GObject = imports.gi.GObject;
```

Here, we import `GLib` and `GObject`. Then we construct a class called `Main`, which is based on `GObject`.

Here is how we do it. The following code says that we subclass `GType` into a new class called `Main` and pass the object structure into the argument.

```
Main = new GType({
    parent: GObject.Object.type,
    name: "Main",
```


The first member of the object is `parent`, which is the parent of our class. We assign it with `GObject.Object.type` to denote that our class is derived from `Object` in the `GObject` module that we imported previously. Then we name our class as `Main`. After that, we put the functions inside the `init` function, which is also the constructor of the class.

The content of the class member is similar to what we've seen in the Vala code and it is quite straightforward.

```
init: function() {
    var counter = 0;
    this.printCounter = function() {
        Seed.printf("%d", counter++);
        return true;
    };
    GLib.timeout_add(0, 1000, this.printCounter);
}
});
```

Then we have the code that is analogous to what we have in Vala's static `main` function. Here we create our `Main` object and create the GLib's main loop.

```
var main = new Main();
var context = GLib.main_context_default();
var loop = new GLib.MainLoop.c_new(context);
loop.run();
```

Have a go hero – stopping the timeout

Our program counts forever. Can you make it stop after the counter reaches 10?



You can just play with the `printCounter` return value.

Or even better, can you make it stop totally, meaning that the program would exit after the counter reaches 10?



You can ignore the return value and rearrange the code, and somehow pass the `loop` object into the `Main` class. In the `printCounter` function, you can call `loop.quit()` whenever it reaches 10 to make the program break the main loop programmatically.

GObject signals

GObject provides a signaling mechanism that we can hook into. In the previous chapter, we have discussed the Vala signaling system. Internally, it is actually using the GObject signaling system, but it is so transparent that it is seamlessly integrated into the language itself.

Time for action – handling GObject signals

Let us see how to do it in JavaScript:

1. Create a new script called `core-signals.js` and fill it with the following code:

```
#!/usr/bin/env seed

GLib = imports.gi.GLib;
GObject = imports.gi.GObject;

Main = new GType({
  parent: GObject.Object.type,
  name: "Main",
  signals: [
    {
      name: "alert",
      parameters: [GObject.TYPE_INT]
    }
  ],
  init: function(self) {
    var counter = 0;

    this.printCounter = function() {
      Seed.printf("%d", counter++);
      if (counter > 9) {
        self.signal.alert.emit(counter);
      }
    }
    return true;
  };

  GLib.timeout_add(0, 1000, this.printCounter);
});

var main = new Main();

var context = GLib.main_context_default();
var loop = new GLib.MainLoop.c_new(context);
```

```
main.signal.connect('alert', function(object, counter) {
    Seed.printf("Counter is %d, let's stop here", counter);
    loop.quit();
});
loop.run();
```

2. Run it and notice the messages printed:

```
0
1
2
3
4
5
6
7
8
9
Counter is 10, let's stop here
```

What just happened?

With the GObject signaling system, we can subscribe for notifications that are emitted by an object. We just need to provide a handler that will perform some action upon receiving the signal.

Here, we declare our signal in an array by putting an object with names and parameters as the content of the object. The parameter type is the type that is known by the GLib system. If our signal does not have any parameters, we can omit it.

```
signals: [
    {
        name: "alert",
        parameters: [GObject.TYPE_INT]
    }
],

main.signal.connect('alert', function(object, counter) {
    Seed.printf("Counter is %d, let's stop here", counter);
    loop.quit();
});
```

Then we subscribe to the signal and provide a closure that just prints the `counter` value and breaks the main loop. Note that the parameter is defined in the second parameter of the closure. The first parameter is reserved for the object itself.

Finally, we emit the signal by calling the signal by its name. `self` is the `Main` class we pass in the `init` function.

```
if (counter > 9) {  
    self.signal.alert.emit(counter);  
}
```

As soon as we call this, the signal will be processed in the main loop and will be delivered to the objects that subscribe to it.

Have a go hero – writing it in Vala

Compared with the previous code, signal declaration, emission, and subscription are easier in Vala, as we've seen it the last time. How about trying to write the previous code in Vala?

GLib properties

Properties are key-value pairs in a storage system that are available in all instances of `GObject`, which is the base class for all objects in the GNOME system. One useful feature of properties is that we can subscribe for changes when the value is changed.

Time for action – accessing properties

We are going to learn how to set and get a value to and from a property as well as monitor the changes.

1. Create a new script called `core-properties.js` and fill it with this code:

```
#!/usr/bin/env seed  
  
GLib = imports.gi.GLib;  
GObject = imports.gi.GObject;  
  
Main = new GType({  
    parent: GObject.Object.type,  
    name: "Main",  
    properties: [  
        {  
            name: 'counter',  
            type: GObject.TYPE_INT,  

```

```

        default_value: 0,
        minimum_value: 0,
        maximum_value: 1024,
        flags: (GObject.ParamFlags.CONSTRUCT
            | GObject.ParamFlags.READABLE
            | GObject.ParamFlags.WRITABLE),
    }
},
init: function(self) {
    this.print_counter = function() {
        Seed.printf("%d", self.counter++);
        return true;
    }

    this.monitor_counter = function(obj, gobject, data) {
        Seed.print("Counter value has changed to " + obj.counter);
    }

    GLib.timeout_add(0, 1000, this.print_counter);
    self.signal.connect("notify::counter", this.monitor_counter);
}
});

var main = new Main();
var context = GLib.main_context_default();
var loop = new GLib.MainLoop.c_new(context);
loop.run();

```

- 2.** And this is the Vala counterpart (you can create a new project called `core-properties` and fill `core_properties.vala` with this code):

```

using GLib;

public class Main : Object
{
    public int counter {
        set construct;
        get;
        default = 0;
    }

    public bool print_counter() {
        stdout.printf("%d\n", counter ++);
        return true;
    }
}

```

```
public void monitor_counter() {
    stdout.printf ("Counter value has changed to %d\n", counter);
}

public Main ()
{
}

construct {
    Timeout.add(1000, print_counter);
    notify["counter"].connect ((obj)=> {
        monitor_counter ();
    });
}

static int main (string[] args)
{
    Gtk.init (ref args);
    var app = new Main ();

    Gtk.main ();
    return 0;
}
}
```

- 3.** Run it and notice the messages printed. Note that you can press the *Ctrl + C* combination keys to stop the program.

```
Counter value has changed to 0
Counter value has changed to 1
0
Counter value has changed to 2
1
Counter value has changed to 3
2
Counter value has changed to 4
3
Counter value has changed to 5
4
Counter value has changed to 6
5..
```

What just happened?

In the JavaScript code, we need to declare the properties inside the `properties` array, and fill it with the property's object.

Here we describe that our property has the name `counter` and is of type integer. It needs to declare the default, minimum, and maximum values. It also needs the flags. From the flags, we can see `GObject.ParamFlags.CONSTRUCT`, which means that the property is initialized in the construction phase. It means that the default value is set when the object is created. We also see that it is readable and writable.

```
properties: [
  {
    name: 'counter',
    type: GObject.TYPE_INT,
    default_value: 0,
    minimum_value: 0,
    maximum_value: 1024,
    flags: (GObject.ParamFlags.CONSTRUCT
           | GObject.ParamFlags.READABLE
           | GObject.ParamFlags.WRITABLE),
  }
]
```

In the following code, we subscribe for changes. We use the signaling system and the name of the signal is constructed with the `notify::` keyword followed by the property's name. After this, every change that happens to the property will trigger the signal handler.

```
self.signal.connect("notify::counter", this.monitor_counter);
```

Here we set the value of the property by increasing its value. Note that here we modify the value; hence the value monitor will be triggered first, and then the actual value is printed by `printf`.

```
this.print_counter = function() {
  Seed.printf("%d", self.counter++);
  return true;
}
```

And the following code shows how to read the value:

```
this.monitor_counter = function(obj, gobject, data) {
  Seed.print("Counter value has changed to " + obj.counter);
}
```

In contrast with the JavaScript code, the properties declaration in Vala is very simple. The declaration is similar to the normal variable declaration with some additions.

In the following code, the `set construct` expression means that it is writable and the default value is initialized in the construction phase. `get` means that it is readable, and `default` defines the default value.

```
public int counter {
    set construct;
    get;
    default = 0;
}
```

However, there is no mechanism to set the minimum and maximum value.

Then we see how reading and writing the property are done like reading and writing a normal variable. From outside the class, we can use the normal way to refer a member variable, which is by using an object name followed by a dot and the property name.

```
public bool print_counter() {
    stdout.printf("%d\n", counter ++);
    return true;
}

public void monitor_counter() {
    stdout.printf ("Counter value has changed to %d\n", counter);
}
```

Subscribing for changes also uses the usual signaling mechanism, with the exception that we insert the property name in square brackets following the signal name, `notify`.

```
notify["counter"].connect ((obj)=> {
    monitor_counter ();
})
```

There is something new in the code; something we have not seen before. It is the `construct` keyword. It is basically an alternative way to construct an object similar to the normal constructors. This style of construction is close to how GObject construction is being carried out in the actual generated C code.

Despite the differences between these JavaScript and Vala codes, both allow the use of a property just like a plain member of the class. So, in both languages, you can access the `counter` property as `main.counter` (assuming that the object's name is `main`).

Pop quiz – why the value of zero is printed out

From the output, we saw this:

```
Counter value has changed to 0
```

Q1. We did not set the counter to 0 explicitly, did we? So, why did it happen?

1. Because the property has the `set` construct keyword defined.
2. Because 0 is the default value.

Have a go hero – making a property read-only

When a property is read-only, we can no longer set its value. Now, let's try to make the `counter` property read-only. Hint: Play with the property flag.

Configuration files

In many cases we need to somehow read from a configuration file in order to customize how our program should behave. Here, we will learn how to use the simplest configuration mechanism in GLib using a configuration file. Imagine that we have a configuration file and it contains the name and version of our application so that we can print it somewhere inside our program.

Time for action – reading configuration files

Here's how to do it:

1. Create a configuration file; let's call it `core-keyfile.ini`. Its content is as follows:

```
[General]
name = "This is name"
version = 1
```

2. Create a new Vala project and name it `core-keyfile`. Put the `core-keyfile.ini` file inside the project directory (but not in `src`).
3. Edit `core_keyfile.vala` to look like this:

```
using GLib;

public class Main : Object
{
    KeyFile keyFile = null;
    public Main ()
```

```
{
    keyFile = new KeyFile();
    keyFile.load_from_file("core-keyfile.ini", 0);
}

public int get_version()
{
    return keyFile.get_integer("General", "version");
}

public string get_name()
{
    return keyFile.get_string("General", "name");
}

static int main (string[] args)
{
    var app = new Main ();
    stdout.printf("%s %d\n", app.get_name(), app.get_version());

    return 0;
}
```

- 4.** The JavaScript code (let's call it `core-keyfile.js`) looks like this (remember to put the `.ini` file in the same directory as the script):

```
#!/usr/bin/env seed

GLib = imports.gi.GLib;
GObject = imports.gi.GObject;

Main = new GType({
    parent: GObject.Object.type,
    name: "Main",
    init: function(self) {

        this.get_name = function() {
            return this.keyFile.get_string("General", "name");
        }

        this.get_version = function() {
            return this.keyFile.get_integer("General", "version");
        }
    }
});
```

```

        this.keyFile = new GLib.KeyFile.c_new();
        this.keyFile.load_from_file("core-keyfile.ini");
    }
});

var main = new Main();
Seed.printf("%s %d", main.get_name(), main.get_version());

```

5. Run the program and look at the output:

```
"This is name" 1
```

What just happened?

The configuration file we are using has a key-value pairs structure conforming to the Desktop Entry Specification document of freedesktop.org. In the GNOME platform, this structure is commonly used, mainly in the `.desktop` files, which are used by the launcher. People using Windows might find this similar to the `.ini` format, which is also used for configuration.

GLib provides the `KeyFile` class to access this type of configuration file. In our constructor, we have this snippet:

```

keyFile = new KeyFile();
keyFile.load_from_file("core-keyfile.ini", 0);

```

It initializes an object of `KeyFile`, and loads the `core-keyfile.ini` file into the object.

If we jump a bit in our `core-keyfile.ini` file, we have a section, as shown, written inside a pair of square brackets.

```
[General]
```

And then all the entries following it can be accessed by specifying the section name. Here we provide two methods, `get_version()` and `get_name()`, as shortcuts to get the value of the `name` and `version` entries in the configuration file.

```

public int get_version()
{
    return keyFile.get_integer("General", "version");
}

public string get_name()
{
    return keyFile.get_string("General", "name");
}

```

Inside the methods, we just get the integer value from the `version` entry and get the string value from the `name` entry. We also see that we obtain the entries under the `General` section. And in these methods we just return the value immediately.

As shown in the following code, we consume the values from the methods and print them:

```
stdout.printf("%s %d\n", app.get_name(), app.get_version());
```

Quite easy, isn't it? The JavaScript code is also easy and straightforward; so it does not need to be explained further.

Have a go hero – multi-section configuration

Let's try adding more sections inside the configuration file and accessing the values. Imagine that we have a specific section called `License` that has `license_file` and `customer_id` as the entries. Imagine that we will use this information later to check whether the customer has the right to use the software.

GIO, the input/output library

In real life, our program must be able to access files wherever they are stored, locally or remotely. Imagine that we have a set of files that we need to read. The files are spread both locally and remotely. GIO will make it easy for us to manipulate these files as it provides an API to interact with our files in an abstract way.

Time for action – accessing files

Let's see how it works:

1. Let's create a new script called `core-files.js`, and fill it with these lines:

```
#!/usr/bin/env seed

GLib = imports.gi.GLib;
Gio = imports.gi.Gio;
GObject = imports.gi.GObject;

Main = new GType({
  parent: GObject.Object.type,
  name: "Main",
  init: function(self) {
    this.start = function() {
      var file = null;
      var files = ["http://en.wikipedia.org/wiki/Text_file",
        "core-files.js"];
```

```

        for (var i = 0; i < files.length; i++) {
            if (files[i].match(/^http:/)) {
                file = Gio.file_new_for_uri(files[i]);
            } else {
                file = Gio.file_new_for_path(files[i]);
            }

            var stream = file.read();
            var data_stream = new Gio.DataInputStream.c_new(stream);
            var data = data_stream.read_until("", 0);

            Seed.print(data)
        }
    }
}
});

var main = new Main();
main.start();

```

- 2.** Alternatively, you can create a Vala project called `core-files`. Fill `src/core_files.vala` with this code:

```

using GLib;

public class Main : Object
{
    public Main ()
    {
    }

    public void start ()
    {
        File file = null;
        string[] files = {"http://en.wikipedia.org/wiki/Text_file",
"src/core_files.vala"};

        for (var i = 0; i < files.length; i++) {
            if (files[i].has_prefix("http:")) {
                file = File.new_for_uri(files[i]);
            } else {
                file = File.new_for_path(files[i]);
            }

            var stream = file.read();
            var data_stream = new DataInputStream(stream);

```

```
        size_t data_read;
        var data = data_stream.read_until("", out data_read);
        stdout.printf(data);
    }
}

static int main (string[] args)
{
    var app = new Main ();
    app.start();
    return 0;
}
```

3. Run the program, and notice that it fetches the Wikipedia page from the Internet as well as the source code of the program from the local directory.



```
Terminal
File Edit View Search Terminal Help
l, true);
}</script>
<script src="/w/index.php?title=Special:BannerController&cache=/cn.js&30
3-4" type="text/javascript"></script>
<script src="//bits.wikimedia.org/en.wikipedia.org/load.php?debug=false&lang
=en&modules=site&only=scripts&skin=vector&*" type="text/javascr
ipt"></script>
<script src="//geoipllookup.wikimedia.org/" type="text/javascript"></script><!--
Served by srv261 in 0.098 secs. -->
</body>
</html>
using GLib;

public class Main : Object
{
    public Main ()
    {
    }

    public void start ()
    {
        File file = null;
        string[] files = {"http://en.wikipedia.org/wiki/Text_file", "src
/core_files.vala"};
```

What just happened?

GIO aims to provide a set of powerful virtual filesystem APIs. It provides a set of interfaces that serve as a foundation to be extended by the specific implementation. For example, here we use the GFile interface that defines the functions for a file. The GFile API does not tell us where the file is located, how the file is read, or other such details. It just provides the functions and that's it. The specific implementation that is transparent to the application developers will do all the hard work. Let's see what this means.

In the following code, we get the file location from the array `files`. Then we check if the location has an HTTP protocol identifier or not; if yes, we create the `GFile` object using `file_new_for_uri`, otherwise we use `file_new_for_path`. We can, of course, use `file_new_for_uri` even for the local file, but we need to prepend the `file://` protocol identifier to the filename.

```
if (files[i].match(/^http:/)) {
    file = Gio.file_new_for_uri(files[i]);
} else {
    file = Gio.file_new_for_path(files[i]);
}
```

This is the only difference between handling the remote file and the local file. And after that we can access files either from the local drive or from a web server by using the same function with GIO.

```
var stream = file.read();
var data_stream = new Gio.DataInputStream.c_new(stream);
var data = data_stream.read_until("", 0);
```

Here we use the `read` function to get the `GFileInputStream` object. Notice here that the API provides the same function wherever the file is.

The resulting object is a stream. A **stream** is a sequence of data that flows from one end to the other. The stream can be passed to an object and can transform it to become another stream or just consume it.

In our case, we get the stream initially from the `file.read` function. We transfer this stream into `GDataInputStream` in order to easily read the data. With the new stream, we ask GIO to read the data until we find nothing, which means it has reached the end of the file. And then we spit the data out onto the screen.

Network access with GIO

GIO provides adequate functions to access the network. Here we will learn how to create socket client and server programs. Imagine that we are building a simple chat program that can send data from one end to another.

Time for action – accessing a network

For brevity, we will do it only in JavaScript now; you can look at the Vala program in `core-server` and `core-client` projects code that accompany this book. Ok, so let's see what are the steps needed to access the network.

1. Create a new script called `core-server.js` and fill it with these lines:

```
#!/usr/bin/env seed

GLib = imports.gi.GLib;
Gio = imports.gi.Gio;
GObject = imports.gi.GObject;

Main = new GType({
  parent: GObject.Object.type,
  name: "Main",
  init: function(self) {
    this.process = function(connection) {
      var input = new Gio.DataInputStream.c_new (connection.get_
input_stream());
      var data = input.read_upto("\n", 1);
      Seed.print("data from client: " + data);
      var output = new Gio.DataOutputStream.c_new (connection.get_
output_stream());
      output.put_string(data.toUpperCase());
      output.put_string("\n");
      connection.get_output_stream().flush();
    }

    this.start = function() {
      var service = new Gio.SocketService();
      service.add_inet_port(9000, null);
      service.start();
      while (1) {
        var connection = service.accept(null);
        this.process(connection);
      }
    }
  }
});

var main = new Main();
main.start();
```


2. Run this script. The program will stay running until we press *Ctrl* + *C*.
3. Then create another script called `core-client.js`; here is the code:

```
#!/usr/bin/env seed

GLib = imports.gi.GLib;
Gio = imports.gi.Gio;
GObject = imports.gi.GObject;

Main = new GType({
  parent: GObject.Object.type,
  name: "Main",
  init: function(self) {

    this.start = function() {
      var address = new Gio.InetAddress.from_string("127.0.0.1");
      var socket = new Gio.InetSocketAddress({address: address,
port: 9000});
      var client = new Gio.SocketClient ();
      var conn = client.connect (socket);

      Seed.printf("Connected to server");

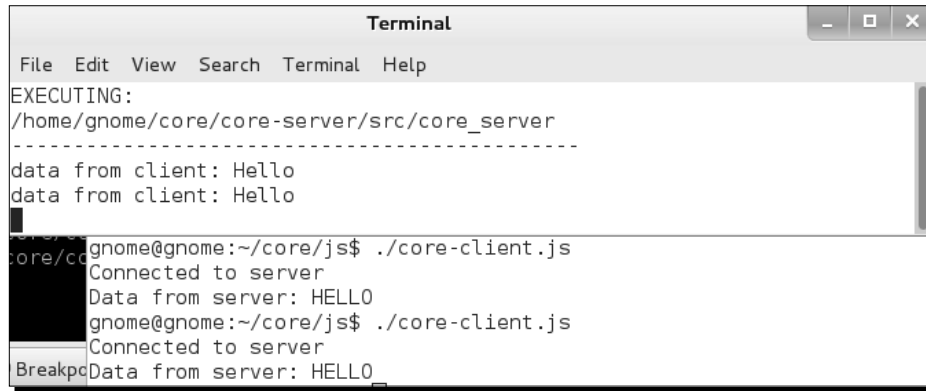
      var output = conn.get_output_stream();
      var output_stream = new Gio.DataOutputStream.c_new(output);

      var message = "Hello\n";
      output_stream.put_string(message);
      output.flush();

      var input = conn.get_input_stream();
      var input_stream = new Gio.DataInputStream.c_new(input);
      var data = input_stream.read_upto("\n", 1);
      Seed.printf("Data from server: " + data);
    }
  }
});

var main = new Main();
main.start();
```

4. Run this program and notice the output of both the server and the client programs. They can talk to each other!



A terminal window titled "Terminal" with a menu bar (File, Edit, View, Search, Terminal, Help). The output shows the server program being executed at `/home/gnome/core/core-server/src/core_server`. It receives two "data from client: Hello" messages. Then, the client program `./core-client.js` is run, showing "Connected to server" and "Data from server: HELLO". This sequence is repeated once more, with a "Breakpoint" message appearing at the end of the second client run.

```
Terminal
File Edit View Search Terminal Help
EXECUTING:
/home/gnome/core/core-server/src/core_server
-----
data from client: Hello
data from client: Hello
gnome@gnome:~/core/js$ ./core-client.js
Connected to server
Data from server: HELLO
gnome@gnome:~/core/js$ ./core-client.js
Connected to server
BreakpointData from server: HELLO
```

What just happened?

GIO provides high-level as well as low-level networking APIs that are really easy to use. Let's take a look at the server first.

Here we open a service in port number 9000. It is an arbitrary number; you can use your own number if you want, with some restrictions:

```
var service = new Gio.SocketService();
service.add_inet_port(9000, null);
service.start();
```

You can't run the service if there is already another service running with a port number that is the same as yours. Also, you have to run your program as root if you want to use a port number below 1024.

And then we enter an infinite loop that is called when the service is accepting an incoming connection. Here, we just call our process function to handle the connection. That's it.

```
while (1) {
    var connection = service.accept(null);
    this.process(connection);
}
```

The server's basic activity is defined as easily as that. The details of the processing is another story.

Then, we create a `GDataInputStream` object based on the input stream coming from the connection. And then we read the data in until we find the end of line character which is `\n`. It is one character, so we put 1 there as well. And then we print the incoming data.

```
var input = new Gio.DataInputStream.c_new (connection.get_
input_stream());
var data = input.read_upto("\n", 1);
Seed.print("data from client: " + data);
```

To make things interesting, we want to return something to the client. Here we create an object of the `GDataOutputStream` class that is coming from the connection object. We change the data coming from the client to uppercase, and we send it back through the stream. In the end, we make sure everything is sent by flushing down the pipe. That's all on the server side.

```
var output = new Gio.DataOutputStream.c_new (connection.get_output_
stream());
output.put_string(data.toUpperCase());
output.put_string("\n");
connection.get_output_stream().flush();
```

On the client side, initially, we make an object of `GInetAddress`. The object is then fed into `GInetSocketAddress` so we can define the port of the address that we want to connect to.

```
var address = new Gio.InetAddress.from_string("127.0.0.1");
var socket = new Gio.InetSocketAddress({address: address, port:
9000});
```

Then we connect the `socket` object with `SocketClient` into `GSocketClient`. After this, if everything is OK, the connection to the server is established.

```
var client = new Gio.SocketClient ();
var conn = client.connect (socket);
```

On the client side, in principle, the process occurs in the opposite way as it would occur on the server side. Here we create `GDataOutputStream` first, based on the stream coming from the connection object. Then we just send the message into it. We also want to flush it so all the remaining data in the pipeline is flushed out.

```
var output = conn.get_output_stream();
var output_stream = new Gio.DataOutputStream.c_new(output);

var message = "Hello\n";
output_stream.put_string(message);
output_stream.flush();
```

Then, we expect to get something from the server; so we create an input stream object. We read from it until we find a newline, and we print the data.

```
var input = conn.get_input_stream();
var input_stream = new Gio.DataInputStream.c_new(input);
var data = input_stream.read_upto("\n", 1);
Seed.printf("Data from server: " + data);
```

Have a go hero – making an echo server

Echo server is a service that returns everything that is sent to it as it is, without any modifications. For example, if we send "Hello", the server will also send back "Hello". Sometimes it is used for checking whether the connection between two hosts is working. How about modifying the server program to be an echo server?

We can put it in an infinite loop, but if we type "quit", the server disconnects.

Understanding GSettings

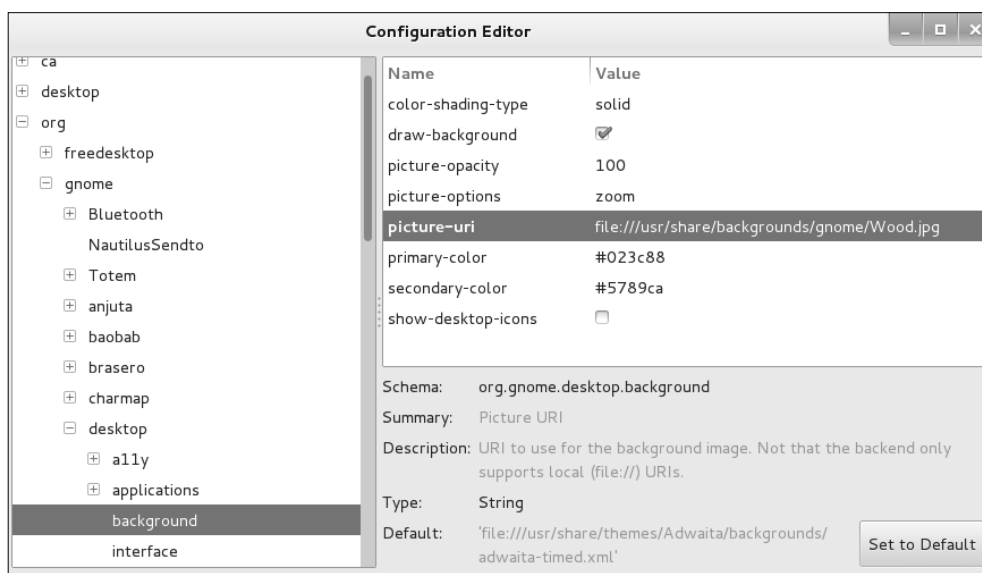
Previously, we have used the GLib configuration parser to read our application configuration. Now we will try to use a more advanced settings system with GSettings. With this, we can access configurations throughout the GNOME platform, including all the applications that use the system.

Time for action – learning GSettings

Let's see what the GSettings configuration system looks like as visualized by the `dconf-editor` tool:

1. Launch a terminal.
2. Run `dconf-editor` from the terminal.

3. Navigate through the `org` tree on the left-hand side of the application, and go through **gnome**, **desktop**, and then **background**.



What just happened?

GSettings is a new introduction in GNOME 3. Before, the configuration was handled with GConf. In GNOME 3, every shipped GNOME application has been migrated to use GSettings. The concept of storing the settings in GConf and GSettings remains the same, that is, by using key-value pairs. However, GSettings contains improvements in many aspects, including more restrictive usage by enforcing schema as metadata. With GConf, we can freely store and read any values from the system.

GSettings is actually only a top-level layer. Underneath, there is a low-level system called dconf, which handles the actual storing and reading of the values. The tool we discuss here shows the keys and values in a hierarchy so we can browse, read, and even write a new value (if the schema says it's writable, of course).

In the screenshot we can see that `org.gnome.desktop.background` has many entries; one of them is `picture-uri`, which contains the URI of the desktop's background image.

GSettings API

In this book, the API is more interesting than the administrative tools. After we see GSettings visually, it is time to access GSettings through API.

Time for action – accessing GSettings programmatically

Imagine that we create a tool to set the background image of our GNOME desktop. Here is how to do it:

1. Create a new Vala project called `core-settings`, and modify `core_settings.vala` with the following:

```
using GLib;

public class Main : Object
{
    Settings settings = null;
    public Main ()
    {
        settings = new Settings("org.gnome.desktop.background");
    }

    public string get_bg()
    {
        if (settings == null) {
            return null;
        }

        return settings.get_string("picture-uri");
    }

    public void set_bg(string new_file)
    {
        if (settings == null) {
            return;
        }
        if (settings.set_string ("picture-uri", new_file)) {
            Settings.sync ();
        }
    }

    static int main (string[] args)
    {
        var app = new Main ();
        stdout.printf("%s\n", app.get_bg());
        app.set_bg ("file:///usr/share/backgrounds/gnome/Wood.jpg");
        return 0;
    }
}
```

2. The JavaScript code is quite straightforward; here we have a snippet of it just to see the adaptation needed from the Vala code:

```
init: function(self) {
    this.settings = null;

    this.get_bg = function() {
        if (this.settings == null)
            return null;

        return this.settings.get_string("picture-uri");
    }

    this.set_bg = function(new_file) {
        if (this.settings == null)
            return;

        if (this.settings.set_string("picture-uri", new_file)) {
            Gio.Settings.sync();
        }
    }

    this.settings = new Gio.Settings({schema: 'org.gnome.desktop.
background'});
}
```

3. Run it and see the change in your current desktop background image. Your current desktop background will change to the file specified in the code.

What just happened?

In this exercise, we use the already installed schema owned by the desktop, which is `org.gnome.desktop.background`, so we can just use the API to access the settings. Let's take a look at the details.

First, we initiate the connection to GSettings by specifying the schema name, which is `org.gnome.desktop.background`, and it returns a GSettings object.

```
settings = new Settings("org.gnome.desktop.background");
```

Then we put a simple safety net just in case the initialization fails. In the real world, we can perform reinitialization rather than just a simple return.

```
if (settings == null) {
    return null;
}
```

After that, we obtain a value of type `string` under the key `picture-uri`, and we can consume it in any way we want.

```
return settings.get_string("picture-uri");
```

Finally, we set the value using the same key. If it is successful, we ask `GSettings` to save it to the disk by calling the `sync` function. Easy, right?

```
if (settings.set_string ("picture-uri", new_file)) {  
    Settings.sync ();  
}
```

Summary

In this chapter, we learned a lot about the GNOME core libraries. Even though we did not touch everything in the libraries, we managed to tackle all the basics and essentials needed to build our GNOME application.

We know now that `GLib` provides a main loop that handles all the events from various sources. We discussed the `GObject` property and the signaling system. We also tried to look into the events processed by the main loop by posting with the timeouts, and signals when a value of a property has changed. Regarding the programming languages, we found out that `Vala` is more integrated with GNOME, and `JavaScript` requires more code to use `GObject` properties or signals.

We had an exercise of accessing files both locally and remotely, and we found out that the API provided by `GIO` is very easy to use because it abstracts the way we access those files wherever they are.

With `GIO`, we also did an experiment of building a simple client and server chat program and we found out that to create such an interesting program requires quite a minimal amount of code, both in `JavaScript` and `Vala`.

Finally, we had a discussion about `GSettings` and tried to read and write the GNOME desktop's background image with it.

After we master the foundation of the GNOME application, the next step is to learn the basics of a graphical program in the next chapter.

Where to buy this book

You can buy GNOME 3 Application Development Beginner's Guide from the Packt Publishing website: <http://www.packtpub.com/gnome-3-application-development-beginners-guide/book>.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our [shipping policy](#).

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.



www.PacktPub.com

For More Information:

www.packtpub.com/gnome-3-application-development-beginners-guide/book