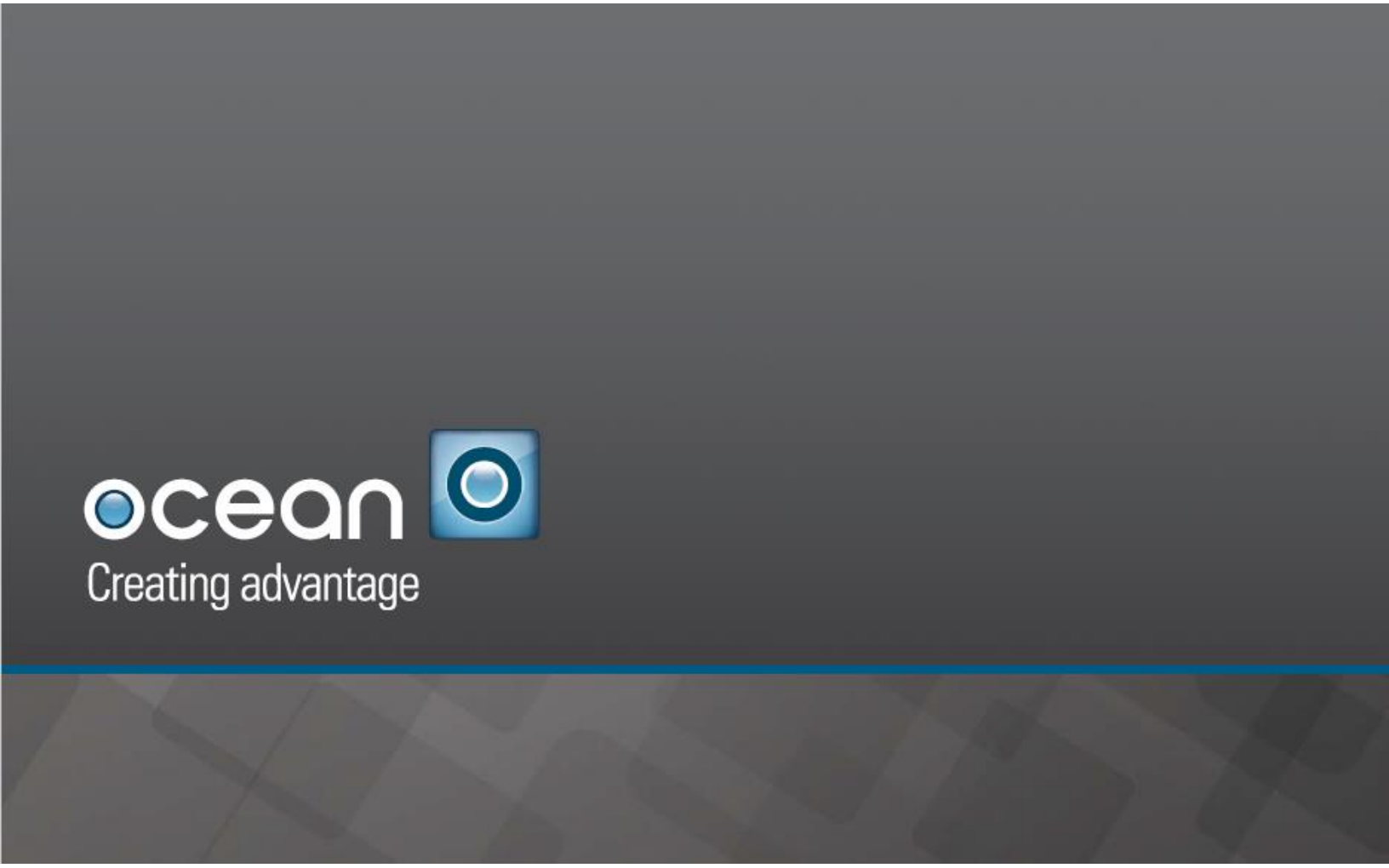
Basics
Volume 1



Ocean Software Development Framework for Techlog
Version 2023

Contents

1 Plug-in information and menu 1-1

Declare plug-in information 1-3

PluginInformation 1-3

Extend Techlog menus 1-7

Tab, group and action created by the plug-in 1-7

PluginMenuTab object 1-9

PluginMenuGroup object 1-12

Use case: Ocean plug-in B inserts a sub-group to a group of Ocean plug-in A 1-13

Use case: Ocean plug-in B inserts an action item and a separator to a group of Ocean plug-in A 1-14

MenuAction object 1-15

PluginMenuAction object 1-16

TechlogMenuAction object 1-18

FileMenuAction object 1-20

PythonAWIMenuAction object 1-23

Group and action created by the plug-in and added to native Techlog tab and group 1-24

Use case: add / insert a PluginMenuGroup to a native Techlog tab 1-26

Use case: add / insert a PluginMenuGroup (sub-group) to a native Techlog group 1-26

Use case: add / insert menu action items to a native Techlog group 1-27

Use case: add / insert menu items to a native Techlog sub-group 1-28

Plug-in menu action management with dependency on other plug-in 1-29

Use case: Conditional external plug-in menu action creation 1-30

Use case: Conditional action menu item creation pointing to an external plugin activity 1-31

2 Techlog convenience classes 2-1

Introduction 2-3

Session class 2-3

Session domain object 2-4

Session signals 2-9

CurrentWorkspaceChanged signal 2-10

EnabledFeaturesChanged signal 2-11

ProjectBrowserFilterChanged signal 2-12

ProjectSaved signal 2-13

Modal and modeless dialogs 2-13
Project class 2-17
Project domain object 2-18
Project properties 2-21
Temporary project 2-23
Workspace class 2-24
Workspace domain object 2-25
Workspace signals 2-28
DepthInteractionChanged signal 2-29
SelectionChanged signal 2-30
PlotCreated signal 2-33
Selection domain object 2-34
Techlog output console 2-37
UnitConverter class 2-38
Unit conversion 2-38
ProjectBrowser class 2-39
ProjectBrowser 2-39
Techlog units management 2-41
Project unit system signal 2-41
Techlog measurements 2-42
Unit catalog 2-43
Unit object 2-44
Display unit 2-44
Plot axes limits and display parameters 2-44
Techlog families management 2-45
Techlog families 2-45
Family object 2-45
MainFamily object 2-45
Family catalog 2-45
Call Python script from an Ocean plug-in 2-48
Progress dialog with Ocean 2-51

3 Accessing domain objects 3-1

General concepts 3-2
Domain object access patterns 3-3
Domain objects 3-3

Common patterns 3-4
Droid 3-4
Find and get patterns 3-5
isA pattern 3-6
Erase pattern 3-7
Name pattern 3-8
Support grouping 3-9
Locking mechanism 3-9
How to use locking mechanism 3-9
Best practices 3-13
Domain object signals 3-17
DomainObjectChanged signal 3-20
DomainObjectErased signal 3-21
DomainObjectRenamed signal 3-21
DomainObjectCollection 3-22
Domain object collection access best practice 3-23
Filter by domain object type 3-25

1 Plug-in information and menu

In This Chapter

Declare plug-in information 1-3
PluginInformation 1-3
Extend Techlog menus 1-7
Tab, group and action created by the plug-in 1-7
PluginMenuTab object 1-9
PluginMenuGroup object 1-12
Use case: Ocean plug-in B inserts a sub-group to a group of Ocean plug-in A 1-13
Use case: Ocean plug-in B inserts an action item and a separator to a group of Ocean plug-in A 1-14
MenuAction object 1-15
PluginMenuAction object 1-16
TechlogMenuAction object 1-18
FileMenuAction object 1-20
PythonAWIMenuAction object 1-23
Group and action created by the plug-in and added to native Techlog tab and group 1-24
Use case: add / insert a PluginMenuGroup to a native Techlog tab 1-26
Use case: add / insert a PluginMenuGroup (sub-group) to a native Techlog group 1-26
Use case: add / insert menu action items to a native Techlog group 1-27
Use case: add / insert menu items to a native Techlog sub-group 1-28
Plug-in menu action management with dependency on other plug-in 1-29
Use case: Conditional external plug-in menu action creation 1-30
Use case: Conditional action menu item creation pointing to an external plugin activity 1-31

Declare plug-in information

An Ocean plug-in must declare information to Techlog. Some specific Ocean plug-in information is required by the Techlog module manager to validate and display the plug-in in the **Ocean plug-ins** node.

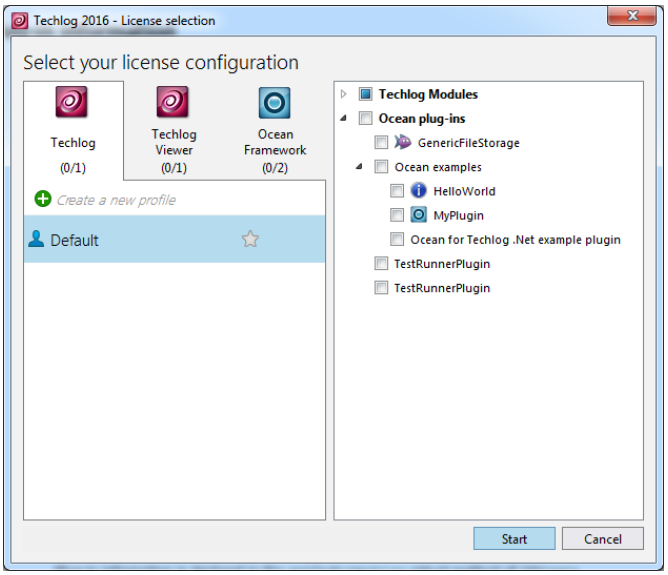


Figure 1-1 Techlog module manager

Set plug-in information in the **`getInformation`** virtual method of the **`IPlugin`** interface, overridden in main plug-in class. The main plug-in class is derived from the **`PluginIdentity`** class, which derived from **`IPlugin`**.

See the "Writing your first plug-in" section in *Getting Started with Ocean for Techlog* for more information on the virtual methods to be overridden in the main plug-in class.

```
class IPlugin
{
public:
  ...
  virtual void getInformation (PluginInformation
                &pluginInformation) const;
};
```

The **`getInformation`** virtual method takes a **`PluginInformation`** argument that allows you to set information about the plug-in.

PluginInformation

Plug-in information is available through the **`PluginInformation`** class. A **`PluginInformation`** object cannot be instantiated; populate the **`PluginInformation`** argument passed to the **`IPlugin::getInformation`** virtual method. This is in the plug-in main class, derived from the **`PluginIndentity`** class.

```
class PluginInformation
```

```
{
public:
  void setVendorName (const QString &name);
  void setName (const QString &name);
  void setDescription (const QString &value);
  void setVersion (const QString &value);
  void setReleaseDate (const QString &value);
  void setIcon (const QIcon &value);
  void setCreator (const QString &value);
  void setSupportEmail (const QString &emailAddress);
  void setCrashDumpEmail (const QString &emailAddress);

  void setLicenseFeature (const QString &seat);
  void addFeatureDependency(const Feature &feature);
  void setDependentLibraryDirectories (const QStringList
       &directories);
  void setPluginBundle(const QString &bundleName);
};
```

The **`PluginMenuInformation`** class allows you to set the plug-in information:

- Vendor name (company/owner) of the plug-in – mandatory information.
- Name of the plug-in – mandatory information.
- Short description of what the plug-in does.
- Version of the plug-in – mandatory information.
- Release date of the plug-in.
- Icon that is displayed next to the plug-in pane in Techlog module manager. An icon is set to the **`PluginInformation`** through a **`QIcon`** object and the path to the icon is relative to the plug-in folder.
- Creator's name (developer's name).
- Support email address.
- Crash dump email address.
- License feature name. This is the private license of the plug-in. Request this license feature through the Ocean store by contacting the Ocean Partner Program.
- Techlog feature dependencies of the plug-in. The plug-in is not enabled if corresponding Techlog features are not activated in the module manager.
- The plug-in bundle property gives the ability to group several plug-ins under a common node in the module manager.

This example shows setting the plug-in information.

```
void Plugin::getInformation(PluginInformation&
pluginInformation) const
{
  pluginInformation.setVendorName("Schlumberger");
  pluginInformation.setName("OceanPlugin");
```

```
  pluginInformation.setDescription("My awsome plug-in");
  pluginInformation.setVersion("1.0");
  pluginInformation.setReleaseDate("7/23/2015");
  pluginInformation.setIcon(QIcon("./images/ocean_32.png"));
  pluginInformation.setCreator("JSmith");
  pluginInformation.setSupportEmail("jsmith@slb.com");
  pluginInformation.setCrashDumpEmail("jsmith@slb.com");
  pluginInformation.setLicenseFeature("O4TL_SLB_ECOVIEW");
  pluginInformation.setPluginBundle("My Plug-ins Group");
  pluginInformation.addFeatureDependency(Feature::WBIFeature);
  pluginInformation.addFeatureDependency(
  Feature::GeophyFeature);

  pluginInformation.setDependentLibraryDirectories(
  QStringList() << Session::pluginDirectory() +
  QDir::separator() + "lib");
}
```

In the Techlog module manager, plug-in information is displayed in the description pane of the plug-in.

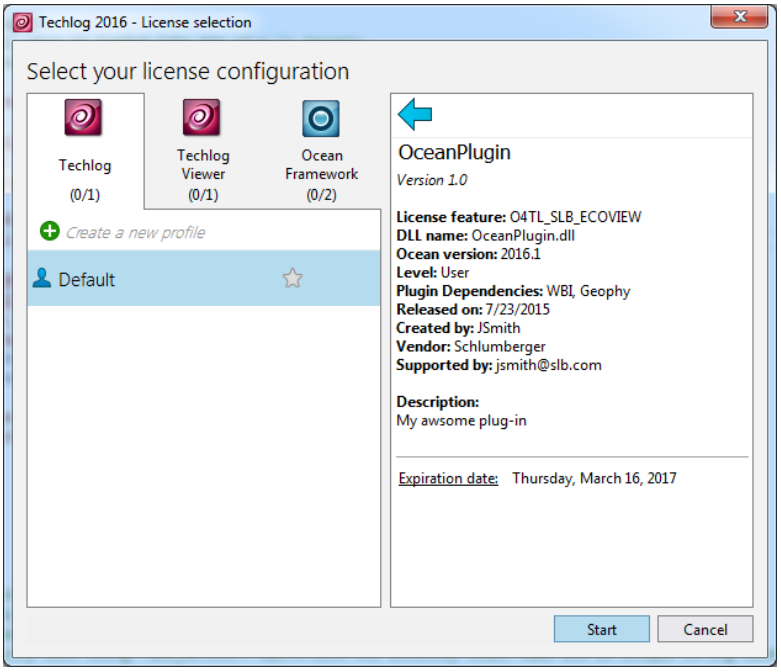


Figure 1-2 Plug-in information displayed in Techlog module manager

Note: The plug-in vendor name, name and version values passed to the **`setVendorName`**, **`setName`** and **`setVersion`** functions of **`PluginInformation`** class must match the plug-in structure folder names **VendorName**/**PluginName**/**TechlogVersion**/**PluginVersion**/. If this structure is not respected, the plug-in is not loaded by the Techlog module manager.

If the plug-in has dependent DLLs that are not deployed in the same folder as the plug-in DLL, you declare a list of directories (absolute paths) through the **`setDependentLibraryDirectories`** method where libraries used by the plug-in are located.

For example, if the dependent DLL is in the "lib" sub-folder of the plug-in folder, you use the **`Session::pluginDirectory()`** method to get the plug-in folder path and set the absolute path of the "lib" folder to the plug-in.

```
pluginInformation.setDependentLibraryDirectories(
QStringList() << Session::pluginDirectory() +
QDir::separator() + "lib");
```

The plug-in project must declare each DLL dependency as delay-loaded otherwise the plug-in cannot be instantiated by the Techlog module manager.

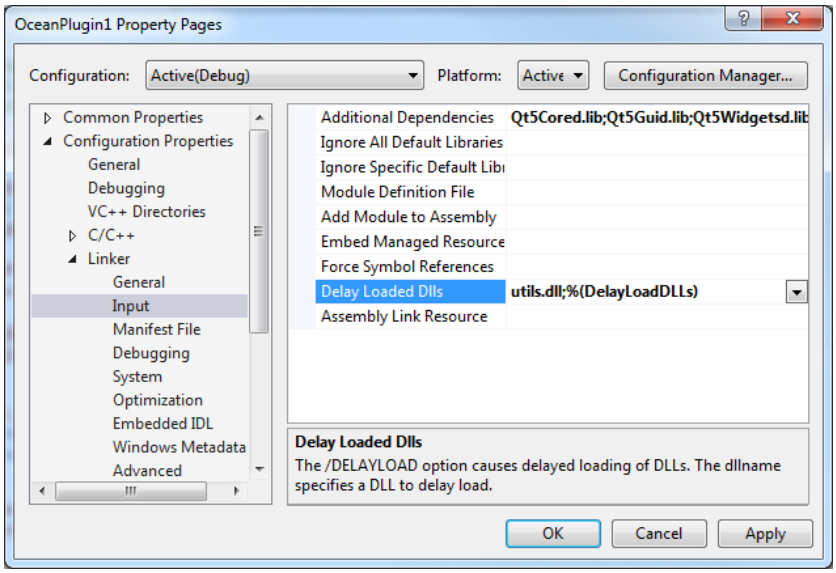


Figure 1-3 DLL dependency declared as Delay-Loaded

Extend Techlog menus

Ocean allows you to add new menus to customize the Techlog ribbon. Techlog tabs, groups and action items are available through **`PluginMenuTab`**, **`PluginMenuGroup`** and **`MenuAction`** classes. Ocean plug-in menus are created in the **`getMenu`** virtual method of **`IPlugin`**. You override this in the main plug-in class, derived from the **`PluginIdentity`** class, that is derived from **`IPlugin`** class.

See the "Writing your first plug-in" section in *Getting Started with Ocean for Techlog* for more information on virtual methods to be overridden in the main plug-in class.

There are two main ways to extend the Techlog ribbon:

- The Ocean plug-in handles tabs, groups and action items creation with their properties (title, tooltip, icon, etc.) and adds the plug-in menu to the Techlog menu.
- The Ocean plug-in instantiates a native Techlog tab and group through existing native Techlog menu ids. Then the Ocean plug-in creates a group and/or an action item and adds it to the appropriate native Techlog tab / group.

Tab, group and action created by the plug-in

Use the **`getMenu`** virtual method of **`IPlugin`** interface to declare an Ocean tab, group and action item and add them to the Techlog menu.

```
class IPlugin
{
public:
  ...
  virtual void getMenu (PluginMenu &menu) const;
};
```

This virtual method takes a **`PluginMenu`** argument that allows you to add plug-in tabs (modeled in Ocean with **`PluginMenuTab`** class) to the Techlog menu.

```
class PluginMenu
{
public:
  void addTab(const PluginMenuTab &menuTab);
  ...
};
```

Plug-in menu tabs are displayed in Techlog in the order that they are added to the list of tabs. Plug-in menu tabs are displayed after native Techlog tabs.

The **`insertTab`** function allows you to set the position of the tab in all **`PluginMenuTab`**s created by plug-ins that are activated in the module manager.

```
class PluginMenu
{
public:
  void insertTab(const PluginMenuTab &menuTab, float index);
  ...
};
```

The position of the tab in the list of tabs is passed to the function as the **`index`** argument. The index argument type is **`float`**; this gives you a fine granularity to insert the tab into existing plug-in menu tabs.

The **`index`** position of a tab in plug-in menus starts at 0.0. This example shows Ocean plug-in A that adds two tabs at the end of the Techlog ribbon when it is activated in the module manager.

```
void OceanPlugin_A::getMenu(PluginMenu& menu) const
{
  PluginMenuTab pluginMenuTab0("OceanPluginATab0");
  pluginMenuTab0.setTitle("OceanPluginA Tab 0");
  menu.addTab(pluginMenuTab0);

  PluginMenuTab pluginMenuTab1("OceanPluginATab1");
  pluginMenuTab1.setTitle("OceanPluginA Tab 1");

  menu.addTab(pluginMenuTab1);
}
```

The index values of Ocean plug-in A tabs are 0.0 and 1.0. If Ocean plug-in B wants to insert a tab between the two tabs of the Ocean plug-in A, it passes an index value between 0.0 and 1.0 to the **`insertTab`** function.

```
void OceanPlugin_B::getMenu(PluginMenu& menu) const
{
  PluginMenuTab pluginMenuTab0("OceanPluginBTab0");
  pluginMenuTab0.setTitle("OceanPluginB Tab 0");

  menu.insertTab(pluginMenuTab0, 0.5);
}
```

These screenshots show how the plug-in menu tab of Ocean plug-in B is inserted between two tabs of Ocean plug-in A following the activation sequence of these two plug-ins in the module manager.

1. Ocean plug-in A is activated in the Techlog module manager:

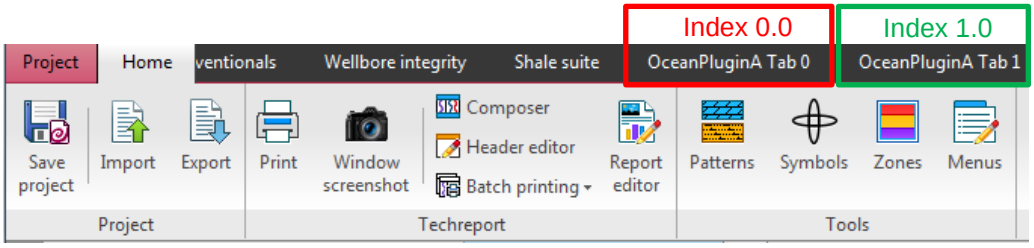


Figure 1-4 Ocean plug-in A adds two tabs to the plug-in menu

2. Ocean plug-in B is activated in the Techlog module manager:

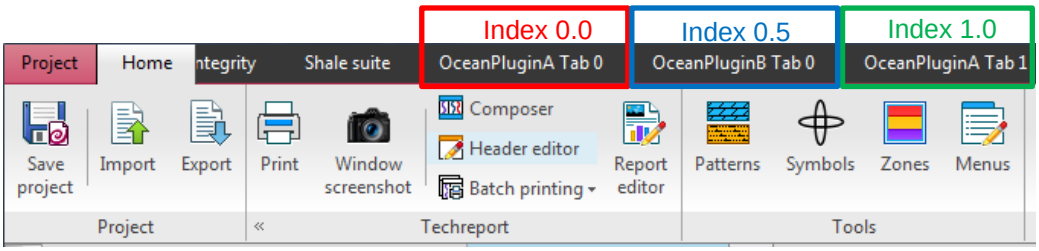


Figure 1-5 Ocean plug-in B inserts a tab to the plug-in menu

The **`PluginMenuTab`** insertion as described in the example works fine if the order of Ocean plug-in A and B activation stays the same. But if Ocean plug-in B happens to be activated before Ocean plug-in A or both plug-ins are activated at the same time, the Ocean plug-in B tab inserted via the **`insertTab`** function is added first to the Techlog ribbon and the index of this tab is set to 0.0. Therefore all tabs that Ocean plug-in A added using the **`addTab`** function are added to the end of the plug-in menu tabs. So if you have a bundle of plug-ins for which you want to control the menu tab insertion order, the best practice is to use the **`insertTab`** function for all plug-in tabs and specify the plug-in tab index explicitly for each **`PluginMenuTab`**.

If a **`PluginMenuTab`** is inserted with an index already used by another **`PluginMenuTab`**, this new **`PluginMenuTab`** is inserted before the existing tab with the same index. For instance, the Ocean plug-in B tab inserted with index value 0.0 is displayed before the OceanPluginATab0 and the new indexes of the two Ocean plug-in A tabs are 1.0 and 2.0.

PluginMenuTab object

The Techlog menu tab is available through the **`PluginMenuTab`** class. Instantiate a **`PluginMenuTab`** object using the constructor of the class. The plugin menu tab id passed to the constructor must be unique in the list of **`PluginMenu::tabs`**.

```
class PluginMenuTab
{
public:
  PluginMenuTab(const QString &areaId);

  void addGroup(const PluginMenuGroup &menuGroup);
  void insertGroup(const PluginMenuGroup &menuGroup,
       float index);

  void setTitle(const QString &value);
  void setTooltip(const QString &value);
};
```

The **`PluginMenuTab`** class allows you to:

- add a plug-in menu group to the plug-in menu tab. Plug-in menu groups are displayed in the plug-in menu tab in the order that they have been added
- insert a plug-in menu group in the plug-in menu tab at a given index position
- set the title and tooltip of the plug-in menu tab

The rules to insert groups in a plug-in menu tab using **`insertGroup`** are the same as the rules to insert a tab in a plug-in menu (see "Tab, group and action created by the plug-in" section on page 1-7).

Given that Ocean plug-in A tab 0 holds two groups, this scenario shows how Ocean plug-in B inserts a **`PluginMenuGroup`** in this same tab at a given index position.

1. The Ocean plug-in A inserts two groups in its tab 0 at positions 0.0 and 1.0

```
void OceanPlugin_A::getMenu(PluginMenu& menu) const
{
  PluginMenuTab pluginMenuTab0("OceanPluginATab0");
  pluginMenuTab0.setTitle("OceanPluginA Tab 0");

  PluginMenuGroup pluginMenuGroup0("OceanPluginATab0Group0");
  pluginMenuGroup0.setTitle("OceanPluginA Group 0");

  PluginMenuAction pluginMenuAction0(OceanActivity1Id);
  pluginMenuAction0.setText("OceanPluginA Action 0");
  pluginMenuAction0.setIcon(QIcon("ocean.png"));

  pluginMenuGroup0.addAction(pluginMenuAction0);

  PluginMenuGroup pluginMenuGroup1("OceanPluginATab0Group1");
  pluginMenuGroup1.setTitle("OceanPluginA Group 1");

  PluginMenuAction pluginMenuAction1(OceanActivity2Id);
  pluginMenuAction1.setText("OceanPluginA Action 1");
  pluginMenuAction1.setIcon(QIcon("ocean.png"));

  pluginMenuGroup1.addAction(pluginMenuAction1);

  pluginMenuTab0.insertGroup(pluginMenuGroup0, 0.0);
  pluginMenuTab0.insertGroup(pluginMenuGroup1, 1.0);

  menu.addTab(pluginMenuTab0);
}
```

2. Ocean plug-in B instantiates a **`PluginMenuTab`** object with the unique id of Ocean plug-in A tab 0 and inserts a group in the tab at the intermediate position 0.5.

```
void OceanPlugin_B::getMenu(PluginMenu& menu) const
{
  PluginMenuTab pluginMenuOceanPluginATab0("OceanPluginATab0");
  pluginMenuOceanPluginATab0.setTitle("OceanPluginA Tab 0");

  PluginMenuGroup pluginMenuGroup0("OceanPluginBTab0Group0");
  pluginMenuGroup0.setTitle("OceanPluginB Group 0");
```

```
  PluginMenuAction pluginMenuAction0(OceanActivity1Id);
  pluginMenuAction0.setText("OceanPluginB Action 0");
  pluginMenuAction0.setIcon(QIcon("ocean.png"));
  pluginMenuGroup0.addAction(pluginMenuAction0);

  pluginMenuOceanPluginATab0.insertGroup(pluginMenuGroup0,
  0.5);
  menu.addTab(pluginMenuOceanPluginATab0);
}
```

3. Activate both Ocean plug-ins A and B in the Techlog module manager

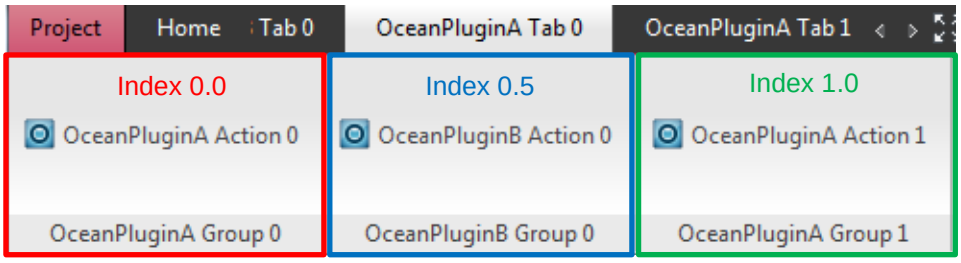


Figure 1-6 Ocean plug-in B inserts a group to a tab of Ocean plug-in A

A few things to note here:

- If Ocean plug-in A adds groups in its tab using the **`addGroup`** function, Ocean plug-in B won't be able to insert a group between Ocean plug-in A's two groups even if the Ocean plug-in A is activated before Ocean plug-in B. You must use the **`insertGroup`** function if you want to share a common **`PluginMenuTab`** with other plug-ins.
- In order to have a valid **`PluginMenuGroup`** to add to your plug-in menu, it must contain at least one **`MenuAction`**. See "MenuAction object" section on page 1-15 for details about the different action items you add to the Techlog ribbon.
- The first plug-in to instantiate an existing **`PluginMenuTab`** (by its unique id or any other technique) is the owner of the **`PluginMenuTab`**., and is the only plug-in that may change its properties. For example, a second plug-in that tries to change its **`title`** will have no effect. The only thing a plug-in that has not instantiated a **`PluginMenuTab`** may do is to modify it by adding or inserting a new **`PluginMenuGroup`** or a new action item in one of its groups, and then it must also notify Techlog of this change by adding the **`PluginMenuTab`** to the **`PluginMenu`** using the **`addTab`** function.
- If Ocean plug-in B is activated before Ocean plug-in A, Ocean plug-in B creates the **`PluginMenuTab`** with the id "OceanPluginATab0" and is the owner of this tab. In order to get rid of plug-in activation order issues, the best practice is to set **`PluginMenuTab`** properties in each plug-in's **`getMenu`** method.

PluginMenuGroup object

The Techlog menu group is available using the **`PluginMenuGroup`** class. Instantiate a **`PluginMenuGroup`** object using the constructor. The plug-in menu group id passed to the constructor must be unique in the list of groups of the **`PluginMenuTab`**.

```
class PluginMenuGroup
{
public:
  PluginMenuGroup(const QString &id);

  void addAction(const MenuAction &action);
  void addGroup(const PluginMenuGroup &pluginMenuGroup);
  void insertAction(const MenuAction &action, float index);
  void insertGroup(const PluginMenuGroup &pluginMenuGroup,
       float index);

  void setTitle(const QString &value);

  void setRowCount(unsigned int value);

  void addSeparator(const QString &title);
  void insertSeparator(const QString &title, float index);

  void setShrunkIcon(const QString &shrunkIconFilePath);
  void setIcon(const QIcon &icon);
  void setIconSmall();
  void setIconBig();
  ...
};
```

The **`PluginMenuGroup`** class allows you to:

- add a plug-in menu action item or a sub-group to the plug-in menu group. Plug-in menu action items and sub-groups are displayed in the plug-in menu group in the order that they have been added. Sub-groups are displayed after action items in the group.
- insert a plug-in menu action item or a sub-group in the plug-in menu group at a given **`index`** position.
- set the title of the plugin menu group.
- set the number of rows in the group on which action items and sub-groups are displayed in the plug-in menu group. The default value is 3 rows. If the number of action items is greater than the row count, at least one new column is added for additional action items. For example, when the row count is 2 and 3 action items are added to the group, then first two action items are displayed on row 1 and row 2 respectively and the third one is added to a new column on row 1.
- add a separator between group menu items. The text passed to the **`addSeparator`** function is only displayed in a sub-group.

- insert a separator in a sub-group at a given **`index`** position. The text passed to the **`insertSeparator`** function is only displayed in a sub-group.
- set the icon file path to use when the plug-in menu group is shrunk by the user in Techlog. The default shrunk icon is the icon of the first action item or sub-group of the group.
- set the icon for a sub-group. Adding an icon to a group that is not a sub-group has no effect. An icon is set for the **`PluginMenuGroup`** through a **`QIcon`** object and the path to the icon is relative to the plug-in folder.
- set the size (big or small) of a sub-group icon for a sub-group that belongs to the group. Calling **`setIconBig`** on a group that is not a sub-group has no effect. Setting a 32x32 icon for the sub-group handles both small and big icon displays. The default icon size is set to small.

Use case: Ocean plug-in B inserts a sub-group to a group of Ocean plug-in A

In this example, Ocean plug-in B instantiates a **`PluginMenuTab`** object and a **`PluginMenuGroup`** object with unique ids of "OceanPluginATab0" and "OceanPluginATab0Group0". Then it inserts a sub-group in the group at the first index position.

```
void OceanPlugin_B::getMenu(PluginMenu& menu) const
{
  PluginMenuTab pluginMenuOceanPluginATab0("OceanPluginATab0");
  pluginMenuOceanPluginATab0.setTitle("OceanPluginA Tab 0");
  PluginMenuGroup
  pluginMenuOceanPluginATab0Group0("OceanPluginATab0Group0");
  pluginMenuOceanPluginATab0Group0.setTitle(
  "OceanPluginA Group 0");

  PluginMenuGroup
  pluginMenuSubGroup0("OceanPluginBTab0Group0SubGroup0");
  pluginMenuSubGroup0.setTitle("OceanPluginB sub-group 0");
  pluginMenuSubGroup0.setIcon(QIcon("ocean.png"));
  pluginMenuSubGroup0.setIconBig();

  pluginMenuOceanPluginATab0Group0.insertGroup(
  pluginMenuSubGroup0, 0.0);
  pluginMenuOceanPluginATab0.addGroup(
  pluginMenuOceanPluginATab0Group0);
  menu.addTab(pluginMenuOceanPluginATab0);
}
```

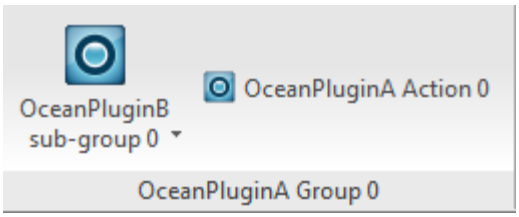


Figure 1-7 Ocean plug-in B inserts a sub-group to a group of Ocean plug-in A

Through the **`insertGroup`** function you have the ability to set the position of a sub-group not only in the list of sub-groups of the group but in all objects that composed the group as sub-groups, actions and separators. Each of these objects has an index in the group.

In order to notify Techlog that **`PluginMenuTab`** and **`PluginMenuGroup`** of Ocean plug-in A have been modified by Ocean plug-in B, the **`PluginMenuGroup`** must be added to the **`PluginMenuTab`** and the **`PluginMenuTab`** added to the **`PluginMenu`** in the **`getMenu`** virtual method of the Ocean plug-in B.

If Ocean plug-in B is activated before Ocean plug-in A, Ocean plug-in B creates a **`PluginMenuTab`** and a **`PluginMenuGroup`** with ids "OceanPluginATab0" and "OceanPluginATab0Group0", respectively. In this case Ocean plug-in B is the owner of the tab and the group. In order to get rid of plug-ins activation order issues, the best practice is to set the **`PluginMenuTab`** and **`PluginMenuGroup`** properties in each plug-in **`getMenu`** method.

There is no limitation in the number of nested groups:

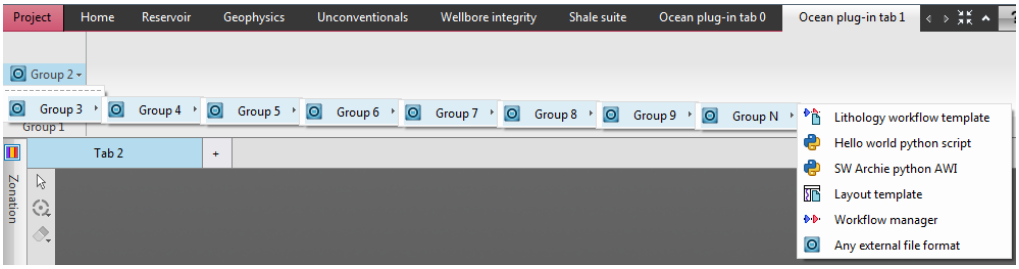


Figure 1-8 Nested groups in a PluginMenuGroup

Use case: Ocean plug-in B inserts an action item and a separator to a group of Ocean plug-in A

In this example, Ocean plug-in B instantiates a **`PluginMenuTab`** object and a **`PluginMenuGroup`** object with the unique ids "OceanPluginATab0" and "OceanPluginATab0Group0". Then it inserts an action item and a separator in the group just after the sub-group inserted in the previous use case.

```
void OceanPlugin_B::getMenu(PluginMenu& menu) const
{
  PluginMenuTab pluginMenuOceanPluginATab0("OceanPluginATab0");
  pluginMenuOceanPluginATab0.setTitle("OceanPluginA Tab 0");
  PluginMenuGroup
  pluginMenuOceanPluginATab0Group0("OceanPluginATab0Group0");
  pluginMenuOceanPluginATab0Group0.setTitle(
  "OceanPluginA Group 0");
```

```
  PluginMenuGroup
  pluginMenuSubGroup0("OceanPluginBTab0Group0SubGroup0");
  pluginMenuSubGroup0.setTitle("OceanPluginB sub-group 0");
  pluginMenuSubGroup0.setIcon(QIcon("ocean.png"));
  pluginMenuSubGroup0.setIconBig();

  pluginMenuOceanPluginATab0Group0.insertGroup(
  pluginMenuSubGroup0, 0.0);

  PluginMenuAction pluginMenuAction1(OceanActivity2Id);
  pluginMenuAction1.setText("OceanPluginB Action 1");
  pluginMenuAction1.setIcon(QIcon("ocean.png"));

  pluginMenuOceanPluginATab0Group0.insertAction(
  pluginMenuAction1, 1.0);
  pluginMenuOceanPluginATab0Group0.insertSeparator(
  "My Separator", 2.0);

  pluginMenuOceanPluginATab0.addGroup(
  pluginMenuOceanPluginATab0Group0);
  menu.addTab(pluginMenuOceanPluginATab0);
}
```

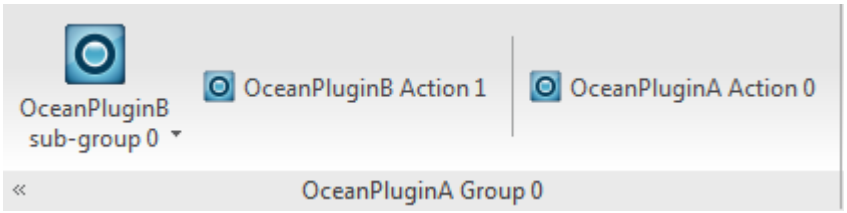


Figure 1-9 Ocean plug-in B inserts an action item and a separator to a group of Ocean plug-in A

MenuAction object

Ocean exposes the ability to add different types of action items to the Techlog ribbon.

1. Create a new action item, defining its title, tooltip, and icon properties. Then link this action item to a plug-in activity through a unique id.

   See the "Writing your first plug-in" section in *Getting Started with Ocean for Techlog* for more information on how to implement a plug-in activity.

2. Instantiate a native Techlog action item and add it to the plug-in menu.
3. Add an action item linked to a Techlog file as python script, template, workflow manager or any external file to the plug-in menu.
4. Create an action item that adds a Python AWI script to an AWI workflow.

These action items are handled by different Ocean classes grouped under the **MenuAction** base class. The class diagram shows these different classes:

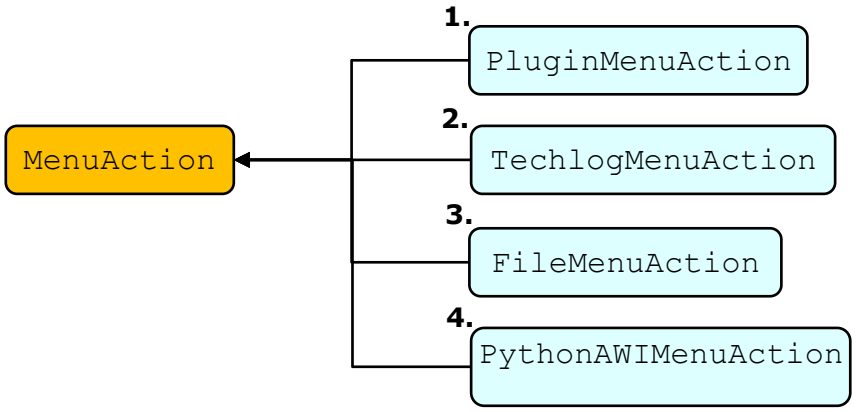


Figure 1-10 MenuAction class diagram

The **MenuAction** class holds properties common to all action items that you can instantiate in the **getMenu** method of your main plug-in class.

```
class MenuAction
{
public:
  void setText(const QString &text);
  void setTooltip(const QString &tooltip);
  void setIcon(const QIcon &icon);
  void setIconSmall();
  void setIconBig();
};
```

The **MenuAction** class allows you to:

- set the text and tooltip of the menu action item
- set an icon for an action item. An icon is set for the **MenuAction** through a **QIcon** object and the path to the icon is relative to the plug-in folder.
- set the size of the action item icon to large or small. Setting a 32x32 icon for the action item handles both small and large icon display. The default icon size is small.

PluginMenuAction object

Instantiate a **PluginMenuAction** object using the constructor of the class. The plug-in menu action id passed to the constructor must be unique in the list of plug-in menu action items. Use a unique action id in the **getActivities** virtual method of **PluginIdentity** class overridden in the main plug-in class. If a **PluginMenuAction** with an action id declared in the **getMenu** method is not linked to a plug-in activity in **getActivities** method, an error is raised at the plug-in runtime.

See the "Writing your first plug-in" section in *Getting Started with Ocean for Techlog* for more information on virtual methods that need to be overridden in the main plug-in class.

```
class PluginMenuAction : MenuAction
{
public:
  PluginMenuAction(const QString &activityId);
  ...
};
```

This example shows a **`PluginMenuTab`** that contains a **`PluginMenuGroup`** with several **`PluginMenuActions`** and a sub-group.

- the sub-group contains two action items separated by a separator
- each of the action items and the sub-group have an icon stored in an images folder at plug-in folder level
- icon size of first action item and sub-group is set to large
- the action items are displayed on two rows in the group

```
void Plugin::getMenu( PluginMenu& menu ) const
{
  // Create a new menu tab for the plug-in like this:
  PluginMenuTab tab("MyPluginTab");
  tab.setTitle("My plugin tab");
  // Create a new menu group for the plug-in like this:
  PluginMenuGroup group("MyPluginGroup");
  group.setTitle("My plugin group");
  group.setRowCount(2);

  PluginMenuAction actionMenu(OceanActivityId);
  actionMenu.setText("Ocean activity 1");
  actionMenu.setTooltip("Ocean activity 1");
  actionMenu.setIcon(QIcon("./images/crossPlot_32.png"));
  actionMenu.setIconBig();

  PluginMenuAction actionMenu1(OceanActivity1Id);
  actionMenu1.setText("Ocean activity 2");
  actionMenu1.setTooltip("Ocean activity 2");
  actionMenu1.setIcon(QIcon("./images/CustomPlot_32.png"));

  PluginMenuAction actionMenu2(OceanActivity2Id);
  actionMenu2.setText("Ocean activity 3");
  actionMenu2.setTooltip("Ocean activity 3");
  actionMenu2.setIcon(QIcon("./images/layout_32.png"));

  // Create a sub group in the group
  PluginMenuGroup subGroup("MyPluginSubGroup");
```

```
  subGroup.setTitle("My sub group");
  subGroup.setIcon(QIcon("./images/ocean_32.png"));
  subGroup.setIconBig();

  PluginMenuAction actionMenu3(OceanActivity3Id);
  actionMenu3.setText("Ocean activity 4");
  actionMenu3.setTooltip("Ocean activity 4");
  actionMenu3.setIcon(QIcon("./images/well_32.png"));

  PluginMenuAction actionMenu4(OceanActivity4Id);
  actionMenu4.setText("Ocean activity 5");
  actionMenu4.setTooltip("Ocean activity 5");
  actionMenu4.setIcon(QIcon("./images/dataset_32.png"));

  // Add the action menu to the plug-in sub-group
  subGroup.addAction(actionMenu3);
  subGroup.addSeparator("MySeparator");
  subGroup.addAction(actionMenu4);

  // Add the action menu to the plug-in group
  group.addAction(actionMenu);
  group.addAction(actionMenu1);
  group.addAction(actionMenu2);
  group.addGroup(subGroup);
  // Add the plug-in group to the plug-in tab
  tab.addGroup(group);

  // Add the plug-in tab to the plug-in menu
  menu.addTab(tab);
}
```

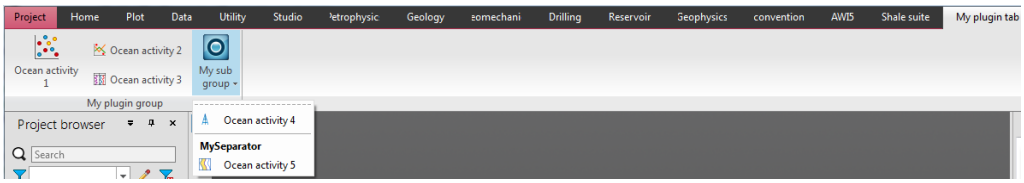


Figure 1-11 Example of tab, group and actions created by the plug-in

TechlogMenuAction object

Instantiate a native Techlog action item through the **`TechlogMenuAction`** class and add it to the plug-in menu.

```
class TechlogMenuAction : MenuAction
{
public:
  TechlogMenuAction(const TechlogMenuAction &activityId);
```

```
  ...
};
```

Instantiate the **`TechlogMenuAction`** object using its constructor, passing an argument for the native Techlog activity id found in the **`TechlogMenuActivities`** namespace.

```
namespace Slb {
  namespace Ocean {
    namespace Techlog {
      namespace TechlogMenuActivities {
        TechlogMenuAction getShaleGasActivity();
        TechlogMenuAction getShaleOilActivity();
        TechlogMenuAction getShaleVolumeActivity();
        ...
} } } }
```

This example shows how to add a Shale Volume **`TechlogMenuAction`** to a **`PluginMenuGroup`**.

```
void MenuActionItemsPlugin::getMenu(PluginMenu& menu) const
{
  PluginMenuTab pluginMenuTab("OceanPluginTab0");
  pluginMenuTab.setTitle("Ocean plug-in tab 0");

  PluginMenuGroup pluginMenuGroup("OceanPluginTab0Group0");
  pluginMenuGroup.setTitle("Group 0");

  TechlogMenuAction techlogActionVSH(
  TechlogMenuActivities::getShaleVolumeActivity());
  techlogActionVSH.setIconBig();

  pluginMenuGroup.addAction(techlogActionVSH);

  pluginMenuTab.addGroup(pluginMenuGroup);
  menu.addTab(pluginMenuTab);
}
```

Setting common properties like the text, tooltip and icon of an action item contained by the **`MenuAction`** base class has no effect on a **`TechlogMenuAction`**. Only setting the size of the icon using **`setIconSmall`** and **`setIconBig`** functions has any effect.

The native Shale volume Techlog action item is added to the plug-in menu and when the end-user clicks on the action item the Techlog Shale Volume activity is triggered, adding the Shale Volume AWI4 method to the Techlog workflow manager:

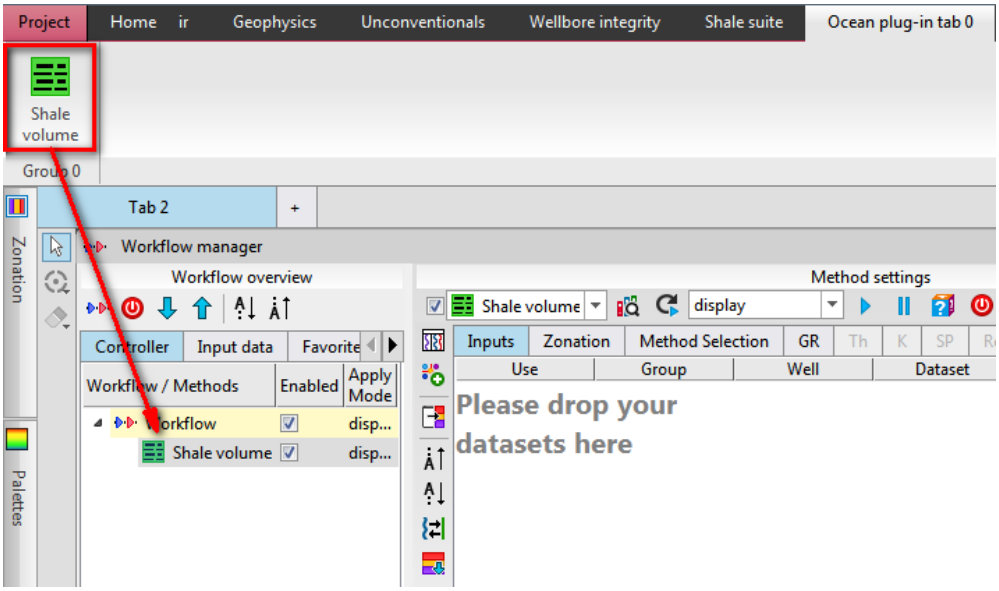


Figure 1-12 Native Techlog Shale Volume action item added to the plug-in menu

FileMenuAction object

Add an action item linked to a file (whether it is a Techlog file or an external file) to the plug-in menu using the **`FileMenuAction`** class.

```
class FileMenuAction : MenuAction
{
public:
  FileMenuAction(const QString &fileName,
    FileMenuActionFileType fileType, StorageLevel level);
  ...
};
```

Instantiate the **`FileMenuAction`** object using its constructor, passing the file name, the type of file, and the storage level of the file (Techlog, Company, User or the plug-in level) as arguments.

File types are available from the **`FileMenuActionFileType`** enum class.

- **`FileMenuActionFileTypeWorkflowTemplate`** is a workflow template XML file stored in a folder named "QuantiTemplates" at a storage level passed to the **`FileMenuAction`** constructor. A workflow template saves only the methods (no data) of one workflow in the workflow manager.
- **`FileMenuActionFileTypePythonScriptSilentMode`** is a python script that is run without opening the Python Editor. The python script is stored in a folder named "PythonScripts" at a storage level passed to the **`FileMenuAction`** constructor.
- **`FileMenuActionFileTypePythonScriptEditMode`** is a python script that is opened in the Python Editor. The python script is stored in a folder named "PythonScripts" at a storage level passed to the **`FileMenuAction`** constructor.

- **`FileMenuActionFileTypeLayoutTemplate`** is a Logview layout template XML file stored in a folder named "LayoutTemplates" at a storage level passed to the **`FileMenuAction`** constructor. A layout template saves only the content of a Logview (variable names and display) and may be applied to other wells and datasets.
- **`FileMenuActionFileTypeWorkflowManager`** is a workflow manager XML file saved in a folder named "WorkflowManager" at a storage level passed to the **`FileMenuAction`** constructor. A workflow manager saves all workflows with their methods (no data). The plug-in storage level isn't supported for this type of Techlog file.
- **`FileMenuActionFileTypeExternalFile`** is any other file type and is stored at a storage level passed to the **`FileMenuAction`** constructor.

Note: Pass a file name with no file extension to the **`FileMenuAction`** constructor for all types other than **`FileMenuActionFileTypeExternalFile`**; for external files, you must include the file extension. You may pass a relative path to this external file using the **`fileName`** argument.

This example shows the different file types to use with the **`FileMenuAction`** object.

```
void MenuActionItemsPlugin::getMenu(PluginMenu& menu) const
{
  PluginMenuTab pluginMenuTab1("OceanPluginTab1");
  pluginMenuTab1.setTitle("Ocean plug-in tab 1");

  PluginMenuGroup pluginMenuGroup1("OceanPluginTab1Group1");
  pluginMenuGroup1.setTitle("Group 1");

  FileMenuAction fileMenuActionWorkflowTemplate(
  "lithoWorkflowTemplate",
  FileMenuActionFileTypeWorkflowTemplate,
  StorageLevelPlugin);
  fileMenuActionWorkflowTemplate.setText(
  "Lithology workflow template");
  fileMenuActionWorkflowTemplate.setIconBig();

  FileMenuAction fileMenuActionPythonScript("HelloWorld",
  FileMenuActionFileTypePythonScriptSilentMode,
  StorageLevelPlugin);
  fileMenuActionPythonScript.setText(
  "Hello world python script");
  fileMenuActionPythonScript.setIconBig();

  FileMenuAction fileMenuActionPythonScriptEditMode(
  "HelloWorld", FileMenuActionFileTypePythonScriptEditorMode,
  StorageLevelPlugin);
  fileMenuActionPythonScriptEditMode.setText(
```

```
  "Hello world python script edit mode");
  fileMenuActionPythonScriptEditMode.setIconBig();

  FileMenuAction fileMenuActionLayoutTemplate("Well9_short",
  FileMenuActionFileTypeLayoutTemplate, StorageLevelPlugin);
  fileMenuActionLayoutTemplate.setText("Layout template");
  fileMenuActionLayoutTemplate.setIconBig();

  FileMenuAction fileMenuActionWorkflowManager(
  "WorkflowManager", FileMenuActionFileTypeWorkflowManager,
  StorageLevelUser);
  fileMenuActionWorkflowManager.setText("Workflow manager");
  fileMenuActionWorkflowManager.setIconBig();

  FileMenuAction fileMenuActionExternalFile(
  "doc/How-to-add-plugin-documentation.html",
  FileMenuActionFileTypeExternalFile, StorageLevelPlugin);
  fileMenuActionExternalFile.setText(
  "Any external file format");
  fileMenuActionExternalFile.setIcon(QIcon("ocean.png"));
  fileMenuActionExternalFile.setIconBig();

  pluginMenuGroup1.addAction(fileMenuActionWorkflowTemplate);
  pluginMenuGroup1.addAction(fileMenuActionPythonScript);
  pluginMenuGroup1.addAction(
  fileMenuActionPythonScriptEditMode);
  pluginMenuGroup1.addAction(fileMenuActionLayoutTemplate);
  pluginMenuGroup1.addAction(fileMenuActionWorkflowManager);
  pluginMenuGroup1.addAction(fileMenuActionExternalFile);

  pluginMenuTab1.addGroup(pluginMenuGroup1);
  menu.addTab(pluginMenuTab1);
}
```

The screenshot shows that all **`FileMenuAction`**s are added to the plug-in menu with a default icon that corresponds to the file type, except for the external file type for which you set the icon. You may overridde the default type icon with any other icon using the **`setIcon`** function of the **`MenuAction`** base class.

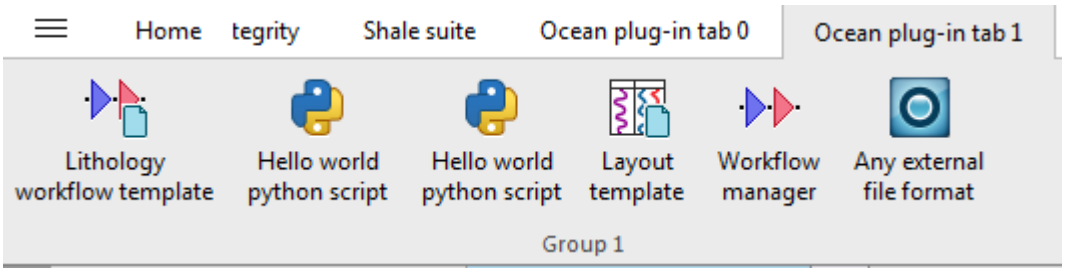


Figure 1-13 File type menu actions

PythonAWIMenuAction object

Create an action item that triggers the add of a Python AWI script to an AWI workflow using the **`PythonAWIMenuAction`** class.

```
class PythonAWIMenuAction : MenuAction
{
public:
  PythonAWIMenuAction (const QString &fileName,
                       StorageLevel level);
  void setWorkstepName(const QString &workstepName);
  void setAWI4SelectionWindowDisplayed(const bool
                       &AWI4SelectionWindowDisplayed);
  void setWorkflowName(const QString &workflowName);
  ...
};
```

Instantiate the **`PythonAWIMenuAction`** object using its constructor, passing the file name and the storage level of the file (Techlog, Company, User or the plug-in level) as arguments.

The **`PythonAWIMenuAction`** class allows you to:

- Define a name for the Python AWI workstep that will be added to the selected AWI workflow.
- Enable or disable (disable by default) the mnemonic selection window that shows up before the Python AWI workstep is added to the selected AWI workflow.
- Set the parent workflow name where is added the Python AWI workstep.

Note: Pass a file name with no file extension to the **`PythonAWIMenuAction`**.

This example shows how to create a **`PythonAWIMenuAction`**.

```
void MenuActionItemsPlugin::getMenu(PluginMenu& menu) const
{
  PluginMenuTab pluginMenuTab1("OceanPluginTab1");
  pluginMenuTab1.setTitle("Ocean plug-in tab 1");
```

```
  PluginMenuGroup pluginMenuGroup1("OceanPluginTab1Group1");
  pluginMenuGroup1.setTitle("Group 1");

  PythonAWIMenuAction pythonAWIMenuAction("SWArchie",
  StorageLevelPlugin);
  pythonAWIMenuAction.setAWI4SelectionWindowDisplayed(true);
  pythonAWIMenuAction.setWorkstepName("WS Archie");
  pythonAWIMenuAction.setWorkflowName("Water Saturation");
  pythonAWIMenuAction.setText("Water Saturation Archie");
  pythonAWIMenuAction.setIconBig();
  pluginMenuGroup1.addAction(pythonAWIMenuAction);

  pluginMenuTab1.addGroup(pluginMenuGroup1);
  menu.addTab(pluginMenuTab1);
}
```

The screenshot shows the **`PythonAWIMenuAction`** added to the plug-in menu with a default Python icon. A click on the **`PythonAWIMenuAction`** triggers the Python AWI script added as a workstep with name "WS Archie" to the selected AWI workflow renamed to "Water Saturation"

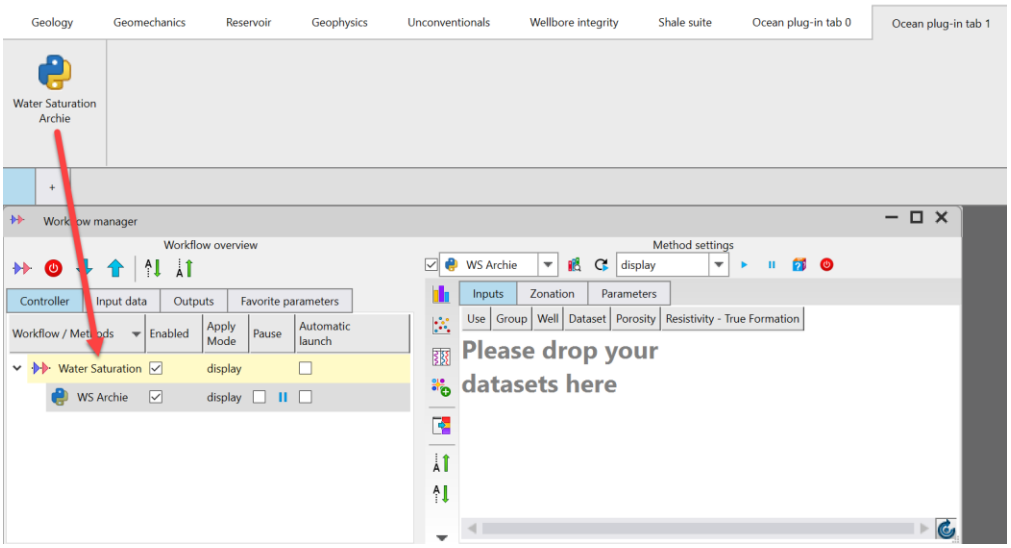


Figure 1-14 Python AWI menu action

Group and action created by the plug-in and added to native Techlog tab and group

You may insert group and action menus in some existing native Techlog tabs or groups. Each native tab and group is identified in Techlog through an id. The complete list of tab and group ids is provided in namespaces in the **`tsdknowntlmenuid`** header file.

`TabsIDs` namespace gives all Techlog tab ids.

```
namespace Slb {
  namespace Ocean {
    namespace Techlog {
      namespace TabsIDs {
        QString home();
        QString plot();
        QString data();
        QString utility();
        QString studio();
        QString petrophysics();
        QString geology();
        QString geomechanics();
        QString drilling();
        QString reservoir();
        QString geophysics();
        QString unconventionals();
        QString wellboreIntegrity();
        QString productionLogging();
} } } }
```

`GroupsIDs` namespace gives all Techlog group ids.

```
namespace Slb {
  namespace Ocean {
    namespace Techlog {
      namespace GroupsIDs {
        QString project();
        QString techReport();
        QString tools();
        QString view();
        QString windows();
        QString plotLayout();
        QString plotSingleWell();
        QString plotMultiWell();
        QString wbi();
        ...
} } } }
```

Instantiate **`PluginMenuTab`** and **`PluginMenuGroup`** objects passing the native Techlog tab and group id that you want to extend with your plug-in menu in the class constructors.

Note: Native Techlog menu properties cannot be modified by the plug-in. Once you get **`PluginMenuTab`** and **`PluginMenuGroup`** instances from Techlog native ids, setting properties on those objects has no effect.

Use case: add / insert a PluginMenuGroup to a native Techlog tab

This example instantiates a **`PluginMenuTab`** object with the **`geology`** native Techlog tab id. Then a **`PluginMenuGroup`** with a **`PluginMenuAction`** is inserted in this tab at the index position 0.5.

```
void MenuActionItemsPlugin::getMenu(PluginMenu& menu) const
{
  PluginMenuTab techlogTabGeology(TabsIDs::geology());

  PluginMenuGroup pluginMenuGroupToGeology(
  "OceanPluginGroupToGeology");
  pluginMenuGroupToGeology.setTitle("Plug-in Menu Group");
  pluginMenuGroupToGeology.setIcon(QIcon("ocean.png"));
  pluginMenuGroupToGeology.setIconBig();

  PluginMenuAction pluginMenuAction(OceanActivityId);
  pluginMenuAction.setText("Plug-in menu action item");
  pluginMenuAction.setIcon(QIcon("ocean.png"));
  pluginMenuAction.setIconBig();

  pluginMenuGroupToGeology.addAction(pluginMenuAction);
  techlogTabGeology.insertGroup(pluginMenuGroupToGeology, 0.5);
  menu.addTab(techlogTabGeology);
}
```

The **`PluginMenuGroup`** is inserted at the second index position of the **Geology** tab.

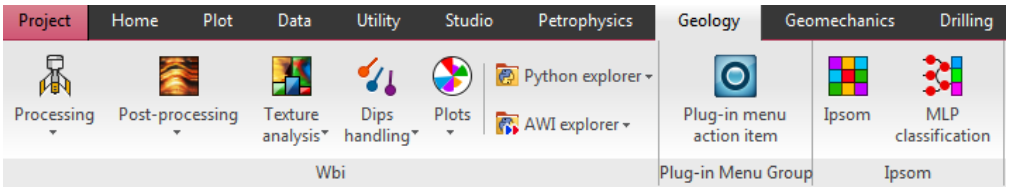


Figure 1-15 PluginMenuGroup inserted to native Techlog tab

Use case: add / insert a PluginMenuGroup (sub-group) to a native Techlog group

This example instantiates a **`PluginMenuTab`** object with the **`geology`** native Techlog tab id and a **`PluginMenuGroup`** object is instantiated with the **`wbi`** native Techlog group id. Then a **`PluginMenuGroup`** with a **`PluginMenuAction`** is inserted in that tab and group at the index position 1.5.

```
void MenuActionItemsPlugin::getMenu(PluginMenu& menu) const
{
  PluginMenuTab techlogTabGeology(TabsIDs::geology());

  PluginMenuGroup pluginMenuGroupToGeology(
  "OceanPluginGroupToGeology");
  pluginMenuGroupToGeology.setTitle("Plug-in Menu Group");
  pluginMenuGroupToGeology.setIcon(QIcon("ocean.png"));
```

```
  pluginMenuGroupToGeology.setIconBig();

  PluginMenuAction pluginMenuAction(OceanActivityId);
  pluginMenuAction.setText("Plug-in menu action item");
  pluginMenuAction.setIcon(QIcon("ocean.png"));
  pluginMenuAction.setIconBig();

  pluginMenuGroupToGeology.addAction(pluginMenuAction);

  PluginMenuGroup techlogGroupWbi(GroupsIDs::wbi());

  techlogGroupWbi.insertGroup(pluginMenuGroupToGeology, 1.5);

  techlogTabGeology.addGroup(techlogGroupWbi);
  menu.addTab(techlogTabGeology);
}
```

The **`PluginMenuGroup`** is inserted as sub-group at the third index position of the **Wbi** group.

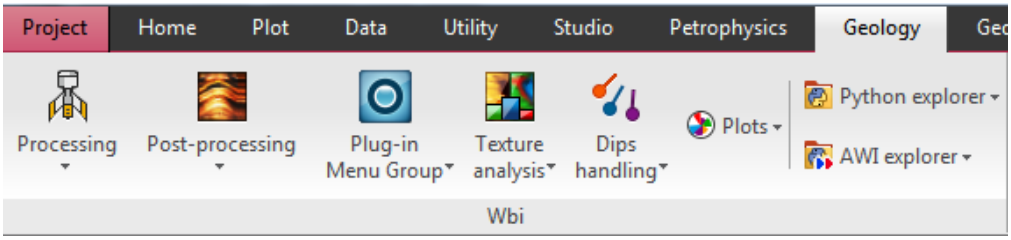


Figure 1-16 PluginMenuGroup (sub-group) inserted to native Techlog group

Use case: add / insert menu action items to a native Techlog group

This example instantiates a **`PluginMenuTab`** object with the **`geology`** native Techlog tab id and a **`PluginMenuGroup`** object is instantiated with the **`wbi`** native Techlog group id. Then a **`PluginMenuAction`** is added to the end of the **Wbi** group. A **`FileMenuAction`** and **`TechlogMenuAction`** are inserted in the **Wbi** native group at the index positions 0.5 and 0.7.

```
void MenuActionItemsPlugin::getMenu(PluginMenu& menu) const
{
  PluginMenuTab techlogTabGeology(TabsIDs::geology());

  PluginMenuGroup techlogGroupWbi(GroupsIDs::wbi());

  PluginMenuAction pluginAction(OceanActivityId);
  pluginAction.setText("Plug-in menu action item");
  pluginAction.setIcon(QIcon("ocean.png"));
  pluginAction.setIconBig();
  techlogGroupWbi.addAction(pluginAction);
```

```
  TechlogMenuAction techlogActionVSH2(
  TechlogMenuActivities::getShaleVolumeActivity());
  techlogGroupWbi.insertAction(techlogActionVSH2, 0.5);

  FileMenuAction fileActionPythonHelloWorld("HelloWorld",
  FileMenuActionFileTypePythonScriptSilentMode,
  StorageLevelPlugin);
  fileActionPythonHelloWorld.setText(
  "Hello world python script");
  fileActionPythonHelloWorld.setIconBig();
  techlogGroupWbi.insertAction(
  fileActionPythonHelloWorld, 0.7);

  techlogTabGeology.addGroup(techlogGroupWbi);
  menu.addTab(techlogTabGeology);
}
```

The different types of menu action items are inserted in the **Wbi** group.

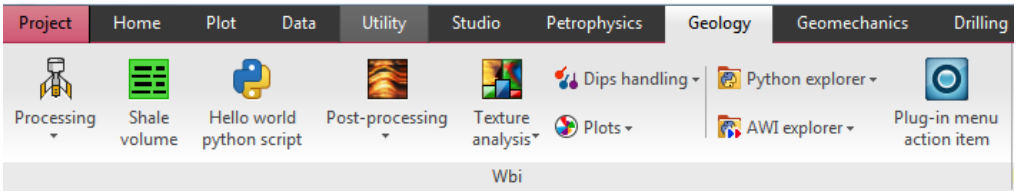


Figure 1-17 Menu action items inserted to native Techlog group

Use case: add / insert menu items to a native Techlog sub-group

Native Techlog sub-groups aren't provided in the **`GroupsIDs`** namespace. In order to get the Techlog sub-group id that allows you to instantiate the **`PluginMenuGroup`** object, contact the Schlumberger Techlog portfolio responsible for the domain where you want to insert your plug-in menu.

This example instantiates a **`PluginMenuGroup`** with the id of the "Post-processing" sub-group in the **Wbi** group. A **`PluginMenuAction`** is inserted in this sub-group.

```
void MenuActionItemsPlugin::getMenu(PluginMenu& menu) const
{
  PluginMenuTab techlogTabGeology(TabsIDs::geology());

  PluginMenuGroup techlogGroupWbi(GroupsIDs::wbi());

  PluginMenuGroup techlogGroupWbiPostProcessing(
  "wbi::postprocessingDyna");

  PluginMenuAction pluginMenuAction(OceanActivityId);
  pluginMenuAction.setText("Plug-in menu action item");
  pluginMenuAction.setIcon(QIcon("ocean.png"));
  pluginMenuAction.setIconBig();
```

```
  techlogGroupWbiPostProcessing.insertAction(pluginAction,
  0.5);
  techlogGroupWbi.addGroup(techlogGroupWbiPostProcessing);

  techlogTabGeology.addGroup(techlogGroupWbi);
  menu.addTab(techlogTabGeology);
}
```

The native Techlog sub-group has the **`PluginMenuAction`** inserted by the plug-in.

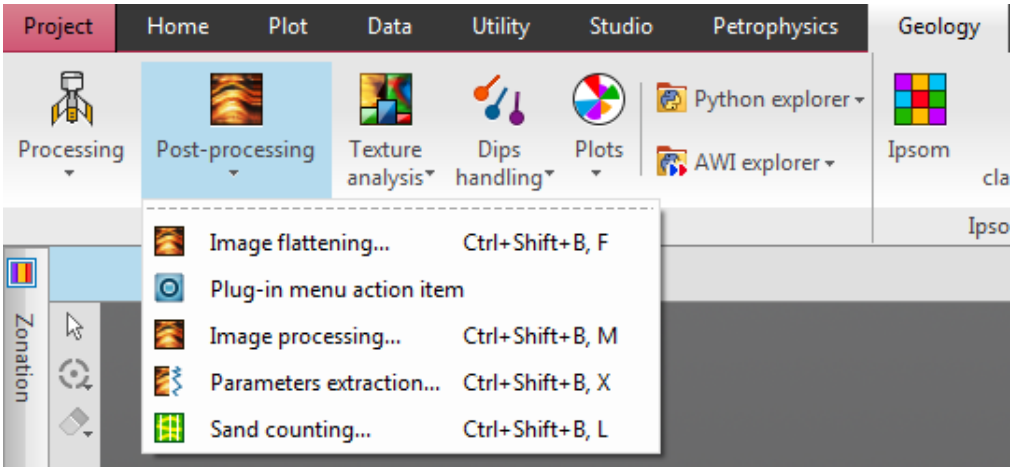


Figure 1-18 Menu action item inserted to native Techlog sub-group

Like other menu action items, you may also add or insert a **`PluginMenuGroup`** in a native Techlog sub-group.

There is no limitation in the number of nested groups that you add or insert in a native Techlog group or sub-group:

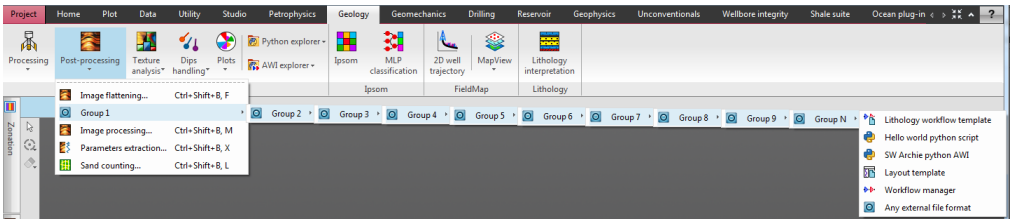


Figure 1-19 Nested groups in a native Techlog group

Plug-in menu action management with dependency on other plug-in

You may need to insert a menu action of your plug-in in a menu holds by another plug-in only if this plug-in is activated in Techlog or insert to your plug-in menu group a plug-in menu action holds by another plug-in. These two use cases are described below.

Use case: Conditional external plug-in menu action creation

The **`PluginMenuCondition`** constructor allows to build the **`PluginMenuCondition`** object on the availability of the plug-in B.

```
class PluginMenuCondition
{
public:
  PluginMenuCondition(const QString &vendorName, const QString
    &techlogVersion, const QString &pluginName, const QString
    &pluginVersion);

}
```

The **`PluginMenuCondition`** is passed to the **`PluginMenuAction`** of plug-in A through the **`setCondition`** function.

```
class PluginMenuAction : public MenuAction
{
public:
 …
 void setCondition(const PluginMenuCondition &condition);

}
```

The example below shows a plug-in A that needs to inject in a plug-in B menu an activity only if plug-in B is available. The plug-in A action menu item is removed if the plug-in B is desactivated in the module manager.

```
void PluginA::getMenu(PluginMenu& menu) const
{
  // Define B tab / group nodes where action from A needs to be
  inserted into B menu structure
  PluginMenuTab tab(QLatin1String("PluginBTab"));
  tab.setTitle(QLatin1String("B Tab"));

  PluginMenuGroup group(QLatin1String("PluginBGroup"));
  group.setTitle(QLatin1String("B Group"));

  //Construct condition object
  PluginMenuCondition BAvailability("Schlumberger", "2020.1",
  "Plugin B", "1.0");
  //PluginMenuCondition BAvailability("Schlumberger", "*.*", "Plugin
  B", "*.*"); //will not care about Techlog nor plugin versions

  //Create menu action and set condition
  PluginMenuAction actionA(activityAId);
  actionA.setCondition(BAvailability);
  actionA.setText("action from A");

  //As usual, build the structure to be returned
  group.addAction(actionA);
  tab.addGroup(group);
  menu.addTab(tab);
}
```

Use case: Conditional action menu item creation pointing to an external plugin activity

This use case is handled by a new **`PluginMenuAction`** construtor that is pointing on the **`activityId`** defined in another plug-in.

```
class PluginMenuAction : public MenuAction
{
public:
 …
 PluginMenuAction(const QString &activityId, const QString
     &vendorName, const QString &techlogVersion, const QString
     &pluginName, const QString &pluginVersion);

}
```

The example below shows a plug-in A that needs to add to its menu a plug-in menu action that triggers a plug-in B activity. The plug-in action is removed from the menu if the plug-in B is desactived in the module manager.

```
void PluginA::getMenu(PluginMenu& menu) const
{
  // Define A tab and group nodes where action from B needs to be
  inserted into A menu structure
  PluginMenuTab tab(QLatin1String("PluginATab"));
  tab.setTitle(QLatin1String("A Tab"));

  PluginMenuGroup group(QLatin1String("PluginAGroup"));
  group.setTitle(QLatin1String("A Group"));

  //Create menu action to reference external PluginB. It's
  availability will be handled internally
  PluginMenuAction action_fromB("activityBId", "Schlumberger",
  "2020.1", "PluginB", "1.0"));
  action_fromB.setText("action from B");

  //As usual, build the structure to be returned
  group.addAction(action_fromB);
  tab.addGroup(group);
  menu.addTab(tab);
}
```

2 Techlog convenience classes

In This Chapter

Introduction 2-3
Session class 2-3
Session domain object 2-4
Session signals 2-9
CurrentWorkspaceChanged signal 2-10
EnabledFeaturesChanged signal 2-11
ProjectBrowserFilterChanged signal 2-12
ProjectSaved signal 2-13
Modal and modeless dialogs 2-13
Project class 2-17
Project domain object 2-18
Project properties 2-21
Temporary project 2-23
Workspace class 2-24
Workspace domain object 2-25
Workspace signals 2-28
DepthInteractionChanged signal 2-29
SelectionChanged signal 2-30
PlotCreated signal 2-33
Selection domain object 2-34
Techlog output console 2-37
UnitConverter class 2-38
Unit conversion 2-38
ProjectBrowser class 2-39
ProjectBrowser 2-39
Techlog units management 2-41
Project unit system signal 2-41
Techlog measurements 2-42
Unit catalog 2-43
Unit object 2-44
Display unit 2-44

Plot axes limits and display parameters 2-44
Techlog families management 2-45
Techlog families 2-45
Family object 2-45
MainFamily object 2-45
Family catalog 2-45
Call Python script from an Ocean plug-in 2-48
Progress dialog with Ocean 2-51

Introduction

A new session is created at Techlog start-up time. Ocean developers have access to this **`Session`** through a singleton object destroyed when Techlog is closed. Using the properties and methods from the current **`Session`** object provides convenient access to commonly used classes.

The Techlog product family convenience classes are:

- **`Session`** – class associated with the current Techlog session.
- **`Project`** – class associated with the loaded projects.
- **`Workspace`** – entry point to the Techlog Workspace related to a plug-in.
- **`UnitConverter`** – access to methods for unit conversion.
- **`ProjectBrowser`** – class associated with Techlog project browser.

Session class

The **`Session`** class provides access to loaded projects and the plug-in workspace that are associated with the current Techlog session.

`Session` is defined as:

```
class Session : public DomainObject
{
public:
  static Session current();
  static QString pluginDirectory();

  void setCurrentWorkspace(Workspace workspace);
  Workspace currentWorkspace();

  DomainObjectCollection<Workspace> workspaces() const;

  Project mainProject();
  Project importProject();
  Project exportProject();

  Project project(ProjectType projectType);

  static ReturnValue<bool> trySaveProject();

  void openImportBuffer();
  void closeImportBuffer();
  bool isImportBufferOpen() const;
  void openExportBuffer();
  void closeExportBuffer();
  bool isExportBufferOpen() const;
```

```
  QStringList commandLine() const;

  ProjectBrowser projectBrowser();
  ReportEditor findCurrentReportEditor();

  FamilyCatalog familyCatalog() const;
  UnitCatalog unitCatalog() const;

  CoreLicense getCoreLicense() const;
  bool isFeatureActivated(Feature feature) const;

  WorkspaceViewMode currentWorkspaceViewMode() const;
  void setCurrentWorkspaceViewMode(WorkspaceViewMode viewMode);

  void loadWorkspaceTemplate(WorkspaceTemplate
  workspaceTemplate);

  QString pluginFolder() const;
  QString projectFolder() const;
  QString userFolder() const;
  QString companyFolder() const;
  QString techlogFolder() const;

  QString assignmentRulesCatalog() const;
  QString techlogAssignmentRulesCatalog() const;
  QString osddAssignmentRulesCatalog() const;

};
```

Session domain object

The static **`Session::current`** method retrieves the current session open in Techlog. The domain object **`Session`** represents the Techlog session. The **`Session`** class is the main entry point of an Ocean plug-in. The **`Session`** class allows you to:

- retrieve the current session created when Techlog starts
- retrieve the absolute path to the plug-in dll
- get and set the current workspace in the Techlog session
- get the collection of workspaces opened into the Techlog session
- get the main project of the current session
- get the import project of the current session top level
- get the export project of the current session
- save the project open in the current session
- open and close import and export buffers in Techlog
  - throw an exception trying to close a buffer which is not open in Techlog

  - check if buffers are open using the **`isImportBufferOpen`** and **`isExportBufferOpen`** functions.

- get a **`Project`** domain object corresponding to the given **`ProjectType`** enum value. **`ProjectType`** enum class has the following values:

```
enum ProjectType
{
 ProjectTypeMain,
 ProjectTypeExport,
 ProjectTypeImport,
 ProjectTypeTemporary
};
```

- get the command line passed to the main Techlog process. Each entry in the list corresponds to one item in the "argv" list passed to Techlog's main process.
- retrieve the **`ProjectBrowser`** instance that allows you to manipulate objects in the project browser of Techlog to do group management
- return the current **`ReportEditor`**; if it does not exist then null is returned
- instantiate the **`UnitCatalog`** that allows access to Techlog units through the Techlog unit system management. See "Techlog units management" on page 2-41 for more information on how to manipulate Techlog units with Ocean.
- instantiate the **`FamilyCatalog`** that allows access to Techlog families through the Techlog family management. See "Techlog families management" on page 2-45 for more information on how to manipulate Techlog families with Ocean.
- get the core license activated by the end-user at Techlog start. Core licenses are exposed through **`CoreLicense`** enum values.

```
enum CoreLicense
{
    TechlogCore,
    TvCore,
    OceanCore
};
```

The current core license can't be changed programmatically or manually during the Techlog session. Techlog must be restarted.

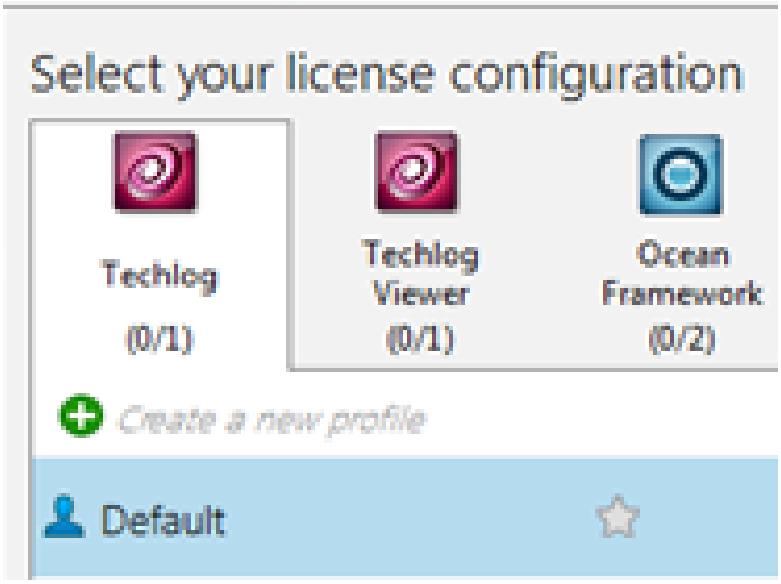


Figure 2-1 Core license selection in Techlog module manager

- check if a Techlog license feature is available. The **`Feature`** enum class gives all Techlog feature names.
- get and set the workspace display mode using **`WorkspaceViewMode`** enum values. A display mode is applied to all workspaces open in the Techlog session.

```
enum WorkspaceViewMode
{
    WorkspaceViewModeDashboard,
    WorkspaceViewModeFloating
};
```

- Load a dashboard template in the Techlog current workspace. In Techlog the end user saves the size and position of the viewers in the current workspace as a template (**Home > Window > Save as template**). That information is saved in an XML file in a folder named "DashboardTemplate" at a Techlog storage level (User, Company, Project or Techlog). The **`loadWorkspaceTemplate`** function allows you to apply a dashboard template to the current Techlog workspace using the **`WorkspaceTemplate`** class to check the existence and retrieve the template at one storage level (Ocean adds the plug-in storage level). Note that the function throws an exception if the current workspace isn't set to the dashboard view mode.
- get absolute paths of the plug-in, user, company, project and Techlog folders

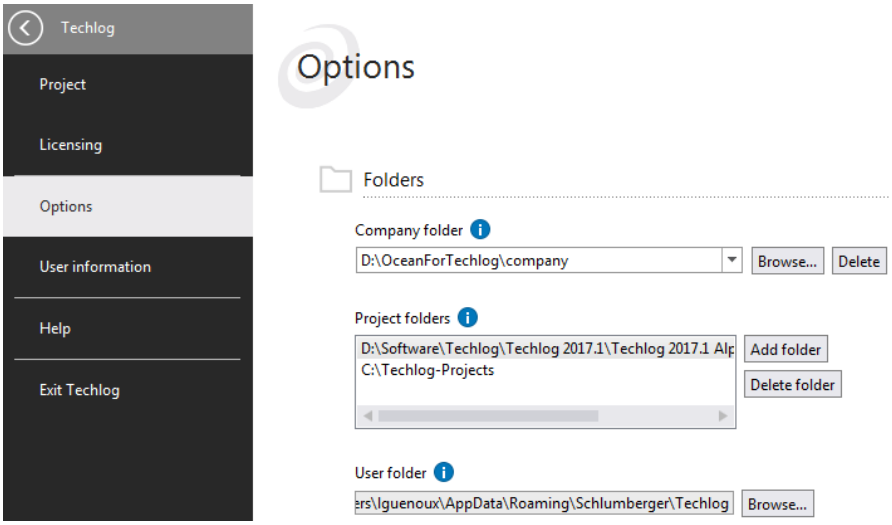


Figure 2-2 Techlog folders

- the **`assignmentRulesCalatalog`** fuction returns the name of the current catalog selected in the Techlog project settings
- **`techlogAssignmentRulesCalatalog`** and **`osddAssignmentRulesCalatalog`** functions returns the names of the catalogs available today in Techlog as "Techlog" and "Schlumberger-OSDD" catalogs

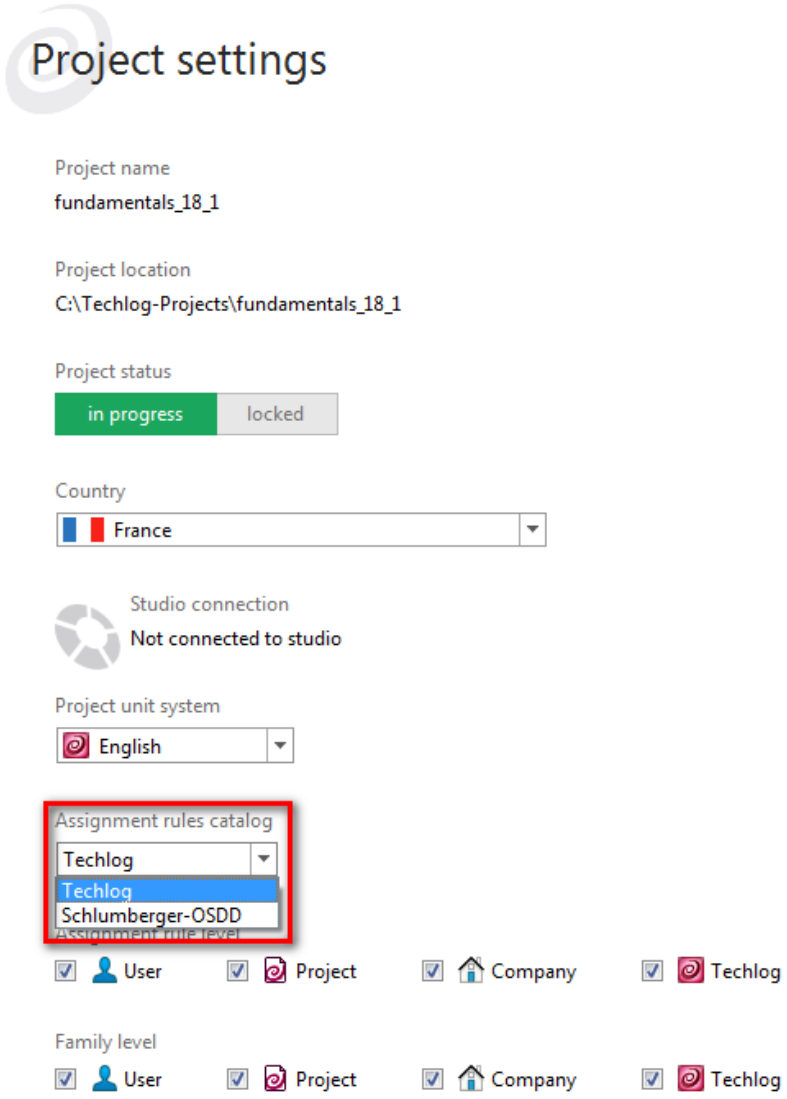


Figure 2-3 Techlog assignement rules catalog

This example shows retrieving workspace and project instances from the **`Session`** and applies a dashboard template to the current workspace saved at the plug-in storage level:

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

// Get the current session
Session session = Session::current();
// gets the current workspace with which the plug-in activity has
been started
Workspace workspace = session.currentWorkspace();
// Get the main project of the current session
Project mainProject = session.mainProject();
// Get the import project of the current session
Project importProject = session.importProject();
// Get the export project of the current session
```

```
Project exportProject = session.exportProject();

// apply a dashboard template to the Techlog current workspace
// the template is stored at the plug-in level
if ((session.currentWorkspaceViewMode() ==
WorkspaceViewModeDashboard)
&& (WorkspaceTemplate::exists(StorageLevelUser,
"MyWorkspaceTemplate")))
{
  WorkspaceTemplate myWorkspaceTemplate =
  WorkspaceTemplate::get(StorageLevelUser,
  "MyWorkspaceTemplate");

   session.loadWorkspaceTemplate(myWorkspaceTemplate);
}

lock.release();
```

Session signals

The **`Session`** domain object enables you to listen for current Techlog session events when you subscribe to its signals.

```
class Session : public DomainObject
{
public:
  ...
  enum EventType
  {
    CurrentWorkspaceChanged,
    EnabledFeaturesChanged,
    ProjectBrowserFilterChanged,
    ProjectSaved
  };
}
```

The signals are emitted whenever:

- **`CurrentWorkspaceChanged`** – the current Techlog workspace is changed by the Techlog end-user in the Techlog workspace window. The current Techlog workspace can't be changed programmatically.
- **`EnabledFeaturesChanged`** – Techlog features enabled in the module manager have changed.
- **`ProjectBrowserFilterChanged`** – The advanced filter in the Techlog project browser has been changed.
- **`ProjectSaved`** – The project is saved by the end-user or programmatically through the **`trySaveProject`**.

This example shows how to include signal arguments and declare slot receivers in the activity header file.

```
#include "tsdkcurrentworkspacechangedargs.h"
#include "tsdkenabledfeatureschangedargs.h"
#include "tsdkprojectbrowserfilterchangedargs.h"
#include "tsdkprojectsavedargs.h"
```

```
private slots:
  void onCurrentWorkspaceChanged(
  const Slb::Ocean::Techlog::CurrentWorkspaceChangedArgs
&args);
  void onEnabledFeaturesChanged(const
  Slb::Ocean::Techlog::EnabledFeaturesChangedArgs &args);
  void onProjectBrowserFilterChanged(const
  Slb::Ocean::Techlog::ProjectBrowserFilterChangedArgs &args);
  void onProjectSaved(const
  Slb::Ocean::Techlog::ProjectSavedArgs &args);
```

This example shows how to connect the **`Session`** domain object to the signals.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Session session = Session::current();

session.connect(Session::CurrentWorkspaceChanged, this,
SLOT(onCurrentWorkspaceChanged(const
Slb::Ocean::Techlog::CurrentWorkspaceChangedArgs&)));
session.connect(Session::EnabledFeaturesChanged, this,
SLOT(onEnabledFeaturesChanged(const
Slb::Ocean::Techlog::EnabledFeaturesChangedArgs&)));
session.connect(Session::ProjectBrowserFilterChanged, this,
SLOT(onProjectBrowserFilterChanged(const
Slb::Ocean::Techlog::ProjectBrowserFilterChangedArgs&)));
session.connect(Session::ProjectSaved, this,
SLOT(onProjectSaved(const
Slb::Ocean::Techlog::ProjectSavedArgs&)));

lock.release();
```

CurrentWorkspaceChanged signal

The **`CurrentWorkspaceChanged`** signal has a **`CurrentWorkspaceChangedArgs`** argument.

```
class CurrentWorkspaceChangedArgs :
      public SignalArgsT<Session>
{
```

```
};
```

The slot handler accesses the needed information from the signal argument.

```
void activity::onCurrentWorkspaceChanged(const
Slb::Ocean::Techlog::CurrentWorkspaceChangedArgs &args)
{
  Session session = args.sender();

  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, session);

  qWarning() << "New current workspace is "
  << session.currentWorkspace().name();

  lock.release();
}
```

EnabledFeaturesChanged signal

The **EnabledFeaturesChanged** signal has an **EnabledFeaturesChangedArgs** argument.

```
class EnabledFeaturesChangedArgs :
      public SignalArgsT<Session>
{
};
```

A possible use case would be to warn the plug-in user that some plug-in functionalities are now disabled because a Techlog feature is disabled; do this in the slot receiver. For example, a 2D well trajectory created by a plug-in activity is no longer available because the Techlog Field Map feature is disabled.

```
void activity::onEnabledFeaturesChanged(const
Slb::Ocean::Techlog::EnabledFeaturesChangedArgs &args)
{
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

  Session session = Session::current();

  if (!session.isFeatureActivated(Feature::FieldMapFeature))
    qWarning() << "Field Map feature is disabled you can't display
anymore a 2D well trajectory plot through this Ocean plug-in
activity";

  lock.release();
}
```

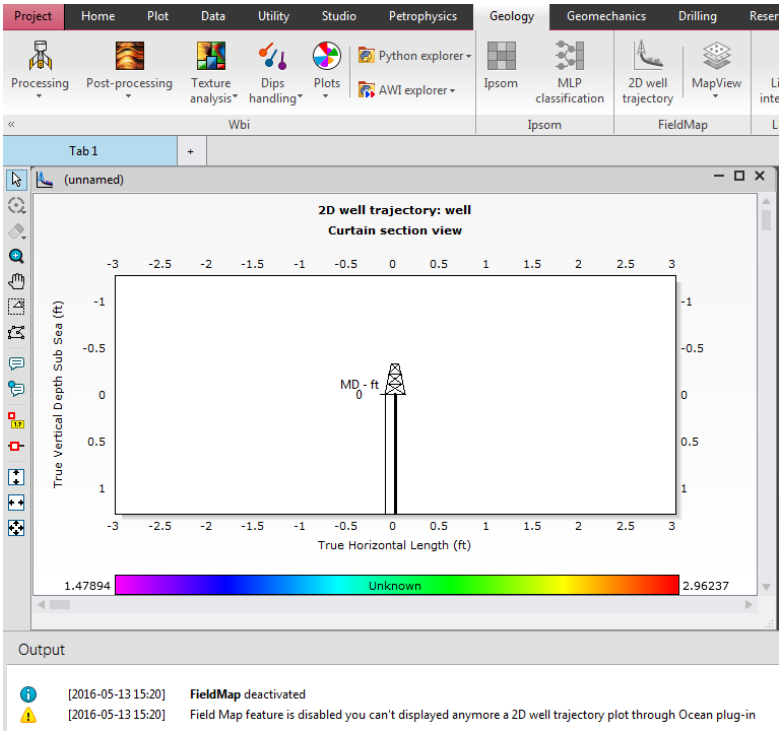


Figure 2-16 Field map feature disabled

ProjectBrowserFilterChanged signal

The **`ProjectBrowserFilterChanged`** signal has a **`ProjectBrowserFilterChangedArgs`** argument.

```
class ProjectBrowserFilterChangedArgs :
      public SignalArgsT<Session>
{
};
```

If a new filter is created into the Techlog project browser this signal is emitted and you can for instance get the new list of filtered wells in the slot receiver as shown in the sample code below:

```
void activity::onProjectBrowserFilterChanged(const
Slb::Ocean::Techlog::ProjectBrowserFilterChangedArgs &args)
{
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

  Project project = Session::current().mainProject();

  qWarning() << "List of new filtered wells:";
  foreach(Well well, project.filteredWells())
  {
    qWarning() << well.name();
  }
```

```
  lock.release();
}
```

ProjectSaved signal

The **`ProjectSaved`** signal includes an argument that gives the **`Session`** instance the slot was connected to through the sender method inherited from the **`SignalArgs`** base class. This argument is modeled in Ocean for Techlog through the **`ProjectSavedArgs`** class.

```
class ProjectSavedArgs :
      public SignalArgsT<Session>
{
};
```

Modal and modeless dialogs

Use **`Session`** class static functions to show Techlog modal dialog windows to the user, like question, information, warning, critical, open file and save file dialogs.

```
class Session : public DomainObject
{
public:
  static int questionDialog(const QString &caption, const QString
    &message, const QStringList &buttons, const int
    defaultButton, const int escapeButton);
  static void informationDialog(const QString &caption, const
    QString &message);
  static void warningDialog(const QString &caption, const QString
    &message);
  static void criticalDialog(const QString &caption, const
    QString &message);
  static QString openFileDialog(const QString &caption, const
    QString &dir, const QString &filter);
  static QString saveFileDialog(const QString &caption, const
    QString &dir, const QString &filter);
  ...
};
```

This is an example:

```
QString result = Session::openFileDialog("Open File Dialog",
Session::pluginDirectory(), "Binary (*.exe *.dll)");

if (!result.isEmpty())
{
  QString message = QString("You just chose to open
file<br>%1").arg(result);
```

```
  QStringList buttons;
  buttons << "Information" << "Warning" << "Critical";
  int id = Session::questionDialog("My question dialog", "Select
a type of message.", buttons, 0, 0);

  switch (id)
  {
  case 0:
    Session::informationDialog("Infomation dialog", message);
    break;
  case 1:
    Session::warningDialog("Warning dialog", message);
    break;
  case 2:
    Session::criticalDialog("Critical dialog", message);
    break;
  }
}
```

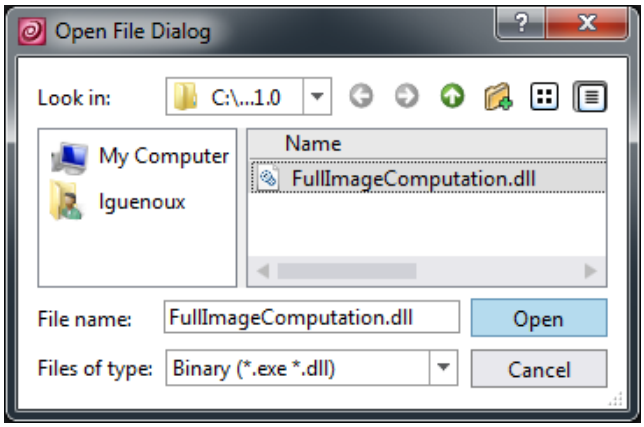


Figure 2-5 Open file dialog

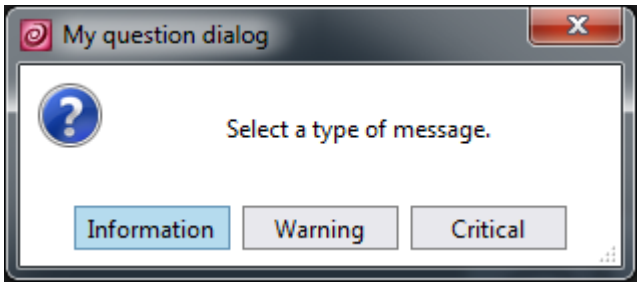


Figure 2-6 Question dialog

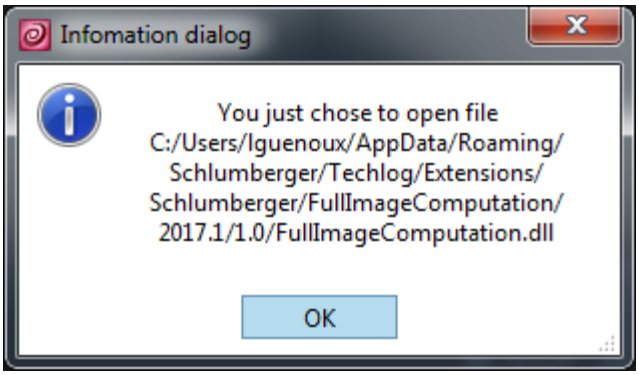


Figure 2-7 Information dialog

Use the **`Session`** class **`addCustomDialog`** static function to show a Techlog custom modal or modeless dialog window to the plug-in end-user.

```
class Session : public DomainObject
{
public:
  static int addCustomDialog(QDialog *customDialog, const
    CustomDialogModality modality, const CustomDialogButtons
    buttons, const CustomDialogResizeability resizeability,
    const CustomDialogDeleteOnClose deleteOnClose);
  ...
};
```

This example shows a custom modal dialog opened from the Techlog main window and a custom modeless dialog opened from the custom modal dialog.

```
void activity::run()
{
  QVBoxLayout *layout = new QVBoxLayout;
  QPushButton *closeThisCustomDialogButton = new
  QPushButton(QString::fromLatin1("Close this Custom Dialog"));
  QPushButton *openModelessCustomDialogButton = new
  QPushButton(QString::fromLatin1("Open nested Modeless Custom
Dialog"));
  QString imagePath = Session::pluginDirectory() +
  "/Ocean_big.png";
  QPixmap pixmap(imagePath);
  QLabel *imageLabel = new QLabel();
  imageLabel->setPixmap(pixmap);

  layout->addWidget(imageLabel);
  layout->addWidget(closeThisCustomDialogButton);
  layout->addWidget(openModelessCustomDialogButton);

  QDialog *mother = new QDialog();
  mother->setWindowTitle(QLatin1String("Modal Custom Dialog"));
  mother->setLayout(layout);
```

```
  mother->adjustSize();

  QObject::connect(openModelessCustomDialogButton,
  &QPushButton::clicked, this,
  &activity::onOpenModelessCustomDialog);
  QObject::connect(closeThisCustomDialogButton,
  &QPushButton::clicked, mother, &QWidget::close);

  Session::addCustomDialog(mother, CustomDialogModalityModal,
  CustomDialogButtonClose, CustomDialogResizeabilityResizeable,
  CustomDialogDeleteOnCloseYes);
}

void activity::onOpenModelessCustomDialog()
{
  QVBoxLayout *layout = new QVBoxLayout;
  QPushButton *closeThisCustomDialogButton = new
  QPushButton(QString::fromLatin1("Close this Custom Dialog"));
  QString imagePath = Session::pluginDirectory() +
  "/Ocean_big.png";
  QPixmap pixmap(imagePath);
  QLabel *imageLabel = new QLabel();
  imageLabel->setPixmap(pixmap);

  layout->addWidget(imageLabel);
  layout->addWidget(closeThisCustomDialogButton);

  QDialog *mother = new QDialog();
  mother->setWindowTitle(QLatin1String("Modeless Custom
Dialog"));
  mother->setLayout(layout);
  mother->adjustSize();

  QObject::connect(closeThisCustomDialogButton,
  &QPushButton::clicked, mother, &QWidget::close);

  Session::addCustomDialog(mother,
  CustomDialogModalityNonModal, CustomDialogButtonClose,
  CustomDialogResizeabilityResizeable,
  CustomDialogDeleteOnCloseYes);
}
```

Figure 2-8 Custom modal and modeless dialogs

Project class

The **`Project`** class provides access to Techlog data. Access the main project opened in the current Techlog session and visible through the project browser using the method **`Session::mainProject()`**. Use the project to access wells, plug-in domain objects and the Techlog zonation model.

See the "Plug-in domain object" section in *Ocean for Techlog Developer Guide - Plugin Domain Object - Importer&Exporter* for more information on **`PluginDomainObject`** class.

`Project` is defined as:

```
class Project : public DomainObject
{
public:
  const QString unitSystem() const;
  QString getUniqueWellName(const QString & baseWellName);

  DomainObjectCollection< Well> wells() const;
  Well findWell(const QString &wellName) const;
  Well getWell(const QString &wellName) const;

  DomainObjectCollection<Well> filteredWells() const;
  Well findCurrentWell() const;


  DomainObjectCollection <ContactGroup> contactGroups() const;
  ContactGroup findContactGroup(const QString &contactGroupName)
    const;
```

```
  ContactGroup getContactGroup(const QString &contactGroupName)
    const;

  DomainObjectCollection <PluginDomainObject>
    pluginDomainObjects() const;
  PluginDomainObject findPluginDomainObject(
    const QString &pluginDomainObjectName) const;
  PluginDomainObject getPluginDomainObject(
    const QString &pluginDomainObjectName) const;

  ProjectZonationModel zonationModel() const;
  ProjectMarkerModel markerModel() const;

};
```

Project domain object

The **`Project`** domain object represents the project. It can be the main, export, import or temporary project. The **`Project`** domain object is the root object of the Techlog project browser.

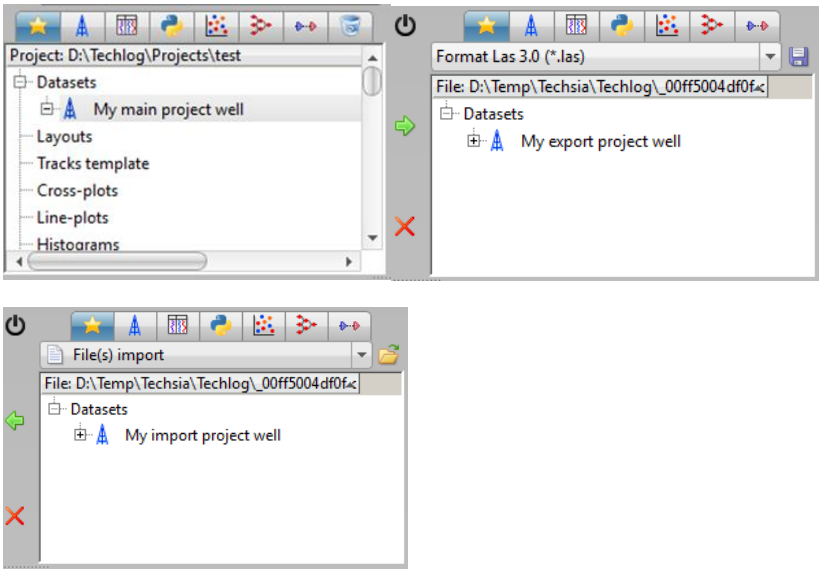


Figure 2-9 Main, export and import Techlog project browsers

The **`Project`** class allows you to:

- Get the unit system currently set for the project. The current unit system can only be changed by the Techlog end-user. See "Techlog units management" on page 2-41 for more information on Techlog unit system.
- Get a unique well name. If the requested base well name is already used by another well, the method appends a number (1, 2, etc.) to make it unique.
- Get the well collection of the project. See the "Well domain object" section in *Ocean for Techlog Developer Guide – Data&Workflow* for more information on how to access well data with Ocean.

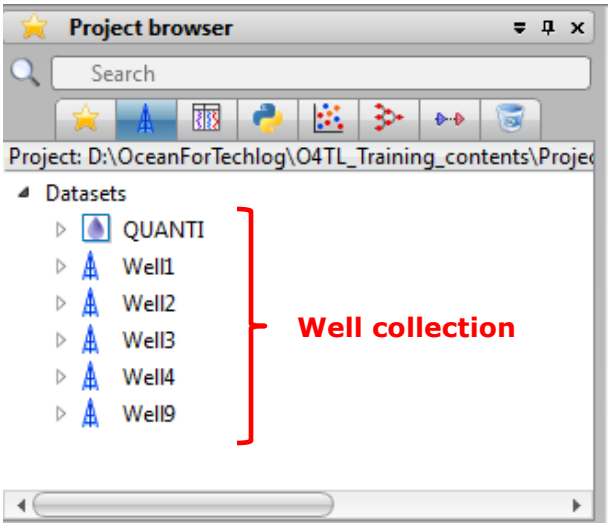


Figure 2-10 Well collection in main Techlog project

- Retrieve a child well by its name using the **`findWell`** and **`getWell`** functions. Find returns a null domain object and get throws an exception if the well does not exist in the collection. See Domain Objects "Find and get patterns" on page 3-5 for more information on Domain Object common patterns.
- Get the filtered wells in the project browser by the Techlog end-user.
- Find the selected well in the project browser through the **`findCurrentWell`** function. In case of a context where no project browser can define the current well, this function returns a null **`Well`** object.
- Get access to a list of **`ContactGroup`** objects (listed for the **`Session::mainProject()`** in the Techlog contacts dock window).
- Retrieve a **`ContactGroup`** by its name through **`findContactGroup`** and **`getContactGroup`** functions.
- Get the collection of plug-in domain objects. See the "Plug-in domain object" section in *Ocean for Techlog Developer Guide - Plugin Domain Object - Importer&Exporter* for more information on how to create a **`PluginDomainObject`**.
- Retrieve a child plug-in domain object by its name using **`findPluginDomainObject`** and **`getPluginDomainObject`** functions.
- Get a list of **`GlobalZonation`** objects (listed for the **`Session::mainProject()`** in the Techlog markers dock window) using the **`ProjectMarkerModel`** class. See the "Zonation creation" section in *Ocean for Techlog Developer Guide – Data & Workflow* for more information on how to create and edit zonations.

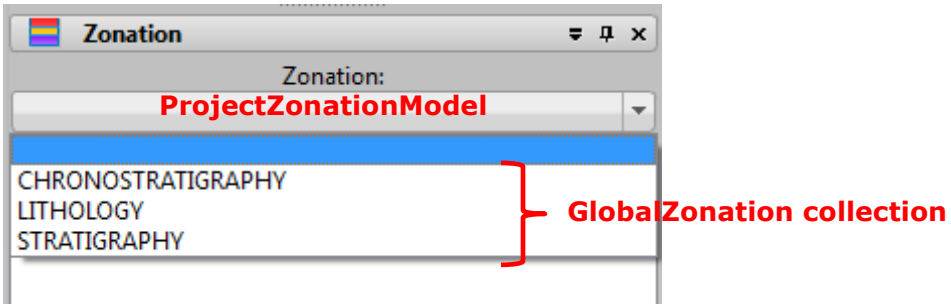


Figure 2-11 GlobalZonation collection from Project zonation model

- Get access to a list of **`GlobalMarkerSet`** objects (listed for the **`Session::mainProject()`** in the Techlog marker dock window) using the **`ProjectMarkerModel`** class. See the "Marker creation" section in *Ocean for Techlog Developer Guide – Data & Workflow* for more information on how to create and edit markers with Ocean.

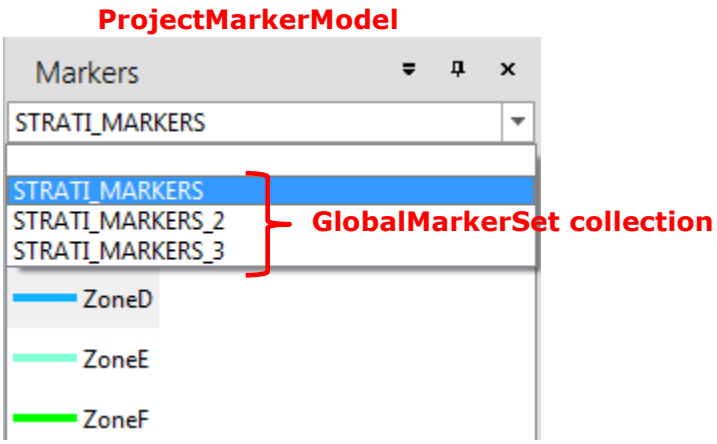


Figure 2-12 GlobalMarkerSet collection from Project marker model

This example uses the **`Project`**.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
// Get the current session
Session session = Session::current();
// Get the main project of the current session
Project mainProject = session.mainProject();
// Iterate on all the wells belonging to the main project
foreach(Well well, mainProject.wells())
{
  qWarning() << well.name();
}
// Iterate on all the groups of contacts belonging to the main
project
foreach(ContactGroup contactGroup, mainProject.contactGroups())
{
  qWarning() << contactGroup.name();
```

```
}
// Iterate on all the zonation belonging to the main project
foreach(GlobalZonation zonation,
mainProject.zonationModel().globalZonations())
{
  qWarning() << zonation.name();
}
// Iterate on all the markerset belonging to the main project
foreach(GlobalMarkerSet markerSet,
mainProject.markerModel().globalMarkerSets())
{
  qWarning() << markerSet.name();
}

lock.release();
```

Project properties

The properties of the project as name, description, country and history are accessible through getters and setters of the **`Project`** domain object.

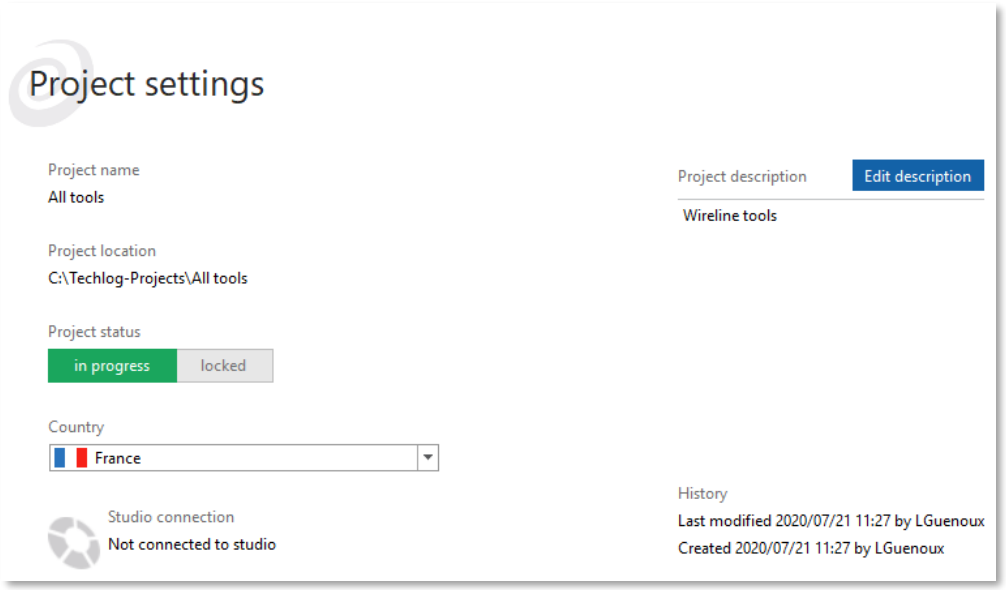


Figure 2-13 Project setting properties

```
class Project : public DomainObject
{
public:
  Country country() const;
  QDateTime creationDate() const;
  QString creator() const;
  QString description() const;
  QDateTime lastModificationDate() const;
```

```
  QString techlogUniqueIdentifier() const;
  QString lastModifier() const;
  QString lastTechlogVersion() const
  QString name() const;

  void setCountry(const Country &country);
  void setDescription(const QString &description);
  void setName(const QString &name);
  ...
};
```

Example:

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Project project = Session::current().mainProject();

qWarning() << QString("Project name: %1").arg(project.name());
qWarning() << QString("Project description:
%1").arg(project.description());

qWarning() << QString("Country:
%1").arg(project.country().name());
qWarning() << QString("History: Last modified %1 by
%2").arg(project.lastModificationDate().toString("yyyy-MM-dd
HH:mm")).arg(project.lastModifier());
qWarning() << QString("-------- Created %1 by
%2").arg(project.creationDate().toString("yyyy-MM-dd
HH:mm")).arg(project.creator());

lock.release();
```

Output:

[2020-07-21 17:28] "Project name: All tools"

[2020-07-21 17:28] "Project description: Wireline tools"

[2020-07-21 17:28] "Country: France"

[2020-07-21 17:28] "History: Last modified 2020-07-21 11:27 by LGuenoux"

[2020-07-21 17:28] "-------- Created 2020-07-21 11:27 by LGuenoux"

Note: Accessing properties from import or export projects return empty objects.

Temporary project

A main, import or export project can't be created programmatically. The **`createTemporary`** function allows you to create a temporary project. This temporary project is erased when the plug-in activity that created it is stopped.

```
class Project : public DomainObject
{
public:
  static Project createTemporary();
  ...
};
```

All the well data that are created in the temporary project aren't displayed in the project browser by Techlog, but they are accessible anytime through Ocean API during the life of the plug-in.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Project project = Project::createTemporary();

Well::create("MyWell", project);

lock.release();
```

Retrieve the temporary project from the **`DomainObjectCollection`** returned by the **`temporaryProjects`** function of the **`Session`** class.

```
class Session : public DomainObject
{
public:
  const DomainObjectCollection <Project> temporaryProjects()
    const;
  ...
};
```

Only a temporary project can be erased. The **`canErase`** function from the **`DomainObject`** base class returns true.

See the "Erase pattern" section on page 3-7 for more information on **`DomainObject canErase`** common pattern.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

if (Session::current().temporaryProjects().count() != 0)
{
  Project project =
  Session::current().temporaryProjects().first();
  if (project.canErase())
```

```
  {
    project.erase();
    qWarning() << "Temporary project erased";
  }
}
else
  qWarning() << "No temporary project";

lock.release();
```

Workspace class

The **`Workspace`** class provides access to all the windows displayed in a Techlog workspace. The windows are either Techlog native plots and workflows (Techlog AWI = Application Workflow Interface) or custom windows (QWidget). This class also gives access to signals and objects which allow interaction between those widgets.

```
class Workspace : public DomainObject
{
public:
  static Workspace create(const QString &name, Session session);

  static void logEvent( const LogLevel severity,
    const QString & msg, const QString & senderName =
    QString::null);

  enum EventType
  {
   DepthInteractionChanged,
   SelectionChanged,
   PlotCreated
  };

  void addCustomWidget(QWidget* customWidget);
  void closeCustomWidget(QWidget* customWidget);
  void setCustomWidgetHelpId(QWidget *customWidget,
    QString guid);

  const bool isCurrent() const;

  const DomainObjectCollection<Plot> plots() const;
  Plot findActivePlot() const;

  const DomainObjectCollection<Selection> selections() const;
```

```
  const Selection currentSelection(const Dataset& dataset)
    const;

  const DomainObjectCollection<Workflow> workflows() const;
  Workflow findWorkflow(const QString &workflowName) const
  Workflow getWorkflow(const QString &workflowName) const;
  const Workflow findSelectedWorkflow() const;

  DipClassification currentDipClassification() const;


  void closeWorkflowManagerIfOpen();

  …
};
```

Workspace domain object

The **`Workspace`** domain object represents a Techlog workspace. In Techlog there may be several workspaces under the **Workspaces** node of the Techlog workspace dialog, available using the **`Session::workspaces()`** function. You change the current workspace with the **`Session::setCurrentWorkspace()`** function. **`Session::currentWorkspace()`** returns the selected workspace in Techlog.

Note: You can only create plots, workflows and custom widgets in the current Techlog workspace; otherwise it throws an exception. **`Workspace::isCurrent()`** property returns false if the **`Workspace`** object is not the current workspace in Techlog.

Create a **`Workspace`** using the static **`create`** function; you provide a unique name across all existing workspaces. If you want to set the created workspace as the current one in the Techlog **`Session`**, call the **`Session::setCurrentWorkspace`** function.

The **`Workspace`** class allows you to:

- log a message in the Techlog output console with a specified severity level.
- add/remove a custom widget to/from the current workspace of the activity providing a pointer to a QWidget object.
- link a custom widget to some Techlog help content deployed at the plug-in storage level (plug-in folder), using the **`setCustomWidgetHelpId`** function. See the "How to add plug-in documentation to Techlog Help Center" tutorial in the **OceanForTechlog.chm** file.
- retrieve the plot collection belonging to this workspace.
- return the **`Plot`** in this **`Workspace`** with the focus. The **`Plot`** returned may be null if the focus is set to a non **`Plot`** object or a plot is not exposed via Ocean.
- retrieve the selection collection belonging to this workspace.
- get the selection belonging to this workspace and a dataset.
- retrieve the workflow collection belonging to this workspace.
- retrieve a child workflow by its name using the **`findWell`** and **`getWell`** functions. Find returns a null domain object and get throws an excepton if the workflow

doesn't exist in the collection. See Domain Objects "Find and get patterns" on page 3-5 for more information on Domain Object common patterns.

- retrieve the selected workflow in this workspace.
- retrieve the current dip classification selected in this workspace.
- The **`closeWorkflowManagerIfOpen`** function closes the workflow manager (AWI) if it is open. Otherwise it does nothing.

See "Workflow Domain Object" in *Ocean for Techlog Developer Guide – Data & Workflow* for more information on workflow.

See "DipClassification object" in *Ocean for Techlog Developer Guide – Geology* for more information on how to handle dip classification selected type with Ocean.

The plot collection returned by the **`Workspace::plots`** method only lists the Techlog plots available from Ocean for Techlog framework. Workflow manager (AWI) and custom widgets are not listed in this collection.

This example shows the use of the **`Workspace`**.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

// Get the current workspace from the current session
Workspace workspace = Session::current().currentWorkspace();
// Enumerate plots from the current workspace
foreach (const Plot &plot, workspace.plots())
{
  workspace.logEvent(LogLevelInformation,
  QString("Plot title = %1").arg((plot.windowTitle())));
}

lock.release();
```

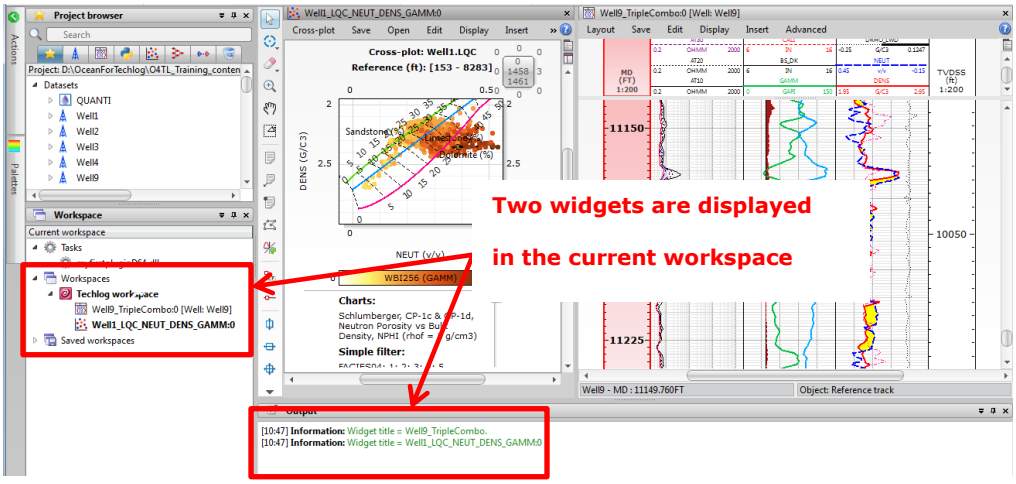


Figure 2-14 Plot collection in the current Techlog workspace

In the Techlog workflow manager (AWI) you can have several workflows. The **`Workspace::workflows`** method lists all those workflows. The

`Workspace::findSelectedWorkflow` method returns the selected workflow in the Techlog AWI.

If there is no workflow in the AWI, this method returns a null object. In this case the next step is to create a new workflow that becomes the selected workflow.

See "Workflow and worksteps" in *Ocean for Techlog Developer Guide – Data & Workflow* for more information on Workflow.

This example shows creating a new workflow.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();

// Enumerate workflows belonging to the current workspace (listed
in the AWI)
foreach(Workflow workflow, workspace.workflows())
{
  workspace.logEvent(LogLevelInformation,
  workflow.name());
}

// Returns the selected workflow in the AWI
Workflow selectedWorkflow = workspace.findSelectedWorkflow();
if (!selectedWorkflow.isNull())
  workspace.logEvent(LogLevelInformation,
  QString("Selected workflow = %1")
  .arg(selectedWorkflow.name()));

lock.release();
```

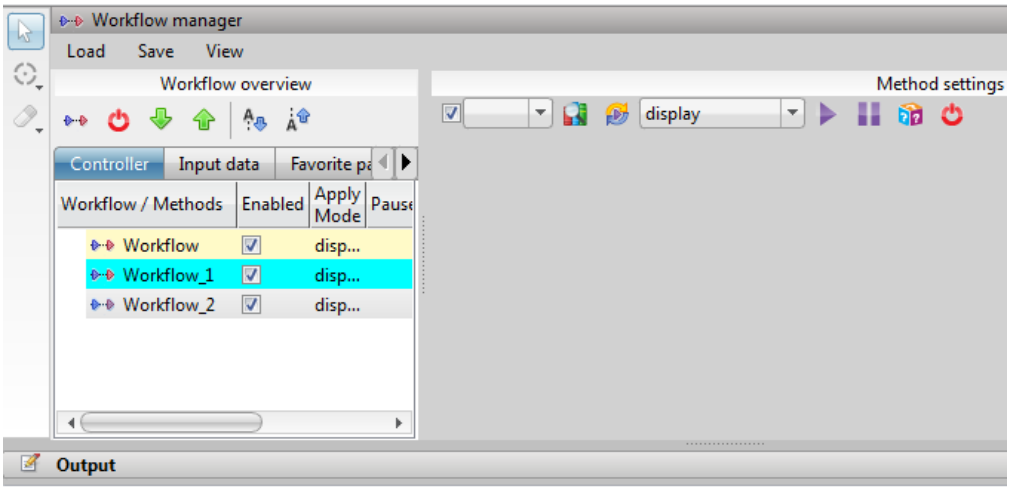


Figure 2-15 Workflow collection and selected workflow in the AWI

Workspace signals

The **Workspace** domain object enables you to listen for current workspace events when you subscribe to its signals.

```
class Workspace : public DomainObject
{
public:
...
  enum EventType
  {
   DepthInteractionChanged,
   SelectionChanged,
   PlotCreated
  };
}
```

The signals are emitted whenever:

- **DepthInteractionChanged** – a depth interaction occurs between plots opened in the workspace
  - Depth interaction is a tool that allows the end user to display and move a depth line or a window along a layout. The line or window then changes accordingly and the tool automatically updates the display of the connected plot/tools that are listening to the depth information sent by **LogView**.
- **SelectionChanged** – dataset indexes selected by Techlog interactive selection tools have changed
- **PlotCreated** – a new plot is created in the workspace

Include signal arguments and declare slot receivers in the activity header file:

```
#include "tsdkdepthinteractionchangedargs.h"
#include "tsdkselectionchangedargs.h"
#include "tsdkplotcreatedargs.h"
```

```
private slots:
  void onDepthInteractionChanged(
  const Slb::Ocean::Techlog::DepthInteractionChangedArgs&
args);
  void onSelectionChanged(
  const Slb::Ocean::Techlog::SelectionChangedArgs& args);
  void onPlotCreated(const Slb::Ocean::Techlog::PlotCreatedArgs
&args);
```

This example shows how to connect the **Workspace** domain object to those signals.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
```

```
Workspace workspace = Session::current().currentWorkspace();


workspace.connect(Workspace::DepthInteractionChanged, this,
SLOT(onDepthInteractionChanged(const
Slb::Ocean::Techlog::DepthInteractionChangedArgs&)));
workspace.connect(Workspace::SelectionChanged, this,
SLOT(onSelectionChanged(const
Slb::Ocean::Techlog::SelectionChangedArgs&)));
workspace.connect(Workspace::PlotCreated, this,
SLOT(onPlotCreated(const
Slb::Ocean::Techlog::PlotCreatedArgs&)));


lock.release();
```

DepthInteractionChanged signal

The **`DepthInteractionChanged`** signal has an argument that gives the depth interaction line value and depth interaction window top and bottom values. This argument is represented by the **`DepthInteractionChangedArgs`** class.

```
class DepthInteractionChangedArgs :
      public SignalArgsT<Workspace>
{
public:
  Well well() const;
  Unit unit() const;
  float depth() const;
  float topDepth() const;
  float bottomDepth() const;
};
```

The slot handler accesses the needed information from the signal argument.

```
void activity::onDepthInteractionChanged(const
Slb::Ocean::Techlog::DepthInteractionChangedArgs &args)
{
  qWarning() << "Depth interaction line value = " << args.depth()
<< " " <<
  args.unit();
  qWarning() << "Depth interaction window top value = " <<
args.topDepth()
  << " " << args.unit();
  qWarning() << "Depth interaction window bottom value = " <<
  args.bottomDepth() << " " << args.unit();
}
```

Here the **Depth interaction** tool is enabled in the **Logview** mousebar, and the depth interaction line and window limits are shown.

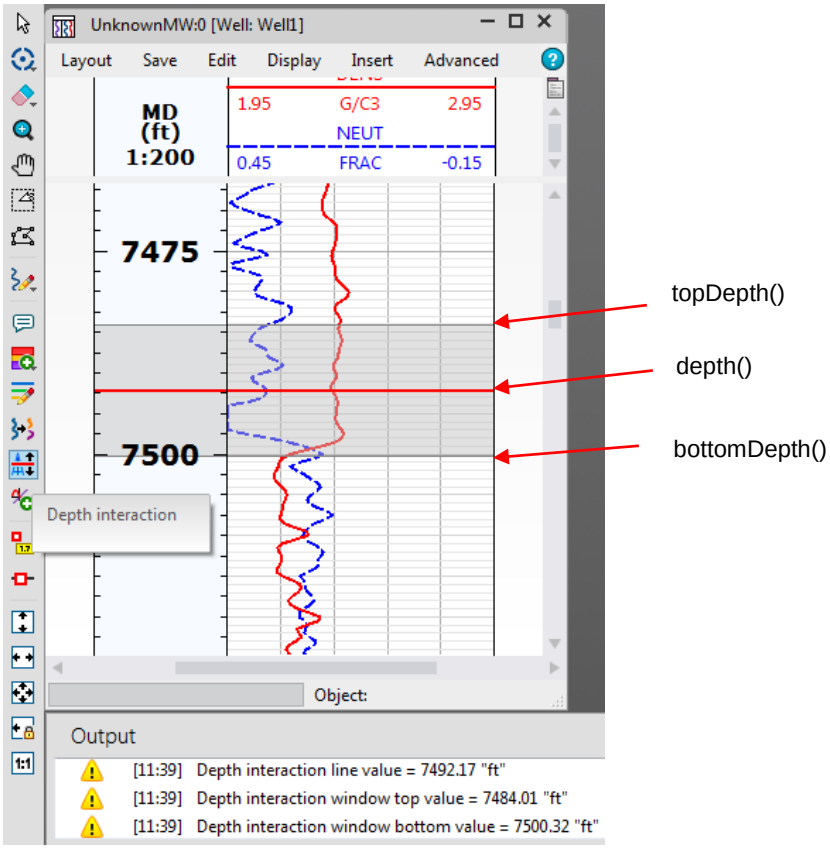


Figure 2-17 Depth interaction

SelectionChanged signal

The **`SelectionChanged`** signal has an argument that returns the new selection through a **`Selection`** domain object, called **`SelectionChangedArgs`**.

See the "Selection domain object" section on page 2-34 for more information on selection.

```
class SelectionChangedArgs : public SignalArgsT<Workspace>
{
public:
  Selection selectionsChanged() const;
};
```

The slot handler accesses the needed information from the signal argument. This example uses the interactive selection color indexes from the **`Selection`** domain object returned by the signal argument. It also gets the ranges of the selected dataset for each color index.

```
void activity::onSelectionChanged(const
Slb::Ocean::Techlog::SelectionChangedArgs &args)
{
  Selection selection = args.selectionsChanged();

  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
selection);
```

```
  Dataset dataset = selection.dataset();
  lock.release();

  lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, dataset);
  Well well = dataset.well();
  lock.release();

  lock = LOCK_CREATE();
  lock.add(selection);
  lock.add(well);
  LOCK_ACQUIRE_OR_RETURN(lock);

  Variable myRef = selection.dataset().findReferenceVariable();
  QList<int> colors = selection.colors();
  QSet<int> colorsSet = colors.toSet();

  if(colorsSet.count() == 1 && colorsSet.contains(-1))
  {
    qWarning() << "-->" + selection.dataset().well().name() + "."
+
    selection.dataset().name() + "." + myRef.name() + " : No
selection";
  }
  else
  {
    qWarning() << "-->" + selection.dataset().well().name() + "."
+
    selection.dataset().name() + "." + myRef.name() + " :";
  }

  foreach(int color, colorsSet)
  {
    // color == -1 is the default value when index not selected
in any color
    if( color != -1 )
    {
      //+1 as on TL interactive selection color count starts at
1.
      qWarning() << "--->Selection Color " +
QString::number(color + 1);
      QString stringColorRange;
      int rangeIndex = -1;
      for(int indexInColors = 0; indexInColors < colors.count();
      indexInColors++)
      {
        if(colors.at(indexInColors) == color)
        {
```

```
          if(!stringColorRange.isEmpty())
          {
            if(rangeIndex == (indexInColors - 1))
            {
              // do nothing in this case, as we want to print only
ranges
              rangeIndex++;
            }
            else
            {
              // new range
              stringColorRange.append(QString(" "));

stringColorRange.append(QString::number(indexInColors));
              rangeIndex = indexInColors;
            }
          }
          else
          {
            // first range

stringColorRange.append(QString::number(indexInColors));
            rangeIndex = indexInColors;
          }
        }
        else
        {
          if(rangeIndex != -1)
          {
            // new color or no color
            // put the previous index to end this color range
            stringColorRange.append(QString("-"));

stringColorRange.append(QString::number(indexInColors - 1));
            rangeIndex = -1;
          }
        }
      }//we might have ended in a range

      if(rangeIndex != -1)
      {
        stringColorRange.append(QString("-"));

stringColorRange.append(QString::number(colors.count()));
      }
      qWarning() << "---->range(s) " + stringColorRange;
    }
```

```
  }
  lock.release();
}
```

The screenshot shows the ranges of dataset indexes per interaction selection color displayed in the Techlog **Output** console when selection is changed in the **Logview**.

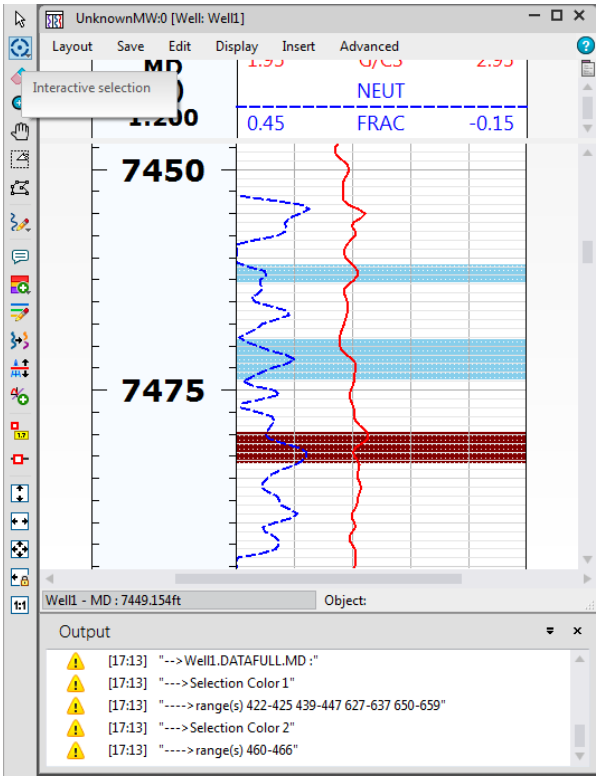


Figure 2-18 Interactive selection

PlotCreated signal

The **`PlotCreated`** signal has an argument called **`PlotCreatedArgs`** that returns the new plot droid and instance created in the workspace.

```
class PlotCreatedArgs : public SignalArgsT<Workspace>
{
public:
  Droid newPlotDroid() const;
  Plot newPlot() const;
};
```

The slot handler accesses the needed information from the signal argument.

Note: The **`Droid`** of the new plot must be locked before calling new plot and its members. Otherwise an exception will be thrown.

```
void activity::onPlotCreated(const
Slb::Ocean::Techlog::PlotCreatedArgs &args)
```

```
{
  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
  args.newPlotDroid());
  qWarning() << "Plot created " << args.newPlot().windowTitle();
  lock.release();
}
```

Selection domain object

The **`Selection`** domain object represents the Techlog interactive selection.

The interactive selection mode allows the end user to create interaction between the plots and other viewers (layouts, tables). The end user selects the points using either polygon selection or interactive selection mode.

```
class Selection : public DomainObject
{
public:
  Dataset dataset() const;
  int color(int index) const;
  void setColor(int index, int val);
  QList<int> colors() const;
  void setColors(const QList<int> &colors);
};
```

Through the **`Selection`** domain object you set dataset indexes to a given interactive selection color. The public methods of **`Selection`** class allow you to:

- get and set the interactive selection color index of a dataset index
  - color = -1 is the default value when the dataset index is not selected in any color
  - interactive selection color count starts at 0 corresponding to color 1 in interactive selection GUI

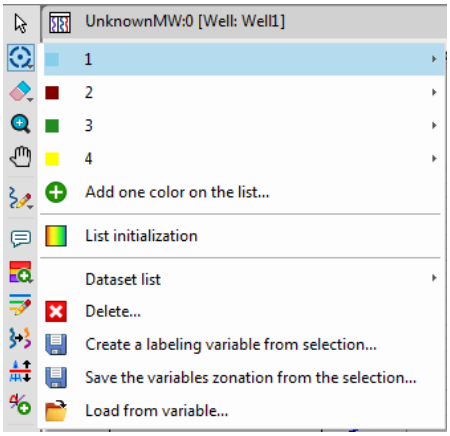


Figure 2-19 Interactive selection color indexes

- get and set the entire colors of selection

The **`Selection`** domain object cannot be created. The selection for a given dataset is returned from the parent **`Workspace`** through **`currentSelection`** public method.

```
class Workspace : public DomainObject
{ ...
public:
  const Selection currentSelection(const Dataset& dataset)
        const;
};
```

This example sets dataset indexes to color sections following depth intervals.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();

Selection selection = workspace.currentSelection(dataset);

Variable ref = dataset.findReferenceVariable();
int rowCount = (int)dataset.rowCount();
for (int i = 0; i < rowCount; i++)
{
  if (ref.getFloatValue(i) >= 7450 && ref.getFloatValue(i) <=
7470)
    selection.setColor(i, 0);
  if (ref.getFloatValue(i) >= 7480 && ref.getFloatValue(i) <=
7500)
    selection.setColor(i, 1);
}

lock.release();
```

This screenshot shows the result in a **Logview** opened in the current workspace and displays variables from the same dataset.

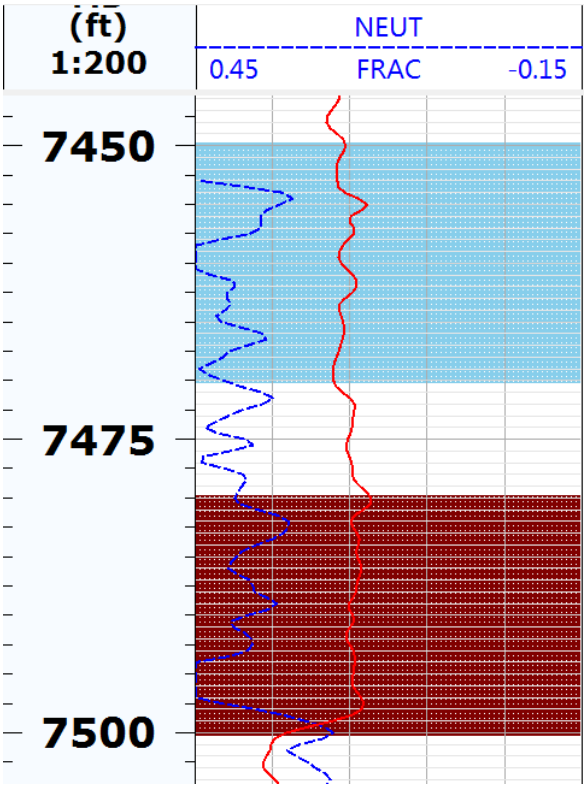


Figure 2-20 Setting dataset indexes to color selections

Techlog output console

The **`Workspace::logEvent`** method issues an information message intended for all end users to see. It displays a string in the Techlog output console UI.

The Techlog output console UI is a window with a scrollable text list. The window remains open until closed by the user. Its contents can be cleared or copied to the clipboard by clicking the items in the right-click contextual menu. The **`logEvent`** method takes two arguments. The first defines the log severity of the message and the second is the text string to display. The different output severity levels are listed in **`LogLevel`** enumeration class:

- Debug
- Information
- Warning
- Error

This example uses the **`logEvent`** method.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();
workspace.logEvent(LogLevelInformation, "Info message");
workspace.logEvent(LogLevelWarning, "Warning message");
workspace.logEvent(LogLevelError, "Error message");
workspace.logEvent(LogLevelDebug, "Debug message");

lock.release();
```

Note: Messages with the log level set to debug are not displayed in the Techlog output console but only in Techlog log file under the **%TEMP%\Schlumberger\Techlog\Logs folder**. The minimum log level in Techlog installation folder **bin64\logger.ini** must be set to Debug (minLogLevel=Debug).

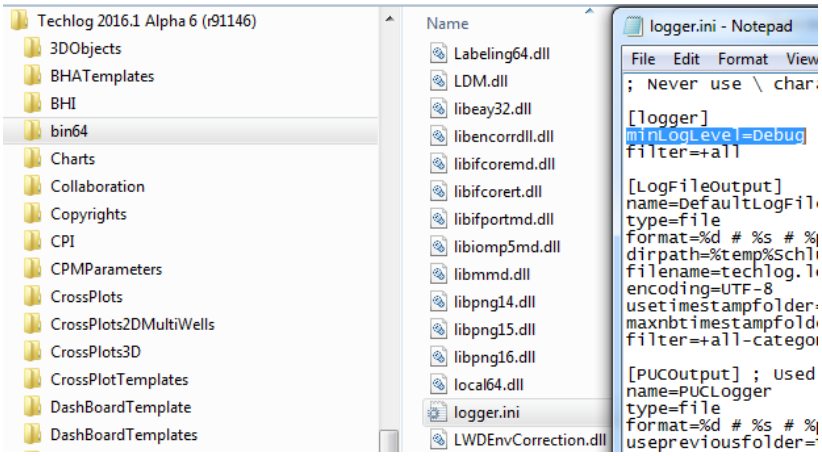


Figure 2-21 Log level set to Debug in logger.ini file

UnitConverter class

The **UnitConverter** class provides access to methods for unit conversion.

The unit converters allow conversion of data when unit conversion is possible. **UnitConverter** is defined as:

```
class UnitConverter
{
public:
  UnitConverter(const Unit & srcUnit, const Unit & dstUnit);
  static bool canConvert(const Unit & srcUnit,
         const Unit & dstUnit);
 float convert(const float srcValue);
 double convert(const double srcValue);
 QVector<float> convert(const QVector<float> srcValues);
 QVector<double> convert(const QVector<double> srcValues);
};
```

Unit conversion

The **UnitConverter::canConvert** static method must be called first to check if the conversion from the source to the target unit is possible.

Get the converter with source and target unit creating the **UnitConverter** object through the constructor of the class (this is one call to Techlog). Afterwards all the conversion calculations are processed locally on the plug-in host side. You can convert a data or a vector of data with the **UnitConverter::convert** public methods.

Example:

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

// Check if can convert from feet to meters
if (!UnitConverter::canConvert("ft","m")) return;
// Get unit converter from feet to meters (one call to Techlog)
UnitConverter unitConverter ("ft","m");

float feetValue = 1.0f;
QVector<float> feetValues;
feetValues << 1.5f << 2.0f << 2.5f << 3.0f;

// Convert from feet to meters (no call to Techlog)
float metersValue;
metersValue = unitConverter.convert(feetValue);
QVector<float> metersValues;
metersValues = unitConverter.convert(feetValues);

lock.release();
```

ProjectBrowser class

The **ProjectBrowser** class provides access to Techlog project browser and allows you to manipulate domain objects as in Techlog project browser GUI.

See the "Accessing domain objects" section on page 3-1 for more information on what is a domain object in Ocean.

```
class ProjectBrowser
{
public:
  void setGroups(DomainObject &childObject, const QStringList
       &groups);
  QStringList getGroups(const DomainObject &childObject);
  void removeFromAllGroups(DomainObject &childObject);
  bool isGrouped(const DomainObject &childObject);
};
```

ProjectBrowser

A **ProjectBrowser** object cannot be created using a constructor; it is only instantiated from the current **Session** domain object.

Session::projectBrowser() returns the **ProjectBrowser** instance.

Currently **ProjectBrowser** is only be used to do group management. The **ProjectBrowser** class allows you to:

- get and set the group for a Techlog domain object. The rules to set a group on a domain object are:
  - the domain object has to support grouping (**DomainObject::supportsGrouping** property returns true)
  - domain objects can belong to only one group at a time
  - a group can be nested in a parent group
  - following characters are forbidden in the group name: ¤ & : \ / * ? " < > |
- remove all the nested groups of a domain object

This example sets different groups with multiple levels to a **Well** domain object.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Session session  = Session::current();
ProjectBrowser projectBrowser = session.projectBrowser();

QStringList multiLevelGroups;
multiLevelGroups << "My first group" << "My second group";
projectBrowser.setGroups(well, multiLevelGroups);

lock.release();
```

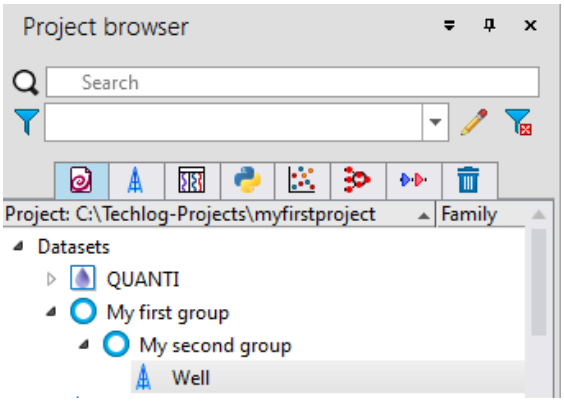


Figure 2-22 Grouping domain object in Techlog project browser

Techlog units management

Techlog supports unit systems since the 2016.1 release, to harmonize units of variables available in a project, both for computation and visualization.

Project unit system signal

A unit system is defined by the end-user at the project creation and can be changed at any time. A plugin cannot change which unit system is selected but can be notified if the unit system is changed by the Techlog end-user through the **`UnitSystemChanged`** event of the **`Project`** class.

```
class Project: public DomainObject
{
public:
  const QString unitSystem() const;

  enum EventType
  {
    UnitSystemChanged
  };
  ...
};
```

Include signal arguments and declare the slot receiver in the activity header file:

```
#include "tsdkunitsystemchangedargs.h"
```

```
private slots:
  void onUnitSystemChanged(
  const Slb::Ocean::Techlog::UnitSystemChangedArgs &args);
```

This example connects the **`Project`** domain object to this signal.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Project project = Session::current().mainProject();

project.connect(Project::UnitSystemChanged, this,
SLOT(onUnitSystemChanged(const
Slb::Ocean::Techlog::UnitSystemChangedArgs &)));

lock.release();
```

The **`UnitSystemChanged`** signal has a **`UnitSystemChangedArgs`** argument. This object returns the new project unit system.

```
class UnitSystemChangedArgs : public SignalArgsT<Project>
{
  QString unitSystem() const;
```

```
};
```

The slot handler accesses the needed information from the signal argument.

```
void activity::onUnitSystemChanged(const
Slb::Ocean::Techlog::UnitSystemChangedArgs &args)
{
  qWarning() << "New project unit system is " << args.unitSystem();
}
```

The Project unit system is only changable by the Techlog end-user through the project setting dialog, so the **`UnitSystemChanged`** signal cannot be triggered programmatically.

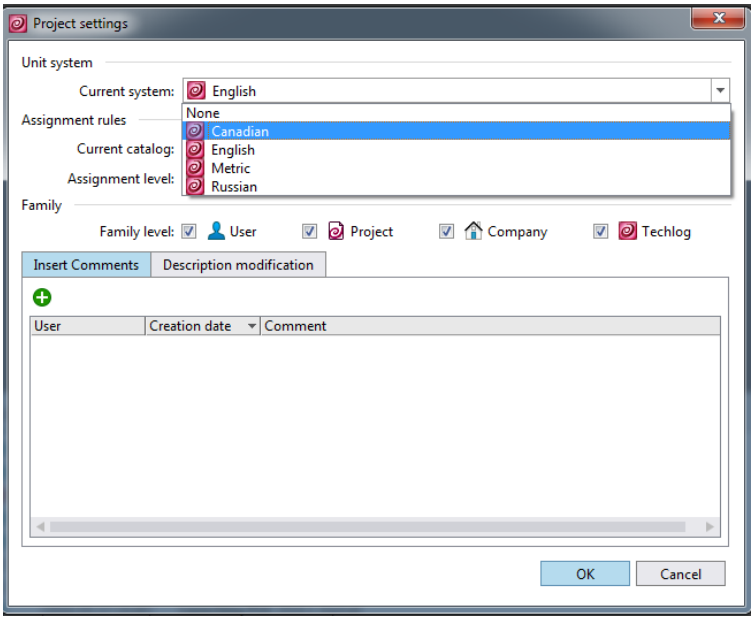


Figure 2-23 Project unit system in project setting dialog

Techlog measurements

A measurement represents a physical quantity (volume, length, etc.), also known as unit quantity in OSDD. A family is associated with a measurement, and the unit system defines the appropriate unit for each measurement.

The **`TechlogMeasurement`** namespace gives well-known measurements exposed through the **`Measurement`** class.

```
namespace Slb {
  namespace Ocean {
    namespace Techlog {
      namespace TechlogMeasurement {
        Measurement getGammaRay();
        Measurement getGammaRayAsCountRate();
        ...
```

```
      }
    }
  }
}
```

```
class Measurement
{
public:
  QString name () const;
  QString description () const;
};
```

Unit catalog

The **`UnitCalalog`** object is instantiated from the **`Session`**.

```
class Session : public DomainObject
{
public:
  UnitCatalog unitCatalog() const;
  ...
};
```

From **`UnitCatalog`** object you can:

- Get the list of compatible units for a given measurement
- Get the list of compatible units for a given unit
- Get the display unit of a measurement for a given unit system
- Get the exaustive list of measurements available in Techlog
- Get the exaustive list of units available in Techlog

```
class UnitCatalog
{
public:

  QList<Unit> getCompatibleUnits(const Measurement &measurement)
       const;
  QList<Unit> getCompatibleUnits(const Unit &unit) const;
  Unit getDisplayUnitFromMeasurement(const Measurement
       &measurement) const;
  QList<Measurement> getMeasurements() const;
  QList<Unit> getAvailableUnits() const;
  ...
};
```

This example retrieves a display unit in the current project unit system for a given measurement.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
```

```
Project project = Session::current().mainProject();

Measurement measurement =
TechlogMeasurement::getMeasuredDepth();

UnitCatalog unitCatalog = Session::current().unitCatalog();

Unit displayUnit =
unitCatalog.getDisplayUnitFromMeasurement(measurement);

qWarning() << "Display unit for measurement " <<
measurement.name() << " is " << displayUnit << " in unit system
" << project.unitSystem();

lock.release();
```

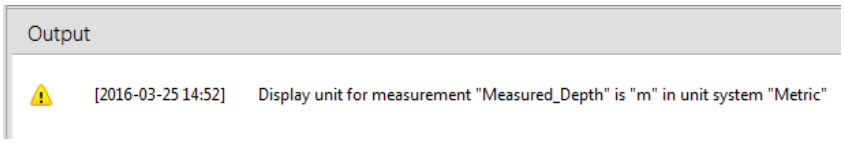


Figure 2-24 "Measured Depth" display unit in "Metric" unit system

Unit object

`Unit` is a typedef of **`QString`** to keep unchanged (not deprecated) legacy APIs dealing with units identified by a string and pass a dedicated **`Unit`** object to the new ones.

Display unit

Every plot has properties to get the display unit of variables plotted on axes. The units can be changed manually with the properties editor but initially the variables display unit is returned by the API.

Plot axes limits and display parameters

Each API to set the limits or display parameters on plot axes has a **`Unit`** object argument. The **`Unit`** informs the plot which unit must be considered regarding the limits or display values passed to the function. The values are converted from the **`Unit`** to the plot axis unit and the converted values are set to limits or display parameters.

Note: The functions throw an exception if the **`Unit`** passed to the function is not compatible with the plot axis unit.

Techlog families management

Techlog has been improving its family management to support unit systems and measurements.

Techlog families

The **`TechlogFamily`** namespace gives well known families exposed with Ocean through the **`Family`** object.

```
namespace Slb {
  namespace Ocean {
    namespace Techlog {
      namespace TechlogFamily {
        Family getGammaRay();
        Family getNeutronPorosity();
        ...
      }
    }
  }
}
```

Family object

`Family` is a typedef of **`QString`** to keep unchanged (not deprecated) the legacy APIs that deal with a family identified by a string and pass a dedidated **`Family`** object to the new ones.

MainFamily object

`MainFamily` is an object with a **`name`** property that can only be retrieved from the Family catalog or the **`FamilyInfo`** of a given **`Family`**.

```
class MainFamily
{
public:
  QString name() const;
  bool isEmpty() const;
  ...
};
```

Note: If **`MainFamily`** object is instantiated through the class constructor, **`isEmpty`** function returns null and this object is considered invalid.

Family catalog

The **`FamilyCalalog`** object is instantiated from the **`Session`**.

```
class Session : public DomainObject
{
public:
```

```
  FamilyCatalog familyCatalog() const;
  ...
};
```

From **`FamilyCatalog`** object you can:

- Get information for a given family defined at the highest priority storage level or for a given storage level
- Retrieve the family from the assignement rule with variable name and unit
- Get the exhaustive list of main families available in Techlog
- Get the exhaustive list of families available in Techlog
- Get the list of families for a given main family

```
class FamilyCatalog
{
public:
  FamilyInfo getFamilyInfo(const Family &family);
  FamilyInfo getFamilyInfo(const Family &family,
        const StorageLevel storageLevel) const;
  Family getFamilyFromCurrentAssignmentRules(const QString
        &variableName, const Unit &unit) const;
  Family findFamilyFromCurrentAssignmentRules(const QString
        &variableName, const Unit &unit) const;
  QList<MainFamily> getMainFamilies() const;
  QList<Family> getFamilies() const;
  QList<Family> getFamiliesFromMainFamily(const MainFamily
        &mainFamily) const;
};
```

From the **`UnitCatalog`** you can retrieve all families associated with a **`Measurement`**.

```
class UnitCatalog
{
public:
  QList<Family> getFamiliesFromMeasurement(const Measurement
       &measurement) const;
  ...
};
```

The main properties of the **`Family`** are exposed with Ocean through the **`FamilyInfo`** object as measurement, unit, variable type and list of main families.

```
class FamilyInfo
{
public:
  Measurement measurement() const;
  QList<MainFamily> mainFamilies() const;
  Unit unit() const;
  VariableType variableType() const;
};
```

This example resolves a **`Family`** from the assignement rule with variable and unit and get the **`FamilyInfo`** according to the Techlog priority order rules.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

FamilyCatalog familyCatalog =
Session::current().familyCatalog();

Family neutFamily =
familyCatalog.findFamilyFromCurrentAssignmentRules("NEUT",
"ft3/ft3");

qWarning() << "Family = " << neutFamily;

if (!neutFamily.isEmpty())
{
  FamilyInfo neutFamilyInfo =
familyCatalog.getFamilyInfo(neutFamily);

  qWarning() << "Variable type (priority level) = " <<
  ArgChecker::toString(neutFamilyInfo.variableType());

  FamilyInfo neutFamilyInfoTechlog =
  familyCatalog.getFamilyInfo(neutFamily, StorageLevelTechlog);

  qWarning() << "Variable type (Techlog level) = " <<
  ArgChecker::toString(neutFamilyInfoTechlog.variableType());
}

lock.release();
```

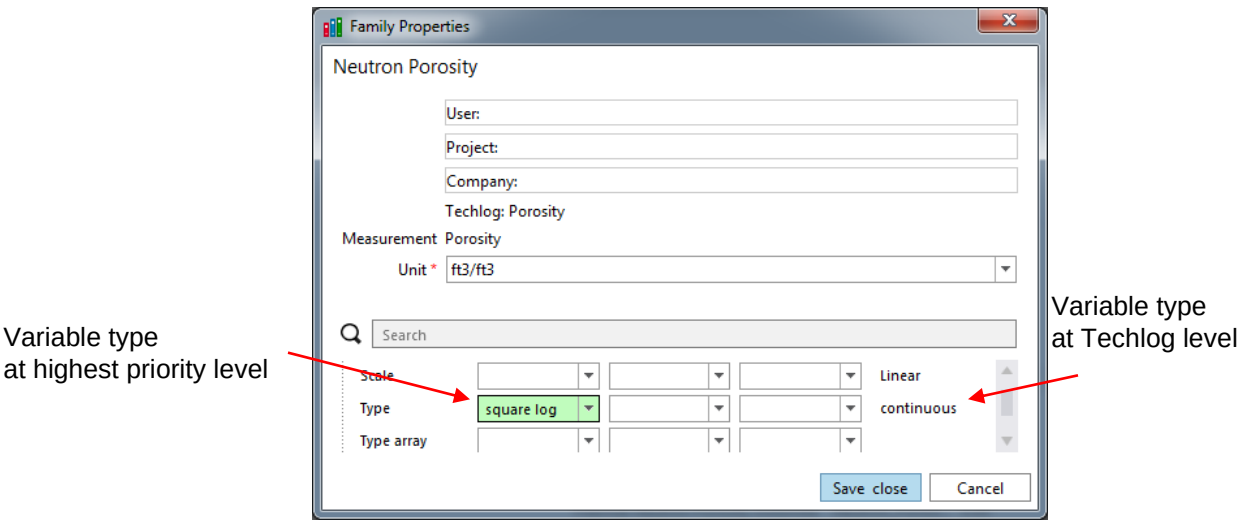


Figure 2-25 Family information for a given storage level

Call Python script from an Ocean plug-in

The static functions of the **`Python`** class allow you to run a Python script from an Ocean plug-in. As a Python script can create and modify Techlog data or create plots in the Techlog workspace, those API can't be called in a **`Lock`**.

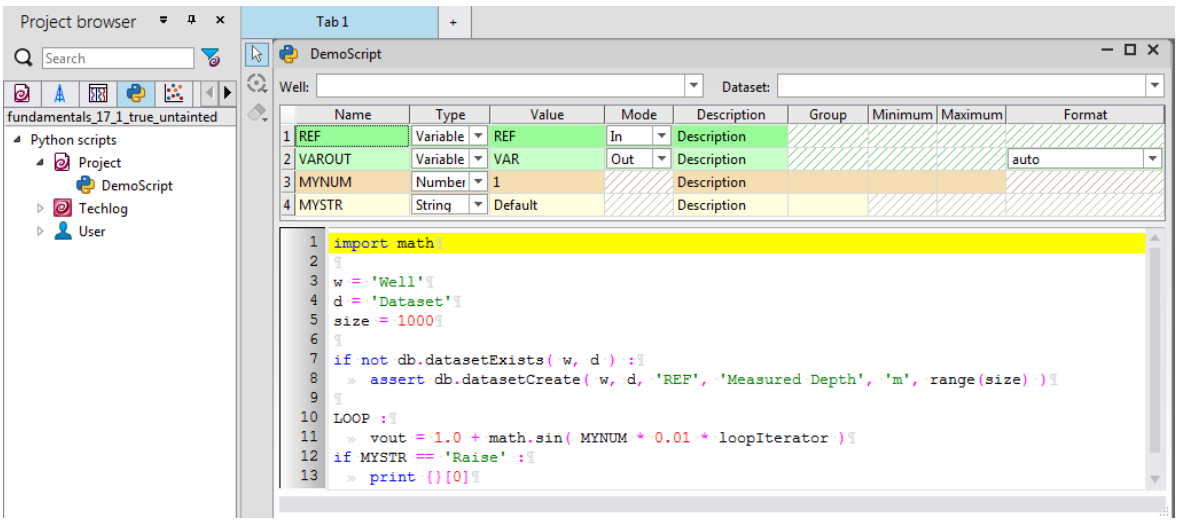
See "Locking mechanism" on page 3-9 for more information on how to lock Techlog objects.

```
class Python
{
public:
  static bool exists(StorageLevel level, const QString
         &scriptName);
  static ReturnValue<bool> run(StorageLevel level, const QString
         &scriptName);
  static ReturnValue<bool> run(StorageLevel level, const QString
         &scriptName, const Dataset &dataset,
         const ScriptArgumentValues &parameters);
};
```

The **`Python`** class allows you to:

- check if the Python script that you want to run exists at a given Techlog storage level (plug-in, user, company, project, or Techlog)
- call the Techlog Python script from Techlog synchronously in background mode without passing any parameters to the script
- call the Techlog Python script from Techlog synchronously in background mode passing parameters to the script

Take as example this Python script stored at the project level:



The script has four parameters:

- a REF paramater of type "Variable" used to create the reference variable with default name REF in the dataset created by the script
- a VAROUT parameter of type "Variable" used to create in the same dataset a variable with default name VAR

- a MYNUM parameter of type "Number" used to compute a sine value at each index of the dataset and populate the VAR variable with those values.
- A MYSTR parameter of type "String" used to raise a script exception at runtime

Through Ocean, call this script with the **`run`** static function modifying the default parameter values with the QMap **`ScriptArgumentValues`** passed to the function. The QMap key is the name of the parameter; the QMap value is the new value of the parameter.

In this example, the name of the variable handled by the VAROUT parameter is modified to ANOTHER_VAR, the sine radian value handled by the MYNUM parameter is set to 3, and the MYSTR parameter value is changed to "Raise" in order to raise a script exception.

Note: The **`run`** function returns false if the Python interpretation ends with an exception, but it doesn't return the Python exception message.

```
// Run a script on a given dataset, giving value of declared
parameters
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
Project project = Session::current().mainProject();
Well                well = project.findWell("Well");
if (well.isNull())
{
  qWarning() << "No well";
  lock.release();
  return;
}

Dataset dataset = well.findDataset("Dataset");
if (dataset.isNull())
{
  qWarning() << "No dataset";
  lock.release();
  return;
}
lock.release();

if (!Python::exists(StorageLevelProject, "DemoScript.py"))
  return;

ScriptArgumentValues scriptParams;
scriptParams["MYNUM"] = "3";
scriptParams[ "VAROUT" ]  = "ANOTHER_VAR";
scriptParams["MYSTR"] = "Raise";

qWarning() << "Run DemoScript from project level ...";
```

```
bool executed = Python::run(StorageLevelProject,
"DemoScript.py", dataset,
scriptParams);

if (!executed)
  Workspace::logEvent(LogLevel::LogLevelWarning, "Could not run
with given parameters...");
```

Progress dialog with Ocean

The **`ProgressDialog`** class shows a progress dialog window to the end-user in order to provide feedback on the progress of a slow operation.

```
class ProgressDialog
{
public:
  ProgressDialog(const QString &text,const unsigned int maximum);
  unsigned int value() const;
  ReturnValue<bool> setValueAndCheckCanceled(
    const unsigned int value);
  ReturnValue<bool> incrementAndCheckCanceled();
  void setText(const QString &text);
};
```

The progress dialog is a modal dialog window that it is automatically closed whenever:

- the operation ends
- the **`Lock`** in which the operation occurs is released
- the **`ProgressDialog`** instance goes out of scope

See "Locking mechanism" on page 3-9 for more information on how to lock Techlog objects.

The progress operation is done in the loop through **`setValueAndCheckCanceled`** or **`incrementAndCheckCanceled`** functions.

This example calls **`setValueAndCheckCanceled`** at each loop iteration.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

ProgressDialog progressDialog("Starting", 100);

for (int i = 0; i < 100; ++i)
{
  if (i == 60)
    progressDialog.setText("Almost done");

  bool wasCanceled = progressDialog.setValueAndCheckCanceled(i);
  if (wasCanceled)
    // Do cleanup or whatever
    break;
}

lock.release(); // Automatically close the progress dialog
```

3 Accessing domain objects

In This Chapter

General concepts 3-2
Domain object access patterns 3-3
Domain objects 3-3
Common patterns 3-4
Droid 3-4
Find and get patterns 3-5
isA pattern 3-6
Erase pattern 3-7
Name pattern 3-8
Support grouping 3-9
Locking mechanism 3-9
How to use locking mechanism 3-9
Best practices 3-13
Domain object signals 3-17
DomainObjectChanged signal 3-20
DomainObjectErased signal 3-21
DomainObjectRenamed signal 3-21

General concepts

In the Ocean API, a *domain object* represents a concept in the Techlog domain; it is something that Techlog users can use when running the application.

A *domain object* can be data objects (well, variable, etc.), plots (logview, cross-plot, etc.), graphical objects (line, polyline, etc.), or workflow objects (workstep, parameter, etc.).

The Ocean Framework libraries model the various domains of Techlog. This chapter explains how to retrieve, modify, and create Techlog domain objects, also known as native Techlog domain objects.

Every domain object representing a Techlog concept has related properties. The properties are non domain objects and are dependent instances that cannot exist without a related independent domain object.

A domain object can participate in any number of binary relationships with other domain objects, and these relationships can be exclusive (composition) or shared (aggregation). For example, a dataset refers to a reference variable object that is really part of the dataset, and a variable can be associated with the x-axis of a crossplot, but it exists outside of that object: if you delete the crossplot, the variable is not deleted.

A domain object is either persisted (like a well) or transient (like a plot), has a unique identifier and is described by a set of attributes.

A domain object cannot be constructed. Using the **`create`** method, you create domain objects that reside in the intended class of the object.

Domain object access patterns

This section describes the basic patterns of domain object access in Ocean for Techlog, covering the relevant portions of the domain object from the perspective of an application developer using the native Techlog domain objects. The native Techlog domain objects API is tailored to the individual domain object and discussed in detail in the following sections.

Domain objects

Domain objects provide a high-level way to access your data. They are often called business objects because they represent concepts that are visible to the end-user. An important point with domain objects is that they are implemented as C++ classes inherited from `QObject` class. The application code is type safe; object properties and types are known and checked by the compiler.

Domain objects contain business logic. Typically, they act as intermediaries between Techlog where the data are stored in the project and application code of the plug-in. They hide database implementation details and differences between database vendors from the application. Domain objects do not require persistence, and it is possible to have memory objects with business logic and not save their state to the Techlog data store.

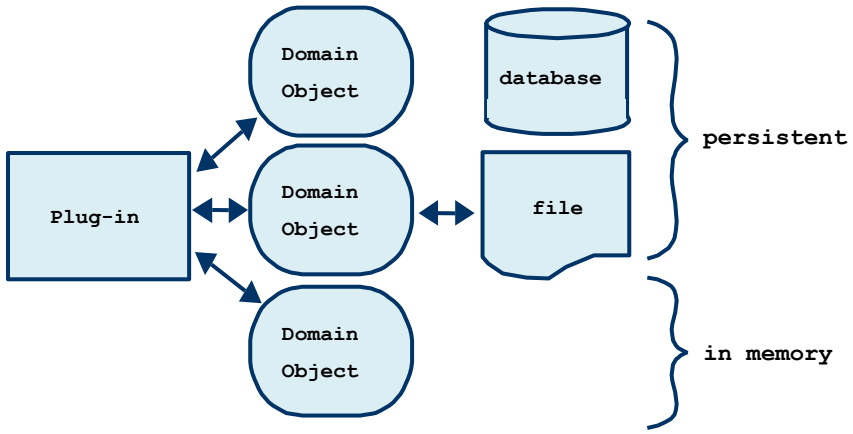


Figure 3-1 Domain Object

Domain object is a reference type object. Ocean for Techlog simplifies the use of C++ reference type objects in the way that you do not have to de-allocate memory used by the object. The memory is de-allocated by Ocean when this object does not have references to other objects. This is called "Smart pointers".

You access domains in Techlog via domain objects. As a user of the native Techlog domain objects, the complexity of implementation and location of Techlog objects is hidden from you. There are only a few general topics that you should be aware of:

- Common patterns
- Locking mechanism
- Signals

All access to native Techlog domain objects must take place on the main thread in order to guarantee the integrity of the data. Access from a thread other than the main will give this run-time error:

```
Call from non-main thread detected. Not supported.
```

Common patterns

There is no general API to create or update native Techlog domain objects. Those API patterns depend on the individual native Techlog domain object.

There is a general API to read and delete native Techlog domain objects using common patterns inherited from the **`DomainObject`** class.

The common patterns that native Techlog domain objects follow are:

- All native Techlog domain objects have a unique identity (**`Droid`**)
- You retrieve native Techlog domain objects by calling **`get`** or **`find`** static methods of the **`DomainObject`** base class.
- You delete native Techlog domain objects by calling an **`erase`** method on the object. All the native Techlog domain objects have this functionality because they inherit it from the **`DomainObject`** base class.
- You create native Techlog domain objects by calling a **`create`** method which is a static method on the domain object type that you intend to create. One argument of the create method is the parent container. There are no public constructors on native Techlog domain objects.
- Domain objects can be grouped.

Droid

Native Techlog domain objects inheriting from the **`DomainObject`** base class have a durable identity via a **`Droid`**.

```
class DomainObject : public IPrintable
{
public:
    ...
    const Droid droid() const;
};
```

A **`Droid`** is a Durable Runtime Object Identifier, which is a reference to an actual domain object. A **`Droid`** eliminates the need to hold on to the object itself. A droid is compact and generic. It can be passed around in an application without impacting performance, and it can be resolved into the object it represents at any time. Methods which receive a **`Droid`** as input and pass it to other methods are not concerned with the type of data the **`Droid`** represents.

Droids and objects represented by droids must be sustainable. This means that they must be persistent in a data store and not just temporary, in-memory only, objects.

Droids are required to be unique. Each **`Droid`** represents a different object. If two droids were identical, it would be difficult to determine which object the droid should resolve to.

Droid creation is handled by the framework when a native Techlog domain object is created.

The **Droid** class is defined as:

```
class Droid : public IPrintable
{
public:
    static Droid create();
    static const Droid& empty();
    QString toString() const;
    const bool isEmpty() const;
    uint qHash() const;
};
```

The **Droid** class allows you to:

- create a new **Droid** object, initialized with a unique value
- create an instance of an empty **Droid**

Find and get patterns

The patterns inherited from **DomainObject** base class allow you to retrieve a native Techlog domain object from its droid.

```
class DomainObject : public IPrintable
{
public:
    ...
    static DomainObject get(const Droid& droid);
    static DomainObject find(const Droid& droid);
    const bool isNull() const;
};
```

Remember that a plug-in runs in its own process. If you need to store domain objects in the plug-in cache, it is safer to store their droids as member variables of your plug-in activity. Then you retrieve the domain objects by using the static **get** or **find**. The **find** pattern returns a null object if a domain object cannot be resolved on the server side. Test the null value of a domain object with the **isNull** method. This method indicates that the domain object cannot be found anywhere.

Note: A **null** is never good. Trying to access the property of a null object will probably throw an exception.

This example shows the best practice to retrieve a domain object from its droid stored as a member of the plug-in activity:

```
static Droid _customPlotDroid;

void Activity::run()
{
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
```

```
  // Get the current workspace from the current session
  Workspace workspace = Session::current().currentWorkspace();

  CustomPlot customPlot;
  // Check if the droid is not empty
  // cannot use a get or find method on an empty droid
  if (_customPlotDroid.isEmpty())
  {
    // Create the domain object
    customPlot = CustomPlot::create(workspace);
    // store the droid value in the plug-in activity cache
    _customPlotDroid = customPlot.droid();
    lock.release();
    return;
  }

  // droid is not empty
  // try to retrieve the domain object on server side
  DomainObject domainObject;
  domainObject = DomainObject::find(_customPlotDroid);
  if (domainObject.isNull())
  {
    // cannot find the domain object, recreate the plot
    customPlot = CustomPlot::create(workspace);
    // store the new droid value in the plug-in activity cache
    _customPlotDroid = customPlot.droid();
  }
  else // domain object loaded on plug-in side and cast as a plot
    customPlot = domainObject.cast<CustomPlot>();

  lock.release();
}
```

isA pattern

The **`isA`** pattern inherited from the **`DomainObject`** base class allows you to check the **`DomainObject`** type before casting the **`DomainObject`** to its type.

```
class DomainObject : public IPrintable
{
public:
   ...
   bool isA () const;
   T tryCast () const;
```

```
  T cast () const;
};
```

Note: The **`tryCast()`** function returns null if the domain object cannot be cast to the required type.

Example:

```
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, droid);

DomainObject domainObject = DomainObject::get(droid);

if (!domainObject.isA<Well>())
  return;
Well well = domainObject.cast<Well>();
// ...

// OR WITH TRYCAST

well = domainObject.tryCast<Well>();
if (well.isNull())
  return;
// ...

lock.release();
```

Erase pattern

The **`erase`** pattern inherited from the **`DomainObject`** base class allows you to delete a native Techlog domain object.

```
class DomainObject : public IPrintable
{
public:
    ...
    const bool isErased() const;
    void erase();
    bool supportsErase() const;
};
```

Call the **`erase`** method on the object that you want to delete. This deletes the object and all its children. If this object has children (for instance a well has child datasets), the children are marked as erased immediately and recursively, and cannot be accessed anymore. In this case the **`isErased`** method returns true for the object and its children.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Project project = Session::current().mainProject();
```

```
Workspace workspace = Session::current().currentWorkspace();
Well well = project.wells().find("MyWell");
Dataset dataset = well.datasets().find("MyDataset");

well.erase();

if (dataset.isErased())
{
  workspace.logEvent(LogLevelWarning,
  QString("Dataset %1 is erased")
  .arg(dataset.droid().toString()));
}

lock.release();
```

The **supportsErase** method checks if the domain object supports being erased or not. Not all native Techlog domain objects can be erased; the **Session** and **Workspace** cannot be deleted.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();
if (!workspace.supportsErase())
{
  workspace.logEvent(LogLevelWarning,
  "This domain object type cannot be erased");
}

lock.release();
```

Note: The **erase** method throws an exception if **supportsErase** returns false.

Name pattern

The **name** property in the **DomainObject** base class is common to all native Techlog domain objects, but not all Techlog objects can have a name. For instance, all data domain objects as **Well**, **Dataset** and **Variable** support a name and plots do not. The **supportsName** method checks if the **DomainObject** type supports having a name or not before calling the getter function **name**.

```
class DomainObject : public IPrintable
{
public:
    const QString name() const;
    bool supportsName() const;
    ...
```

```
};
```

Example:

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();
Droid droid = well.droid();
DomainObject wellDomainObject = DomainObject::get(droid);
if (wellDomainObject.supportsName())
  workspace.logEvent(LogLevelWarning,
  QString("Wells support name").arg(wellDomainObject.name()));

lock.release();
```

Note: The **`name`** method throws an exception if **`supportsName`** returns false.

Support grouping

The **`supportsGrouping`** method checks if the domain object supports being grouped or not. Not all native Techlog domain objects can be grouped in the Techlog project browser. For instance **`Session`** and **`Workspace`** cannot be grouped.

`Well`, **`Dataset`** and **`Variable`** domain objects currently support grouping.

```
class DomainObject : public IPrintable
{
public:
    ...
    bool supportsGrouping() const;
};
```

See the "ProjectBrowser " section on page 2-39 for more information on how to group domain objects in the Techlog project browser.

Locking mechanism

A lock represents a group of reads or edits performed on data (domain objects). Locks are required for every operation on native Techlog domain objects: create, read, update, or delete the data.

How to use locking mechanism

1. Create a **`Lock`** object.
2. Lock the objects that you want to change.
3. Modify the objects.
4. Release the locked objects.

The locking mechanism uses the **`Lock`** class.

```
class Lock : public IPrintable
```

```
{
public:
 …
 static Lock create(const QString &file, const int& line);
 void add(const Droid& droid);
 void add(const DomainObject& domainObject);
 void addAll();
 ReturnValue<bool> tryAcquire();
 QList<Droid> droidsThatFailedToLock() const;
 QString failedToLockMsg() const;
 void release();
};
```

You may create locks through the static **`create`** function but it is recommended to use the **`LOCK_CREATE`** macro instead of calling this function directly.

```
Lock lock = LOCK_CREATE();
```

The lock does not need to be disposed of when the system no longer needs it; releasing the lock is enough.

The rules for locking domain objects in Techlog are:

- First, you must lock the objects that you want to read or update. If the object is not locked and an attempt is made to access the object, then the plug-in throws an exception. Since the plug-in is not running in the same process as the Techlog process, it is important to read data through a lock acquiring the object that you want to read.
- When you want to create an object as the child of another object or delete an object, you must lock the parent object before the creation or deletion.
- When you create an object inside a lock, it is implicitly locked. For example, if you create an object and then want to modify a property of this object, you don't need to lock it. The lock you placed on its parent before the creation carries through to the new child.

Some further things to note with locking mechanism include:

- **`tryAcquire`** function attempts to lock all the **`DomainObject`** instances that have been added to this **`Lock`**. The only cases in which this function returns false are if one of the objects has already been erased or the droid does not exist. If some of the objects are already locked by another actor, the call will wait until those objects are released. If the return value of the function is not checked by the calling code, it will throw an exception.
- There is no rollback; the data is changed when the line of code for the change is executed.
- If you create a lock, add a domain object to this lock, acquire and change the object and you omit the call to **`release`**, the changes to the data will not occur, and the plug-in throws an exception.
- Each plug-in may only have one open lock at a time. If you try to nest locks in the activity, the plug-in throws an exception.

If you want to access data from the Techlog project you need to lock the current **`Session`** (the parent of a Techlog project). It means locking everything in the lock. In order to achieve this, Ocean for Techlog provides the **`LOCK_CREATE_AND_ACQUIRE_ALL`** macro.

Note: Locks are recursive. The **`LOCK_CREATE_AND_ACQUIRE_ALL`** macro locks the **`Session`** object and all the objects underneath it so everything is locked.

For instance, in the **`run`** method of the plug-in activity you want to retrieve a well with its name from the main project and set its color property to red. In this case you need to lock the parent session because searching the well by name in the main project is not safe. If the parent session of the main project is not locked, this main project could be closed or changed by another actor other than the plug-in.

This example demonstrates this scenario.

```
// this macro will create a lock object
// and lock the entire Techlog session
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
Project project = Session::current().mainProject();
Well well = project.wells().get("MyWell");
well.setColor(Qt::red);
// release objects and commit changes to Techlog
lock.release();
```

When the well droid or the well instance is stored as a member of your plug-in activity then in a callback you may lock only the droid or the object (locking the droid locks the domain object and all its children), avoiding the need to search for the well in the main project and locking everything.

```
Well _myWell = createMyWell();

// create a lock object
Lock lock = LOCK_CREATE();
// add the well to the lock
lock.add(_myWell);
// try to acquire the well
if (!lock.tryAcquire())
{
  // this code will be executed if the well has been erased
  return;
}
_myWell.setColor(Qt::blue);
// release objects and commit changes to Techlog
lock.release();
```

If you lock several domain objects and one cannot be locked because it has been erased, the **`failedToLockMsg`** function returns a string designed for the end-user describing why the locking failed and the **`droidsThatFailedToLock`** function returns the list of droids that failed to lock**.**

```
Lock lock = LOCK_CREATE();
// locking a droid which does not exist on Techlog side will fail.
lock.add(Droid::create());
if (!lock.tryAcquire())
{
  Workspace::logEvent(LogLevelError, lock.failedToLockMsg());

  Workspace::logEvent(LogLevelError, "List of droid that failed
to lock:");
  foreach (Droid droid, lock.droidsThatFailedToLock())
    Workspace::logEvent(LogLevelError, droid.toString());

  return;
}

lock.release();
```

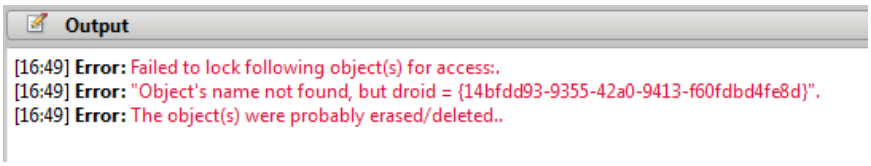


Figure 3-2 Failed to lock message

Note: If the locking failed, do not call **`release`**. Calling the **`release`** method if the **`tryAcquire`** function returns false throws an exception.

Some other macros allow you to simplify the code. For instance you can lock or return using the **`LOCK_ACQUIRE_OR_RETURN`** macro. If you access only one object, you can create the lock, add your object, acquire or return using the **`LOCK_CREATE_THEN_ACQUIRE_OR_RETURN`** macro.

This example is the original code.

```
// create a lock object
Lock lock = LOCK_CREATE();
// add the object to lock
lock.add(myWell);
// acquire or return
if (!lock.tryAcquire()) return;
// modify the locked object
myWell.setColor(Qt::blue);
// release objects and commit changes to Techlog
lock.release();
```

Simplify the example using the **`LOCK_ACQUIRE_OR_RETURN`** macro.

```
Well myWell = createMyWell();

// create a lock object
Lock lock = LOCK_CREATE();
```

```
// add the object to lock
lock.add(myWell);
// acquire or return
LOCK_ACQUIRE_OR_RETURN(lock);
// modify the locked object
myWell.setColor(Qt::blue);
// release objects and commit changes to Techlog
lock.release();
```

If you access only one object, further simplify your code:

```
// create a lock, add the object and acquire or return
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, myWell);
// modify the locked object
myWell.setColor(Qt::blue);
// release objects and commit changes to Techlog
lock.release();
```

Best practices

All access to Ocean APIs requires locking but an attempt to lock may fail under certain conditions. This scenario must not be considered exceptional and must be handled gracefully by all plug-ins. To avoid implementing complicated patterns of return values, structure and layer your code according to these guidelines and best practices.

1. Create/release the lock in the outermost level of the callstack

You should attempt to locate all Lock handling code in the immediate layer called by the external code.

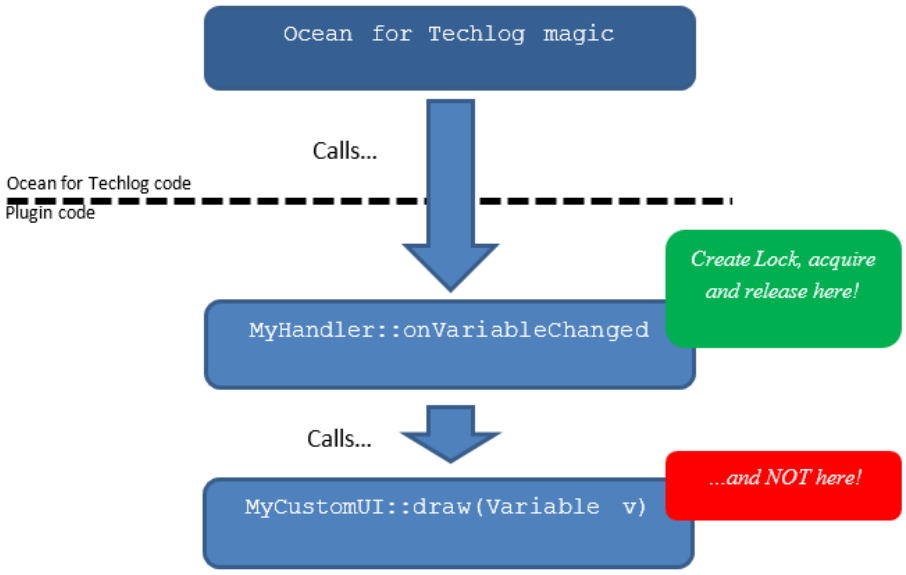


Figure 3-3 Lock code location

The guidelines for Ocean locks are to always create/release the lock in the outermost level of the callstack. This is a good way to structure the plugin code.

2. Avoid locking everything

It is sometimes difficult to avoid locking everything (the Techlog session) in a Lock. You may need to access some data from the main Techlog project or in a callback, you may need to access some objects from the parent of the object that emitted the signal.

It is recommended to split the Lock in different blocks, trying to lock objects with the highest granularity. This is a best practice when you have a call to a complex algorithm in the lock block running on a large amount of data. This best practice is also strongly recommended when you need to run several plug-ins at the same time.

```
void activity::onGraphicsItemMovingNOK(const
Slb::Ocean::Techlog::GraphicsItemInteractionArgs &args)
{
  MarkerLabel markerLabel = args.sender().cast<MarkerLabel>();

  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

  qreal x = markerLabel.center().x();
  qreal y = markerLabel.center().y();

  GraphicsScene graphicsScene = markerLabel.graphicsScene();

  // run complex algorithm that compute shapes from huge variable
values following shape position and add them to the graphics scene
  runComplexAlgorithm(x, y, graphicsScene, _hugeVariable);

  lock.release();
}

void activity::onGraphicsItemMovingOK(const
Slb::Ocean::Techlog::GraphicsItemInteractionArgs &args)
{
  MarkerLabel markerLabel = args.sender().cast<MarkerLabel>();

  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
  markerLabel);

  qreal x = markerLabel.center().x();
  qreal y = markerLabel.center().y();

  GraphicsScene graphicsScene = markerLabel.graphicsScene();

  lock.release();

  lock = LOCK_CREATE();
  lock.add(graphicsScene);
  lock.add(_hugeVariable);
  LOCK_ACQUIRE_OR_RETURN(lock);
```

```
  // run complex algorithm that compute shapes from huge variable
values following shape position and add them to the graphics scene
  runComplexAlgorithm(x, y, graphicsScene, _hugeVariable);

  lock.release();
}
```

3. Locking objects in a loop

Do not open a lock, acquire objects and release them in a loop; this can negatively affect performance.

4. Don't pass a lock to a sub-routine

Consider the following example where the function **`runComplexAlgorithm`** exposes a complex Techlog calculation using the **`Variable`** data as input.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Variable variable =
well.datasets().get("MyDataset").variables().get("MyVariable")
;

setMyData(variable);

// ...

float result = runComplexAlgorithm(variable);

// ...

lock.release();
```

Unfortunately, when the algorithm is called, the data still only exists in the local cache of the plug-in host process while the actual implementation of the algorithm is in the Techlog process. The variable changes must be committed on Techlog side to run the complex algorithm.

```
Well well = createMyWell();

Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, well);

Variable variable =
well.datasets().get("MyDataset").variables().get("MyVariable")
;

// ...

setMyData(variable);
```

```
// ...

lock.release();

lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

float result = runComplexAlgorithm(variable);

// ...

lock.release();
```

This code could be structured using helper functions; one pitfall would be that it must pass the **`Lock`** to the sub routine which then does a commit through a call to the **`release`** function, followed by a **`LOCK_CREATE_AND_ACQUIRE_ALL`**.

```
void Activity::onMyVariableCreatedNOK(const
Slb::Ocean::Techlog::VariableCreatedArgs &args)
{
  Variable variable = args.newVariable();
  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
variable);

  helperFunctionA(variable);
  helperFunctionB_NOK(lock, variable);

  lock.release();
}

void Activity::helperFunctionA(Variable variable)
{
  // ...
  setMyData(variable);
  // ...
}
void Activity::helperFunctionB_NOK(Lock &lock, Variable
variable)
{
  lock.release();

  lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

  float result = runComplexAlgorithm(variable);
  qWarning() << result;
  // ...
}
```

The best practice is to not pass the **Lock** to helper function B and do not release and open a new Lock in the sub-method (see "Create/release the lock in the outermost level of the callstack" on page 3-13).

```
void Activity::onMyVariableCreatedOK(const
Slb::Ocean::Techlog::VariableCreatedArgs &args)
{
  Variable variable = args.newVariable();

  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
variable);

  helperFunctionA(variable);

  lock.release();

  lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

  helperFunctionB_OK(variable);

  lock.release();
}

void Activity::helperFunctionB_OK(Variable variable)
{
  float result = runComplexAlgorithm(variable);

  qWarning() << result;
  // ...
}
```

5. Do not call **QCoreApplication::processEvents**

Plug-ins should **NEVER** call **QCoreApplication::processEvents** method from inside a **Lock**.

According to Qt documentation, this method processes all pending events for the calling thread until there are no more events to process.

This may cause re-entry issues and a second **Lock** may be created on the Techlog main thread which is forbidden by locking mechanism rules (only one **Lock** open at the time).

This results in a plug-in crash.

Domain object signals

A plug-in interacts with Techlog by manipulating (creating, reading, updating or deleting) domain objects, and registering with signals on those objects to provide interactive behavior. Each native Techlog domain object inherits **connect** and

disconnect from the **DomainObject** base class. These methods allow you respectively to subscribe to and unsubscribe from a signal on the **DomainObject**.

```
class DomainObject : public IPrintable
{
public:
    ...
    enum EventType
    {
     DomainObjectChanged,
     DomainObjectErased,
     DomainObjectRenamed
    };

    void connect (const EVENT_TYPE eventType, QObject * receiver,
         const char *slot) const;
    void disconnect (const EVENT_TYPE eventType,
         QObject * receiver, const char *slot) const;
};
```

All native Techlog domain objects implement **DomainObjectChanged**, **DomainObjectErased** and **DomainObjectRenamed** event types and publish **Changed**, **Deleted** and **Renamed** signals to allow you to monitor lifecycle activity. Each **DomainObject** type that exposes any signal has its own **EventType**.

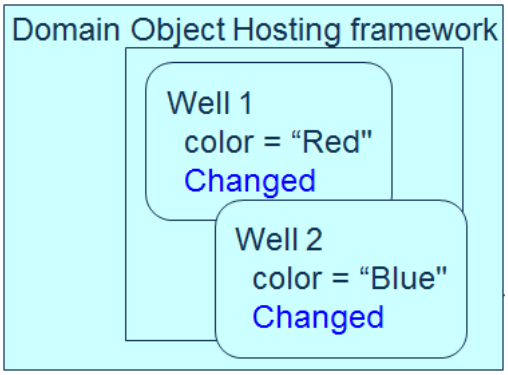


Figure 3-4 Changed Event

Include signal arguments and declare slot receivers in the activity header file.

```
#include "tsdkdomainobjectchangedargs.h"
#include "tsdkdomainobjecterasedargs.h"
#include "tsdkdomainobjectrenamedargs.h"
public slots:
  void onWellChanged(
  const Slb::Ocean::Techlog::DomainObjectChangedArgs &args);

  void onWellErased(
  const Slb::Ocean::Techlog::DomainObjectErasedArgs &args);

  void onWellRenamed(
```

```
  const Slb::Ocean::Techlog::DomainObjectRenamedArgs &args);
```

Subscribe to the signals on a domain object instance.

```
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, well);

well.connect(Well::DomainObjectChanged, this,
SLOT(onWellChanged(const
Slb::Ocean::Techlog::DomainObjectChangedArgs &)));

well.connect(Well::DomainObjectErased, this,
SLOT(onWellErased(const
Slb::Ocean::Techlog::DomainObjectErasedArgs &)));

well.connect(Well::DomainObjectRenamed, this,
SLOT(onWellRenamed(const
Slb::Ocean::Techlog::DomainObjectRenamedArgs &)));

lock.release();
```

Unsubscribe from signals.

```
Well well = createMyWell();

Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, well);

well.disconnect(Well::DomainObjectChanged, this,
SLOT(onWellChanged(const
Slb::Ocean::Techlog::DomainObjectChangedArgs &)));

well.disconnect(Well::DomainObjectErased, this,
SLOT(onWellErased(const
Slb::Ocean::Techlog::DomainObjectErasedArgs &)));

well.disconnect(Well::DomainObjectRenamed, this,
SLOT(onWellRenamed(const
Slb::Ocean::Techlog::DomainObjectRenamedArgs &)));

lock.release();
```

Subscribing to a signal or unsubscribing from a signal modifies the domain object. This is why it must be done in a lock block where you add, acquire and release the domain object.

Ocean **`DomainObject connect`** and **`disconnect`** functions are build on top of the Qt4 signals and slots connection / disconnection mechanism. Which means that signals and slots connection / disconnection syntax done through those function aren't verified at the build time. The slot receiver function passed through the **`SLOT`** macro to **`connect`** and **`disconnect`** functions is viewed by the compiler as a string character. If the signature of the slot receiver function is wrong this generates only an error at the plug-in runtime.

In order to verify connections / disconnections of Ocean domain objects at the build time, you can use the new signal and slot connection syntax provided by Qt5 and exposed with Ocean through **`CONNECT`** and **`DISCONNECT`** macros.

```
CONNECT(emitter, SIGNAL_OBJECT_CLASS, signal, receiver,
SLOT_OBJECT_CLASS, slot)


DISCONNECT(emitter, SIGNAL_OBJECT_CLASS, signal, receiver,
SLOT_OBJECT_CLASS, slot)
```

Subscribe to the signals on a domain object instance.

```
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, well);


CONNECT(well, DomainObject, DomainObjectChanged, this, Activity,
onWellChanged);


CONNECT(well, DomainObject, DomainObjectErased, this, Activity,
onWellErased);


CONNECT(well, DomainObject, DomainObjectRenamed, this, Activity,
onWellRenamed);


lock.release();
```

Unsubscribe from signals.

```
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, well);


DISCONNECT(well, DomainObject, DomainObjectChanged, this,
Activity, onWellChanged);


DISCONNECT(well, DomainObject, DomainObjectErased, this, Activity,
onWellErased);


DISCONNECT(well, DomainObject, DomainObjectRenamed, this,
Activity, onWellRenamed);


lock.release();
```

DomainObjectChanged signal

The **DomainObjectChanged** signal has a **DomainObjectChangedArgs** argument that gives the object changed.

```
class DomainObjectChangedArgs : public SignalArgsT<DomainObject>
{
};
```

The slot handler accesses the needed information from the signal argument.

```
void Activity::onWellChanged(const
Slb::Ocean::Techlog::DomainObjectChangedArgs &args)
{
  DomainObject domainObject = args.sender();
```

```
  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
domainObject);

  Well well = domainObject.cast<Well>();

  QColor wellColor = well.color();

  lock.release();
}
```

DomainObjectErased signal

Similarly, the **DomainObjectErased** signal has a **DomainObjectErasedArgs** argument that gives the affected object.

```
class DomainObjectErasedArgs : public SignalArgsT<DomainObject>
{
};
```

The slot handler accesses the needed information from the signal argument. The signal argument returns the sender which is the **DomainObject** that has been erased.

Note: In the slot receiver, the only property accessible from the erased **DomainOject** is its **Droid**. Trying to access any other properties throws a plug-in exception.

```
void Activity::onWellErased(const
Slb::Ocean::Techlog::DomainObjectErasedArgs &args)
{
  DomainObject domainObject = args.sender();

  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
domainObject);

  Well well = domainObject.cast<Well>();

  Droid wellDroid = well.droid();

  lock.release();
}
```

An asynchronous **DomainObjectErased** signal is raised at the end of the lock release which contains the erased domain object.

DomainObjectRenamed signal

The **DomainObjectRenamed** signal is only emitted when domain object name has changed.

Note: You connect this signal to domain objects that support a name.

The signal has a **`DomainObjectRenamedArgs`** argument that gives the affected object.

```
class DomainObjectRenamedArgs : public SignalArgsT<DomainObject>
{
};
```

The slot handler accesses the needed information from the signal argument.

```
void Activity::onWellRenamed(const
Slb::Ocean::Techlog::DomainObjectRenamedArgs &args)
{
  DomainObject domainObject = args.sender();

  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
domainObject);

  Well well = domainObject.cast<Well>();

  QString newName = well.name();

  lock.release();
}
```

DomainObjectCollection

The find and get patterns are also accessible from a **`DomainObjectCollection`**. As suggested by its name this class represents a collection of domain objects from which you retrieve a domain object through get and find patterns, passing either the domain object **`Droid`** or the domain object name as a **`QString`** value.

```
class DomainObjectCollection : public IPrintable
{
public:
  int count() const;
  bool contains( const T & t ) const ;
  bool contains( const Droid & droid ) const;
  bool contains( const QString & name ) const;
  T find( const QString & name ) const;
  T get( const QString & name ) const;
  T find( const Droid & droid ) const;
  T get( const Droid & droid ) const ;
   T at( const int & index ) const;
  T first() const;
  virtual QString toString() const override;
  QList<T> toList() const;
  QList<TFILTER> toListFilteredByType() const;
```

```
  bool isEmpty() const;
};
```

This example uses the **DomainObjectCollection**.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

// Get the main project from the current session
Project project = Session::current().mainProject();
Workspace workspace = Session::current().currentWorkspace();
// try to retrieve the domain object on server side
DomainObjectCollection<Well> wellCollection = project.wells();
Well well = wellCollection.find("MyWell");
if (well.isNull())
  workspace.logEvent(LogLevelWarning, "MyWell does not exist");

lock.release();
```

Domain object collection access best practice

Ocean follows the smart pointer mechanism, which cleans an object from the plug-in cache as soon as the plug-in no longer has a strong reference to this object.

Accessing the same collection of objects many times, but without maintaining a reference to those objects in their current scope can result in performance degradation.

This example may be affected by this performance issue:

```
Well well = createMyWell();

Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, well);

QList<QString> vList;
vList << QLatin1String("var1") << QLatin1String("var2") <<
QLatin1String("var3") << QLatin1String("var4") <<
QLatin1String("var5");

foreach(const QString variableName, vList)
{
  //The collection of datasets should have been loaded only once
  //Now, the collection is loaded for each variableName
  //because no reference is kept on this collection between two
iterations
  foreach(const Dataset dataset, well.datasets())
  {
    //Same thing here with the collection of variables
    if (dataset.variables().contains(variableName))
      performXYZ(well, dataset);
  }
}
```

```
lock.release();
```

The workaround is to create a 'cache' on the plug-in side:

```
Well well = createMyWell();

Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, well);

// caching domain object collections on the plug-in side
DomainObjectCollection<Dataset> datasetCollection =
well.datasets();
QList<DomainObjectCollection<Variable>> variableCollectionList;
foreach(const Dataset dataset, well.datasets())
{
  variableCollectionList << dataset.variables();
}

QList<QString> vList;
vList << QLatin1String("var1") << QLatin1String("var2") <<
QLatin1String("var3") << QLatin1String("var4") <<
QLatin1String("var5");

foreach(const QString variableName, vList)
{
  int datasetIndex = 0;
  // As long as datasetCollection and variableCollectionList stay
in the scope there is no performance degradation.
  foreach(const Dataset dataset, datasetCollection)
  {
    if
(variableCollectionList.at(datasetIndex).contains(variableName
))
      performXYZ(well, dataset);

    datasetIndex++;
  }
}

lock.release();
```

For each Ocean object with child domain objects that support name, use find and get child methods accessible directly in the parent class. The methods improve the performance of getting a domain object from its parent by its name.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
Project project = Session::current().mainProject();

DomainObjectCollection<Well> wellCollection = project.wells();
Well well = wellCollection.find("MyWell");
```

```
// OR
// Best performance using findWell method instead to reload the
whole collection in order to retrieve a well by its name
well = project.findWell("MyWell");

lock.release();
```

Filter by domain object type

A domain object collection can contain several domain object types grouped under a domain object base class. For instance, the graphics item collection may contain different type of shapes.

See "GraphicsItem domain objects" section in *Ocean for Techlog Developer Guide - Plots* for more information on different shape types available with Ocean.

From the collection you query a specific domain object type through the templated function **`toListFilteredByType`**.

```
class DomainObjectCollection : public IPrintable
{
public:
    ...
    QList<TFILTER> toListFilteredByType() const;
};
```

This example uses **`toListFilteredByType`**.

```
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
graphicsScene);

DomainObjectCollection<GraphicsItem> graphicsItems =
graphicsScene.items();

QList<Rectangle2d> rectangles =
graphicsItems.toListFilteredByType<Rectangle2d>();

qWarning() << "Number of rectangles in the graphics item collection
is " << rectangles.count();

lock.release();
```