Activité : déploiement d'une application (installateur)

Thierry Vaira <tvaira@free.fr>

9 décembre 2015

Table des matières

Site : tvaira.free.fr

Déploiement d'une application (installateur)

Expression du besoin

Dans le cadre du **déploiement** d'une application, on a besoin de réaliser un programme **d'installation** de celle-ci.

Un installateur (ou installeur) désigne un programme permettant d'installer un logiciel. Ces programmes interagissent généralement avec l'utilisateur à l'aide de boîtes de dialogue. Leur but est de procéder automatiquement à la copie sur l'ordinateur cible (et parfois au paramétrage) des programmes et fichiers composant le logiciel, afin d'assurer son bon fonctionnement.

Déploiement

Pour réaliser une procédure d'installation d'une application, il faudra décomposer celle-ci en trois parties :

- fabriquer une version finale (*release*) de l'application (`make`) en édition statique ou dynamique
- fabriquer un installateur pour l'application (`setup`)
- déployer cette application sur une machine

Ces trois parties sont parfois dépendantes de la plateforme et des outils utilisés.

Une autre solution consisterait à réaliser un **paquet** ou paquetage (*package*) pour la distribution de votre choix.

Dans le contexte des systèmes UNIX, on appelle paquet (ou parfois paquetage, en anglais package) une archive (fichier compressé) comprenant les fichiers informatiques, les informations et procédures nécessaires à l'installation d'un logiciel sur un système d'exploitation au sein d'un agrégat logiciel, en s'assurant de la cohérence fonctionnelle du système ainsi modifié. Actuellement, on distingue deux types de paquets : deb qui est le format des paquets logiciels de la distribution Debian GNU/Linux (et presque toutes les distributions basées sur Debian) et rpm (Red Hat Package Manager) qui est un format ouvert initié par la distribution Red Hat GNU/Linux.

Voir aussi :

- Tutoriel RPM
- Tutoriel makeself
- Tutoriel INNO SETUP

Fabriquer une version finale de l'application (release)

De manière générale, il faut résoudre un certain nombre de problèmes et répondre à quelques choix.

- Droits et licence ? Les droits et les fichiers à déployer (consulter le contrat de licence de Qt par exemple) et les droits et les fichiers à appliquer à son application. Lire : www.gnu.org/licenses/.
- Statique ou dynamique ? Il faudra choisir entre fabriquer un exécutable indépendant (statique) ou non (dynamique). Sous Linux, les dépendances sont nombreuses : architecture multi-platerforme, bibliothèques, etc ... Lire : linux-deployment.html.
- Librairies dynamiques ? Les bibliothèques dynamiques contiennent du code exécutable (des fonctions formant une API) qui sera susceptible d'être utilisé par un (ou plusieurs) programmes au moment de leur exécution. En cas de besoin, la librairie dynamique sera chargée en mémoire et son code sera alors utilisable par le programme demandeur. Il en résulte les avantages suivants : la taille du programme est réduite (puisque le code dont il a besoin se trouve dans la librairie), la possibilité de faire évoluer la librairie sans avoir à recompiler (si le prototype des fonctions définies dans la librairie reste inchangé). L'inconvénient majeur reste l'obligation de la présence de la librairie sur le système cible pour que le programme puisse s'exécuter. L'extension usuelle d'une librairie dynamique sous Linux est `.so` (pour Windows, ce sera `.dll`)
- Dépendances ? Il y a plusieurs techniques pour connaître les dépendances externes d'une application : utiliser un utilitaire qui recherche ces dépendances (la commande `ldd` sous Linux) ou appliquer une démarche (rudimentaire !) qui teste l'application sur une machine vierge (une machine virtuelle par exemple) et qui résout les dépendances les unes après les autres.
- L'emplacement ? Il faudra déterminer où installer l'application (dans les chemins partagés `/usr/bin/` et `/usr/local/bin`, séparés dans `/opt/` ou dans `$HOME` pour Linux). On peut aussi envisager de créer une entrée dans le menu Application de l'environnement de bureau et aussi un raccoruci.
- Les fichiers de l'application à déployer ? Il faut évidemment recenser les fichiers (ainsi que l'arborescence) qui composent l'application et qui devront être déployer avec celle-ci. De manière générale, on trouve : l'exécutable, l'icône (`.ico`), des fichiers de configuration (comme les `.ini`), des fichier multimédia (comme des images), des fichiers d'aide (comme les `.chm` ou les pages `man`, des fichiers `.html`, ...), un fichier licence, un fichier readme, etc ...

On désire au final regrouper toutes ces actions et tous ces fichiers dans un même et seul fichier exécutable : l'installateur.

Fabriquer un installateur avec qt

Pour le déploiement de l'application, notre choix se porte sur l'utilitaire fourni par Qt : Qt-Installer-Framework.

Le Qt-Installer-Framework fournit un ensemble d'outils et utilitaires pour créer des installateur pour les plates-formes Linux, Microsoft Windows, et Mac OS X. Il est utilisé notamment pour les installateurs de Qt Creator et du SDK de Qt.

Généralement, les logiciels s'installent via un installateur qui prend en charge de manière automatique l'installation. Tout au plus l'utilisateur devra répondre à quelques questions à choix multiples. Cette procédure est plus ou moins complexe selon l'ordinateur et le système d'exploitation.

Téléchargement : https ://download.qt.io/

Vous devez avoir installé au préalable l'archive `qt-installer-framework-opensource-2.0.0-x64.run`. Vous pouvez l'installer par exemple dans votre répertoire personnel (`$HOME`) :

```
$ wget https://download.qt.io/official_releases/qt-installer-framework/2.0.0/qt-installer-framework-opensou
$ chmod +x qt-installer-framework-opensource-2.0.0-x64.run
$ ./qt-installer-framework-opensource-2.0.0-x64.run
```

Les fichiers ont été installés dans `$HOME/Qt/QtIFW2.0.0`. La documentation se trouve dans le répertoire `doc` et des exemples dans `examples`.

Documentation : doc.qt.io/qtinstallerframework/

Le principe d'une installation avec Qt-Installer-Framework suivra le cycle ci-dessous :

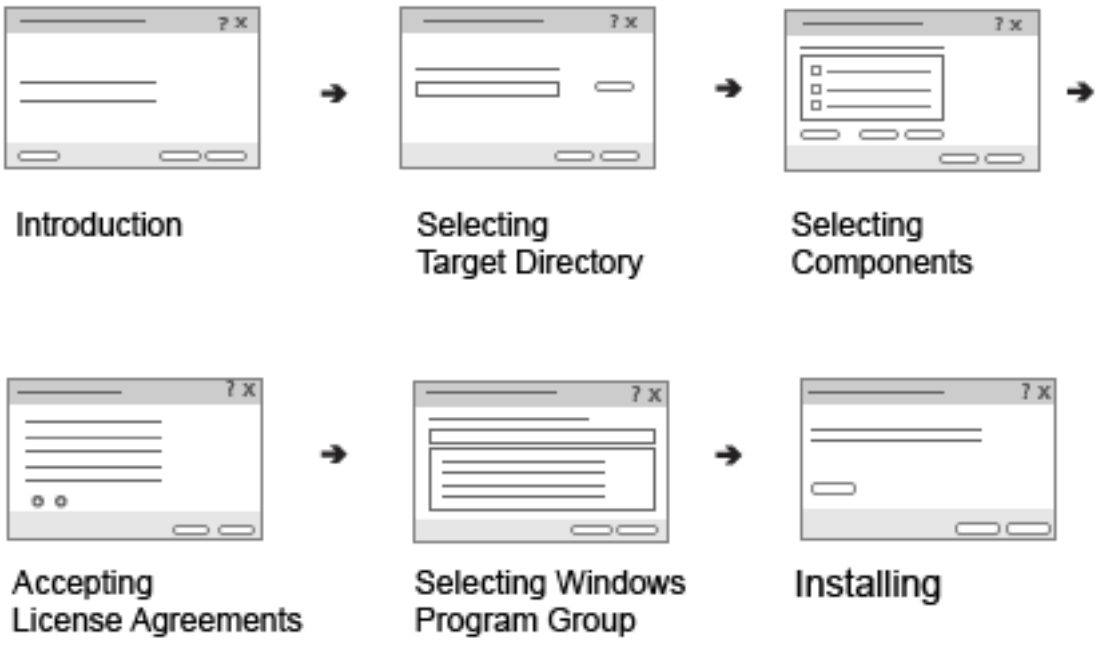

FIGURE *1*

Une fois notre projet préparé et configuré, il suffira de créer l'installateur avec l'outil `binarycreator`.

Exemple n°1

On désire créer un installateur `setup` qui déploiera le(s) fichier(s) suivant(s) : l'exécutable `helloworld`.

On va commencer par créer la racine du projet pour notre installateur :

```
$ cd $HOME/Qt/QtIFW2.0.0
$ mkdir helloworld
$ cd helloworld
```

On doit ensuite créer les deux branches suivantes :

– le répertoire `config` qui contiendra le fichier de configuration générale `config.xml`
– le répertoire `packages` qui contiendra tous les fichiers de configuration et les composants de votre application à installer

On va commencer par créer le fichier de configuration générale `config.xml` :

```
$ cd $HOME/Qt/QtIFW2.0.0/helloworld
$ mkdir config
$ cd config
$ touch config.xml
```

On édite maintenant le fichier `config.xml` afin d'y paramétrer les éléments essentiels :

```
<?xml version="1.0" encoding="UTF-8"?>
<Installer>
    <Name>Le nom</Name>
    <Version>1.0.0</Version>
    <Title>Un titre</Title>
    <Publisher>tv</Publisher>
    <ProductUrl>http://tvaira.free.fr</ProductUrl>
    <RunProgram>@TargetDir@/helloworld</RunProgram>
    <MaintenanceToolName>uninstaller</MaintenanceToolName>
    <StartMenuDir>Helloworld</StartMenuDir>
    <TargetDir>@HomeDir@/helloworld</TargetDir>
</Installer>
```


Remarque : ici on choisit de modifier le nom par défaut du déinstallateur (`maintenancetool` en `uninstaller`). Si vous inséréez l'élément RunProgram, l'installateur vous proposera de lancer votre exécutable en fin d'installation. Le reste est classique. Le détail de l'ensemble des éléments utilisables dans un fichier `config.xml` sont décrits ici.

Dans notre exemple, on ne gérera qu'un seul composant à installer. Sinon, chaque composant à installer doit avoir un répertoire dans la branche `packages`.

```
$ cd $HOME/Qt/QtIFW2.0.0/helloworld
$ mkdir packages
$ cd packages
$ mkdir helloworld.composant1
$ cd helloworld.composant1
```

Un composant possèdera généralement deux répertoires :

- le répertoire `meta` qui contiendra le fichier de configuration du composant nommé `package.xml` et éventuellement des scripts à exécuter au moment de l'installation (voir exemple n°2)
- le répertoire `data` qui contiendra les fichiers à installer

On va créer l'arborescence pour ce composant :

```
$ cd $HOME/Qt/QtIFW2.0.0/helloworld/packages/helloworld.composant1
$ mkdir data
$ mkdir meta
$ cd meta
$ touch package.xml
```

On édite ensuite le fichier de configuration du composant `package.xml` :

```
<?xml version="1.0"?>
<Package>
    <DisplayName>helloworld</DisplayName>
    <Description>bla bla ...</Description>
    <Version>1.0.0</Version>
    <ReleaseDate>2015-12-02</ReleaseDate>
    <Default>true</Default>
    <Licenses>
        <License name="GNU GENERAL PUBLIC LICENSE Version 3" file="gpl3.txt"/>
    </Licenses>
</Package>
```

Ici, on ajoute une étape concernant la licence. Le fichier `gpl3.txt` doit être placé dans le répertoire `meta` et sera affiché au moment de l'installation.

C'est le moment de placer les fichiers à déployer (pour ce composant) dans le dossier `data`. Il seront copiés dans le répertoire **TargetDir** pendant l'installation.

```
$ ls -l $HOME/Qt/QtIFW2.0.0/helloworld/packages/helloworld.composant1/data
-rwxr-xr-x 1 tv tv 387314 avril 17  2010 helloworld
```

Pour finaliser la préparation, on va créer un fichier de projet Qt `.pro` pour automatiser la fabrication de l'installateur :

```
$ cd $HOME/Qt/QtIFW2.0.0/helloworld/
$ touch helloworld.pro
```

On édite ce fichier en affectant quelques variables de QMake :

```
TEMPLATE = aux

INSTALLER = setup

INPUT = $$PWD/config/config.xml $$PWD/packages
```



```
exemple.input = INPUT
exemple.output = $$INSTALLER
exemple.commands = ../bin/binarycreator -c $$PWD/config/config.xml -p $$PWD/packages ${QMAKE_FILE_OUT}
exemple.CONFIG += target_predeps no_link combine

QMAKE_EXTRA_COMPILERS += exemple
```

`Veillez à bien indique le chemin exact vers l'utilitaire binarycreator !`

Au final, on a donc l'arborescence suivante :

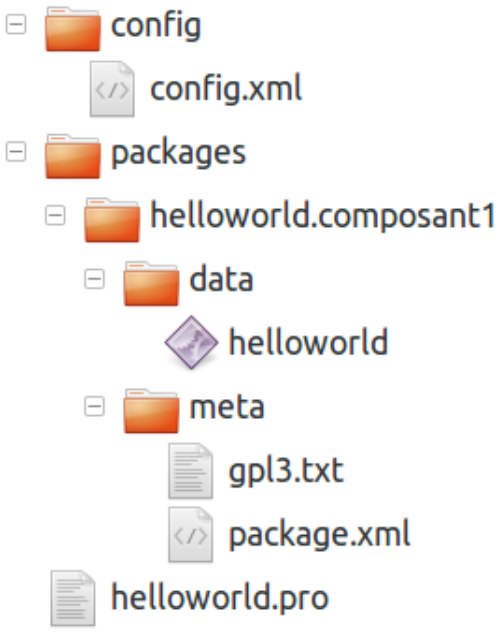

FIGURE *2*

On est prêt à créer l'installateur de nom `setup` ici :

```
$ qmake
$ make
  ../bin/binarycreator -c /home/tv/Qt/QtIFW2.0.0/helloworld/config/config.xml
                       -p /home/tv/Qt/QtIFW2.0.0/helloworld/packages setup
  rm -f helloworld
```

On peut tester notre installateur `setup` :

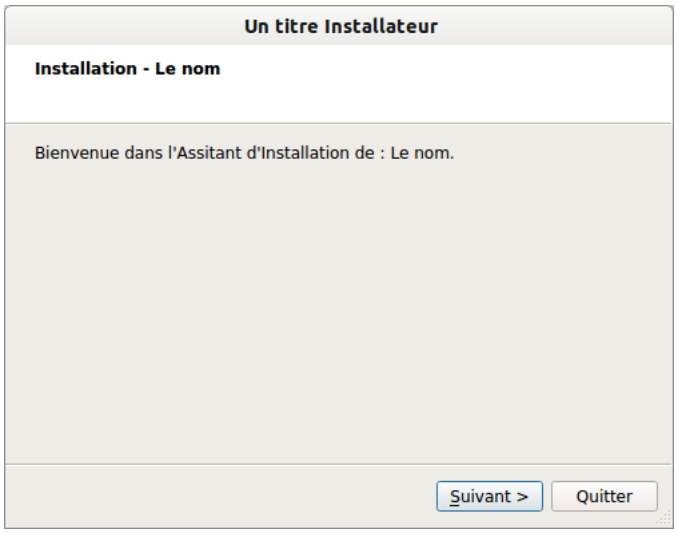

FIGURE *3*

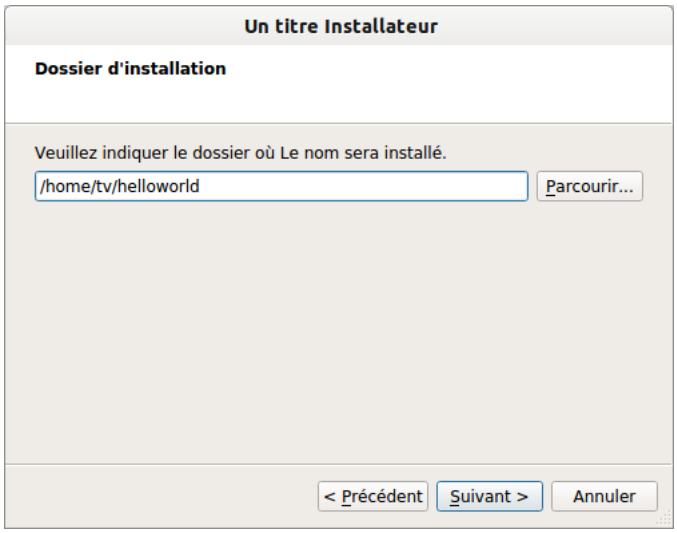

FIGURE 4

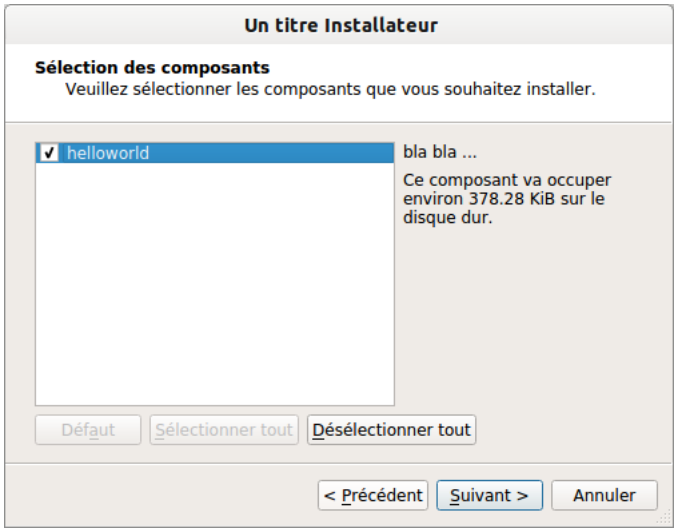

FIGURE 5

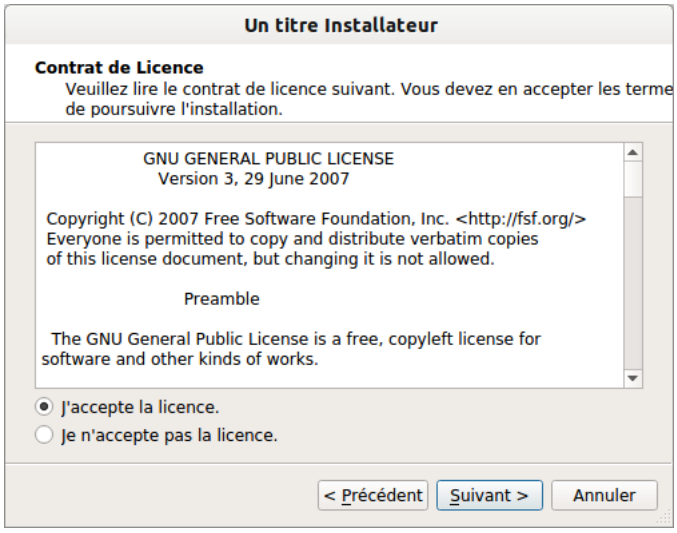

FIGURE 6

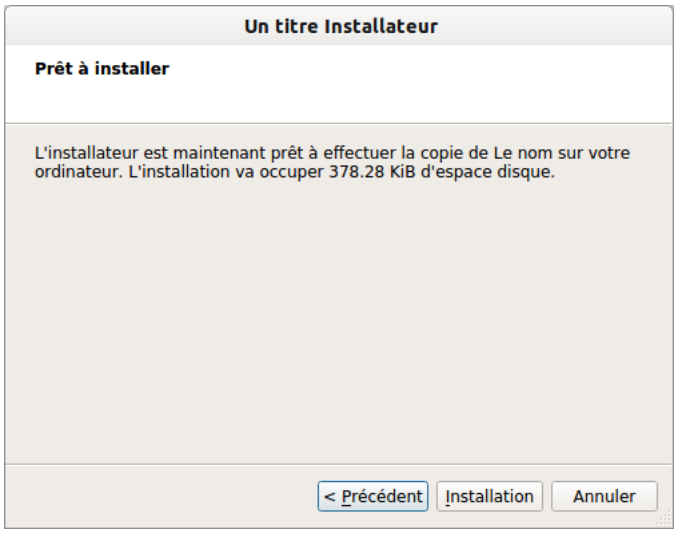

FIGURE 7

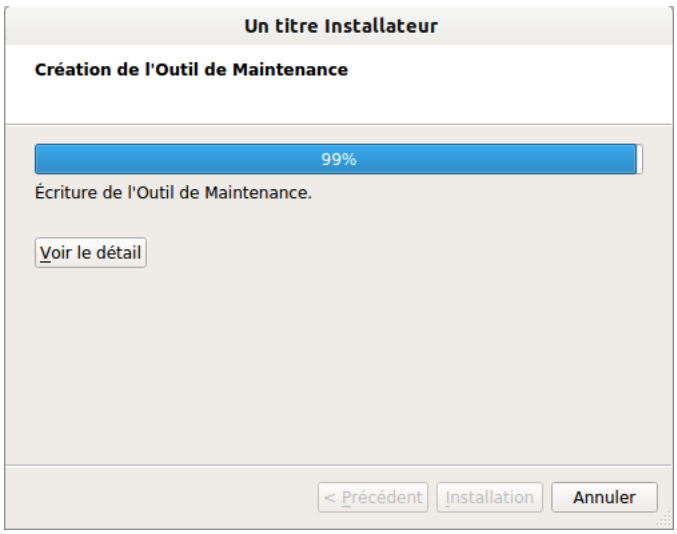

FIGURE 8

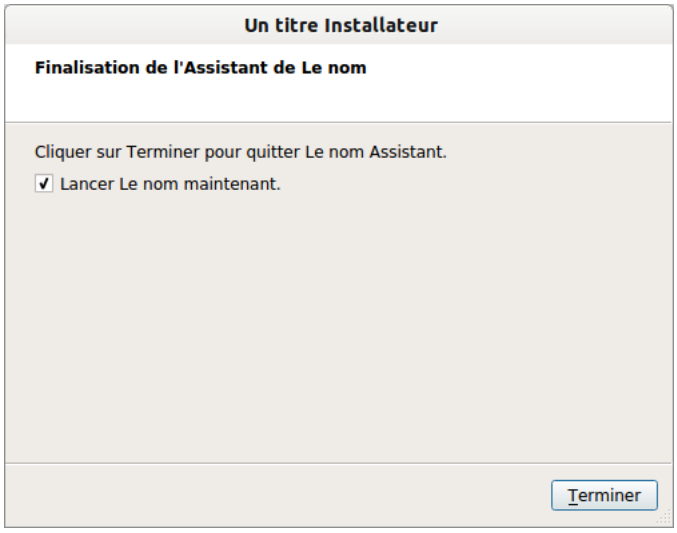

FIGURE 9

L'application `helloworld` est bien lancée :

FIGURE *10*

Le répertoire où a été installée notre application :

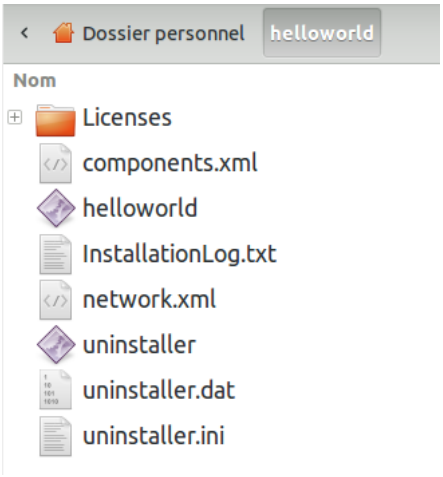

FIGURE *11*

Un désinstallateur a aussi été créé :

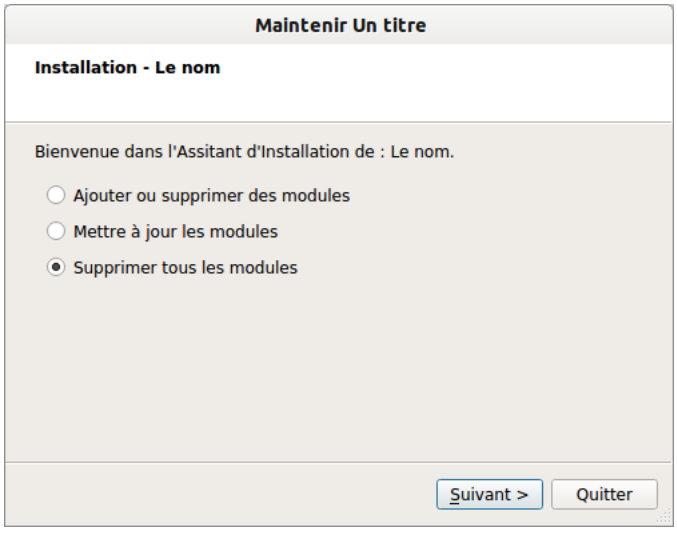

FIGURE *12*

FIGURE *13*

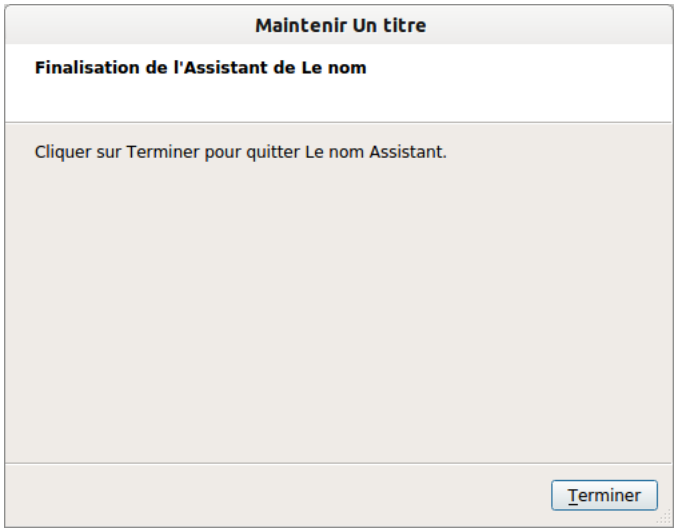

FIGURE *14*

Exemple n°2

Ce deuxième exemple va permettre d'utiliser les possibilités offertes par `Qt-Installer-Framework` pour vérifier si l'application peut être installée sur la machine cible.

Globalement, `Qt-Installer-Framework` permet de personnaliser la procédure d'installation en intégrant des scripts en JavaScript (avec le moteur `QJSEngine`) pour ajouter des opérations.

Ici, l'exécutable à déployer a été fabriqué et testé sur une plateforme 64 bits avec Ubuntu 12.04 LTS. On va donc vérifier ces paramètres au moment de l'installation avant de prévenir l'utilisateur.

On copie l'arborescence de l'exemple précédent pour la modifier ensuite :

```
$ cd $HOME/Qt/QtIFW2.0.0/
$ cp -r helloworld helloworld2
$ cd helloworld2
```

On peut consulter la documentation sur l'écriture de script dans `Qt-Installer-Framework`.

Pour chaque composant, il est possible de spécifier un script (ici `installscript.qs`) qui prépare les opérations à effectuer par l'installateur. Pour cela, il faut modifier le fichier `package.xml` pour ajouter l'élément `Script` :

```
$ cd $HOME/Qt/QtIFW2.0.0/helloworld2/
```



```
$ cd packages/helloworld.composant1/meta
$ touch installscript.qs
```

Puis on édite le fichier `package.xml` :

```
<?xml version="1.0"?>
<Package>
    <DisplayName>helloworld</DisplayName>
    <Description>bla bla ...</Description>
    <Version>1.0.0</Version>
    <ReleaseDate>2015-12-02</ReleaseDate>
    <Default>true</Default>
    <Licenses>
        <License name="GNU GENERAL PUBLIC LICENSE Version 3" file="gpl3.txt"/>
    </Licenses>
    <Script>installscript.qs</Script>
</Package>
```

Le script `installscript.qs` doit contenir un objet `Component` que le programme d'installation crée quand il chargera le script. Par conséquent, le script doit contenir au moins la fonction `Component()` qui effectuera l'initialisation (comme la connexion signaux et slots).

Le script ci-dessous utilise la variable prédéfinie `systemInfo` qui fournit des informations sur l'OS (`kernelType`, `kernelVersion`, `productType`, `productVersion`, ...).

```
function cancelInstaller(message)
{
    installer.setValue("Erreur", "true");
    // Pour désactiver le "Lancer ... maintenant"
    var widget = gui.pageById(QInstaller.InstallationFinished);
    if (widget != null)
    {
        widget.RunItCheckBox.checked = false;
    }
    installer.setDefaultPageVisible(QInstaller.Introduction, false);
    installer.setDefaultPageVisible(QInstaller.TargetDirectory, false);
    installer.setDefaultPageVisible(QInstaller.ComponentSelection, false);
    installer.setDefaultPageVisible(QInstaller.ReadyForInstallation, false);
    installer.setDefaultPageVisible(QInstaller.StartMenuSelection, false);
    installer.setDefaultPageVisible(QInstaller.PerformInstallation, false);
    installer.setDefaultPageVisible(QInstaller.LicenseCheck, false);

    var abortText = "<font color='red'>" + message +"</font>";
    installer.setValue("FinishedText", abortText);
}

function majorVersion(str)
{
    return parseInt(str.split(".", 1));
}

function Component()
{
    console.log("OS: " + systemInfo.productType);
    console.log("Version: " + systemInfo.productVersion);
    console.log("Kernel: " + systemInfo.kernelType + "/" + systemInfo.kernelVersion);

    var validOs = false;
    installer.setValue("Erreur", "false");

    if (systemInfo.kernelType === "winnt")
```



```
    {
        if (majorVersion(systemInfo.kernelVersion) >= 6)
            validOs = false;
    }
    else if (systemInfo.kernelType === "darwin")
    {
        if (majorVersion(systemInfo.kernelVersion) >= 11)
            validOs = false;
    }
    else
    {
        if (systemInfo.productType !== "ubuntu" || systemInfo.productVersion !== "12.04")
        {
            QMessageBox["warning"]("os.warning", "Installer",
                                   "Ce programme n'a été testé que sur Ubuntu 12.04.5 LTS.",
                                   QMessageBox.Ok);
        }
        validOs = true;
    }

    if (!validOs)
    {
        cancelInstaller("L'installation sur " + systemInfo.prettyProductName + " n'est pas supportée");
        return;
    }

    console.log("CPU Architecture: " +  systemInfo.currentCpuArchitecture);

    if (systemInfo.currentCpuArchitecture !== "x86_64")
    {
        var result = QMessageBox["critical"]("architecture.critical", "Installateur",
                                   "Ce programme ne fonctionne que sur une architecture x86_64.",
                                   QMessageBox.Ok);
        if (result == QMessageBox.Ok)
        {
            cancelInstaller("L'installation sur " + systemInfo.currentCpuArchitecture + " n'est pas supporte
            return;
       }
    }

    // On peut aussi tester la présence de fichiers
    /*if (!installer.fileExists("/usr/lib/libpcan.so"))
    {
        var result = QMessageBox["critical"]("library.critical", "Installateur",
                                   "Ce programme necessite la présence de la bibliothèque pcan.",
                                   QMessageBox.Ok);
    }*/

    console.log("Erreur: " + installer.value("Erreur"));
}
```

On fabrique l'installateur :

```
$ cd $HOME/Qt/QtIFW2.0.0/helloworld2/
$ make clean
$ qmake
$ make
```

Et on le lance :

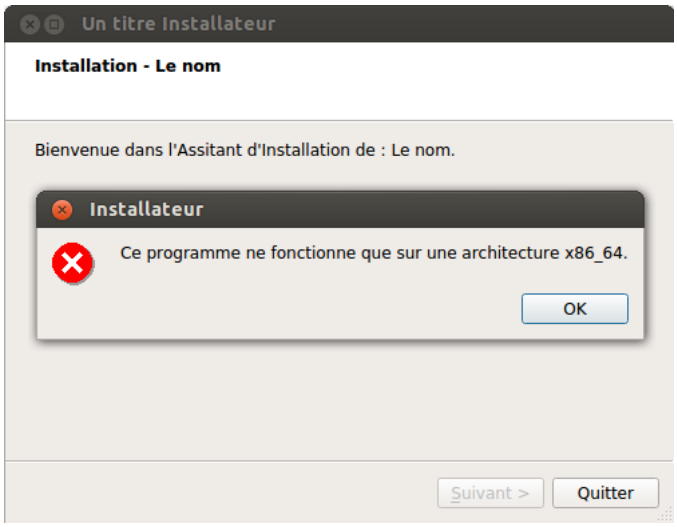

FIGURE *15*

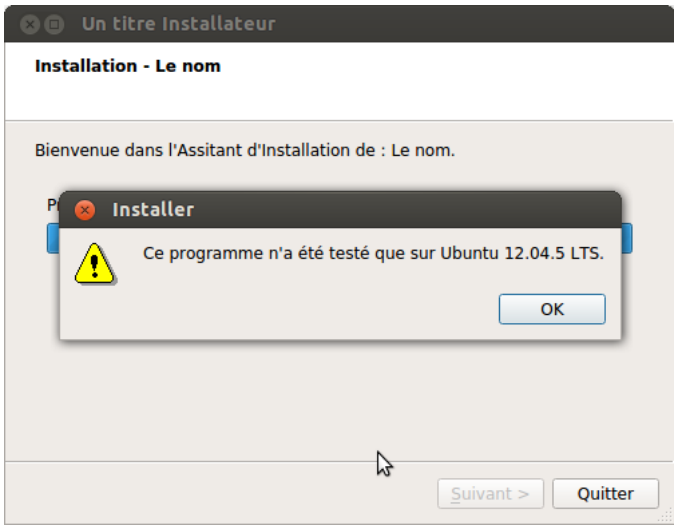

FIGURE *16*

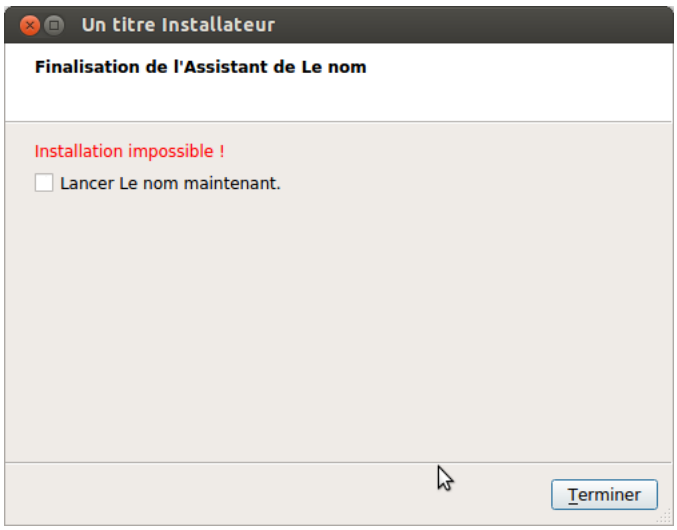

FIGURE *17*

Exemple n°3

Ce troisième exemple va permettre d'utiliser une autre possibilité offerte par `Qt-Installer-Framework` pour personnaliser graphiquement l'installateur.

Globalement, `Qt-Installer-Framework` permet de personnaliser la procédure d'installation en ajoutant des fenêtres créées avec QtDesigner et intégrées à l'installateur (voir l'exemple changeuserinterface).

Ici, on va intégrer dans notre installateur l'exemple README fourni par Qt.

On copie l'arborescence de l'exemple précédent pour la modifier ensuite :

```
$ cd $HOME/Qt/QtIFW2.0.0/
$ cp -r helloworld2 helloworld3
$ cd helloworld3
```

On va ajouter à l'arborescence un composant :

```
$ cd $HOME/Qt/QtIFW2.0.0/helloworld3/packages/
$ mkdir helloworld.readme
$ cd $HOME/Qt/QtIFW2.0.0/helloworld3/packages/helloworld.readme
$ mkdir data
$ cd data
$ touch readme.html
$ cd ..
$ mkdir meta
$ cd meta
$ touch package.xml
```

On édite ensuite le fichier de configuration du composant `package.xml` :

```
<?xml version="1.0" encoding="UTF-8"?>
<Package>
    <DisplayName>README</DisplayName>
    <Description>Manuel d'utilisation</Description>
    <Version>1.0.1</Version>
    <ReleaseDate>2015-12-02</ReleaseDate>
    <AutoDependOn>helloworld.composant1</AutoDependOn>
    <Script>installscript.qs</Script>
    <UserInterfaces>
        <UserInterface>readmecheckboxform.ui</UserInterface>
    </UserInterfaces>
    <SortingPriority>1</SortingPriority>
</Package>
```

On crée une autodépendance (`<AutoDependOn>`) avec le composant principal `helloworld.composant1`, on ajoute la fenêtre GUI (`<UserInterface>`) `readmecheckboxform.ui` crée avec Qt Designer :

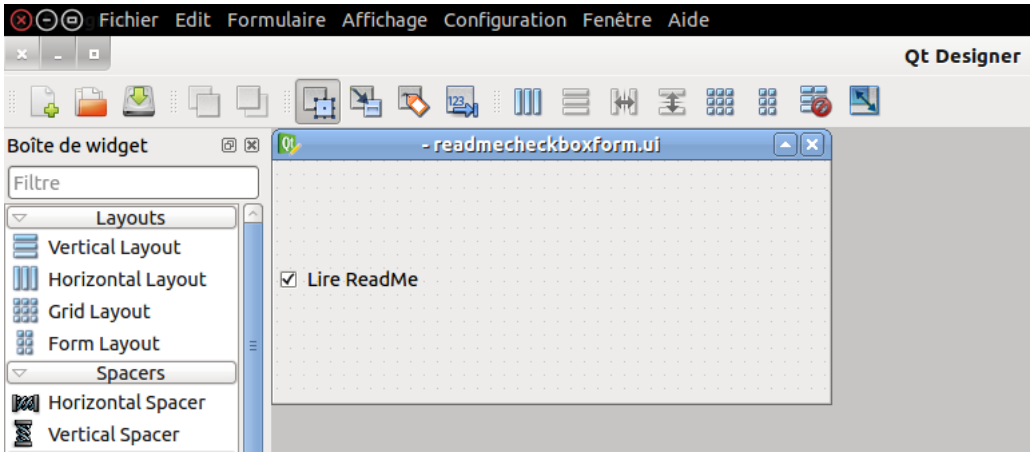

FIGURE *18*

L'élément `SortingPriority` permet de fixer un niveau de priorité dans les composants qui permet surtout de contrôler l'ordre d'affichage dans la fenêtre de sélection des composants.

On termine par ajouter le script `installscript.qs` qui gèrera cette nouvelle étape dans l'installation. On crée une fonction `installationFinished` qui vérifiera si la case a été cochée puis lancera l'ouverture du fichier README (ici une page HTML) avec l'appel à `QDesktopServices.openUrl()` :

```
$ cd $HOME/Qt/QtIFW2.0.0/helloworld3/packages/helloworld.readme/meta
$ touch installscript.qs
```

Puis on édite le fichier `installscript.qs` :

```
function Component()
{
    // En cas d'erreur dans l'installation, on décoche la case
    var erreur = installer.value("Erreur");
    if (erreur === "true")
    {
        component.userInterface( "ReadMeCheckBoxForm" ).readMeCheckBox.checked = false;
        return;
    }
    installer.installationFinished.connect(this, Component.prototype.installationFinishedPageIsShown);
    installer.finishButtonClicked.connect(this, Component.prototype.installationFinished);
}

Component.prototype.createOperations = function()
{
    component.createOperations();
}

Component.prototype.installationFinishedPageIsShown = function()
{
    try {
        if (installer.isInstaller() && installer.status == QInstaller.Success) {
            installer.addWizardPageItem( component, "ReadMeCheckBoxForm", QInstaller.InstallationFinished )
        }
    } catch(e) {
        console.log(e);
    }
}

Component.prototype.installationFinished = function()
{
    try {
        if (installer.isInstaller() && installer.status == QInstaller.Success) {
            var isReadMeCheckBoxChecked = component.userInterface( "ReadMeCheckBoxForm" ).readMeCheckBox.che
            if (isReadMeCheckBoxChecked)
            {
                QDesktopServices.openUrl("file:///" + installer.value("TargetDir") + "/readme.html");
            }
        }
    } catch(e) {
        console.log(e);
    }
}
```

On fabrique l'installateur :

```
$ cd $HOME/Qt/QtIFW2.0.0/helloworld3/
$ make clean
$ qmake
$ make
```


Et on le lance :

. . .

Le nouveau composant README apparaît dans la fenêtre :

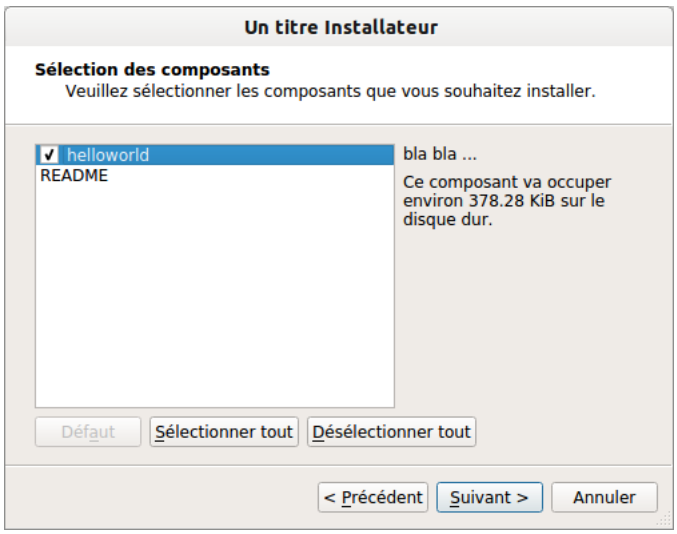

FIGURE *19*

. . .

La case à cocher est intégrée à la fenêtre finale :

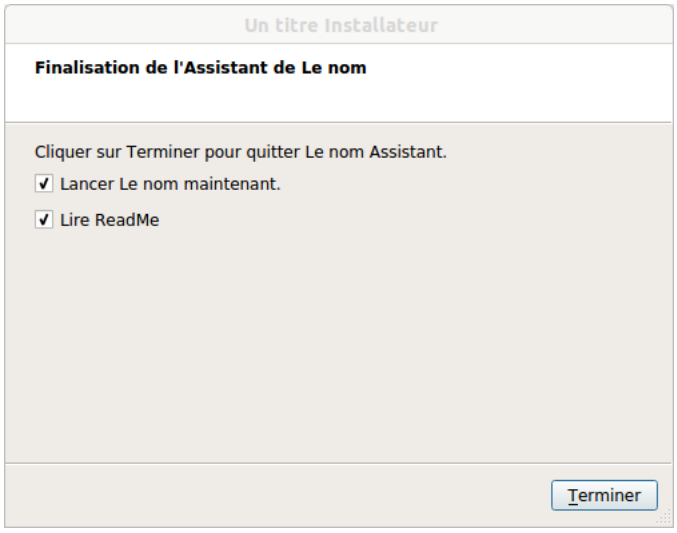

FIGURE *20*

Retour au sommaire

