Getting your application ready for KF6

2022/10/01, Akademy

Nicolas Fella & Alexander Lohnau

Brace yourselves

About Nicolas

- KDE developer since 2017
- Working on almost everything from Frameworks to packaging
- One of the main people behind Qt6 and KF6 porting
- Already talked about this at last Akademy

About Alexander

- KDE Developer since 2020
- Works on Frameworks, Plasma, some apps
- Maintainer of KRunner
- One of the main people behind Qt6 and KF6 porting

Where are we at?

- Qt 6.0 released in 2020
- Qt 6.4 released just the other day
- No KDE Frameworks 6 yet
- Many KDE project already build against Qt6

https://iskdeusingqt6.org/

Why port now?

- If there is no KF6 release, why talk about porting?
- Many things can be done now
- Do things as early as possible
- Identify blockers/missing things in Qt6/KF6

How to approach a port

- Check used Qt/KF5 modules, port away from deprecated modules
- Port away from deprecated Qt/KF5 API
- Prepare build system
- Address remaining build issues
- Add Qt6 CI

Deprecated Modules

- Qt QuickControls 1
- QtScript
- QtGraphicalEffects
- KDELibs4Support
- Kross
- KJS
- KInit
- KHTML
- KDEWebkit

Port away from deprecated API

- Use CMake flags to enable warning and disable deprecated API
- Can be managed through ECMDeprecationSettings

```
ecm_set_disabled_deprecation_versions(
        QT 5.15.2
        KF 5.98.0
        KCOREADDONS 5.92.0 # needed for compatibility
    )
```

Prepare Build System

- Goal is to allow building against Qt5 and Qt6
- Replace 5 with ${QT_MAJOR_VERSION}
- Use version-less targets/functions/variables
- Don’t use version-less targets in libraries
- Avoid deprecated install variables
- KF5 targets are not renamed yet

Qt6 CI

- We have Qt6 CI for all CI platforms
- e.g. linux → linux-qt6
- Also builds without deprecated KF5 API

Plugin Loading – How we used to do it

- Application provides ServiceType definition
- Loading of services KServiceTypeTrader
- Filtering through trader constraint strings
- Services can point to a library, this gets loaded using KPluginLoader
- From the loaded KPluginFactory we create the object
- Multiple plugin of same type registered using keywords

Plugin Loading – How we used to do it

- Multiple ways for plugin representation:
  - KService
  - KPluginInfo
  - KPluginMetaData

Plugin Loading - How we do it now

- Plugins are loaded from specific directories (plugin namespaces)
- KPluginMetaData::findPlugins
- KPluginMetaData::findPluginById
- Metadata is embedded as JSON using the MOC
- Loading of plugins is done with QPluginLoader

Plugin Loading - How we do it now

- Utility methods in KPluginFactory simplify loading

```
KPluginFactory::Result<KPluginFactory> factoryResult = KPluginFactory::loadFactory(metaData);
```

- Result has the following fields:

```
T *plugin = nullptr;
/// translated, user-visible error string
QString errorString;
/// untranslated error text
QString errorText;
ResultErrorReason errorReason = NO_PLUGIN_ERROR;
```

Plugin Loading - How we do it now

- To directly create a plugin object we can use:

```
auto result = KPluginFactory::instantiatePlugin<PackageStructure>(metaData, nullptr, args);
if (!result) {
    // Error handling
}
PackageStructure *structure = result.plugin;
```

Plugin Loading - How to port

- Load plugins from a specific namespace using KPluginMetaData

```
KPluginMetaData::findPlugins(QStringLiteral("myapp/parsers"));
```

- Filtering can be done using optional function

```
KPluginMetaData::findPlugins(QStringLiteral("myapp/parsers"), [myMimeType]
    (const KPluginMetaData &data) {
        return data.supportsMimeType(myMimeType);
    });
```

Plugin Loading - How to port

- Desktop files need to be converted to JSON format
- Either using kcoreaddons_desktop_to_json CMake function
- Or preferably keep the converted json files in VSC:
  - “desktoptojson -i metadatafile.desktop”
  - “rm metadatafile.desktop”

Plugin Loading - How to port

- JSON metadata needs to be embedded in plugin

  K_PLUGIN_CLASS_WITH_JSON(MyClass, “metadatafile.json”)

- For having multiple plugin classes or no metadata different macros are available

- To install the plugin we can use a helper method

  kcoreaddons_add_plugin(myplugin SOURCES myplugin.cpp INSTALL_NAMESPACE myapp/parsers)

Plugin Loading – Perks of the new System

- KPluginMetaData::isEnabled checks if plugins are enabled
- Useful in combination with KCMUtils classes
- Support for loading static plugins

  kcoreaddons_add_plugin(myplugin STATIC SOURCES ... INSTALL_NAMESPACE myapp/parsers)

  kcoreaddons_target_static_plugins(myapp myapp/parsers)

  KPluginMetaData/KPluginFactory methods still work the same
- Less dependencies overall

Porting KToolInvocation

- Utility methods of KToolInvocation class
  - invokeTerminal
  - invokeMailer
  - startServiceByDesktopName

Porting KToolInvocation

- API has a few issues:
- Depends on klauncher from kinit
- Unpleasant error handling

```
static int startServiceByName(const QString &_name,
                              const QString &URL,
                              QString *error = nullptr,
                              QString *serviceName = nullptr,
                              int *pid = nullptr,
                              const QByteArray &startup_id = QByteArray(),
                              bool noWait = false);
```

Porting KToolInvocation

- Job-based alternatives in KIO:
  - KIO::CommandLauncherJob
  - KIO::ApplicationLauncherJob
  - KTerminalLauncherJob
  - KEMailClientLauncherJob
- Provide better API and consistent error handling
- KF6 plan: Move these to KService

We Learned

- Why we should think about porting now
- How to approach a port
- How to adapt the build system to support Qt5 and Qt6
- How to port away from some API

Future

- No exact timeline for KF6 yet
- Will be discussed at BoF
- Depends on porting progress

Questions?