QFace Documentation

Release 2.0.8

JRyannel

Apr 13, 2022

Contents

Note: Repository is hosted at https://github.com/pelagicore/qface

QFace is a flexible API generator inspired by the Qt API idioms. It uses a common IDL format (called QFace interface document) to define an API. QFace is optimized to write a custom generator based on the common IDL format.

Several code generators for common use cases have already been implemented. These can be used as is or can be used as a base for a custom generator.

CHAPTER 1

Motivation

QFace is an attempt to establish a common interface description language with an easy to use code generator framework. While QFace as an interface description language which is Qt friendly, it is not limited to Qt usage. The vision is that many projects can agree on this interface language and many different generators will be created. In the end we all can learn from how other projects generate code from the same IDL.

1.1 The IDL

The IDL uses common API concept such as `modules`, `interfaces`, `properties`, `structs` and `enums`/`flags`. Additionally it knows about `lists`, `maps` and `models`. A list is an array of primitive or complex types. A map is an associative array of key/value pairs. A model is an indicator for large data sets which are typically used using a streaming concept, e.g. via a defined interface or via pagination.

```
module org.example 1.0

interface Echo {
    string message;
    void echo(string message);
    signal broadcast(string message);
    Status status;
}

enum Status {
    Null, Loading, Ready, Error
}
```

The data types provided by QFace can be divided into primitive and complex types:

Primitive Types

The primiteve types are mostly modeled after the JSON data types (see https://www.json.org/).

All exact data types (e.g. 32bit, 64bit, etc depend on the generator the IDL).

- `bool` - true/false
- `int` - represents integer type
- `real` - floating point number
- `string` - unicode string
- `var` - placeholder for a data type

Complex Types

A complex type is a composition of other types and add specific semantic to the type.

- `Interface` - collection of properties, operations and signals
- `Struct` - typed data package
- `Enum` - enumeration of integer values
- `Flag` - enumeration with n^2 values
- `List` - array of primitive or complex data types
- `Map` - collection of primitive or complex value. Key type is defined by generator. E.g. string is recommended.
- `Model` - A stream of primitive or complex data values.

1.2 Why another IDL

Many IDLs are already in existence. But most of them are bound to a certain technology or library or are limited for a specific use. Only a few IDLs exist which are independent from a technology. From these few technology independent IDLs which are known to the author satisfied the requirement to be Qt compatible and easy to use. Also the IDL should be easy to install and be extendable. The unique mix of technologies used in QFace allows it to provide a solid stable IDL with a powerful generation framework. - The base for your own code generator.

1.3 Defining APIs

There are many opinions how to define APIs and what would be the best way. The idea of QFace is that many projects find the IDL useful and use it to create their own code generator. Consequently, there will be a large set of generators and finally APIs can be compared and unified, even if they will be used with different technologies.

Inside one technology area there are often discussions between developers or teams about how an interface shall be coded. QFace allows the different parties to create their own generators based on the same API. Ideally at the end the knowledge how an interface shall be best coded will reside in the provided generator.

1.4 Large Projects

In larger projects there is the need to make a large set of operating services available to an user interface layer. It is less about defining new visual items in C++, more about creating an abstraction of a service and make it available to the UI developer.

This can be a challenge when you create many plugins and in the middle of the project you figure out that you have issues with your current design or if the customer in the next project wants to use a different HMI technology. All the knowledge is inside these plugins.

With QFace these companies can be ensured that QFace does not lock them into the UI technology and smaller API design issues can be fixed by fixing the used code generator.

1.5 Remote Services

Some projects use network communication to communicate from the HMI to the services, which might run on a different process or even a networked device. QFace was not designed for remote services as it does not define any storage types (e.g. int32, int16, int64), it only knows an `int` and does not define how large the `int` shall be. For this QFace needs to rely on the author of the generators to have a defined protocol to exchange data using QFace common data types.

1.6 Complex APIs

QFace is purposely designed to have limited features. The goal is to make QFace easy to use with an easy to remember syntax so that you don't need to be an expert to write interface files.

QFace does not support unions or structs that extend other structs. If you look for these features, QFace is probably the wrong choice.

1.7 Limitations

Like other code generation tools, QFace is limited by how much information you can place inside your interface files. In excessive cases code generation might not make sense and hence QFace will also not help you.

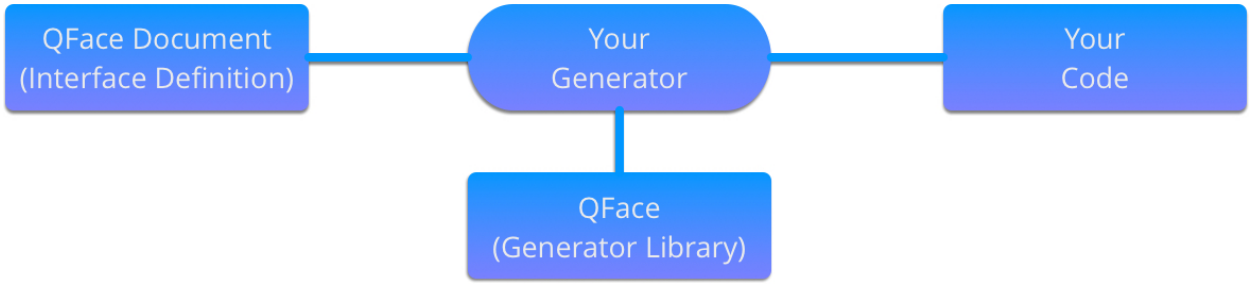
QFace allows you to use annotations which can add meta information to the interface files. But the generator needs to be designed to understand this meta information. QFace only defined the the structure of these annotations not the information and semantic they carry. Annotations might help you to add information to an interface document to better control the code generation process.

CHAPTER 2

Usage

2.1 Concept

QFace requires one or more IDL files as input file and a custom generator to produce output files. The IDL files are named QFace interface documents.



To use QFace you need to write your own generator. A generator is a small rules document which reads the QFace document and writes code using template files.

```
# rules-qface.yaml
project:
    interface:
        - {{interface}}.h: interface.h
        - {{interface}}.h: interface.cpp
        - Makefile: Makefile
```

You then call the script using the qface executable.

```
qface --rules rules-qface.yaml --target output echo.qface
```

2.2 Code Generation Principle

The code generation is driven by a rules document which applies the domain model and writes files using the Python Jinja template language.

Note: Refer to http://jinja.pocoo.org and particularly the template designer documentation at http://jinja.pocoo.org/docs/dev/templates/.

The initial folder structure should have a rules file and in the templates folder the required templates.

```
qface-rules.yml
templates/report.tpl
```

The rules document provides the rules for code-generation.

```
# qface-rules.yml
project:
    system:
        - project_report.csv: report.tpl
```

The qface executable reads the input qface files and converts them into a domain model. The domain model is then passed into the rules document. Inside the rules document you specify scopes and matches. If a `system` is specified as the match the `system` is passed into the given template documents.

```
{# templates/report.tpl #}
{% for module in system.modules %}
    {%- for interface in module.interfaces -%}
    INTERFACE, {{module}}.{{interface}}
    {% endfor -%}
    {%- for struct in module.structs -%}
    STRUCT , {{module}}.{{struct}}
    {% endfor -%}
    {%- for enum in module.enums -%}
    ENUM   , {{module}}.{{enum}}
    {% endfor -%}
{% endfor %}
```

The template iterates over the domain objects and generates text which is written into the output file in the given target folder.

You call the yaml document by calling the qface executable and provide the rules document as also the output document. The domain model is created based on the given input files.

```
qface --rules rules-qface.yaml --target output echo.qface
```

The output would then look like this.

```
output/project_report.csv
```

More ...

To know more about the different options just ask the help of qface.

```
qface --help

Usage: qface [OPTIONS] [SOURCE]...

Options:
  --rules PATH
  --target DIRECTORY
  --reload / --no-reload      Auto reload script on changes
  --scaffold / --no-scaffold  Add extrac scaffolding code
  --watch DIRECTORY
  --feature TEXT
  --run TEXT                  run script after generation
  --force / --no-force        forces overwriting of files
  --help                      Show this message and exit.
```

CHAPTER 3

Reference Generators

QFace does provide some real world reference generators which are hosted as separated projects. Their purpose is merely to showcase how to write a code generator using QFace. They are working and complete examples of general purpose generators.

qface-qtcpp

: The QtCPP generator generates a Qt C++ plugin with a QML API ready to be used in your project.

  Hosted at: https://github.com/Pelagicore/qface-qtcpp

qface-qtqml

: The QtQml generator generates a QML only API which ready to be used.

  Hosted at: https://github.com/Pelagicore/qface-qtqml

qface-qtro

: The RO (RemoteObjects) generator generates a client and server project using the Qt5 QtRemoteObejcts library

  Hosted at: https://github.com/Pelagicore/qface-qtro

From the QML user interface perspective the QtCPP and QtQML generators both provide the same API and are interchangeable.

CHAPTER 4

Grammar

QFace (Quick Interface Language) is an Interface Description Language (IDL). While it is primarily designed to define an interface between Qt, QML and C++, it is intended to be flexible enough also to be used in other contexts.

The grammar of QFace is well defined and is based on the concept of modules as larger collections of information.

A module can have several interfaces, structs and/or enums/flags. Here a not complete unformal grammar.

```
module <module> <version>

import <module> <version>

interface <Identifier> {
    <type> <identifier>
    <type> <operation>(<parameter>*)
    signal <signal>(<parameter>*)
}

struct <Identifier> {
    <type> <identifier>
}

enum <Identifier> {
    <name> = <value>,
}

flag <Identifier> {
    <name> = <value>,
}
```

A QFace document always describes one module. It is convention that a qface document is named after the module. If the same module appears in different documents the behavior is not defined currently. Each document can contain one or more interfaces, structs, flags or enums. Each document can import other modules using the import statement.

4.1 Module

A module is identified by its name. A module should be normally a URI where all parts are lowercase (e.g. *entertainment.tuner*). A module may import other modules with the primary purpose being to ensure that dependencies are declared inside the QFace file.

```
// org.example.qface
module org.example 1.0

import org.common 1.0
```

Note: The parser will not validate if the module exists yet. It will just provide the reference to the module and will try to resolve the module on code-generation runtime.

4.2 Interface

An interface is a collection of properties, operations and signals. Properties carry data, whereas the operations normally modify the data. Signals are used to notify the user of changes.

```
interface WeatherStation {
    real temperature;
    void reset();
    signal error(string message);
}
```

QFace allows to extend interfaces using the `extends` keyword after the interface name.

Note: It is in the responsibility of the author to ensure the order of interfaces are correct. E.g. the base interface should come before the depending interface. QFace does not try to re-order interfaces based on dependency. The order they appear in the document is the order they are passed to the code generator.

```
interface Station {
    void reset();
    signal error(string message);
}

interface WeatherStation extends Station {
    real temperature;
}
```

Note: For the sake of simplicity, as an API designer you should carefully evaluate if an extension is required. The typical way in QFace to allow extensions is normally to write your own code-generator and use type annotations for kind of interfaces.

```
@kind: Station
interface WeatherStation {
    real temperature;
}
```

The API reader does not need to know the internals of the API. The station behavior would be automatically attached by the custom generator.

4.3 Struct

The struct resembles a data container. It consist of a set of fields where each field has a name and a data type. Data types can be primitive of complex types.

```
struct Error {
    string message;
    int code;
};
```

Structs can also be nested. A struct can be used everywhere where a type can be used.

```
interface WeatherStation {
    real temperature;
    Error lastError;
    void reset();
    signal error(Error error);
}
```

Note: When you nest structs, ensure the used struct comes before the using structs and there are no circular dependencies. The order struct appear is the same order they are passed to the code generator.

4.4 Enum/Flag

An enum and flag is an enumeration type. The value of each member is automatically assigned if missing and starts with 0.

```
enum State {
    Null,       // implicit 0
    Loading,    // will be one
    Ready,      // will be two
    Error       // will be three
}
```

The value assignment for the enum type is sequential beginning from 0. To specify the exact value you can assign a value to the member. The value can also be written in hex form (e.f. 0xN).

```
enum State {
    Null = 0,
    Loading = 0x1,
    Ready = 2,
    Error = 3
}
```

The flag type defines an enumeration type where different values are treated as a bit mask. The values are in the sequence of the 2^n.

```
flag Cell {
    Null,   // starting value is one
    Box,    // value is two
    Wall,   // value is four
    Figure  // value is eight
}
```

4.5 Types

Types are either local and can be referenced simply by their names, or they are from external modules. In the latter case they need to be referenced with the fully qualified name (<module>.<symbol>). A type can be an interface, struct, enum or flag. It is also possible to reference the inner members of the symbols with the fragment syntax (<module>.<symbol>#<fragment>).

A module consists of either one or more interfaces, structs and enums/flags. They can come in any number or combination. The interface is the only type which can contain properties, operations and signals. The struct is merely a container to transport structured data. An enum/flag allows the user to encode information used inside the struct or interface as data-type.

Below is an example of a QFace file.

```
module entertainment.tuner 1.0;

import common 1.0

/*! Service Tuner */
interface Tuner {
    /*! property currentStation */
    readonly Station currentStation;
    /*! operation nextStation */
    void nextStation();
    /*! operation previousStation */
    void previousStation();
    /*! operation updateCurrentStation */
    void updateCurrentStation(int stationId);

    list<int> primitiveList;
    list<Station> complexList;
    map<int> simpleMap;
    map<Station> complexMap;
    model<int> primitiveModel;
    model<Station> complexModel;
}

/*! enum State */
enum State {
    /*! value State.Null */
    Null=0,
    /*! value State.Loading */
    Loading=1,
    /*! value State.Ready */
    Ready=2,
    /*! value State.Error */
    Error=3
}
```

(continues on next page)

(continued from previous page)

```
/*! enum Waveband */
enum Waveband {
    /*! value Waveband.FM */
    FM=0,
    /*! value Waveband.AM */
    AM=1
}

flag Features {
    Mono = 0x1,
    Stereo = 0x2,
}

/*! struct Station */
struct Station {
    /*! member stationId */
    int stationId;
    /*! member name */
    string name;
    /*! last time modified */
    common.TimeStamp modified;
}
```

4.6 Nested Types

A nested type is a complex type which nests another type. These are container types, e.g. list, map or model.

```
list<Color>  colors
map<Station> stations
model<WeatherInfo> weather
```

A list is an array of the provided value type. A map specifies only the value type. The key-type should be generic (e.g. a string type) and can be freely chosen by the generator. This allows for example the generator to add an id to each structure and use it as a key in the map.

A model is a special type of a list. It should be able to stream (e.g. add/change/remove) the data and the changes should be reflected by a more advanced API. Also the data could in general grow infinitely and the generator should provide some form of pagination or window API. You should use a model if you expect the data it represents to grow in a way that it may influence the performance of your API.

4.7 Annotations

Annotations allow the writer to add meta data to an interface document. It uses the @ notation followed by valid YAML one line content.

```
@singleton: true
@config: { port: 1234 }
interface Echo {
}
```

More information on annotations can be found in the annotations chapter.

4.8 Comments

Comments use the JavaDoc convention of using an @ sign as prefix with the keyword followed by the required parameters.

Currently only brief, description, see and deprecated are supported doc tags.

The QtCPP built-in generator generates valid Qt documentation out of these comments.

4.9 Default Values

QFace supports the assignment of default values to properties and struct fields. A default values is a text string passed to the generator.

```
interface Counter {
    int count = "0";
    Message lastMessage;
}

struct Message {
    string text = "NO DATA";
}
```

You can use quotes ‘ or double-quotes “ as a marker for text. There is no type check on QFace side. The text-content is directly passed to the generator.

CHAPTER 5

Annotations

Annotations allow to add meta information to your interface definition. It is available to each symbol in the interface.

With annotations an interface author can extend the existing interface with additional meta information, called tags, aka annotations. One or several annotations can precede a module, `interface`, `struct` or `enum`. They are also allowed before an `operation`, `property` or `signal`. Everywhere where a documentation comment is allowed you can also add annotations.

An annotation looks like this

```
@service: {port: 12345}
interface Tuner {
}
```

An embedded annotation precedes a symbol and it starts with an `@` sign. A symbol can have more than one annotation line. Each line should be one individual annotation. The content is YAML content. All `@` signs preceding a symbol are collected and then evaluated using a YAML parser.

For larger annotations you can use the external annotation document feature (see below).

```
@singleton: yes
@data: [1,2,3]
@config: { values: [LEFT, RIGHT, TOP] }
```

This will be result into a YAML content of

```
singleton: yes
data: [1,2,3]
config: { values: [LEFT, RIGHT, TOP] }
```

And the result as Python object would be

```
{
  "data": [ 1, 2, 3 ],
  "singleton": true,
  "config": {
```

(continues on next page)

(continued from previous page)

```
    "values": [ "LEFT", "RIGHT", "TOP" ]
  }
}
```

5.1 Annotation Documents

QFace allows also to specify these annotations in external documents using the *YAML* syntax. For this you need to create a document with the same name as the QFace document but with the extension *.yaml*. It should have roughly the following format

```
com.pelagicore.ivi.Tuner:
    service:
        port: 12345
```

On the root level should be a fully qualified name of a symbol. The symbol will be looked up and the following annotation information merged with the existing annotations from the QFace document.

5.2 Merging Annotations

The external annotations will be merged on top of the embedded annotations per symbol. Dictionaries will be merged. If a merge can not be done then the external document based annotations will override the embedded annotations.

5.3 Generators

Annotations are available later when navigating the domain model.

```
{% if "service" in interface.tags %}
interface {{interface}} is served on port: {{interface.tags.service.port}}
{% else %}
interface {{interface}} is not served
{% endif %}
```

Note: QFace does not specify specific annotations, but defines just the annotation format. The set of annotations supported must be defined and documented by the generator.

CHAPTER 6

YAML Primer

This page provides a basic overview of the YAML syntax as used by QFace in the embedded annotations and the external annotations document.

According to the official YAML website, YAML is "a human friendly data serialization standard for all programming languages".

6.1 YAML Foundation

For QFace every YAML file is a dictionary of values.

```
@singleton: true
@base: QObject
interface Heater {
}
```

A dictionary in YAML is expressed like this

In an external YAML file the key on the root level is the fully qualified name of the symbol

```
org.example.Heater:
    singleton: true
    base: QObject
```

6.2 Dictionary

A dictionary is a simple `key:  value` pair with a colon followed by a space (the space is mandatory).

```
key: value
key2: value
key3:
```

(continues on next page)

(continued from previous page)

```
    key31: value
    key32: value
```

A nested dictionary can be achieved by a new level of indentation.

An alternate form for a dictionary is this

```
key3: { key31: value, key32: value }
```

In a template the dictionay can be used as attributes of an object

```
{% if interface.tags.key == 'value' %}YES{% endif %}
```

To test is a key exists you can use the key in dictionary form

```
{% if 'key' in interface.tags %}YES{% endif %}
```

6.3 List

A list is an array of values

```
- item1
- item2
- item3:
    - item31
    - item32
```

A nested list can be created by indenting the list and postfixing the parent entry with a colon.

An alternate form is

6.3.1 Comments

YAML only knows line comments. A comment starts with a # and ends with line.

```
# this is the key for the value
key: value
```

6.3.2 Primitive Types

YAML understands different primitive types.

string

YAML understands strings either as an identifier or quoted using " or '.

You can use code blocks using the | sign. The block continues until the indentation ends. Or the > folding block, where each new line is replaced with a space.

number

YAML understands different number formats, here is a short list of the most important ones

```
# an integer
value: 10

# an hex value
value: 0xFF

# a float
value: 1.01
```

boolean

YAML understand different values as true/false.

```
positive: yes
positive: true
negative: no
negative: false
```

Besides these words it understand different writing forms (e.g. YES, Yes, Y). Same applies for the negative version.

CHAPTER 7

JSON Meta Export

QFace allows you to easily export the domain model as a JSON document. This enables you to parse the domain information to be used with other tooling.

Inside your generator you need to register the filter first

```
from qface.filters import jsonify


generator = Generator(search_path=search_path)
generator.register_filter('jsonify', jsonify)
```

Then inside the template you can transform any symbol into a JSON string using the `jsonify` filter.

```
{{module|jsonify}}
```

Depending on your need you might want to create a JSON document from the whole system or from each interface or you are just interested on a JSON representation of an enumeration. The portion of the domain model exported to JSON really depends on your custom code generator and on which doamin element you apply the `jsonify` filter.

7.1 JSON Format

Taking the example QFace document

```
module org.example 1.0;

interface Echo {
    readonly string currentMessage;
    void echo(Message message);
}

struct Message {
    string text;
```

(continues on next page)

(continued from previous page)

```
}

enum Status {
    Null,
    Loading,
    Ready,
    Error
}
```

The following JSON output is generated

```
{
  "name": "org.example",
  "version": "1.0",
  "interfaces": [
    {
      "name": "Echo",
      "properties": [
        {
          "name": "currentMessage",
          "type": {
            "name": "string",
            "primitive": true
          },
          "readonly": true
        }
      ],
      "operations": [
        {
          "name": "echo",
          "parameters": [
            {
              "name": "message",
              "type": {
                "name": "Message",
                "complex": true
              }
            }
          ]
        }
      ],
      "signals": []
    }
  ],
  "structs": [
    {
      "name": "Message",
      "fields": [
        {
          "name": "text",
          "type": {
            "name": "string",
            "primitive": true
          }
        }
      ]
    }
```

(continues on next page)

(continued from previous page)

```
  ],
  "enums": [
    {
      "name": "Status",
      "enum": true,
      "members": [
        {
          "name": "Null",
          "value": 0
        },
        {
          "name": "Loading",
          "value": 1
        },
        {
          "name": "Ready",
          "value": 2
        },
        {
          "name": "Error",
          "value": 3
        }
      ]
    }
  ]
}
```

CHAPTER 8

Domain Model

The domain model resembles the structure of our system as objects. It is build by the parser and is the input into the generator.

It is important to understand the domain model as it is the main input for the template generation.

The IDL is converted into an in memory domain model (see qface/idl/domain.py)

```
- System
    - Module
        - Import
        - Interface
            - Property
            - Operation
            - Signal
        - Struct
        - Enum
        - Flag
```

The domain model is the base for the code generation. You traverse the domain tree and trigger your own code generation. Below you see a python example to traverse the model. Normally you do not need todo this, as the rules generator does the traversing for you.

```
from qface.generator import FileSystem

system = FileSystem.parse('./interfaces')

for module in system.modules:
    print(module.name)

    for interfaces in module.interfaces:
        print(interfaces.name)

    for struct in module.structs:
        print(struct.name)
```

The rules generator calls the template directive for each domain element found and it is up to the template to place the code in the right place.

```
project:
    system:
        documents:
            - '{{system}}.txt': 'system.txt'
    module:
        documents:
            - '{{module}}.txt': 'module.txt'
    interface:
        documents:
            - '{{interface}}.txt': 'interface.txt'
    struct:
        documents:
            - '{{struct}}.txt': 'struct.txt'
    enum:
        documents:
            - '{{enum}}.txt': 'enum.txt'
```

CHAPTER 9

Extending

QFace is easy to use and easy to extend with your own generator. The standard way is to write a rules document to control the code generation. The more complicated path is to write your generator using just a small python script which uses qface as library.

9.1 Rules Extensions

The rules extension uses a YAML based document to control the code generation. The document is structured roughly like this:

```
<scope>:
    when: <feature-list>
    context: <context map update>
    path: <target path prefix>
    source: <source prefix>
    <qualifier>:
        when: <feature-list>
        context: <context map update>
        path: <target path prefix>
        source: <source prefix>
        documents:
            - <target>: <source>
        preserved:
            - <target>: <source>
```

scope entry

Scope is a logical distribution of generator. For example if you write a client/server generator you may want to have a `client` and `server` scope. This enables you also to switch a scope off using the `when` condition.

qualifier entry

The qualifier defines the domain model type this code generation section shall be applied to. Valid qualifiers are `system`, `module`, `interface`, `struct` and `enum`.

when entry

The *when* entry defines a condition when this part of the code generation is enabled. For example you may have some code generation parts, which create a scaffold project. By passing in the scaffold flag or by enabling the scaffold feature this part would then also be evaluated. By default when is true.

context entry

The contexct map allows you to extend the cotnext given to the template. Each context key will then accessible in the template.

path entry

The path is the path appended to the target directory. So the full export path for a template is `<target>/<path>/<template>`.

source entry

The source prefixed to the template name. For example to not to repeat the `server` folder for the next templates you can set the `source` to `server`.

documents entry

The documents section is a list of target, source templates. The source defines the template document used to produce the target document. The target document can have a fully qualified template syntax, for example `{{interface}}.h`, where the interface name is looked up using the given context. When generating existing documents will be overriden.

preserve entry

Very similar to the documents section. The only difference is that existing documents will be preserved. You can overrule this using the `--force` command line option.

Preserved is useful when generated document shall be edited by the user and a new run of the generator shall not overwrite (preserve) the edited document.

Example

Below is a more complex rules document from the qtcpp generator using one scope called `project`.

```
project:
    system:
        documents:
            - '{{project}}.pro': 'project.pro'
```

(continues on next page)

(continued from previous page)

```
            - '.qmake.conf': 'qmake.conf'
            - 'CMakeLists.txt': 'CMakeLists.txt'
    module:
        path: '{{module|identifier}}'
        documents:
            - 'CMakeLists.txt': 'plugin/CMakeLists.txt'
            - 'qmldir': 'plugin/qmldir'
            - 'generated/generated.pri': 'plugin/generated/generated.pri'
            - 'generated/{{module|identifier}}_gen.h': 'plugin/generated/module.h'
            - 'generated/{{module|identifier}}_gen.cpp': 'plugin/generated/module.cpp'
            - 'docs/plugin.qdocconf': 'plugin/docs/plugin.qdocconf'
            - 'docs/plugin-project.qdocconf': 'plugin/docs/plugin-project.qdocconf'
            - 'docs/docs.pri': 'plugin/docs/docs.pri'
        preserve:
            - '{{module|identifier}}.pro': 'plugin/plugin.pro'
            - 'plugin.cpp': 'plugin/plugin.cpp'
            - 'plugin.h': 'plugin/plugin.h'
    interface:
        preserve:
            - '{{interface|lower}}.h': 'plugin/interface.h'
            - '{{interface|lower}}.cpp': 'plugin/interface.cpp'
```

9.2 Script Extensions

The script iterates over the domain model and writes files using a template language.

See template engine documentation:

- http://jinja.pocoo.org
- http://jinja.pocoo.org/docs/dev/templates

```
from qface.generator import FileSystem, Generator

def generate(input, output):
    # parse the interface files
    system = FileSystem.parse(input)
    # setup the generator
    generator = Generator(search_path='templates')
    # create a context object
    ctx = {'output': output, 'system': system}
    # apply the context on the template and write the output to file
    generator.write('{{output}}/modules.csv', 'modules.csv', ctx)
```

This script reads the input directory returns a system object from the domain model. This is used as the root object for the code generation inside the template. The context object is applied to the file path as also on the named template document. The output of the template is then written to the given file path.

Below is a simple template which generates a CSV document of all interfaces, structs and enums.

```
{% for module in system.modules %}
    {%- for interface in module.interfaces -%}
    INTERFACE, {{module}}.{{interface}}
    {% endfor -%}
    {%- for struct in module.structs -%}
```

(continues on next page)

(continued from previous page)

```
    STRUCT , {{module}}.{{struct}}
    {% endfor -%}
    {%- for enum in module.enums -%}
    ENUM   , {{module}}.{{enum}}
    {% endfor -%}
{% endfor %}
```

The template code iterates over the domain objects and generates text using a mixture of output blocks `{{}}` and control blocks `{%%}`.

9.3 Rule Base Generation

The *RuleGenerator* allows you to extract the documentation rules into an external yaml file. This makes the python script more compact.

```
from qface.generator import FileSystem, RuleGenerator
from path import Path

here = Path(__file__).dirname()

def generate(input, output):
    # parse the interface files
    system = FileSystem.parse(input)
    # setup the generator
    generator = RuleGenerator(search_path=here/'templates', destination=output)
    generator.process_rules(here/'docs.yaml', system)
```

The rules document is divided into several targets. Each target can have an own destination. A target is typical for example and *app*, *client* or *server*. Each target can have rules for the different symbols (system, module, interface, struct, enum). An each rule finally consists of a destination modifier, additional context and a documents collection.

```
<target>:
    <symbol>:
        context: { <key>: <value> }
        destination: <path>
        documents:
            <target>:<source>
        preserve:
            <target>:<source>
```

- `<target>` is a name of the current target (e.g. client, server, plugin)
- `<symbol>` must be either system, module, interface, struct or enum

Here is an example (`docs.yaml`)

```
global:
    destination: '{{dst}}'
    system:
        documents:
            '{{project}}.pro': 'project.pro'
            '.qmake.conf': 'qmake.conf'
            'CMakeLists.txt': 'CMakeLists.txt'
plugin:
    destination: '{{dst}}/plugin'
```

(continues on next page)

(continued from previous page)

```
    module:
        context: {'module_name': '{{module|identifier}}'}
        documents:
            '{{module_name}}.pro': 'plugin/plugin.pro'
            'CMakeLists.txt': 'plugin/CMakeLists.txt'
            'plugin.cpp': 'plugin/plugin.cpp'
            'plugin.h': 'plugin/plugin.h'
            'qmldir': 'plugin/qmldir'
    interface:
        documents:
            '{{interface|lower}}.h': 'plugin/interface.h'
            '{{interface|lower}}.cpp': 'plugin/interface.cpp'
    struct:
        documents:
            '{{struct|lower}}.h': 'plugin/struct.h'
            '{{struct|lower}}.cpp': 'plugin/struct.cpp'
```

The rule generator adds the `dst`, `project` as also the corresponding symbols to the context automatically. On each level you are able to change the destination or update the context.

Features

The rules document allows to conditional write files based on a feature set. The feature set must be a set of tags indicating the features which will then be checked in the `when` section of a rule. The `when` tag needs to be a list of feature switched.

The features are passed to the generator in your custom generator code. The existence of a feature tells the rules engine to check if a `when` section exists conditionally execute this rule.

```
plugin:
    when: [plugin_enabled]
    destination: '{{dst}}/plugin'
    module:
        ...
```

Here the plugin rule will only be run when the feature set contains a 'plugin_enabled' string.

Preserving Documents

Documents can be moved to the `preserve` tag to prevent them to be overwritten. The rules documents has an own marker for this called `preserve`. This is the same dictionary of target/source documents which shall be be marked preserved by the generator.

```
plugin:
    interface:
        documents:
            '{{interface|lower}}.h': 'plugin/interface.h'
        preserve:
            '{{interface|lower}}.cpp': 'plugin/interface.cpp'
```

In the example above the preserve listed documents will not be overwritten during a second generator run and can be edited by the user.

Destination and Source

The `destination` tag allows you to specify a prefix for the target destination of the document. It should always contain the `{{dst}}` variable to be placed inside the project folder.

The `source` tag specifies a prefix for the templates resolving. If the template name starts with a `/` the prefix will be ignored.

Destination and source tags are allowed on the target level as also on each system, module and other symbol levels. A tag on a parent symbol will be the default for the child symbols.

Implicit symbol hierarchy

This is the implicit logical hierarchy taken into account:

```
<target>
    <system>
        <module>
            <interface>
            <struct>
            <enum>
```

Typical you place the destination prefix on the module level if your destination depends on the module symbol. For generic templates you would place the destination on the system level. On the system level you can not use child symbols (such as the module) as at this time these symbols are not known yet.

9.4 Parsing Documentation Comments

The comments are provided as raw text to the template engine. You need to parse using the *parse_doc* tag and the you can inspect the documentation object.

See below for a simple example

```
{% with doc = property.comment|parse_doc %}
\brief {{doc.brief}}

{{doc.description}}
{% endwith %}
```

Each tag in the JavaDoc styled comment, will be converted into a property of the object returned by *parse_doc*. All lines without a tag will be merged into the description tag.

9.5 Language Profiles

QFace supports the notion of profile. A profile is a set of features supported by the named profile. The intention of a profile is to make it easier for generator writers to stick to a limited set of language features, also if the overall language is evolving.

Currently there exists three language profiles:

- Micro - A limited set of languages features. The base profile. It does not allow importing of other modules or extending an interface, neither does it support maps.
- Advanced - Builds upon micro and allows imports, maps, interface extension.

- Full - Builds up on advanced and will also contain experimental language features.

The current features defined are: - const oeprations - const properties - imports - maps - interface extensions

The profiles and features are defined in the *qface.idl.profile* module.

```
from qface.generator import FileSystem
from qface.idl.profile import EProfile

system = FileSystem.parse(input=input, profile=EProfile.MICRO)
```

CHAPTER 10

Rules Mode

In the rules mode, qface is used as the qface exectuable. In this mode the code generator consits of a rule document and template files and optionally a filter module.

Whereas normally the generator writer create an own python package in this module only some documents are needed and the qface rules is used.

10.1 Setup

To get started create a *qface-rules.yml* document and a templates folder:

```
qface-rules.yml
templates/
```

In the rules file you provide the code generation rules according to the rule generator documentation. The templates folder will contain the required templates.

10.2 Filters

To provide extra filder you need to create a *filters.py* document with the declaration of your filters:

```
# a filter takes in a domain element or string
# and returns a string
def echo(s):
    return '{} World!'.format(s)

def get_filters():
    # returns a dict of new filters
    return {
        'echo': echo
    }
```

The filters module will be loaded by qface and all entries to the filters dictionary are added to the global lists of Jinja filters. You can now use it like any other Jinja filter.

```
{{ "Hello" | echo }}
```

Will resolve to `Hello World!`.

10.3 Running

To run now the generator you can simply call:

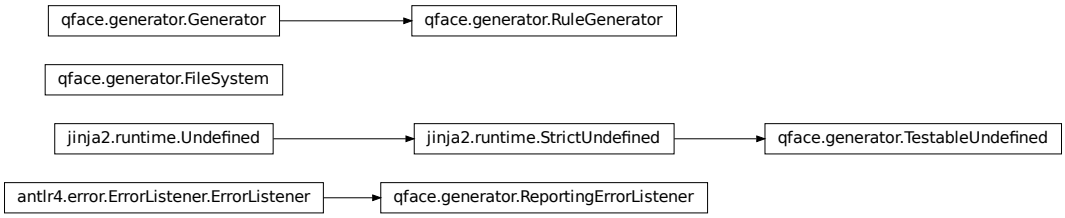
```
qface --rules qface-rules.yml --target out counter.qface
```

This will take your rules and generate the files inside the out folder based on the *counter.qface* interface file.

CHAPTER 11

API

11.1 Generator API



11.1.1 Generator Class

Provides an API for accessing the file system and controlling the generator

11.1.2 FileSystem Class

class qface.generator.**FileSystem**

Bases: object

QFace helper functions to work with the file system

static merge_annotations(*system*, *document*)

Read a YAML document and for each root symbol identifier updates the tag information of that symbol

static parse(*input*, *identifier: str = None*, *use_cache=False*, *clear_cache=True*, *pattern='*.qface'*, *profile=<EProfile.FULL: 'full'>*)
Input can be either a file or directory or a list of files or directory. A directory will be parsed recursively. The function returns the resulting system. Stores the result of the run in the domain cache named after the identifier.

Parameters

- **path** – directory to parse
- **identifier** – identifies the parse run. Used to name the cache
- **clear_cache** – clears the domain cache (defaults to true)

strict = False
enables strict parsing

class qface.generator.**Generator**(*search_path*, *context={}*, *force=False*)
Bases: object

Manages the templates and applies your context data

destination
destination prefix for generator write

get_template(*name*)
Retrieves a single template file from the template loader

register_filter(*name*, *callback*)
Register your custom template filter

render(*name*, *context*)
Returns the rendered text from a single template file from the template loader using the given context data

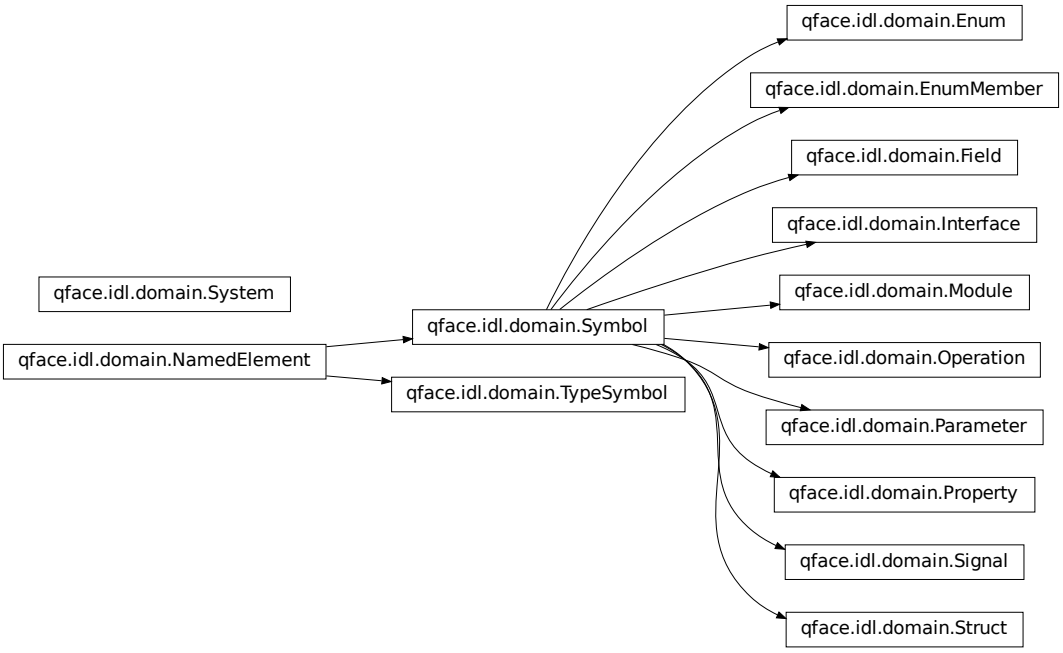
source
source prefix for template lookup

strict = False
enables strict code generation

write(*file_path*, *template*, *context={}*, *preserve=False*, *force=False*)
Using a template file name it renders a template into a file given a context

11.2 Template Domain API

This API is exposed to the Jinja template system.



11.2.1 High Level Classes

class qface.idl.domain.**System**
Bases: object

The root entity which consist of modules

lookup(*name: str*)
lookup a symbol by fully qualified name.

modules
returns ordered list of module symbols

class qface.idl.domain.**Module**(*name: str*, *system: qface.idl.domain.System*)
Bases: *qface.idl.domain.Symbol*

Module is a namespace for types, e.g. interfaces, enums, structs

enums
returns ordered list of enum symbols

imports
returns ordered list of import symbols

interfaces
returns ordered list of interface symbols

lookup(*name: str*, *fragment: str = None*)
lookup a symbol by name. If symbol is not local it will be looked up system wide

majorVersion
returns the major version number of the version information

minorVersion
returns the minor version number of the version information

module_name
returns the last part of the module uri

name_parts
return module name splitted by '.' in parts

structs
returns ordered list of struct symbols

11.2.2 Interface Related Classes

class qface.idl.domain.**Interface**(*name: str*, *module: qface.idl.domain.Module*)
Bases: *qface.idl.domain.Symbol*

A interface is an object with operations, properties and signals

extends
returns the symbol defined by the extends interface attribute

operations
returns ordered list of operations

properties
returns ordered list of properties

signals
returns ordered list of signals

class qface.idl.domain.**Operation**(*name: str*, *interface: qface.idl.domain.Interface*)
Bases: *qface.idl.domain.Symbol*

An operation inside a interface

interface = None
the interface the operation is part of

is_const = None
reflects is the operation was declared as const operation

parameters
returns ordered list of parameters

qualified_name
return the fully qualified name (<*module*>.<*interface*>#<*operation*>)

class qface.idl.domain.**Parameter**(*name: str*, *operation: qface.idl.domain.Operation*)
Bases: *qface.idl.domain.Symbol*

An operation parameter

class qface.idl.domain.**Property**(*name: str*, *interface: qface.idl.domain.Interface*)
Bases: *qface.idl.domain.Symbol*

A typed property inside a interface

is_complex_model
true if type is a model of nested complex types

is_model
true if type is a model

is_primitive_model
true if type is a model of nested primitive types

qualified_name
return the fully qualified name (<*module*>.<*interface*>#<*property*>)

11.2.3 Struct Related Classes

class qface.idl.domain.**Struct**(*name: str*, *module: qface.idl.domain.Module*)
Bases: *qface.idl.domain.Symbol*

Represents a data container

fields
returns ordered list of members

class qface.idl.domain.**Field**(*name: str*, *struct: qface.idl.domain.Struct*)
Bases: *qface.idl.domain.Symbol*

A member in a struct

qualified_name
return the fully qualified name (<*module*>.<*struct*>#<*field*>)

Enum/Flag Related Classes

class qface.idl.domain.**Enum**(*name: str*, *module: qface.idl.domain.Module*)
Bases: *qface.idl.domain.Symbol*

An enum (flag) inside a module

members
returns ordered list of members

class qface.idl.domain.**EnumMember**(*name: str*, *enum: qface.idl.domain.Enum*)
Bases: *qface.idl.domain.Symbol*

A enum value

qualified_name()
return the fully qualified name (<*module*>.<*enum*>#<*member*>)

11.2.4 Base Classes

class qface.idl.domain.**NamedElement**(*name*, *module: qface.idl.domain.Module*)
Bases: object

module = None
module the symbol belongs to

name = None
symbol name

qualified_name
return the fully qualified name (<*module*>.<*name*>)

class qface.idl.domain.**Symbol**(*name: str*, *module: qface.idl.domain.Module*)
Bases: *qface.idl.domain.NamedElement*

A symbol represents a base class for names elements

add_attribute(*tag*, *name*, *value*)
add an attribute (nam, value pair) to the named tag

add_tag(*tag*)
add a tag to the tag list

attribute(*tag*, *name*)
return attribute by tag and attribute name

comment = None
comment which appeared in QFace right before symbol

contents
return general list of symbol contents

kind = None
the associated type information

system
returns reference to system

tag(*name*)
return tag by name

class qface.idl.domain.**TypeSymbol**(*name: str*, *parent: qface.idl.domain.NamedElement*)
Bases: *qface.idl.domain.NamedElement*

Defines a type in the system

is_bool
checks if type is primitive and bool

is_complex = None
if type represents a complex type

is_enum
checks if type is an enumeration and reference is enum

is_enumeration
checks if type is complex and instance of type Enum

is_flag
checks if type is an enumeration and reference is flag

is_int
checks if type is primitive and int

is_interface
checks if type is interface

is_list = None
if type represents a list of nested types

is_map = None
if type represents a map of nested types. A key type is not defined

is_model = None
if type represents a model of nested types

is_primitive = None
if type represents a primitive type

is_real
checks if type is primitive and real

is_string
checks if type is primitive and string

is_struct
checks if type is complex and struct

is_valid
checks if type is a valid type

is_var
checks if type is primitive and var

is_void = None
if type represents the void type

nested = None
nested type if symbol is list or model

parent = None
the parent symbol of this type

reference
returns the symbol reference of the type name

type
return the type information. In this case: self

11.2.5 Utility Modules

qface.watch.**monitor**(*args*, *watch*)
reloads the script given by argv when src files changes

qface.shell.**sh**(*args*, ***kwargs*)
runs the given cmd as shell command

CHAPTER 12

Features

The list fo features is split between features which are based on the chosen IDL and features which are provided by the generator itself.

IDL Features

- Common modern IDL
- Scalable through modules
- Structured data through structs, enums, flags
- Interface API with properties, operations and signals
- Annotations using YAML syntax
- Fully documentable

Generator Features

- Easy to install using python package manager
- Designed to be extended
- Well defined domain objects
- Template based code generator
- Simple rule based code builder
- Well documented

CHAPTER 13

Quick Start

QFace is a generator framework and is bundled with several reference code generators.

To install qface you need to have python3 installed and typically also pip3

```
pip3 install qface
```

This installs the python qface library onto your system.

You can verify that you have qface installed with

```
qface --help
```

13.1 Custom Generator

To write a custom generator it is normally enough to write a generator rules and the used templates. We use a QFace interface file (here called "sample.qface") as an example.

The QFace document could look like this

```
// interfaces/sample.qface
module org.example 1.0

interface Echo {
    string echo(string msg);
}
```

We need now to write our templates for the code generation. In our example we would simple print out for each module the interfaces and it's operations.

```
{# templates/module.tpl #}
{% for interface in module.interfaces %}
{{module}}.{{interface}}
{% endfor %}
```

This will write for each interface in the module the text `<module>.<interface>`. The rules file will define what shall be generated and where.

```
# qface-rules.yaml

project:
  module:
    documents:
      - {{module}}.csv: module.tpl
```

The first entry defined a scope (e.g. project). Then for each module we geenrated documents. We use the module.tpl document from the templates folder and generate a CSV document based on the module name.

Now you can simple call your rules document

```
qface --rules qface-rules.yaml --target output interfaces
```

And a "org.example.csv" file named after the module should be generated.

See Also

- *Extending*
- *Grammar*
- *Domain Model*
- *API*

13.2 Generators

QFace has several generator maintained by the qface team. They are maintained and documented in their own repositories.

- https://github.com/pelagicore/qface-qtcpp
- https://github.com/pelagicore/qface-qtqml

Python Module Index

q

Index