How to setup the project and to use the UI PluginAPI

Create a new UI Plugin

Handbook for creating a new UI plugin for the openTwin plattform

Contents

1 Difference between a service and a UI plugin

A UI Plugin is mainly used to add custom widgets to the UI frontend. The plugin API is designed for a service and plugin combination. The plugin should be used to extend the user interface experience. A service can request to load a plugin based on the UI Plugin API to the UI frontend. The plugin must be located on the same machine where the UI frontend is running. To control the plugin a message interface is provided for the service and the plugin by the UI Plugin API.

2 Setting up the Project

The easiest way to create a new UI Plugin for openTwin is to use the service template provided with the openTwin plattform. The template is located at "../Libraries/UIPluginTemplate".

a) Copy the whole folder to a directory of your choice
b) Rename the folder to your desired project name
c) Rename the project files to the same name of the project folder
    a. *.sln
    b. *.vcxproj
    c. *.vcxproj.filter
    d. *.vcxproj.user
d) If the project folder contains the folder ".vs" and/or "x64", delete them with all their contents
e) First adjust the batch files for editing and building the project:
   Replace the project path to the new path the *.vcxproj file of your new service is located at

```
"%DEVENV_ROOT%\devenv.exe" "%SIM_PLAT_ROOT%\Libraries\ServiceTemplate\UIPluginTemplate.vcxproj" %TYPE% "Debug|x64" /Out buildLog_Debug.txt
```

-or-

```
START "" "%DEVENV_ROOT%\devenv.exe" "%SIM_PLAT_ROOT%\Libraries\ServiceTemplate\UIPluginTemplate.vcxproj"
```

f) Now you can open the project by running the edit.bat file.
g) Build the project
h) In some cases it is necessary to restart the IDE and build the project again to get all syntax highlighting right

3 How to use the UIPluginAPI

3.1 Service

If a UI frontend is connected to the session a plugin can be requested by providing either its name or its full path.

```
void Application::uiConnected(ot::components::UiComponent * _ui)
{
	_ui->requestUiPlugin("UIPluginTemplate.dll");
}
```

The plugin path is "*<UI frontend executable path>\uiPlugins*". If the frontend is running in debug mode Another plugin path can also be provided.

```
void Application::uiConnected(ot::components::UiComponent * _ui)
{

#ifdef _DEBUG
	_ui->addPluginSearchPath("<Path to add>");
#endif // _DEBUG
}
```

If the frontend is running in Release mode, the Plugin needs to be copied to the default UI plugin path.

After the plugin was successfully loaded to the UI frontend, the service will be notified.

```
void Application::uiPluginConnected(ot::components::UiPluginComponent * _plugin) {
	_plugin->sendQueuedMessage("Hello from the service");
}
```

3.2 Plugin

The plugin is controlled by the PluginCore class. When the initialize function is called the plugin can create all required widgets and place them in the UI by using the UI interface.

```
bool PluginCore::initialize(void) {
	// Display info message
	m_uiInterface->appenInfoMessage("[PluginCore] Initializing UI plugin");

	// Create a text edit widget
	m_textEdit = new ak::aTextEditWidget;

	// Add the widget to the tab view in the UI frontend
	m_uiInterface->addNewTab("Test", m_textEdit);

	// Return true: Initialization successful
	return true;
}
```

Received messages will be forwarded to the PluginCore::messageRecieved function.

```
bool PluginCore::messageRecieved(const std::string& _message) {
	// Display received message in the text edit
	m_textEdit->append("Message received: " + QString::fromStdString(_message));

	// Return true: Initialization successful
	return true;
}
```