guiliani.de

Custom Extensions

Using Custom Extensions with the Guiliani Streaming Editor (GSE)

Product:	Guiliani Streaming Editor (GSE)
Release version:	2.2
Release date:	August 10, 2018

Table of contents

1. Introduction 2
2. Intended audience 2
3. Preparation and compilation 2
   3.1. Dependencies 2
   3.2. General Information 2
   3.3. Platform-specific differences 4
4. Using the Custom Extension Generator 4
5. Manually adding Custom Extensions 5
   5.1. What needs to be implemented 5
   5.2. The interface: CustomExtensionFuncs.h 5
   5.3. Example: A custom control 7
   5.4. Custom factories 8
   5.5. Custom enum values 8
6. Application dependencies 9

TES Electronic Solutions GmbH
www.guiliani.de

1. Introduction

This manual explains the steps that must be followed to extend the Guiliani Streaming Editor (GSE) with user functionality. User functionality means controls (widgets), commands, behaviors and layouters that are not part of the Guiliani core. Additionally, any enum declarations in these Custom Extensions can be announced to GSE to make it easier for the user to recognize them when editing the GUI.

2. Intended audience

This manual is aimed at programmers that implement new extended Guiliani functionality. A basic understanding of the Guiliani core concepts is necessary to follow this manual.

3. Preparation and compilation

3.1. Dependencies

Before considering compiling your own GSE and StreamRuntime, please make sure that these platform-specific dependencies are installed:

Windows:
  Microsoft Visual Studio

Linux:
```
gcc-c++
SDL_mixer-devel
freetype-devel
chmsee
```

3.2. General Information

In order to be able to use your own CustomExtensions, the GSE as well as the StreamRuntime both need to be recompiled with your code present in the build process. On all supported platforms CMake is used to setup the build environment for the specific toolchain. Adding your source-files to this environment is done by placing them inside the source-folder and re-running CMake.

On Windows and Linux PCs, the GSE as well as the StreamRuntime are managed in one CMake project. After installing cmake-gui from http://www.cmake.org/ or (optionally in case of Linux) your distribution's repository, proceed as follows:

1. Select the GSE-folder as "Source" in cmake-gui.

2. Create a new directory as the "Build" directory. This will contain files generated by CMAKE (e.g. the solution for Visual Studio or project for Code::Blocks)

NOTE:
As the source directory is used in a CMake-generated #define-statement, the path must be preprocessor-safe, i.e. it must not contain special characters like "#".

3. Clicking on "Configure" will open the toolchain selection window where the default can be used on PCs (using "make" on Linux and Visual Studio on Windows).

4. After that, click "Configure" twice to accept the preset default values for the project's variables – these need only be adjusted for specific build scenarios, e.g. when a release build is to be made to evaluate performance on an embedded device.

5. Then a click on "Generate" will create the platform-specific build files; on Windows a ".sln" file and its dependencies will be created, on Linux it will be a Makefile.

NOTE:
The "Configure" and "Generate" steps need to be repeated every time new files are added to the "StreamRuntime/Source/CustomExtension" and "StreamRuntime/Include/CustomExtension" directories, CMake will then detect .cpp and .h files accordingly and modify the build project accordingly.

6. On Linux, run "make" in the "Build" directory to generate the GSE and StreamRuntime binaries;
   On Windows, open the .sln file, set the Startup-Project to "GSE" with Visual Studio and build the project.

7. Note that the GSE binary will be placed in "Binary/win/pc" or "Binary/Linux/pc" according to the current platform. The StreamRuntime binary is placed into the "Resources" directory of the GSE.

8. The GSE binary placed in the root directory (used during "Getting started") will not be replaced by the build process; it is a copy of the default evaluation build and will not be able to open GSE projects with Custom Extensions. If you want to use the newly built GSE copy the binary into the root-folder.

NOTE:

When starting the GSE from within your IDE, please be sure to set the working-directory to the GSE-folder, so that it finds its resources.

3.3. Platform-specific differences

When compiling the StreamRuntime to be run on an embedded target device, use the “StreamRuntime” directory as the “Source”. You also need to select a different “Build”-folder to avoid any conflicts of the generated files.

When the toolchain selection window has opened choose the appropriate file from “CMake/Common/Toolchain” as the toolchain.

Building is then again done by running “make” in the “Build” directory, however the StreamRuntime binary will in this case be placed into “StreamRuntime/Binary/<platform-dependant path>” when the StreamRuntime is built on its own.

4. Using the Custom Extension Generator

Instead of manually adding Custom Extensions as described below, it is recommended to use the wizard functionality within GSE. Choose ***“Custom Extensions -> Create custom extension…”*** from the main screen to open the Custom Extension Generator.

In the window which appears, choose the type of extension you would like to generate from the combo-box. Afterwards enter a name for the newly generated class in the respective input field. GSE will automatically propose an ID for this name. This ID will be accessible from code through an enumeration. Hit “Ok” to let GSE generate the skeleton code for you.

You will find the generated code in your StreamRuntime’s “custom_extension”-directory. If for instance you chose to generate a Control named *MyControl*, then you will find a file names MyControl.cpp in this location. Feel free to modify and extend it as required.

NOTE:

Do not forget to re-run CMAKE and restart GSE after adding new Custom Extensions!

5. Manually adding Custom Extensions

If you wish to add your extensions manually into the project, instead of using the Custom Extension Generator, this chapter explains in detail the steps you need to take.

5.1. What needs to be implemented

For Custom Extensions to properly work with GSE and the target application, you must implement Guiliani's *streaming* correctly. Streaming is a concept that allows for serialization of Guiliani GUIs, for instance for reading GUI definitions from XML files at runtime. It is explained in detail in the Guiliani documentation. In a nutshell, it is all about implementing the virtual methods `ReadFromStream` and `WriteToStream` in each Custom Extension class to correctly read and write all attributes that belong to objects of the class.
The *class IDs* that are needed by Guiliani for dynamic creation of objects of the Custom Extension classes at runtime must be put into special header files. These are listed in the table below.

Type of extension	Class ID header file
Control (widget)	`CustomControlResource.h`
Command	`CustomCommandResource.h`
Behaviour	`CustomBehaviourResource.h`
Layouter	`CustomLayouterResource.h`

5.2. The interface: CustomExtensionFuncs.h

The interface for announcing new extensions is described in the header file `CustomExtensionFuncs.h`. It contains declarations of functions that are called by GSE after start-up. The implementations of these functions are located in the file `CustomExtensionFuncs.cpp`. This is the place where your new Custom Extensions must be put into the respective function implementations.

Each of the functions receives a vector to which *descriptors* of the Custom Extensions have to be appended. A descriptor is a simple class that contains some members including the Custom Extension's name and an instance pointer. The following table provides an overview of the types of Custom Extensions and the respective descriptor class.

Type of extension	Descriptor class
Control (widget)	`ControlDescriptor`
Command	`CommandDescriptor`
Behaviour	`BehaviourDescriptor`
Layouter	`LayouterDescriptor`

Detailed documentation of the descriptor classes can be found in `CustomExtensionFuncs.h` where they are declared.

5.3. Example: A custom control

Let’s assume you have implemented a custom control (sometimes called widget) that is called `CMyControl` and is declared in the header file `MyControl.h`. Here are the steps to be followed to make this control available to GSE users:

- Open the file `CustomExtensionFuncs.cpp`. At the top of this file, add a new include for your control:
  ```
  #include "MyControl.h"
  ```

- Edit the function `GetCustomControls` further down in the same file. As you can see from the declaration of this function, it receives a reference to a vector of `ControlDescriptor`s. The descriptor for your new control has to be added to this vector:

```
rkControls.push_back(ControlDescriptor(
   CTL_MY_CONTROL, "my control",
   new CMyControl(NULL, eC_FromInt(0), eC_FromInt(0),
         eC_FromInt(100), eC_FromInt(100))));
```

Let’s have a detailed look at this example. It calls the vector’s `push_back` method with an instance of `ControlDescriptor` that is created anonymously here. The most interesting details are in the construction of this `ControlDescriptor`:

- The first parameter is the control’s *class ID*. This ID must be defined for the streaming mechanism to work correctly. In this case, it must be added to the header file `CustomControlResource.h`. If you have already implemented your custom control with streaming, this ID is most likely already defined.

- The second parameter is a string that represents the human-readable name of the control. This name is used in GSE’s GUI.

- The third parameter is a pointer to an instance of the custom control. It is created anonymously here, but you may also create this object in advance and pass in the pointer. Just make sure that the instance you create here is in a valid state as it acts as a template for later usage. That means if a user creates an instance of this custom control later in the GUI, the control will look exactly like you describe it here. For instance, in this example the size is set to 100 by 100 pixels. Note that the parent pointer is `NULL` and no object ID is set (defaults to `NO_HANDLE`) because all this must be set by the GSE user after creating the object in the GUI.

For other extensions, the `GetCustom…` functions and descriptor classes work similarly.

5.4. Custom factories

You will have to include your header files at the beginning of the CustomExtensionFactory.cpp to compile. The procedure is identical no matter if you are adding controls, commands, or other extensions.

NOTE:

With release 2.1 of Guiliani an easier way to add Custom Extensions was introduced. Due to the new way class ids are defined (see doxygen Guiliani documentation about helper macros) the factory entries are generated automatically by the preprocessor. Therefore including your new control header file will be sufficient to generate the needed factory entry.

5.5. Custom enum values

If one of your Custom Extensions uses an internal enumeration and uses streaming for reading and writing of values of this enumeration type, it would be nice to allow users to select values of this enum with the respective enum value names instead of the literal integer values represented by them. Assume for instance that the example control (see above) has an internal enum that is declared like this:

```
enum MyEnum { MY_VALUE_1, MY_VALUE_2 };
```

and that it has a member variable of type `MyEnum` (called `m_eValue`) that it writes to a stream in its `WriteToStream` method like this:

```
GETOUTPUTSTREAM.WriteInt(m_eValue, "MyEnumValue");
```

It is possible to announce this enumeration and all of its values to GSE to have them represented with nice strings in the GUI. The function `GetCustomEnumMappings` has to be extended to do this. It works similarly to the other custom extension functions, and there is a separate descriptor class called `EnumMapping`. For the aforementioned example, the two lines that have to be added to the function would look like this:

```
    rkEnumMappings.push_back(EnumMapping(
        "MyEnumValue", MyControl::MY_VALUE_1, "MY_VALUE_1"));
    rkEnumMappings.push_back(EnumMapping(
        "MyEnumValue", MyControl::MY_VALUE_2, "MY_VALUE_2"));
```

The parameters of the `EnumMapping` constructor are:

- First, the XML tag that is used when streaming the value (see the `WriteInt` example above).
- Second, the actual value. It is recommended to reference the value like this (instead of hard-coding integer values) so the compiler produces an error in case you have to change the enum during development.
- And third, the value as a human-readable string. To avoid confusion it is recommended to use the same text as the enum value identifier.

6. Application dependencies

Since Guiliani's streaming mechanism works through class IDs it is imperative that the *same* ID header files that are used for GSE are also used for your application. This is easily achieved by using the same physical files in both projects.

NOTE:
In other words: If you are using IDs generated in GSE within your application Source-Code you must make sure to copy the header files exported by GSE over the respective *UserXXXXXXResource.h* files in the StreamRuntime's *include/GUIConfig* folder.