

# **Seed Modules Reference Manual**

May 20, 2009

---

**Seed Modules Reference Manual**

# Contents

<b>1</b>	<b>readline module.</b>	<b>1</b>
1.1	API Reference . . . . .	1
1.2	Examples . . . . .	1
<b>2</b>	<b>SQLite module.</b>	<b>3</b>
2.1	API Reference . . . . .	3
2.2	Examples . . . . .	3
<b>3</b>	<b>GtkBuilder module.</b>	<b>5</b>
3.1	API Reference . . . . .	5
3.2	Examples . . . . .	5
<b>4</b>	<b>Sandbox module.</b>	<b>7</b>
4.1	API Reference . . . . .	7
4.2	Examples . . . . .	7



# Chapter 1

## readline module.

Robert Carr

### 1.1 API Reference

The readline module allows for basic usage of the GNU readline library, in Seed. More advanced features may be added a a later time. In order to use the readline module it must be first imported.

```
readline = imports.readline;
```

**readline.readline (prompt)** Prompts for one line of input on standard input using *prompt* as the prompt.

**prompt** A string to use as the readline prompt

**Returns** A string entered on standard input.

**readline.bind (key, function)** Binds *key* to *function* causing the function to be invoked whenever *key* is pressed

**key** A string specifying the key to bind

**function** The function to invoke when *key* is pressed

**readline.done ()** Indicates that readline should finish the current line, and return from *readline.readline*. Can be used in callbacks to implement features like multiline editing

**readline.buffer()** Retrieve the current readline buffer

**Returns** The current readline buffer

**readline.insert (string)** Inserts *string* in to the current readline buffer

**string** The string to insert

### 1.2 Examples

Below are several examples of using the Seed readline module. For additional resources, consult the examples/ folder of the Seed source

This demonstrates a simple REPL using the readline module

```
readline = imports.readline;
while (1){
  try{
    eval(readline.readline("> "));
  }
  catch(e) {
    Seed.print(e.name + " " + e.message);
  }
}
```

---

# Chapter 2

## SQLite module.

Robert Carr

### 2.1 API Reference

The `sqlite` module allows for manipulation and querying of `sqlite` databases.

```
sqlite = imports.sqlite;
```

The `SQLite` module provides a selection of status enums, to be used as the return values of functions. For meanings, consult the `SQLite C` documentation.

```
sqlite.[OK, ERROR, INTERNAL, PERM ABORT, BUSY,  
        LOCKED, NOMEM, READONLY, INTERRUPT, CORRUPT,  
        NOTFOUND, FULL, CANTOPEN, PROTOCOL, EMPTY,  
        SCHEMA, TOOBIG, CONSTRAINT, MISMATCH, MISUSE,  
        NOLFS, AUTH, FORMAT, RANGE, NOTADB, ROW, DONE]
```

**`new sqlite.Database(filename)`** Constructs a new `sqlite.Database`

**`filename`** The `SQLite` database to be opened (or constructed if it does not exist)

**Returns** A new `sqlite.Database` object, the `status` property will be one of the `SQLite` status enums

**`database.exec(command, callback)`** Executes the `SQLite` `command` on the given database. If `callback` is defined, it is called with each table entry from the given `command`, with a single argument. The argument has properties for each value in the returned table entry.

Keep in mind that, just like in C, it is necessary to sanitize user input in your SQL before passing it to the database.

**`command`** The `SQLite` command to evaluate

**`callback`** The callback to invoke, should expect one argument and return nothing. *optional*

**Returns** An `SQLite` status enum representing the result of the operation

**`database.close()`** Closes an `SQLite` database and syncs

### 2.2 Examples

Below are several examples of using the Seed `sqlite` module. For additional resources, consult the `examples/` folder of the Seed source

This demonstrates creating a new table, populating it, and querying it for results

```
sqlite = imports.sqlite;
var db = new sqlite.Database("people.db");
db.exec("create table people (key INTEGER PRIMARY KEY, name TEXT, " +
        "age INTEGER, phone TEXT);");
db.exec("insert into people(name, age, phone) " +
        "values('John Smith', 24, '555-123-4567');");

function cb_print_phone(results){
    Seed.print(results.phone);
}

db.exec("select from people where name='John Smith';", cb_print_phone);
db.close();
```

---

## Chapter 3

# GtkBuilder module.

Robert Carr

### 3.1 API Reference

The GtkBuilder extends `Gtk(GtkBuilder.prototype)` to implement a custom automatic signal connection function, which is useful in Seed. It does not provide any methods or types, so there is no need to save it's namespace, as of such it can be imported as follows.

```
imports.gtkbuilder;
```

**`builder.connect_signals(object, user_data)`** Connects the signals present in the GtkBuilder to the functions present in *object*. That is to say, a signal with handler name, 'ok\_button\_clicked' will be connected to the 'ok\_button\_clicked' property of object.

***object*** The object containing the signal handlers

***user\_data*** The user\_data to use in connecting the signals

### 3.2 Examples

Below are several examples of using the Seed GtkBuilder module. For additional resources, consult the `examples/` folder of the Seed source

```
<interface>
  <object class="GtkDialog" id="dialog1">
    <child internal-child="vbox">
      <object class="GtkVBox" id="vbox1">
        <property name="border-width">10</property>
        <child internal-child="action_area">
          <object class="GtkHButtonBox" id="hbuttonbox1">
            <property name="border-width">20</property>
            <child>
              <object class="GtkButton" id="ok_button">
                <property name="label">gtk-ok</property>
                <property name="use-stock">TRUE</property>
                <signal name="clicked" handler="ok_button_clicked"/>
              </object>
            </child>
          </object>
        </child>
      </object>
    </child>
  </object>
</interface>
```

```
#!/usr/local/bin/seed
Gtk = imports.gi.Gtk;
GtkBuilder = imports.gtkbuilder;

handlers = {
    ok_button_clicked: function(button) {
        Seed.quit();
    }
};

Gtk.init(Seed.argv);

b = new Gtk.Builder();
b.add_from_file("test.ui");
b.connect_signals(handlers);

d = b.get_object("dialog1");

d.show_all();

Gtk.main();
```

# Chapter 4

## Sandbox module.

Robert Carr

### 4.1 API Reference

The sandbox module allows the creation of isolated JSCore contexts with individual global objects. It is useful as a literal "sandbox" or in a variety of other contexts.

```
sandbox = imports.sandbox;
```

**new sandbox.Context()** Creates a new sandbox context object, which wraps a Seed JavaScript context with it's own global object. By default this global object contains only default JSCore globals (Array, Object, etc...) and has no ability to interact with the outside system. Note the context must be manually destroyed with the *destroy* method.

**context.eval(source)** Evaluates a string *source* with context, returns the result.

**source** The code to evaluate

**context.add\_globals()** Adds the default Seed globals to the context, including the 'Seed' global object, and the imports object.

**context.destroy()** Destroys the internal context object, and any further usage of the wrapper is an exception

**context.global** A project, representing the literal global object of the context, may be freely read from and assigned to

### 4.2 Examples

Below are several examples of using the Seed Sandbox module. For additional resources, consult the examples/ folder of the Seed source

---

```
sandbox = imports.sandbox;

ctx = new sandbox.Context();
ctx.eval("b = 2+2");
Seed.print(ctx.global.b); //4
ctx.global.b = new Gtk.Window(); // Possible to expose objects to the context.
ctx.eval("b.show()");
```

---