GTK Development Blog

All things GTK

A primer on GtkListView

Some of the early adopters of GTK4 have pointed out that the new list widgets are not the easiest to learn. In particular, GtkExpression and GtkBuilderListItemFactory are hard to wrap your head around. That is not too surprising – a full list widget, with columns, and selections and sorting, and tree structure, etc is a complicated beast.

But lets see if we can unpack things one-by-one, and make it all more understandable.

Overview

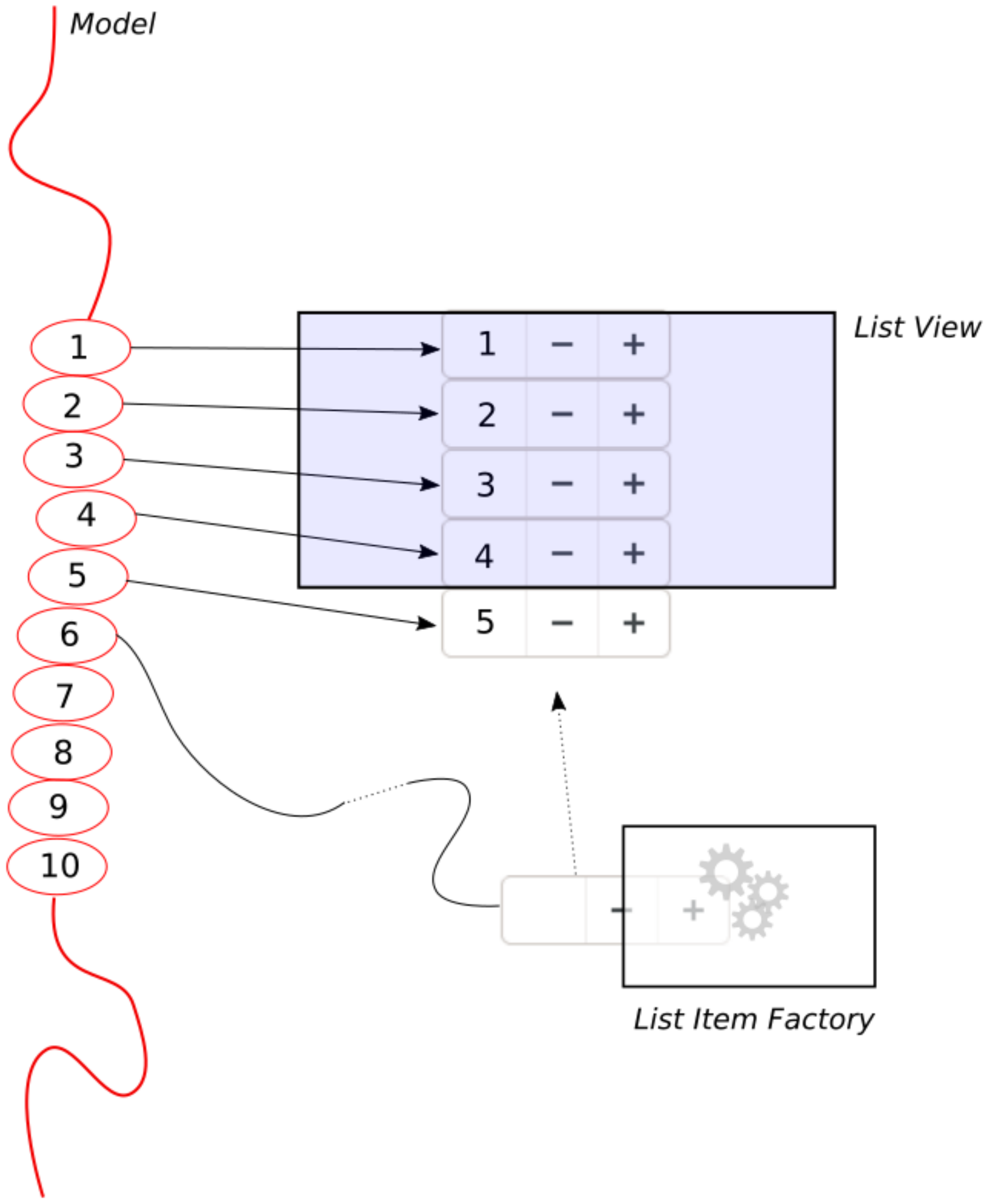
Lets start with a high-level view of the relevant components and their interactions: the model, the list item factory and the list view.

They are the three things that occur when we create a list view:

```
view = gtk_list_view_new (model, factory);
```

The models we use are GListModels. These always contain GObjects, so you will have to provide your data in the form of objects. This is a first notable difference from GtkTreeview, which is using GtkTreeModels directly containing basic types.

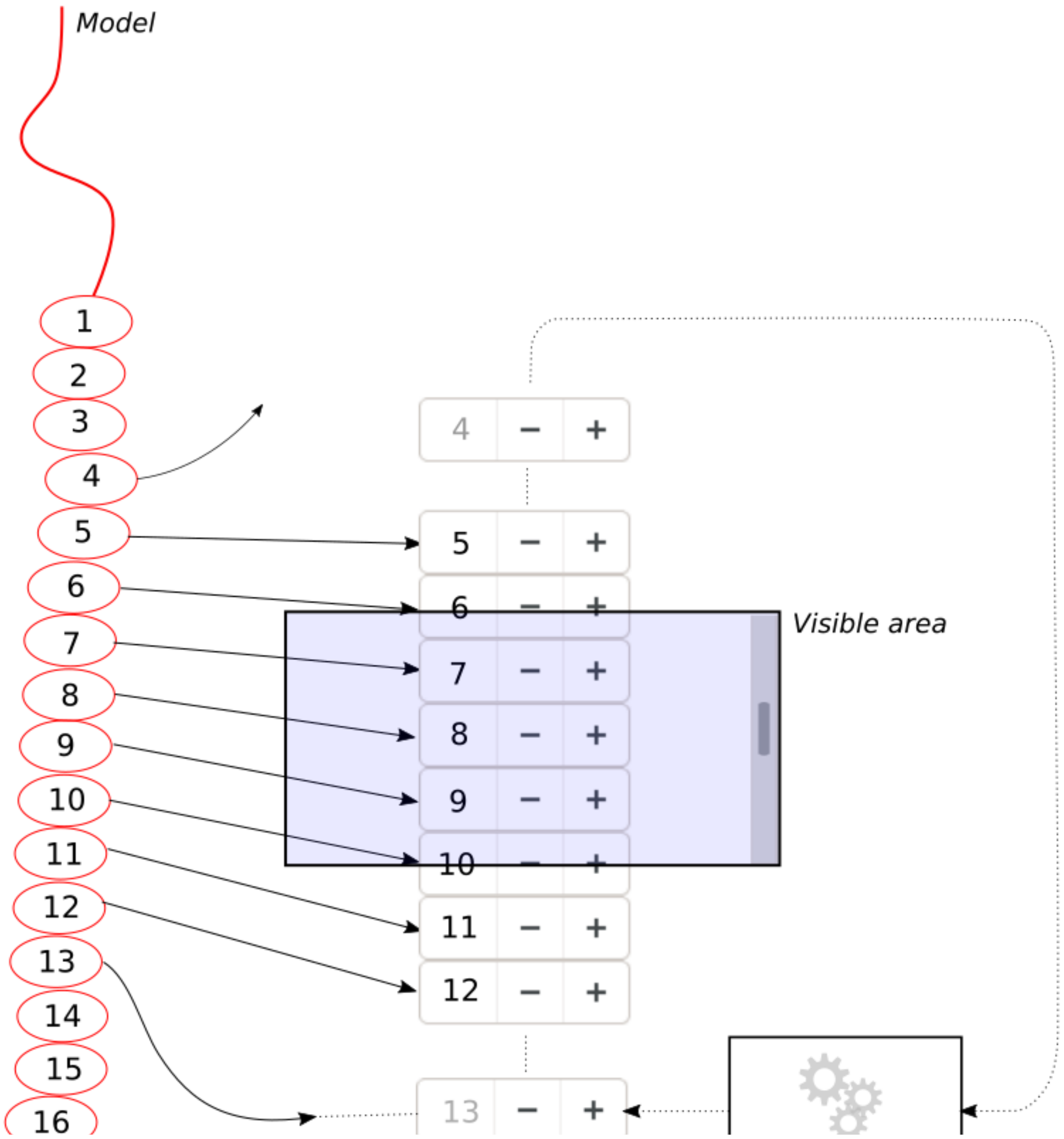
For some simple cases, GTK provides ready-made models, such as GtkStringList. But in general, you will have to make your own model. Thankfully, GListModel is a much simpler interface than GtkTreeModel, so this is not too hard.

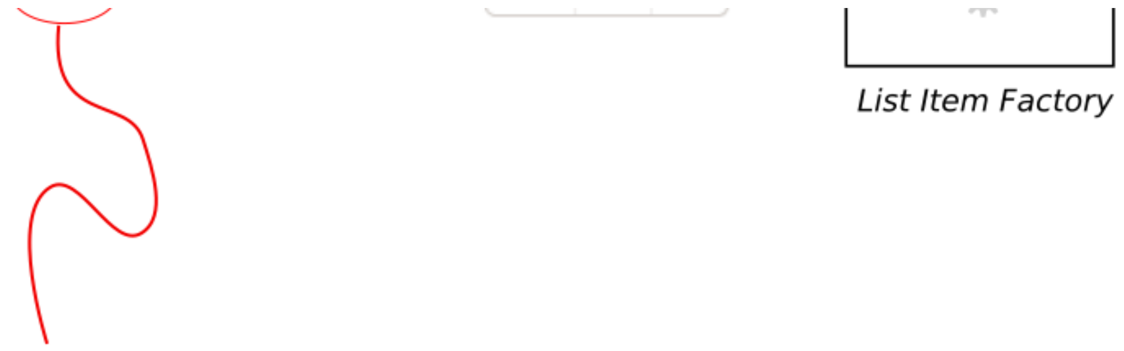


The responsibility of the *list item factory* is to produce a row widget and connect it to an item in the model, whenever the list view needs it.

The list view will create a few more rows than it needs to fill its visible area, to get a better estimate for the size of the scrollbars, and in order to have some "buffer"for when you decide to scroll the view.

Once you do scroll, we don't necessarily need to ask the factory to make more rows—we can recycle the rows that are being scrolled out of view on the other end.





Thankfully, all of this happens automatically behind the scenes. All you have to do is provide a list item factory.

Creating items

GTK offers two different approaches for creating items. You can either do it manually, with GtkSignalListItemFactory, or you can instantiate your row widgets from a ui file, using GtkBuilderListItemFactory.

The manual approach is easier to understand, so lets look at that first.

```
factory = gtk_signal_list_item_factory_new ();
g_signal_connect (factory, "setup", setup_listitem_cb, NULL);
g_signal_connect (factory, "bind", bind_listitem_cb, NULL);
```

The "setup" signal is emitted when the factory needs to create a new row widget, "bind" is emitted when a row widget needs to be connected to an item from the model.

Both of these signals take a GtkListItem as argument, which is a wrapper object that lets you get at the model item (with gtk_list_item_get_item()) and also lets you deliver the new row widget (with gtk_list_item_set_child()).

```
static void
setup_listitem_cb (GtkListItemFactory *factory,
                   GtkListItem        *list_item)
{
  GtkWidget *label = gtk_label_new ("");
  gtk_list_item_set_child (list_item, label);
}
```

Typically, your rows will be more complicated than a single label. You can create complex widgets and group them in containers, as needed.

```
static void
bind_listitem_cb (GtkListItemFactory *factory,
                  GtkListItem        *list_item)
{
  GtkWidget *label;
  MyObject *obj;

  label = gtk_list_item_get_child (list_item);
  obj = gtk_list_item_get_item (list_item);
  gtk_label_set_label (GTK_LABEL (label),
                       my_object_get_string (obj));
}
```

If your “bind” handler connects to signals on the item or does other things that require cleanup, you can use the “unbind” signal to do that cleanup. The “setup” signal has a similar counterpart called “teardown”.

The builder way

Our “setup” handler is basically a recipe for creating a small widget hierarchy. GTK has a more declarative way of doing this: GtkBuilder ui files. That is the way GtkBuilderListItemFactory works: you give it a ui file, and it instantiates that ui file whenever it needs to create a row.

```
ui = "<interface><template class="GtkListItem">...";
bytes = g_bytes_new_static (ui, strlen (ui));
gtk_builder_list_item_factory_new_from_bytes (scope, bytes);
```

You might now be wondering: Wait a minute, you are parsing an xml file for each of the thousands of items in my model, isn't that expensive?

There are two answers to this concern:

- We are not literally parsing the xml for each item; we parse it once, store the callback sequence, and replay it later.
- The most expensive part of GtkBuilder is actually not the xml parsing, but the creation of objects; recycling rows helps for this.

It is relatively easy to see how a ui file can replace the "setup" handler, but what about "bind"? In the example above, the bind callback was getting properties of the item (the MyObject:string property) and using their values to set properties of the widgets (the GtkLabel:label property). In other words, the "bind" handler is doing property bindings. For simplicity, we just created a one-time binding here, but we could have just as well used g_object_bind_property() to create a lasting binding.

GtkBuilder ui files *can* set up property bindings between objects, but there is one problem: The model item is not 'present' in the ui file, it only gets associated with the row widget later on, at "bind" time.

This is where GtkExpression comes in. At its core, GtkExpression is a way to describe bindings between objects that don't necessarily exist yet. In our case, what we want to achieve is:

```
label->label = list_item->item->string
```

Unfortunately, this gets a little more clumsy when it is turned into xml as part of our ui file:

```
<interface>
  <template class="GtkListItem">
    <property name="child">
      <object class="GtkLabel">
        <binding name="label">
          <lookup name="string">
            <lookup name="item">GtkListItem</lookup>
          </lookup>
        </binding>
      </object>
    </property>
  </template>
</interface>
```

Remember that the classname (GtkListItem) in a ui template is used as the “this” pointer referring to the object that is being instantiated.

So, <lookup name=”item”>GtkListItem</lookup> means: the value of the “item” property of the list item that is created. <lookup name=”string”> means: the “string” property of that object. And <binding name=”label”> says to set the “label” property of the widget to the value of that property.

Due to the way expressions work, all of this will be reevaluated when the list item factory sets the “item” property on the list item to a new value, which is exactly what we need to make recycling of row widgets work.

Expert level confusion

GtkBuilderListItemFactory and GtkExpression can get *really* confusing when you nest things—list item factories can be constructed in ui files themselves, and they can get given their own UI files as a property, so you end up with constructions like

```
<object class="GtkListView">
  <property name="factory">
    <object class="GtkBuilderListItemFactory">
      <property name="bytes"><![CDATA[
<?xml version="1.0" encoding="UTF-8"?>
<interface>
<template class="GtkListItem">
  <property name="child">
  ...
]]>
      </property>
    </object>
  </property>
...
```

This can be confusing even to GTK experts.

My advice would be avoid this when starting out with GtkListView—you don't have to create the list item factory in the UI file, and you can specify its UI template as a resource instead of embedding it directly.

Going deeper

Everything we've described here today applies to grid views as well, with minimal adjustments.

So far, we've focused on the view side of things. There's a lot to say about models too.

And then there is the column view, which deserves a post of its own.

mclasen / September 5, 2020 / Uncategorized / gtk4, guides, in depth, lists