guiliani.de

Application Binding

A detailed description how to bind your application to Guiliani to make full use of its features

Product:	Guiliani-SDK
Release version:	2.4
Release date:	April 14, 2021

Table of contents

1 Introduction

This manual explains how your application can be enhanced using Guiliani-features. It also describes how to bind business-logic to the GUI.

2 Prerequisites

2.1 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. It is also recommended to have basic knowledge of the GSE and how to do things with it.

2.2 Preparation and compilation

Since for most parts of this document a compilation of the application or even the GSE is needed, please refer to the appropriate document for your platform to see how to set up your build-environment.

2.3 Reference

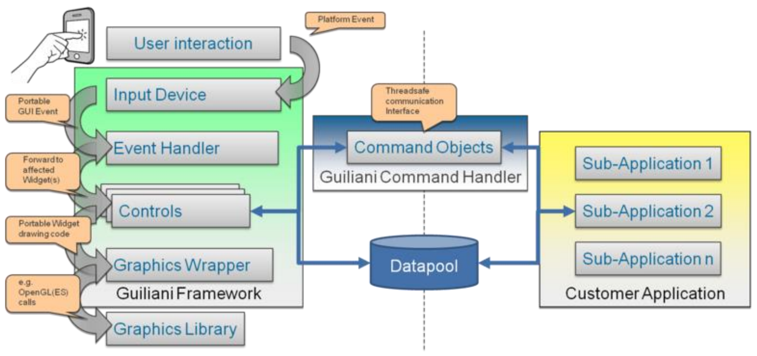
For detailed information on the topic described in this document and a complete reference please refer to the Guiliani API-documentation.

3 How application and GUI work together

What a useless piece of code would be an application when it could not interact with the GUI and vice versa. Fortunately Guiliani offers various ways – each with its good and bad sides – to help communicate between business-logic and GUI.

3.1 Event-flow of Guiliani

The following image depicts how the events go through the various parts of Guiliani and will eventually reach the application by Commands and DataPool.



The basic way of exchanging data and react to events is to use DataPool and commands.

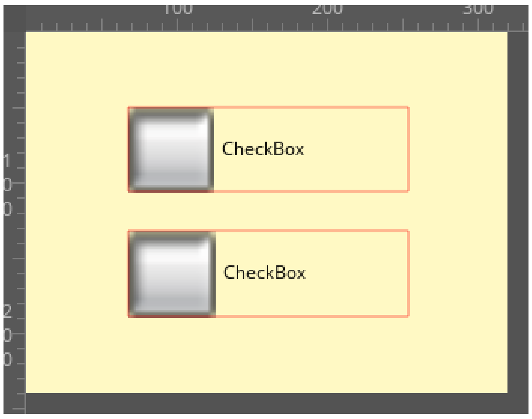
3.2 DataPool

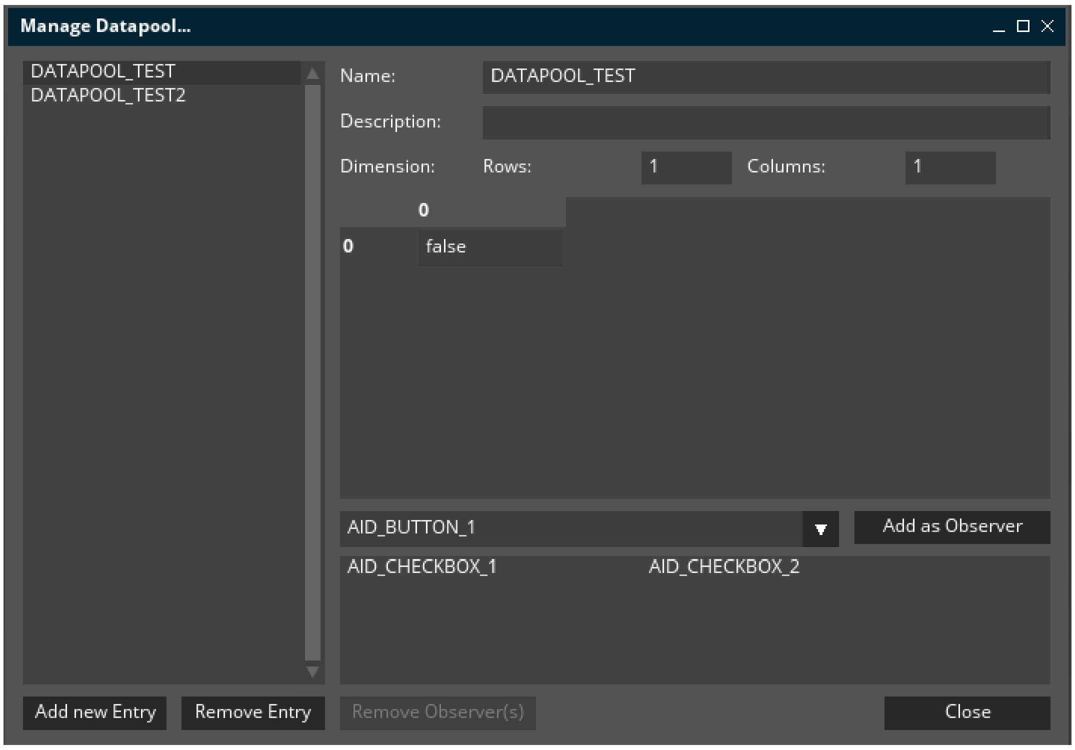
The DataPool is a way of exchanging data between the business-logic of your application and the GUI. To have this work a DataPool-item needs to be present in the application, created either manually by code or inside the GSE.

Also different controls in your application can be hooked to the same DataPool and so share the exact same information. This can be used to synchronize controls in your GUI.

3.2.1 Examples

Place two checkboxes into a dialog in the GSE and hook them to the same DataPool.





DATAPOOL_TEST
DATAPOOL_TEST2

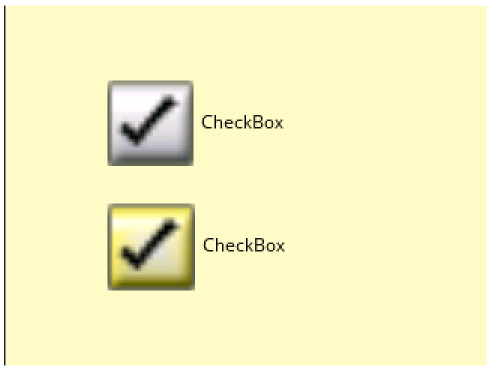
Name: DATAPOOL_TEST
Description:
Dimension: Rows: 1 Columns: 1

	0
0	false

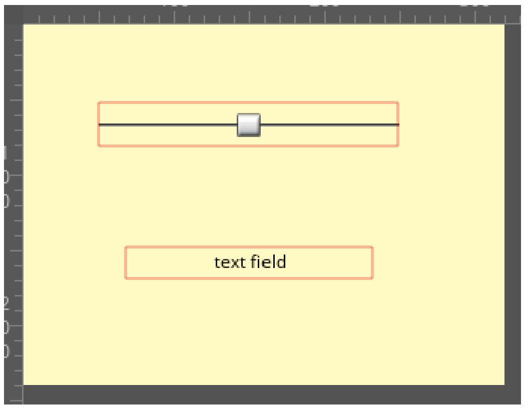
AID_BUTTON_1 ▼ Add as Observer

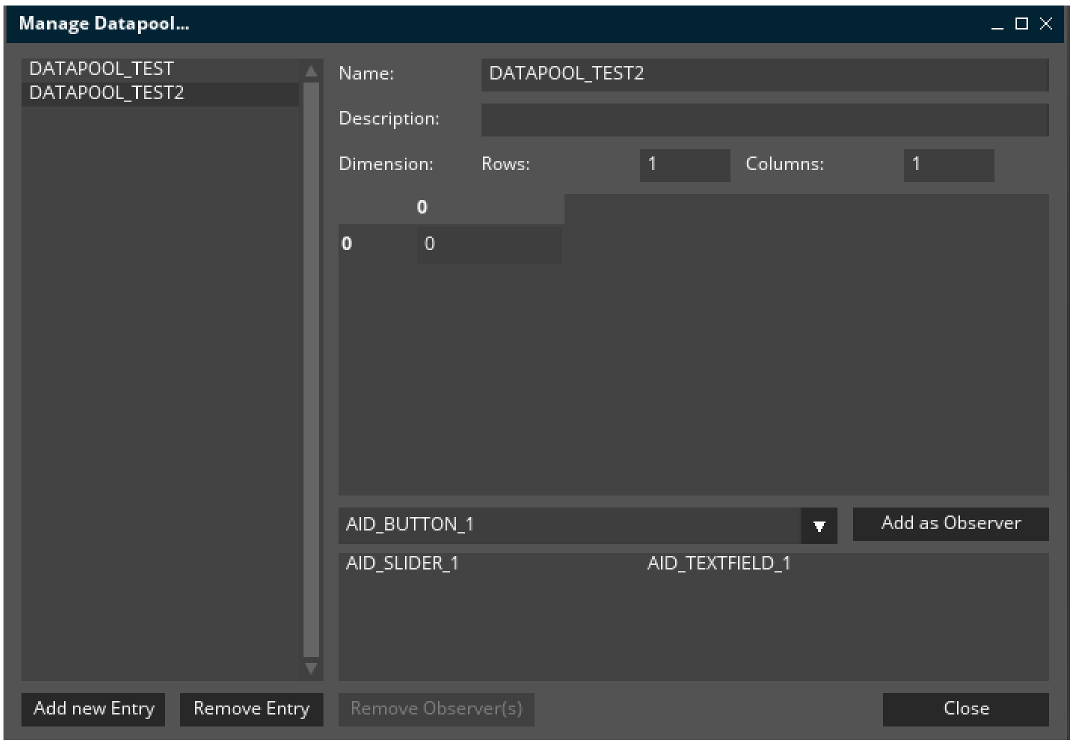
AID_CHECKBOX_1 AID_CHECKBOX_2

When you click on one of them the other one will set the value accordingly.



Now insert a Slider and a TextField and attach them to another DataPool.




Manage Datapool...

DATAPOOL_TEST
DATAPOOL_TEST2

Name: DATAPOOL_TEST2

Description:

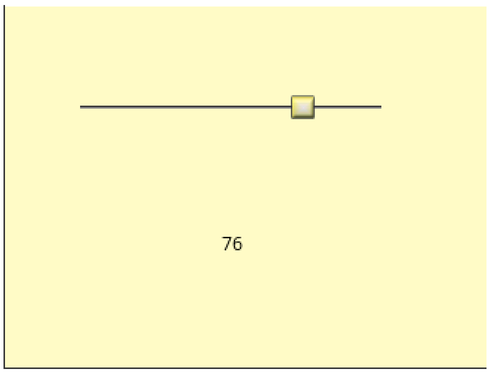
Dimension: Rows: 1 Columns: 1

	0
0	0

AID_BUTTON_1 — Add as Observer

AID_SLIDER_1 AID_TEXTFIELD_1

When you move the slider the TextField will mirror the value.



Of course it is not necessary that all controls which are attached to a DataPool will have to be in the same dialog. If a new dialog is created, Guiliani checks for the IDs of new controls and sets them according to the value of the DataPool.

3.2.2 Set DataPool from application

To set a DataPool to a specific value you basically call the following code:

```
CGUIDataPool::Set(DATAPOOL_ID, VALUE, X, Y);
```

Where DATAPOOL_ID is the ID which is used in your GSE-project to identify the DataPool and VALUE is a CGUIValue-instance containing the desired value. If you have a two-dimensional DataPool-item you can also specify where this value should be written to.

The next time the GUI-loop goes through the DataPool-processing it will take all waiting changes and apply them to the appropriate DataPool-items. After that the attached objects are notified and will update according to the sent value.

3.3 Commands

Commands can be used to trigger certain actions, either in the application or in the GUI. Inside a Command it is safe to manipulate the GUI directly by the application, since the Command will get executed inside the context of the GUI-thread. However, the time between the insertion of the command into the command-queue of Guiliani and its execution may vary based on the workload of the GUI and overall resources. But in most cases it is a slight delay of a few milliseconds.

Note: Commands are not guaranteed to be executed immediately. But it is guaranteed to be safe within the Command to access the GUI.

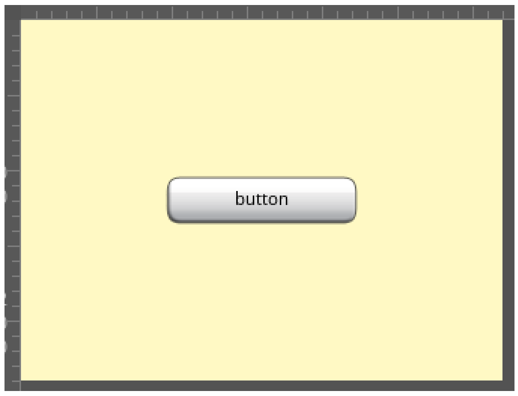
The core of a command is the method Do() which will be called by the command-handler of Guiliani when it is time to be executed.

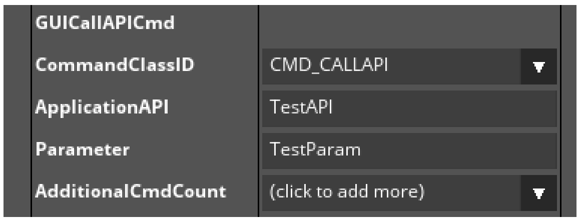
3.3.1 CMD_CALLAPI

The simplest way of triggering the application via a command is the CallAPI-Command. Here you just specify two string-parameters which can be used to distinguish what to do in the application.

When this command is attached to a control and is triggered, a short time later the method DoCallAPI of your application is called and you can react to the call according to the parameters kAPI and kParam.

Insert a button and attach a CMD_CALLAPI to it. Enter "TestAPI" for the first attribute and "TestParam" for the second one.



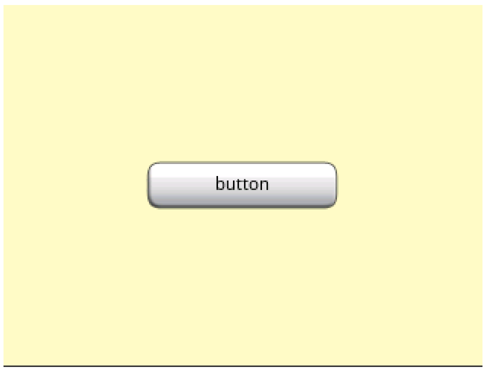


GUICallAPICmd	
CommandClassID	CMD_CALLAPI
ApplicationAPI	TestAPI
Parameter	TestParam
AdditionalCmdCount	(click to add more)

Now go into your application source-code at the file MyGUI_SR.cpp to the method "DoCallAPI". Insert the following code-snippet into it:

```
    if ((kAPI == "TestAPI") && (kParam == "TestParam"))
    {
        GETCMDHDL.Execute(new CGUIQuitCmd());
    }
```

Re-compile and execute the application from within your IDE. The button will be shown.



Now set a breakpoint at the code you have just entered and click on the button in the dialog. The execution will stop at the breakpoint in CMyGUI::DoCallAPI.
If you will now continue with execution a new instance of CGUIQuitCmd will be created and inserted into the command-queue of Guiliani. Shortly after this the application will exit.

3.3.2 CMD_CUSTOM

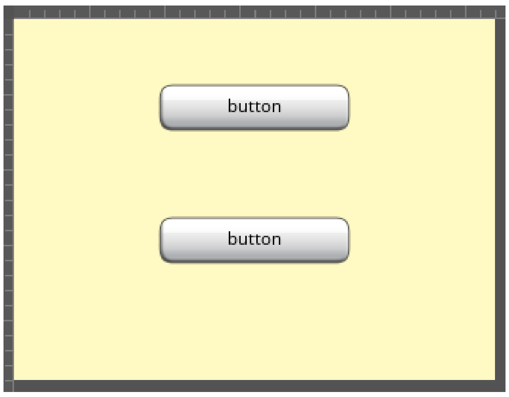
This command shows how the GUI can be manipulated with a command, while it doesn't matter if it was triggered from within the GUI or the application.

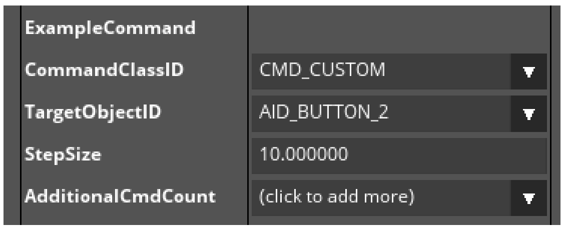
You find the file ExampleCommand.cpp in the folder "Source/Common/CustomExtension"

```
void ExampleCommand::Do()
{
    // when using GetObjectByID this define must be checked
#if defined(STREAMRUNTIME_APPLICATION)
    if (m_eTargetObject == NO_HANDLE || m_vStepSize == eC_FromInt(0))
    {
        return;
    }

    CGUIObject* pkObj = GETGUI.GetObjectByID(m_eTargetObject);
    if (pkObj == NULL)
    {
        return;
    }

    // Do not allow setting width or height less than zero.
    if (pkObj->GetWidth() + eC_Mul(m_vStepSize, eC_FromInt(2)) <
eC_FromInt(0) ||
        pkObj->GetHeight() + eC_Mul(m_vStepSize, eC_FromInt(2)) <
eC_FromInt(0))
    {
        return;
    }

    pkObj->InvalidateArea();
    CGUIRect kRect = pkObj->GetRelRect();
    kRect.Expand(m_vStepSize);
    pkObj->SetRelRect(kRect);
    pkObj->InvalidateArea();
#endif
}
```

In this command a target-object will be resized according to some value. After some checks to prevent unexpected behaviour (e.g. objects with negative sizes), the target-object will be invalidated with its current dimensions. Then it will be resized and again invalidated.

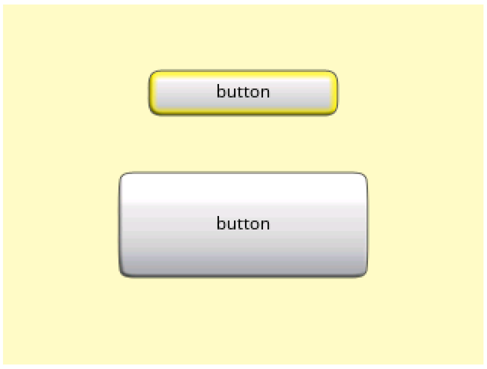
Place two buttons in a dialog. Attach the CMD_CUSTOM to the first one and use the ID of the second one for the attribute “TargetObjectID”. Enter the value of 10.00000 for the attribute “StepSize”





ExampleCommand	
CommandClassID	CMD_CUSTOM
TargetObjectID	AID_BUTTON_2
StepSize	10.000000
AdditionalCmdCount	(click to add more)

Run the application and click on the first button. You will see that the other will grow bigger every time you click on it.



4 Extend the functionality of Guiliani

To extend the functionality of Guiliani and use it inside your application, you can either do this manually by sub-classing a Guiliani-class (e.g. CGUIBehaviour) or creating a Custom-Extension.

The advantage when using Custom-Extensions is that the designer can use them directly within the GSE and play around with them without any programming knowledge.
The programmer just fills the important parts of the generated shell with custom code for visual representation and behaviour and updates the GSE and application.

But also the “traditional” Guiliani-extension by sub-classing has its charm, since it can be used inside the application without bothering the designer of its use in the GSE.

Note: when using Custom-Extensions during project-design you are strongly encouraged to use versioning inside the extensions to avoid breaking the project.

4.1 Create new features by sub-classing

Extending a Guiliani-feature is as easy as sub-classing anything else in the world. So we are covering the more interesting part of Custom-Extensions here.

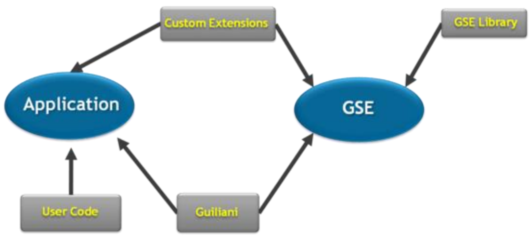
4.2 Custom-Extensions

There are a few pre-build Custom-Extensions (one for each type) shipped with Guiliani. You can find them inside the folders "*Common/Source/CustomExtension*" and "*Common/Include/CustomExtension*" of your Guiliani-application.

Custom-Extensions can directly be used in the GSE to have an even better WYSIWYG-experience. While the application is executed the Custom-Extensions will be created automatically when needed and filled with the attributes as in the GSE-project. This also means that the same code will be used for the application and the GSE.

Note: please be sure to always update and re-compile both application and GSE after changing to avoid any problems.

The following image shows how the code is used inside the application and the GSE



4.2.1 Generating a Custom-Extension in the GSE

Choose ***"Custom Extensions -> Create custom extension…"*** from the main screen to open the Custom Extension Generator.

In the opening window, choose the type of extension you would like to generate from the combo-box. Then 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. Finally hit the "OK-button" to let GSE generate the skeleton code for you and place it in the folders according to the project-settings.

Note: The folder where the generated files are placed is set in "Settings->Project Settings->Simulator Folder".

Note: The license limits for Custom Extensions apply to the usability within the GSE. Guiliani can, of course, be extended with new customized functionality with no restrictions.

4.2.2 Use Custom-Enumeration

If one of the Custom-Extensions uses an internal enumeration and uses streaming for reading and writing of values of this enumeration type, it can be used to allow the designer in the GSE 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_t
{
    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.

To do this the function `GetCustomEnumMappings` in the file CustomExtensionFuncs.cpp (in the folder “Common/Source/CustomExtension” of the application) has to be extended

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 of the attribute 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.