SysInfraEGCppCommonAPISpecification

C++ Common API Specification

- What is C++ Common API?
- C++ Common API Reference
    - Introduction
        - Deployment
        - General design
        - Namespaces
    - Data types
        - Basic types
        - Arrays
        - Structures
        - Enumerations
        - Maps
        - Unions
        - Type aliases
        - Complete data types example (struct, array, enum and maps)
    - Interfaces
        - Basics
        - Methods
        - Attributes
        - Events
        - Attributes and Methods Example
    - Addresses, service discovery, connect/disconnect
        - Proxy availability
        - Runtime
            - Linking the Middleware at Compile Time
            - Linking the Middleware at Runtime
        - Factory
            - Example usage
        - Interface Factory
            - Example usage
    - Provider
        - Stubs
            - Methods
            - Broadcasts
            - Attributes
            - Default implementation
            - Example generated Stub
        - Interface Adapter
            - CommonAPI Interface Adapter class
    - Threading Model
        - Standard Threading
        - Single Threaded (Mainloop integration)
    - Configuring CommonAPI
        - Change the behavior of interfaces
        - Configuration Files
            - Available categories
                - Well known names of specific middleware bindings
            - Environment Variables
    - Known Limitations

What is C++ Common API?

Common API allows that applications (i.e., clients and servers using C++) developed against Common API can be linked against different Common API IPC backends without any changes to the application code. Thus, components which have been developed for a system which uses IPC X can be deployed for a system which uses IPC Y easily, just by generating some glue code and rebuilt. The actual interface definitions will be created using Franca IDL, which is the Common IDL solution favored by GENIVI.

Common API will not be restricted to GENIVI members (cf. new GENIVI open source policy). It will be available as an open source specification and some code generation tooling. The latter will be part of the open source Franca distribution. In the code repository, there is a prototype Common API generator implementation (work in progress, see **dev_CommonAPI** branch).

This page represents an intermediate status of the C++ Common API specification, with features up to CommonAPI 2.1. It summarizes the discussion documented on a separate wiki page, which is still ongoing, though not only on this page.

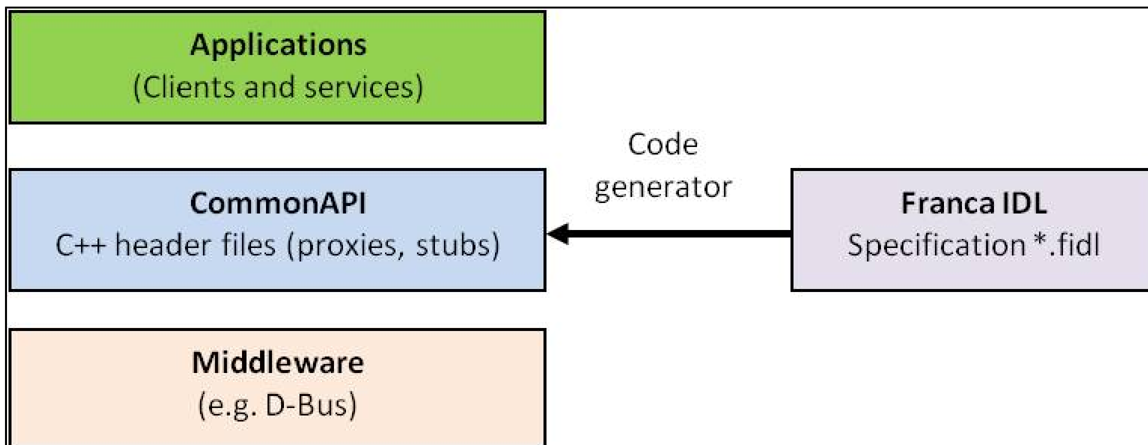
C++ Common API Reference

Introduction

CommonAPI is a standardized **C++ API** specification for the development of distributed applications which communicate via a middleware for interprocess communication. The intention is to make the C++ interface for the applications independent from the underlying middleware.

Other basic assumptions are:

- The applications use the client-server communication paradigm.
- The C++ API is based on the common interface description language Franca IDL which provides the possibility to specify interfaces independent from the platform, middleware or programming language. That means that the application specific part of the API is generated via a code generator from a Franca IDL specification file.
- In principle, the *CommonAPI* should be platform independent. However, this is without any restrictions very difficult to realize. Therefore it is agreed that CommonAPI attempts to use only features supported from the gnu C++ compiler version <= 4.4.2.



Deployment

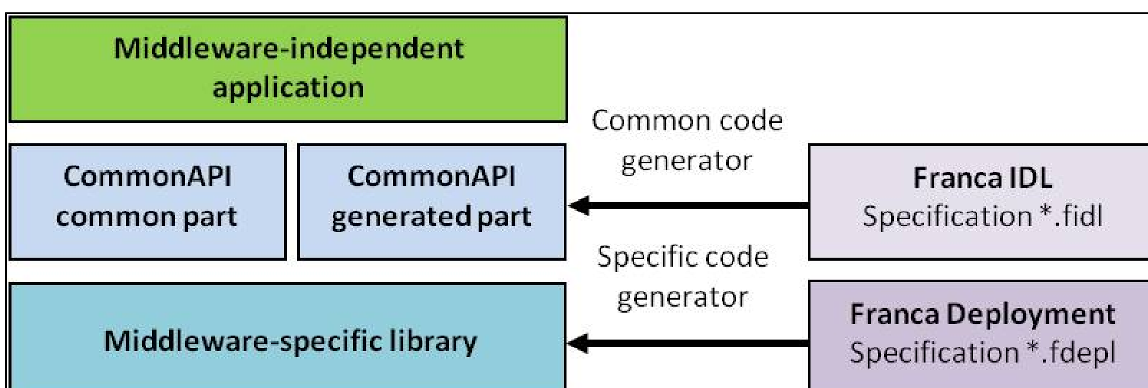
One problem with definition of a middleware-independent C++ API is that depending on the middleware different configuration parameters for parts of the API could be necessary. Examples:

- QoS parameter
- Maximum length of arrays or strings
- Endianness of data
- Priorities

The Franca IDL offers the possibility to specify these kind of parameters which depend on the used middleware in a middleware-specific or platform-specific deployment model (*.depl file). The deployment parameters can be specified arbitrarily.

But as indicated above it is an explicit goal that an application written against CommonAPI can be linked against different CommonAPI IPC backends without any changes to the application code. This goal brings an important implicit restriction:

The interface defined in Franca is the only information that can be used to generate the CommonAPI header that defines the implementation API. Deployment models are specific to the IPC backend, therefore they must not affect the generated API.

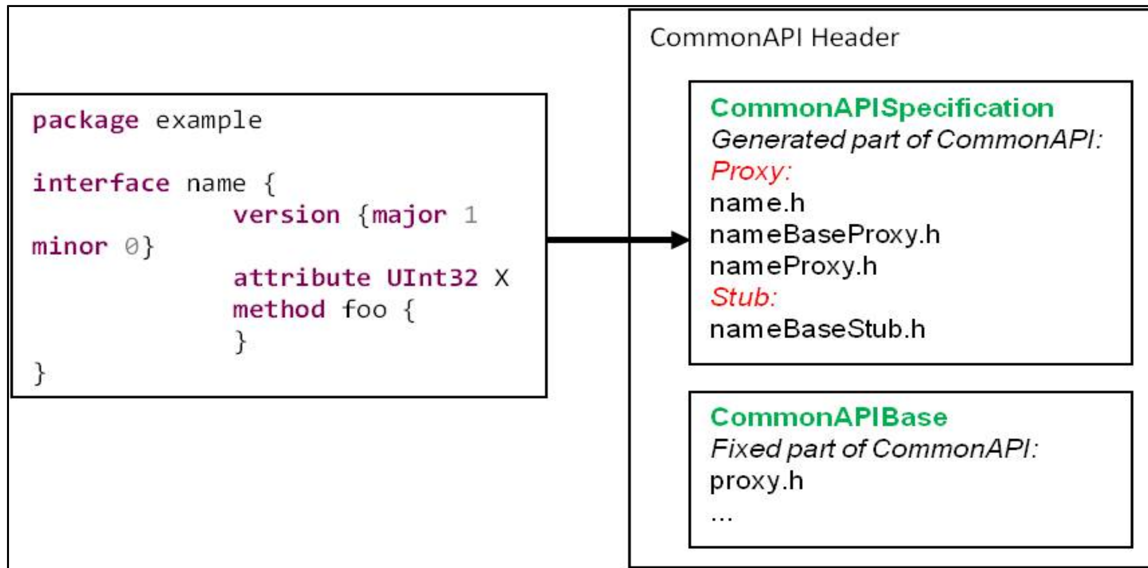


General design

The CommonAPI specification can be divided up into two parts:

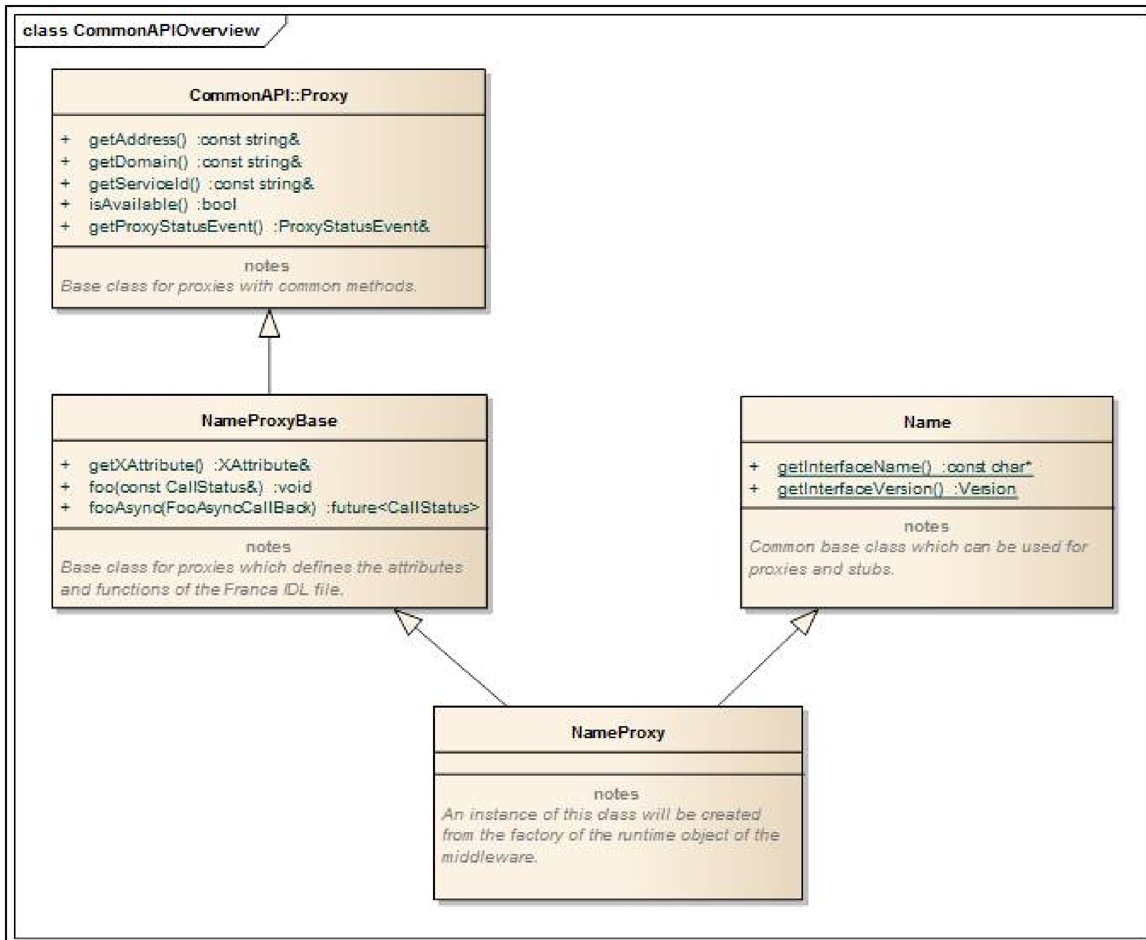
- The first part (**CommonAPISpecification**) refers to the variable (generated) part of the logical interface. That is the part of the interface which depends on the specifications in the Franca IDL file (data types, arrays, enumerations and interface basics as attributes, methods, callbacks, error handling, broadcast).

- The second fixed part (**CommonAPIBase**) which is mainly independent from the interface specifications refers to the CommonAPI library functions as service discovery, connect/disconnect, and address handling. Furthermore this part contains common type definitions and base classes.



The headerfiles of the CommonAPIBase are:

Attribute.h	Base classes for attributes
AttributeExtension.h	Base class for extensions
ByteBuffer.h	Typedef for ByteBuffer
Event.h	Eventclass for framework event
Proxy.h	Base class with proxy methods
ProxyFactory.h	Proxy factory class
InterfaceAdapter.h	Base class for method call handling on provider side
InterfaceAdapterFactory.h	Interface adapter factory class
Runtime.h	Base class, class loading for specific middleware starts here
SerializableStruct.h	Base class for structs, allows for correct handling of inheritance and serialization
SerializableVariant.h	Base class for variants, allows for correct handling of inheritance and serialization
Stream.h	readValue methods
Types.h	AvailabilityStatus, CallStatus, Version



The CommonAPI library provides a proxy factory which can be used to get a proxy instance of the interface specific proxy class (in the example this is the class nameProxy for the interface name). This proxy can be used to call interface methods (see section Connect below).

Namespaces

The **namespace** of the CommonAPIBase functions is CommonAPI; the namespace of the CommonAPISpecification depends on the the qualified package name of the interface specification.

fidl	C++

`package example.user`	`namespace example {`
	`namespace user {`
	
	`}`
	`}`

Data types

Basic types

The integer datatypes used by Common API are defined in stdint.h

Franca	CommonAPI	Remark
Int8	int8_t	
Int16	int16_t	
Int32	int32_t	
Int64	int64_t	

UInt8	uint8_t	
UInt16	uint16_t	
UInt32	uint32_t	
UInt64	uint64_t	
Boolean	bool	
Double	double	
Float	float	
String	std::string	Franca has only one String datatype, and if necessary the wire format / encoding can be specified via deployment model. The proxies always expect and deliver UTF-8.
ByteBuffer	std::vector<uint8_t>	

Example for all basic data types and how they are mapped to CommonAPI test-predefined-types.fidl

```
package commonapi.tests
typeCollection PredefinedTypeCollection {
 typedef TestUInt8 is UInt8
 typedef TestUInt16 is UInt16
 typedef TestUInt32 is UInt32
 typedef TestUInt64 is UInt64
 typedef TestInt8 is Int8
 typedef TestInt16 is Int16
 typedef TestInt32 is Int32
 typedef TestInt64 is Int64
 typedef TestBoolean is Boolean
 typedef TestByteBuffer is ByteBuffer
 typedef TestDouble is Double
 typedef TestFloat is Float
 typedef TestString is String
}
```

CommonAPI *.h	CommonAPI *.cpp

<pre>#ifndef COMMONAPI_TESTS_PREDEFINED_TYPE_COLLECTION_H
#define COMMONAPI_TESTS_PREDEFINED_TYPE_COLLECTION_H
 	
#include <CommonAPI/ByteBuffer.h>
#include <CommonAPI/types.h>
#include <cstdint>
#include <string>
 	
namespace commonapi {
namespace tests {
namespace PredefinedTypeCollection {
 	
typedef uint8_t TestUInt8;
typedef uint16_t TestUInt16;
typedef uint32_t TestUInt32;
typedef uint64_t TestUInt64;
typedef int8_t TestInt8;
typedef int16_t TestInt16;
typedef int32_t TestInt32;
typedef int64_t TestInt64;
typedef bool TestBoolean;
typedef CommonAPI::ByteBuffer TestByteBuffer;
typedef double TestDouble;
typedef float TestFloat;
typedef std::string TestString;
 	
static inline const char* getTypeCollectionName() {
    return "commonapi.tests.PredefinedTypeCollection";
}
 	
} // namespace PredefinedTypeCollection
} // namespace tests
} // namespace commonapi
 	
#endif // COMMONAPI_TESTS_PREDEFINED_TYPE_COLLECTION_H</pre>
	<pre>n.a.</pre>

Franca has only one string data type, and if necessary the wire format / encoding can be specified via deployment model. The Proxies always expect and deliver UTF-8.

Arrays

Franca array types (in explicit and implicit notation) are mapped to std::vector<T>. While explicitly defined Array types will be made available as typedef with the name as it was given in Franca IDL, the implicit version will just be generated as std::vector<T> wherever needed.

Structures

Franca struct types are mapped to C++ struct types.

Example:

fidl	C++

fidl	C++

```
struct
TestStruct {
 UInt16
uintValue
 String
stringValue
}
```

```
struct TestStruct: CommonAPI::SerializableStruct {
 TestStruct() = default;
 TestStruct(const uint16_t& uintValue, const std::string&
stringValue);

 virtual void readFromInputStream(CommonAPI::InputStream&
inputStream);
 virtual void writeToOutputStream(CommonAPI::OutputStream&
outputStream) const;

 uint16_t uintValue;
 std::string stringValue;
};
```

One problem is the possibility to inherit structures in Franca IDL. This feature can be mapped 1:1 to C++ inheritance for structs. Due to the limitations of the C++ language, we can make use of the C++ virtual methods to guarantee proper serialization of derived struct types. For that we define *InputStream* and *OutputStream* classes which are part of the CommonAPI library. Each basic struct (the topmost parent in inheritance) must derive from *SerializableStruct* which is also defined within the CommonAPI library. This will force each struct to implement two virtual methods: *readFromInputStream()* and *writeToOutputStream()*. The CommonAPI library also defines the two input and output stream operators on *SerializableStruct*: operator<<() and operator>>(), whose purpose is to signal the *InputStream* and *OutputStream* implementations that a struct is about to be serialized, and call the internal *readFromInputStream()* or *writeToOutputStream()* methods. The latter are CommonAPI specific and must not be accessible from the application or the individual bindings to avoid confusion. A typical application will create a struct instance, fill in the values and call a proxy method with the struct instance. The proxy method will be dispatched to the apropriate binding, which will eventually serialize the struct using the stream operators. This will result in calling the binding specific *InputStream* or *OutputStream* implementations through *SerializableStruct*'s virtual methods *readFromInputStream()* or *writeToOutputStream()*.

Enumerations

Franca enumerations will be mapped to C++ strongly typed enums. Enum backing datatype and wire format by default is uint_32. If needed, the wire format can be specified via deployment model, but proxy only delivers and expects the default.

fidl	C++

```
enumeration MyEnum {
 E_UNKNOWN = "0x00"
}

enumeration
MyEnumExtended extends
MyEnum {
 E_NEW = "0x01"
}
```

```
enum class MyEnum: int32_t {
 E_UNKNOWN = 0
};

//Definition of a comparator  is necessary for
GCC 4.4.1, topic is fixed since 4.5.1
struct MyEnumComparator;

enum class MyEnumExtended: int32_t {
 E_UNKNOWN = MyEnum::E_UNKNOWN,
 E_NEW = 1
};
```

To enable comparisons between enumerations in an inheritance hierarchy comparators have to be generated for the C++ types, as C++ does not support enum-inheritance natively.

Maps

For efficiency reasons std::unordered_map<K,V> are recommended. They are supported by gcc 4.4.

fidl	C++

`map MyMap { UInt32 to String }`	`typedef std::unordered_map<uint32_t, std::string> MyMap;`

Unions

Franca union types are implemented as a typedef of CommonAPI generic templated C++ variant class.

fidl	C++
`union MyUnion { UInt32 MyUInt String MyString }`	`typedef Varaint<uint32_t, std::string> MyUnion;`

This uses a variadic template to define the possible options, and implements operators in the expected fashion.

Assignment works by constructor or assignment operator:

```
MyUnion union = 5;
MyUnion stringUnion("my String");
```

Geting the contained value is done via a get method templated to the type desired for type safety. This results in a compile error if an impossible type is attempted to be fetched.
In case of fetching a type which can be contained but is not an exception is thrown. The choice of an exception at this point is made for the following reasons:

- Returning pointers is inconvenient, especially in case of primitives
- Returning a temporary reference in case of failure is dangerous due to potential for segmentation faults in case of accidental use
- Returning a null heap object will be a memory leak if not deleted by the user

```
MyUnion union = 5;

int a = union.get<uint32_t>(); //Works!

std::string b = union.get<std::string>(); //Throws exception
```

Also available is an templated isType method to test for the contained type:

```
MyUnion union = 5;

bool contained = union.isType<uint32_t>(); //True!
contained = union.isType<std::string>(); //False!
```

To enable comparisons between variants in an inheritance hierarchy comparators have to be generated for the C++ types, as C++ as all variants are instances of the same generic class.

Type aliases

Franca typedefs are mapped to C++ typedef.

Complete data types example (struct, array, enum and maps)

Franca-IDL test-derived-types.fidl

```
package commonapi.tests
import commonapi.tests.* from "test-predefined-types.fidl"
typeCollection DerivedTypeCollection {
 <** @description : Common errors. **>
 enumeration TestEnum {
   <** @description : default **>
  E_UNKNOWN = "0x00"
  <** @description : no error - positive reply **>
  E_OK = "0x01"
  <** @description : value out of range **>
  E_OUT_OF_RANGE = "0x02"
     <** @description : not used **>
  E_NOT_USED = "0x03"
 }

 enumeration TestEnumMissingValue {
   <** @description : default **>
  E1 = "A"
  E2
  E3 = "2"
 }

 enumeration TestEnumExtended extends TestEnum {
  <** @description : new error **>
  E_NEW = "0x04"
 }

 enumeration TestEnumExtended2 extends TestEnumExtended {
  <** @description : new error **>
  E_NEW2 = "0x05"
 }

 struct TestStruct {
  <** @description : the name of the property **>
  PredefinedTypeCollection.TestString testString

  <** @description : the actual value **>
  UInt16 uintValue
 }

 struct TestStructExtended extends TestStruct {
  TestEnumExtended2 testEnumExtended2
 }

 array TestArrayUInt64 of UInt64
 array TestArrayTestStruct of TestStruct

 map TestMap { UInt32 to TestArrayTestStruct }
}
```

CommonAPI *.h	CommonAPI *.cp

`#ifndef COMMONAPI_TESTS_DERIVED_TYPE_COLLECTION_H`
`#define COMMONAPI_TESTS_DERIVED_TYPE_COLLECTION_H`
 	
`#include <CommonAPI/SerializableStruct.h>`
`#include <CommonAPI/Stream.h>`	

```
#include <CommonAPI/types.h>
#include <commonapi/tests/PredefinedTypeCollection.h>
#include <cstdint>
#include <unordered_map>
#include <vector>

namespace commonapi {
namespace tests {

namespace DerivedTypeCollection {

enum class TestEnum: int32_t {
    E_UNKNOWN,
    E_OK,
    E_OUT_OF_RANGE,
    E_NOT_USED
};

// XXX Definition of a comparator still is necessary for GCC 4.4.1,
topic is fixed since 4.5.1
struct TestEnumComparator;

enum class TestEnumMissingValue: int32_t {
    E1 = 10,
    E2,
    E3 = 2
};

// XXX Definition of a comparator still is necessary for GCC 4.4.1,
topic is fixed since 4.5.1
struct TestEnumMissingValueComparator;

enum class TestEnumExtended: int32_t {
    E_UNKNOWN = TestEnum::E_UNKNOWN,
    E_OK = TestEnum::E_OK,
    E_OUT_OF_RANGE = TestEnum::E_OUT_OF_RANGE,
    E_NOT_USED = TestEnum::E_NOT_USED
    ,
    E_NEW
};

// XXX Definition of a comparator still is necessary for GCC 4.4.1,
topic is fixed since 4.5.1
struct TestEnumExtendedComparator;

enum class TestEnumExtended2: int32_t {
    E_UNKNOWN = TestEnum::E_UNKNOWN,
    E_OK = TestEnum::E_OK,
    E_OUT_OF_RANGE = TestEnum::E_OUT_OF_RANGE,
    E_NOT_USED = TestEnum::E_NOT_USED,

    E_NEW = TestEnumExtended::E_NEW
    ,
    E_NEW2
};

// XXX Definition of a comparator still is necessary for GCC 4.4.1,
topic is fixed since 4.5.1
struct TestEnumExtended2Comparator;

struct TestStruct: CommonAPI::SerializableStruct {
    TestStruct() = default;
    TestStruct(const PredefinedTypeCollection::TestString&
```

```
#include

namespac
namespac
namespac

TestStru
Predefin
uint16_t


{
}

void Tes
inputStr
    inpu
    inpu
}

void Tes
outputSt
    outp
    outp
}
TestStru
Predefin
uint16_t
testEnum


{
}

void
TestStru
inputStr
    Test
    inpu
}

void
TestStru
outputSt
    Test
    outp
}

} // nam
} // nam
} // nam
```

```
testString, const uint16_t& uintValue);

    virtual void readFromInputStream(CommonAPI::InputStream&
inputStream);
    virtual void writeToOutputStream(CommonAPI::OutputStream&
outputStream) const;

    PredefinedTypeCollection::TestString testString;
    uint16_t uintValue;
};

struct TestStructExtended: TestStruct {
    TestStructExtended() = default;
    TestStructExtended(const PredefinedTypeCollection::TestString&
testString, const uint16_t& uintValue, const TestEnumExtended2&
testEnumExtended2);

    virtual void readFromInputStream(CommonAPI::InputStream&
inputStream);
    virtual void writeToOutputStream(CommonAPI::OutputStream&
outputStream) const;

    TestEnumExtended2 testEnumExtended2;
};

typedef std::vector<uint64_t> TestArrayUInt64;
typedef std::vector<TestStruct> TestArrayTestStruct;
typedef std::unordered_map<uint32_t, TestArrayTestStruct> TestMap;

inline CommonAPI::InputStream& operator>>(CommonAPI::InputStream&
inputStream, TestEnum& enumValue) {
    return inputStream.readEnumValue<int32_t>(enumValue);
}

inline CommonAPI::OutputStream& operator<<(CommonAPI::OutputStream&
outputStream, const TestEnum& enumValue) {
    return
outputStream.writeEnumValue(static_cast<int32_t>(enumValue));
}

struct TestEnumComparator {
    inline bool operator()(const TestEnum& lhs, const TestEnum& rhs)
const {
        return static_cast<int32_t>(lhs) < static_cast<int32_t>(rhs);
    }
};

inline CommonAPI::InputStream& operator>>(CommonAPI::InputStream&
inputStream, TestEnumMissingValue& enumValue) {
    return inputStream.readEnumValue<int32_t>(enumValue);
}

inline CommonAPI::OutputStream& operator<<(CommonAPI::OutputStream&
outputStream, const TestEnumMissingValue& enumValue) {
    return
outputStream.writeEnumValue(static_cast<int32_t>(enumValue));
}

struct TestEnumMissingValueComparator {
    inline bool operator()(const TestEnumMissingValue& lhs, const
TestEnumMissingValue& rhs) const {
        return static_cast<int32_t>(lhs) < static_cast<int32_t>(rhs);
    }
```

```
};

inline CommonAPI::InputStream& operator>>(CommonAPI::InputStream&
inputStream, TestEnumExtended& enumValue) {
    return inputStream.readEnumValue<int32_t>(enumValue);
}

inline CommonAPI::OutputStream& operator<<(CommonAPI::OutputStream&
outputStream, const TestEnumExtended& enumValue) {
    return
outputStream.writeEnumValue(static_cast<int32_t>(enumValue));
}

struct TestEnumExtendedComparator {
    inline bool operator()(const TestEnumExtended& lhs, const
TestEnumExtended& rhs) const {
        return static_cast<int32_t>(lhs) < static_cast<int32_t>(rhs);
    }
};

inline bool operator==(const TestEnumExtended& lhs, const TestEnum&
rhs) {
    return static_cast<int32_t>(lhs) == static_cast<int32_t>(rhs);
}
inline bool operator==(const TestEnum& lhs, const TestEnumExtended&
rhs) {
    return static_cast<int32_t>(lhs) == static_cast<int32_t>(rhs);
}
inline bool operator!=(const TestEnumExtended& lhs, const TestEnum&
rhs) {
    return static_cast<int32_t>(lhs) != static_cast<int32_t>(rhs);
}
inline bool operator!=(const TestEnum& lhs, const TestEnumExtended&
rhs) {
    return static_cast<int32_t>(lhs) != static_cast<int32_t>(rhs);
}
inline CommonAPI::InputStream& operator>>(CommonAPI::InputStream&
inputStream, TestEnumExtended2& enumValue) {
    return inputStream.readEnumValue<int32_t>(enumValue);
}

inline CommonAPI::OutputStream& operator<<(CommonAPI::OutputStream&
outputStream, const TestEnumExtended2& enumValue) {
    return
outputStream.writeEnumValue(static_cast<int32_t>(enumValue));
}

struct TestEnumExtended2Comparator {
    inline bool operator()(const TestEnumExtended2& lhs, const
TestEnumExtended2& rhs) const {
        return static_cast<int32_t>(lhs) < static_cast<int32_t>(rhs);
    }
};

inline bool operator==(const TestEnumExtended2& lhs, const TestEnum&
rhs) {
    return static_cast<int32_t>(lhs) == static_cast<int32_t>(rhs);
}
inline bool operator==(const TestEnum& lhs, const TestEnumExtended2&
rhs) {
    return static_cast<int32_t>(lhs) == static_cast<int32_t>(rhs);
}
inline bool operator!=(const TestEnumExtended2& lhs, const TestEnum&
```

```
rhs) {
    return static_cast<int32_t>(lhs) != static_cast<int32_t>(rhs);
}
inline bool operator!=(const TestEnum& lhs, const TestEnumExtended2&
rhs) {
    return static_cast<int32_t>(lhs) != static_cast<int32_t>(rhs);
}

inline bool operator==(const TestEnumExtended2& lhs, const
TestEnumExtended& rhs) {
    return static_cast<int32_t>(lhs) == static_cast<int32_t>(rhs);
}
inline bool operator==(const TestEnumExtended& lhs, const
TestEnumExtended2& rhs) {
    return static_cast<int32_t>(lhs) == static_cast<int32_t>(rhs);
}
inline bool operator!=(const TestEnumExtended2& lhs, const
TestEnumExtended& rhs) {
    return static_cast<int32_t>(lhs) != static_cast<int32_t>(rhs);
}
inline bool operator!=(const TestEnumExtended& lhs, const
TestEnumExtended2& rhs) {
    return static_cast<int32_t>(lhs) != static_cast<int32_t>(rhs);
}

static inline const char* getTypeCollectionName() {
    return "commonapi.tests.DerivedTypeCollection";
}
} // namespace DerivedTypeCollection
} // namespace tests
```

```
} // namespace commonapi
#endif // COMMONAPI_TESTS_DERIVED_TYPE_COLLECTION_H
```

Interfaces

Basics

For the Franca interface *name* a class *name* is generated which provides the methods *getInterfaceName* and *getInterfaceVersion*. The Version is mapped to a struct CommonAPI::Version (source code here).

A basic example shows the generated header file *name.h* (see the section General Design and Namespaces above).

Methods

Franca IDL supports the definition of methods and broadcasts. Methods can have several in and out parameters; if an additional flag fireAndForget is specified, no out parameters are permitted.   Broadcasts can have only out parameters. Methods without the fireAndForget flag can return an error which can be specified in Franca IDL as an enumeration.

TBD: Selective Broadcasts

In Franca IDL there is no difference between an asynchronous or synchronous call of methods; the CommonAPI will provide both. The user of the API can decide which variant he calls.

The example below shows the definition of a method and a broadcast. For the method without the fireAndForget flag the synchronous variant of the function call (getProperty) and the asynchronous variant (getPropertyAsync) is generated. For both variants an additional return value *CallStatus* is provided which is defined as enumeration:

```
enum class CallStatus {
 SUCCESS,
 OUT_OF_MEMORY,
 NOT_AVAILABLE,
 CONNECTION_FAILED,
 REMOTE_ERROR
};
```

The *CallStatus* defines the transport layer result of the call, i.e. it returns

- SUCCESS, if the remote call returned successfully.
- OUT_OF_MEMORY, if sending the call or receiving the reply could not be completed due of a lack of memory.
- NOT_AVAILABLE, if the corresponding service for the remote method call is not available.
- CONNECTION_FAILED, if there is no connection to the communication medium available.
- REMOTE_ERROR, if the sent remote call does not return (in time).
    - NOT considered to be a remote error is an application level error that is defined in the corresponding Franca interface, because from the point of view of the transport layer the service still returned a valid answer.
    - It IS considered to be a remote error if no answer for a sent remote method call is returned within a defined time. It is discouraged to allow the sending of any method calls without a defined timeout. This timeout may be middleware specific. This timeout may also be configurable by means of a Franca Deployment Model. It is NOT configurable at runtime by means of the Common API.

For the return parameters a *function* object is created which is passed to the asynchronous method call. This *function* object can then be used directly in the client application as function pointer to a callback function or be bound to a function with a different signature. The usage of std::bind is not enforced but must be possible.

The bound callback *function* object will be called in any case:

- If the call returns successfully: Once the remote method call successfully returns, the callback *function* object is called with *SUCCESS* for its *CallStatus* and any received parameters.
- If a transport layer error occurs: If an error occurs that would trigger the method to return anything other but *SUCCESS* for its *CallStatus*, the callback has to be called with the corresponding *CallStatus* value. All other values that are input to the callback may remain unitialized in this case.

The asynchronous call returns the *CallStatus* as future object. This allows the synchronization of asynchronous calls to a defined time. The future object will attain its value at the same time at which the callback *function* object is called.

Example:

fidl	C++

<pre>package example.user
 	
interface name {
 version {major 1
minor 0}
 	
 method getProperty {
  in {
   UInt32 ID
  }
  out {
   String Property
  }
 }
 	
 method newMessage
fireAndForget {
  in {
   String MessageName
  }
 }
 	
 broadcast
signalChanged {
  out {
   UInt32 NewValue
  }
 }
}</pre>
	<pre>namespace example {
	namespace user {
	
	class nameProxyBase: virtual public
	CommonAPI::Proxy {
	 public:
	    typedef CommonAPI::Event<uint32_t>
	SignalChangedEvent;
	    typedef std::function<void(const
	CommonAPI::CallStatus&, const std::string&)>
	GetPropertyAsyncCallback;
	
	    virtual SignalChangedEvent&
	getSignalChangedEvent() = 0;
	
	    virtual void getProperty(const uint32_t& ID,
	CommonAPI::CallStatus& callStatus, std::string&
	Property) = 0;
	    virtual std::future<CommonAPI::CallStatus>
	getPropertyAsync(const uint32_t& ID,
	GetPropertyAsyncCallback callback) = 0;
	
	    /**
	     * @invariant Fire And Forget
	     */
	    virtual void newMessage(const std::string&
	MessageName, CommonAPI::CallStatus& callStatus) =
	0;
	};
	
	} // namespace user
	} // namespace example</pre>

The implementation of broadcasts uses an event class which is defined in the file event.h of the CommonAPI. The class Event provides the methods subscribe and unsubscribe to the client application.
An addition error return value is implemented as additional out parameter (see example).

fidl	C++

<pre>method
getProperty {
  error {
    OK
    NOT_OK
  }
}</pre>
	<pre>enum class getPropertyError: int32_t {
	  OK,
	  NOT_OK
	};
	
	virtual void getProperty(CommonAPI::CallStatus&
	callStatus,
	                                  getPropertyError&
	methodError) = 0;</pre>

The CommonAPI does not provide the possibility to cancel asynchronous calls.

Attributes

An attribute of an interface is defined by name and type. Additionally the specification of an attribute can have two flags:

- noSubscriptions

- readonly

CommonAPI provides a basic implementation of the attribute interface and a mechanism for so-called extensions. The basic implementation is shown in the example below. For the four possible combinations of flags:

- standard attributes with no additional flag
- readonly attributes (readonly flag is set)
- non observable attributes (noSubscription flag)
- and non observable and non writable attributes (both flags are set)

Template classes for each of those four types of attributes are defined in the header file Attribute.h. The CommonAPI provides a getter function which returns a reference to an instance of the appropriate attribute template class (see example below).

<table>
<tr><th>fidl</th><th>C++</th></tr>
<tr><td>

```
package example.user

interface name {
 version {major 1
minor 0}
 attribute UInt32 A
 attribute UInt32 B
noSubscriptions
 attribute UInt32 C
readonly
        attribute
UInt32 D readonly
noSubscriptions
}
```

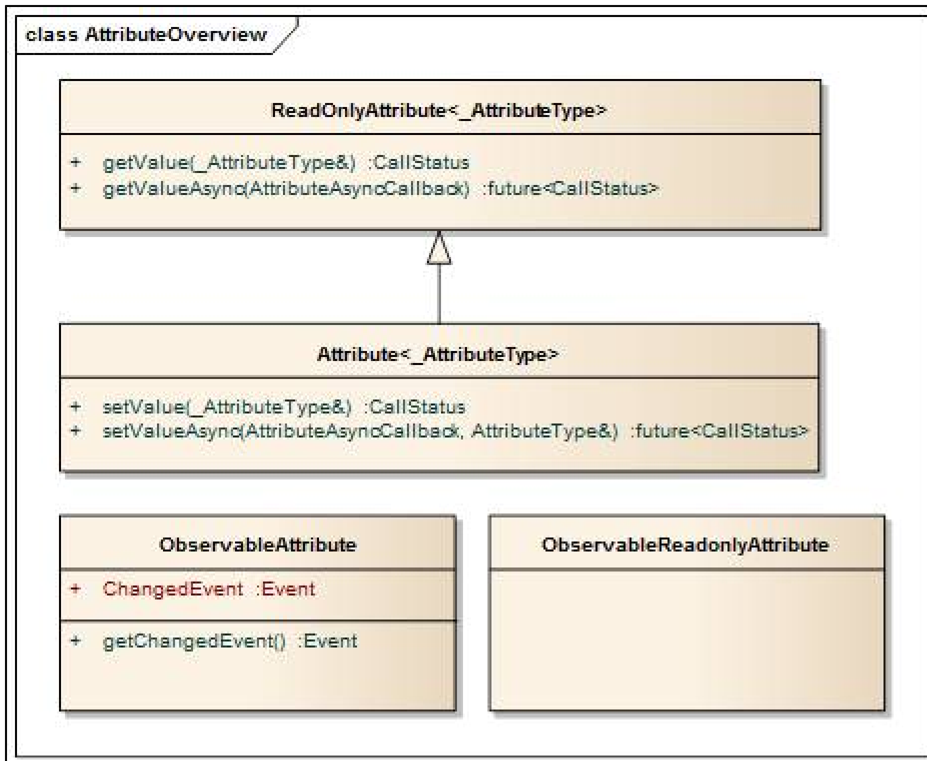
</td><td>

```
namespace example {
namespace user {

class nameProxyBase: virtual public
CommonAPI::Proxy {
 public:
    typedef
CommonAPI::ObservableAttribute<uint32_t>
AAttribute;
    typedef CommonAPI::Attribute<uint32_t>
BAttribute;
    typedef
CommonAPI::ObservableReadonlyAttribute<uint32_t>
CAttribute;
    typedef
CommonAPI::ReadonlyAttribute<uint32_t>
DAttribute;

    virtual AAttribute& getAAttribute() = 0;
    virtual BAttribute& getBAttribute() = 0;
    virtual CAttribute& getCAttribute() = 0;
    virtual DAttribute& getDAttribute() = 0;};
} // namespace user
} // namespace example
```

</td></tr>
</table>



Observable attributes provide a ChangedEvent which can be used to subscribe to updates to the attribute. This Event works exactly as all other events, described in SysInfraEGCppCommonAPISpecification#Events

By default, the attributes are not cached in client side. Creating a cache on client side is not an implementation-specific detail that should be a part of the logical interface specification, nor is it a platform- or middleware-dependent parameter. Moreover, the requirements for an attribute cache can be very different depending on the application specific use case. Differences in points of view include, but are not limited to:

- Is the cache value to be updated on any value changed event or is it to be updated periodically?
- Should calls to getters of potentially cached values be blocking or non-blocking?
- Should caching be configurable per attribute or per proxy, or should caching always be enabled?
- Is getting a cached value a distinct method call or is it to be included transparently within the standard getter methods?

Because of this, there should be a general scheme to include individual extensions in order to provide any additional features for attributes. This would prevent an exponential growth of configuration possibilities within the Common API and also relieve Common API developers from the necessity to
always implement all specified features for their specific middleware, regardless of whether the feature is supported by the middleware or not. On the other hand, it gives complete freedom to application developers to add an implementation for their specific needs to attribute handling.

The basic principle is that the user of the API has to implement an extension class that is derived from the base class AttributeExtension. The AttributeExtension is packed in a wrapper class which in turn is generated for each attribute the Proxy has. A wrapper for a given attribute only then is mixed into the proxy if an extension for this given attribute is defined during construction time.

The wrapper forwards the correct attribute to the constructor of the extension, so that the extension sees nothing but the attribute it should extend. Wrappers are written as templates, so that all wrappers can be reused for all attributes of the same category. As soon as an extension for an attribute is defined during construction time, the extension class will be instantiated and a method to retrieve the extended attribute will be added to the proxy.

Such an solution requires the proxy to be made ready for *mixins*. The proxy inherits from all *mixins* that are defined during construction time, so that their interface is added directly to the proxy itself. The interface that would be added to the proxies in our case would be the interface of the defined attribute extension wrappers, which in turn provide access to the actual attribute extensions.

By using variadic templates the amount of possible *mixins* is arbitrary. However, because a given proxy may not inherit from the same class twice, **only one extension per attribute per proxy is possible**.

The base class for extensions is defined in *AttributeExtension.h*.

The following examples show an implementation of the extension class for an attribute and the generated code in the proxy class. For the implementation in the proxy factory, see *ProxyFactory.h*.

C++ (Extension class for caching attributes)

```
template<typename _AttributeType>
class AttributeCacheExtension: public AttributeExtension<_AttributeType> {
    typedef AttributeExtension<_AttributeType> __baseClass_t;

public:
    typedef typename _AttributeType::value_t value_t;
    typedef typename _AttributeType::AttributeAsyncCallback
AttributeAsyncCallback;

    AttributeCacheExtension(_AttributeType& baseAttribute) :
                    AttributeExtension<_AttributeType>(baseAttribute),
                    isCacheValid_(false)
   {
      auto changedEvent = baseAttribute.getInternalChangedEvent();
      changedEvent.subscribe(std::bind(
                        &AttributeCacheExtension::onValueUpdate,
                        this,
                        std::placeholders::_1));
   }
   ~AttributeCacheExtension() {}

   bool getCachedValue(value_t& value) const {
        if (isCacheValid_) {value = cachedValue_;}
        return isCacheValid_;
   }

private:

    void onValueUpdate(const value_t& t) {
        isCacheValid_ = true;
        cachedValue_ = t;
    }
    mutable bool isCacheValid_;
    mutable value_t cachedValue_;
};
```

The definition of the specific wrapper class in the proxy:

C++ (Wrapper class for attribute A)

```
template <template <typename > class _ExtensionType>
class AAttributeExtension {
public:
  typedef _ExtensionType<nameProxyBase::AAttribute> extension_type;

  static_assert(std::is_base_of<typename
CommonAPI::AttributeExtension<nameProxyBase::AAttribute>,
                     extension_type>::value, "Not CommonAPI Attribute
Extension!");

        AAttributeExtension(nameProxyBase& proxy):
attributeExtension_(proxy.getAttribute()) {}

        inline extension_type& getAAttributeExtension() {return
attributeExtension_;}

private:
        extension_type attributeExtension_;
};
```

For the implementation in the proxy factory, see *ProxyFactory.h*.

Events

Events provide an asyncronous interface to remotely triggered actions. This covers broadcast methods in Franca IDL and change events for attributes Every proxy also provides an availabity event which can be used for notifications of the proxies status. The Events provide a subscribe and unsubscribe method which allow registration and de-registration of callbacks.

The public interface of the event class is as follows:

```
class Event {
public:
        typedef std::function<void(const _Arguments&...)> Listener;
        typedef std::list<Listener> ListenersList;
        typedef typename ListenersList::iterator Subscription;

        Subscription subscribe(const Listener& listener);
        void unsubscribe(Subscription listenerSubscription);
        virtual ~Event();
}
```

Attributes and Methods Example

test-interface-proxy.fidl

```
package commonapi.tests
import commonapi.tests.* from "test-derived-types.fidl"

interface TestInterface {
 version { major 1 minor 0 }

 attribute UInt32 TestPredefinedTypeAttribute
 attribute DerivedTypeCollection.TestStructExtended TestDerivedStructAttribute
 attribute DerivedTypeCollection.TestArrayUInt64 TestDerivedArrayAttribute

 method testVoidPredefinedTypeMethod {
  in {
   UInt32 uint32Value
   String stringValue
  }
 }

 method testPredefinedTypeMethod {
  in {
   UInt32 uint32InValue
   String stringInValue
  }
  out {
   UInt32 uint32OutValue
   String stringOutValue
  }
 }

 method testVoidDerivedTypeMethod {
  in {
   DerivedTypeCollection.TestEnumExtended2 testEnumExtended2Value
   DerivedTypeCollection.TestMap testMapValue
  }
 }

 method testDerivedTypeMethod {
  in {
   DerivedTypeCollection.TestEnumExtended2 testEnumExtended2InValue
   DerivedTypeCollection.TestMap testMapInValue
  }
  out {
   DerivedTypeCollection.TestEnumExtended2 testEnumExtended2OutValue
   DerivedTypeCollection.TestMap testMapOutValue
  }
 }

 broadcast TestPredefinedTypeBroadcast {
  out {
   UInt32 uint32Value
   String stringValue
  }
 }
}
```

TestInterface.h

```
namespace commonapi {
namespace tests {

class TestInterface {
 public:
    virtual ~TestInterface() { }

    static inline const char* getInterfaceName();
    static inline CommonAPI::Version getInterfaceVersion();
};

const char* TestInterface::getInterfaceName() {
    return "commonapi.tests.TestInterface";
}

CommonAPI::Version TestInterface::getInterfaceVersion() {
    return CommonAPI::Version(1, 0);
}


} // namespace tests
} // namespace commonapi
```

TestInterfaceProxyBase.h	TestInterfa

```
namespace commonapi {
namespace tests {

class TestInterfaceProxyBase: virtual public CommonAPI::Proxy {
 public:
    typedef CommonAPI::ObservableAttribute<uint32_t>
TestPredefinedTypeAttributeAttribute;
    typedef
CommonAPI::ObservableAttribute<DerivedTypeCollection::TestStructExtended>
TestDerivedStructAttributeAttribute;
    typedef
CommonAPI::ObservableAttribute<DerivedTypeCollection::TestArrayUInt64>
TestDerivedArrayAttributeAttribute;
    typedef CommonAPI::Event<uint32_t, std::string>
TestPredefinedTypeBroadcastEvent;
    typedef std::function<void(const CommonAPI::CallStatus&)>
TestVoidPredefinedTypeMethodAsyncCallback;
    typedef std::function<void(const CommonAPI::CallStatus&, const
uint32_t&, const std::string&)> TestPredefinedTypeMethodAsyncCallback;
    typedef std::function<void(const CommonAPI::CallStatus&)>
TestVoidDerivedTypeMethodAsyncCallback;
    typedef std::function<void(const CommonAPI::CallStatus&, const
DerivedTypeCollection::TestEnumExtended2&, const
DerivedTypeCollection::TestMap&)> TestDerivedTypeMethodAsyncCallback;

    virtual TestPredefinedTypeAttributeAttribute&
getTestPredefinedTypeAttributeAttribute() = 0;
    virtual TestDerivedStructAttributeAttribute&
getTestDerivedStructAttributeAttribute() = 0;
    virtual TestDerivedArrayAttributeAttribute&
getTestDerivedArrayAttributeAttribute() = 0;
```

```
nar
nar

ter
cla
_At
 pu

Cor

cor

Cor

cor

tes

Der
```

```
    virtual TestPredefinedTypeBroadcastEvent&
getTestPredefinedTypeBroadcastEvent() = 0;


    virtual void testVoidPredefinedTypeMethod(const uint32_t& uint32Value,
const std::string& stringValue, CommonAPI::CallStatus& callStatus) = 0;
    virtual std::future<CommonAPI::CallStatus>
testVoidPredefinedTypeMethodAsync(const uint32_t& uint32Value, const
std::string& stringValue, TestVoidPredefinedTypeMethodAsyncCallback
callback) = 0;

    virtual void testPredefinedTypeMethod(const uint32_t& uint32InValue,
const std::string& stringInValue, CommonAPI::CallStatus& callStatus,
uint32_t& uint32OutValue, std::string& stringOutValue) = 0;
    virtual std::future<CommonAPI::CallStatus>
testPredefinedTypeMethodAsync(const uint32_t& uint32InValue, const
std::string& stringInValue, TestPredefinedTypeMethodAsyncCallback
callback) = 0;

    virtual void testVoidDerivedTypeMethod(const
DerivedTypeCollection::TestEnumExtended2& testEnumExtended2Value, const
DerivedTypeCollection::TestMap& testMapValue, CommonAPI::CallStatus&
callStatus) = 0;
    virtual std::future<CommonAPI::CallStatus>
testVoidDerivedTypeMethodAsync(const
DerivedTypeCollection::TestEnumExtended2& testEnumExtended2Value, const
DerivedTypeCollection::TestMap& testMapValue,
TestVoidDerivedTypeMethodAsyncCallback callback) = 0;

    virtual void testDerivedTypeMethod(const
DerivedTypeCollection::TestEnumExtended2& testEnumExtended2InValue, const
DerivedTypeCollection::TestMap& testMapInValue, CommonAPI::CallStatus&
callStatus, DerivedTypeCollection::TestEnumExtended2&
testEnumExtended2OutValue, DerivedTypeCollection::TestMap&
testMapOutValue) = 0;
    virtual std::future<CommonAPI::CallStatus>
testDerivedTypeMethodAsync(const DerivedTypeCollection::TestEnumExtended2&
testEnumExtended2InValue, const DerivedTypeCollection::TestMap&
testMapInValue, TestDerivedTypeMethodAsyncCallback callback) = 0;
};
```

```
tes


cor
Der
tes

Der
tes

 pr

};

nar






Cor
ext



att










Cor
ext



att
```

```
} // namespace tests
} // namespace commonapi
```

```
Cor
ext

att

} ,

//
//
//
ter
Tes

}

ter
Tes

}

ter
tyr
Tes

}

ter
tyr
Tes

}

ter
tyr
Tes

}

ter
tyr
Tes

}

ter
vo:
cor
```

```
}

ter
stc
Tes
cor

}

ter
vo:
stc
st1

st1
}

ter
stc
Tes
stc

}

ter
vo:
De1
tes

}

ter
stc
Tes
De1
tes

}

ter
vo:
De1
tes
tes

tes
}

ter
stc
De1
tes

}


} /
} /

nar
ter
st1
```

```
};
}
```

Addresses, service discovery, connect/disconnect

A client application which needs to get access to a certain interface of a service has to call the proxy factory (see *ProxyFactory.h*) which delivers an instance of the proxy class (see *Proxy.h*). The proxy factory can be obtained from the CommonAPI runtime environment. The proxy factory implements the following functions:

- getRuntime()
- isServiceAvailable(serviceInstanceID, serviceInterfaceName, serviceDomainName)
- isServiceAvailable(serviceInstanceID, serviceDomainName)
- getAvailableServiceInstances(serviceInterfaceName, serviceDomainName)

The proxy itself (*proxy.h*) implements:

```
class Proxy {
 public:
 virtual ~Proxy() { }

  // The addressing scheme has the following format: "domain:service:instance"
  virtual std::string getAddress() const = 0;

  // i.e. "local"
  virtual const std::string& getDomain() const = 0;

  // i.e. "com.bmw.infotainment"
  virtual const std::string& getServiceId() const = 0;

  // i.e. "com.bmw.infotainment.low"
  virtual const std::string& getInstanceId() const = 0;

 virtual bool isAvailable() const = 0;

 virtual ProxyStatusEvent& getProxyStatusEvent() = 0;

 virtual InterfaceVersionAttribute& getInterfaceVersionAttribute() = 0;
};
}
```

Proxy availability

isAvailable is a non-blocking check whether the remote service for this proxy currently is available. Always returns *false* until the availability of the proxy is determined. The proxy actively determines its availability status asynchronously and ASAP as soon as it is created, and maintains the correct state afterwards.

- Calls to synchronous methods will block until the initial availability status of the proxy is determined. As soon as the availability status has been determined at least once, calls to synchronous methods will return *NOT_AVAILABLE* as value for the *CallStatus* whenever *isAvailable()* would return *false*.
- Calls to asynchronous methods do not wait for the initial availability status to be determined. Calls to asynchronous methods will instantly call the given callback with *NOT_AVAILABLE* as value for the *CallStatus* whenever *isAvailable()* would return *false*.
- You may subscribe to the *ProxyStatusEvent* in order to have a callback notified whenever the availability status of the proxy changes. It is guaranteed that the callback is notified of the proxy's currently known availability status at the same instant in which the subscription is done (i.e. the callback will most likely be called with a value of *false* if you subscribe for this event right after the proxy has been instantiated).

Runtime

The Common API Runtime (source code here) is the base class from which all class loading starts. The Common API Runtime accesses a config file to determine which specific middleware runtime library shall be loaded. Middleware libraries are either linked statically or are provided as shared objects (file extension .so), so they can be loaded dynamically. To make dynamic loading controllable, additional configuration parameters are available, see the chapter on "Configuration Files" below.

Linking the Middleware at Compile Time

A specific Middleware runtime registers itself with the Common API Runtime during startup time via static initialization. Static initialization is done via a statically defined method in a .cpp file with *__attribute((constructor))* as prefix. Within this method, *registerRuntimeLoader* of the Common API Runtime will be called. A specific middleware runtime needs to register itself with a well known string identifier, e.g. "DBus" for a D-Bus middleware.

The architecture was chosen this way in order to ease later support of dynamic linkage. No template based architecture was used to cross the boundary from Common API to a specific middleware, because template parameters can not be substituted dynamically.

Example: Static initialization within DBusRuntime.cpp

```
__attribute__((constructor)) void registerDBusMiddleware(void) {
    Runtime::registerRuntimeLoader("DBus", &DBusRuntime::getInstance);
}
```

Linking the Middleware at Runtime

CommonAPI supports the loading of Middleware specific libraries at runtime, without linking them to the executable beforehand. For this purpose, each Middleware binding library provides a struct defined as *extern "C"*, which provides information on the well known name of the Middleware, plus a function pointer to its runtime loader.

The struct itself is defined in **CommonAPI/MiddlewareInfo.h**, and is called **CommonAPI::MiddlewareInfo**. The function template is **typedef std::shared_ptr<Runtime> (*MiddlewareRuntimeLoadFunction) ()** and is defined in the same header file just above the mentioned struct.

Middleware header file	Middleware source file
`extern "C" const CommonAPI::MiddlewareInfo middlewareInfo;`	`const CommonAPI::MiddlewareInfo middlewareInfo(<MiddlewaresWellKnownName>, &CommonAPI::<MiddlewareNamespace>::<MiddlewareRunt:`

Factory

The CommonAPI::Factory (source code here) class builds the proxies and registers stubs for a specific instance of a commonapi runtime (i.e. a particular binding) and connection. If there are multiple connections to the rpc mechanism (not a specific service) multiple instances of Factory are needed. This class provides templated build methods which return particular instances of proxies according to the templates and passed address. Examples of how to use these methods and of the config files which can be provided are included below.

Example usage

```
int main(void) {

    //Loads default or first runtime binding defined in config file
    std::shared_ptr<CommonAPI::Factory> factory =
                    CommonAPI::Runtime::load()->createFactory();

    //Loads a named runtime binding, either according to a well known name defined
in the binding or a defined arbitrary alias as stated in the config file
    std::shared_ptr<CommonAPI::Factory> factory2 =
                    CommonAPI::Runtime::load("namedBinding")->createFactory();

    //Loads a Proxy with no attribute extensions
    auto defaultTestProxy =
factory->buildProxy<com::bmw::test::TestProxy>("local:com.bmw.test.EchoService:com.bm
auto allCachedTestProxy = factory->buildProxyWithDefaultAttributeExtension<
                    com::bmw::test::TestProxy,

CommonAPI::Extensions::AttributeCacheExtension>("local:com.bmw.test.EchoService:com.b
auto specificAttributeExtendedTestProxy = factory->buildProxy<
                    com::bmw::test::TestProxy,

com::bmw::test::TestProxyTest3AttributeExtension<CommonAPI::Extensions::AttributeCach
>
            ("local:com.bmw.test.EchoService:com.bmw.test.EchoInstance");
}
```

Interface Factory

The Factory class provides a registerService() and unregisterService() Method. These allow the activation and deactivation of services and stubs via a complete common api address and a provided stub. The usage is similar to that for Proxies, except that no object is returned. Included below is the header for Common API factory.

Example usage

```
int main(void) {

//Loads default or first runtime binding defined in config file
auto runtime_ = CommonAPI::Runtime::load();

std::shared_ptr<CommonAPI::Factory> factory = runtime_->createFactory();
const std::string serviceAddress_ =
"local:commonapi.tests.TestInterface:commonapi.tests.TestInterface";

auto myStub = std::make_shared<commonapi::tests::TestInterfaceStubDefault>();
bool success = factory->registerService(myStub, serviceAddress_);

//Now use the stub...

}
```

Provider

A provider is made up of two parts, the *interface adapter* and its helpers and the *stub*. The *interface adapters* need to be generated individually for each service, but they remain invisible to the application developer, except when designing a derived stub class.

Stubs

Stubs are the methods to be implemented by a provider. This includes the remotely callable methods defined in the interface and the callbacks for attribute accessors. These are provided as a default stub class with a default blank implementation for all methods and storage and handling of attribute values inside the stub, which allows the application developer to overwrite the methods in a subclass as needed.

Alternatively the pure virtual root class can be implemented directly, which also allows delegation of attribute handlers to other parts of the code.

A given instance of a stub can be appended to one and only one interface adapter.

Methods

For all methods, only a single callback function is provided. The name is equal to the name provided in the defining franca file.

Broadcasts

Broadcasts are provided as fire and forget methods, named *fire<BroadcastName>*, within the handler. In order to keep the interaction of the application developer with the generated interface in one place, those methods are also exposed in the service stub. Thereby transparent forwarding of the calls to the adapter is realized via the shared pointer to the adapter held by the stub.

Attributes

For Attributes, three callback methods are provided in the stub:

The first, "validate<AttributeName>RequestedValue", allows for verification of the attribute value before it actually is set. This callback receives the new requested value and should return true if the operation is possible or false if the operation cannot be completed at all. In the default case, this callback will return true.

The second, "trySet<AttributeName>Attribute", setting and modification of the attribute value. This callback receives the new requested value as a reference and modifies this as needed. Additionally it returns true if the value was modified before setting, or false if it is set as requested. In the default case, this callback will return the same value and true (i.e. the new value for the attribute is accepted without further modifications).

The third callback, "onRemote<AttributeName>Changed", is called if the new value of the attribute differs from the old, and allows the service to do arbitrary work in response to such an attribute change. This is called after the clients are informed of the change.

These are protected to avoid exposing them to the API code, as they should only be called by the adapter. This is done via a generated class, the <interfaceName>StubRemoteEvent, which has access to these methods.

The CommonAPI library will transmit any "attribute changed" event notifications to remote listeners after the verify callback but before the change callback, if and only if the value of the attribute actually has changed after verification.

Default implementation

The default implementation of a generated stub does

- nothing on method calls
- return the new value of an attribute on a verify callback
- do nothing on an attribute changed callback

Example generated Stub

test-interface-proxy.fidl

```
class TestInterfaceStubDefault : public TestInterfaceStub {
 public:
    TestInterfaceStubDefault();

    TestInterfaceStubRemoteEvent* initStubAdapter(const
std::shared_ptr<TestInterfaceStubAdapter>& stubAdapter);

    virtual const uint32_t& getTestPredefinedTypeAttributeAttribute();
    void setTestPredefinedTypeAttributeAttribute(uint32_t value);

    virtual const DerivedTypeCollection::TestStructExtended&
getTestDerivedStructAttributeAttribute();
    void
setTestDerivedStructAttributeAttribute(DerivedTypeCollection::TestStructExtended
value);

    virtual const DerivedTypeCollection::TestArrayUInt64&
getTestDerivedArrayAttributeAttribute();
    void
setTestDerivedArrayAttributeAttribute(DerivedTypeCollection::TestArrayUInt64
```

```
value);


    virtual void testVoidPredefinedTypeMethod(uint32_t uint32Value, std::string
stringValue);

    virtual void testPredefinedTypeMethod(uint32_t uint32InValue, std::string
stringInValue, uint32_t& uint32OutValue, std::string& stringOutValue);

    virtual void testVoidDerivedTypeMethod(DerivedTypeCollection::TestEnumExtended2
testEnumExtended2Value, DerivedTypeCollection::TestMap testMapValue);

    virtual void testDerivedTypeMethod(DerivedTypeCollection::TestEnumExtended2
testEnumExtended2InValue, DerivedTypeCollection::TestMap testMapInValue,
DerivedTypeCollection::TestEnumExtended2& testEnumExtended2OutValue,
DerivedTypeCollection::TestMap& testMapOutValue);


    virtual void fireTestPredefinedTypeBroadcastEvent(const uint32_t& uint32Value,
const std::string& stringValue);

 protected:
    void onRemoteTestPredefinedTypeAttributeAttributeChanged();
    bool trySetTestPredefinedTypeAttributeAttribute(uint32_t value);
    bool validateTestPredefinedTypeAttributeAttributeRequestedValue(const uint32_t&
value);

    void onRemoteTestDerivedStructAttributeAttributeChanged();
    bool
trySetTestDerivedStructAttributeAttribute(DerivedTypeCollection::TestStructExtended
value);
    bool validateTestDerivedStructAttributeAttributeRequestedValue(const
DerivedTypeCollection::TestStructExtended& value);

    void onRemoteTestDerivedArrayAttributeAttributeChanged();
    bool
trySetTestDerivedArrayAttributeAttribute(DerivedTypeCollection::TestArrayUInt64
value);
    bool validateTestDerivedArrayAttributeAttributeRequestedValue(const
DerivedTypeCollection::TestArrayUInt64& value);


 private:
    class RemoteEventHandler: public TestInterfaceStubRemoteEvent {
     public:
        RemoteEventHandler(TestInterfaceStubDefault* defaultStub);

        virtual bool onRemoteSetTestPredefinedTypeAttributeAttribute(uint32_t
value);
        virtual void onRemoteTestPredefinedTypeAttributeAttributeChanged();

        virtual bool
onRemoteSetTestDerivedStructAttributeAttribute(DerivedTypeCollection::TestStructExten
value);
        virtual void onRemoteTestDerivedStructAttributeAttributeChanged();

        virtual bool
onRemoteSetTestDerivedArrayAttributeAttribute(DerivedTypeCollection::TestArrayUInt64
value);
        virtual void onRemoteTestDerivedArrayAttributeAttributeChanged();


     private:
```

```
        TestInterfaceStubDefault* defaultStub_;
    };

    RemoteEventHandler remoteEventHandler_;
    std::shared_ptr<TestInterfaceStubAdapter> stubAdapter_;

    uint32_t testPredefinedTypeAttributeAttributeValue_;
    DerivedTypeCollection::TestStructExtended
testDerivedStructAttributeAttributeValue_;
    DerivedTypeCollection::TestArrayUInt64
testDerivedArrayAttributeAttributeValue_;
};
```

```
} // namespace tests
}
```

Interface Adapter

An interface adapter is responsible for serialization and deserialization of messages, as well as the dispatching of the (de-)serialized messages to a registered stub. The stub forwards the broadcasts that are fired by a call to the "fire<BroadcastName>" methods to the adapter via the held shared pointer in order to send them.

An interface adapter for a specific middleware needs to be generated in order to provide dispatching and callback handling that matches the stub. The procedure to attain the correct interface adapter is equivalent to the procedure to attain the correct proxy on client side.

A given instance of an interface handler can serve one and only one stub at a time. The connection from the adapter to the stub is established during build time by a call to the predefined *initStubAdapter* method of the Stub template class.

CommonAPI Interface Adapter class

The common ancestor of any middleware specific handler (CommonAPI::StubAdapter) and stub (CommonAPI::Stub) classes defines a basic interface to retrieve general information about the service (source code here).

The generated TestInterfaceAdapter inherits from the same TestInterface as TestInterfaceProxy.

Threading Model

CommonAPI supports multithreaded execution (standard threading) as well as single threaded execution (i.e. Mainloop integration). The decision which of both is desired happens when *CommonAPI::Runtime::createFactory* is called. If single threaded execution is desired, a *CommonAPI::MainLoopContext* has to be created, and then has to be passed as an argument to this method. All objects that are created by a factory that was instantiated this way will be controllable via the mainloop context that was handed to this factory. If a factory is not given this parameter during instantiation, standard threading will be set up for all objects that are created by this factory.

Standard Threading

To create a factory that uses standard threading, a factory has to be created this way:

```
std::shared_ptr<CommonAPI::Runtime> runtime = CommonAPI::Runtime::load();
std::shared_ptr<CommonAPI::Factory> factory = runtime->createFactory();
```

Single Threaded (Mainloop integration)

A *CommonAPI::MainLoopContext* provides nothing but hooks for callbacks that will be called on specific binding internal events. Internal events may be

- (De-)Registration of a *CommonAPI::DispatchSource*
- (De-)Registration of a *CommonAPI::Watch*
- (De-)Registration of a *CommonAPI::Timeout*
- Issuing of a wakeup call

Each of these calls has to be mapped to an appropriate method in the context of the actual Mainloop that does the single threaded execution. CommonAPI does **NOT** provide a fully fledged implementation for a Mainloop!

What the mainloop related interfaces are meant for is this:

- *CommonAPI::DispatchSource*: Hooks that may have work ready that is to be done, e.g. dispatching a method call to a stub, dispatching a method return to a proxy callback and the like. There is no constraint on what kind of work may be represented by a DispatchSource, **BUT** a dispatch source may **NOT** be directly related to a file descriptor that is used to actually read or write the incoming or outgoing transmission from or to a transport!
- *CommonAPI::Watch*: Work that is related to a file descriptor that is used for reading from or writing to a transport is represented by Watches. **ANY** work that is not directly related to such a file descriptor may **NOT** be represented by watches!
- *CommonAPI::Timeout*: Represents the work that has to be done when a timeout occurs (e.g. deleting the structures that are used to identify an answer to an asynchronous call and calling the callback that is waiting for it with an appropriate error flag). A timeout stores internally both the interval of time within which it is to be dispatched, and the next moment in time the timeout is to be dispatched.

A *binding developer* **MUST** provide his own implementations for at least *CommonAPI::DispatchSource* and *CommonAPI::Watch* in order to ensure the functionality of his respective CommonAPI binding in the single threaded case, and must ensure that the appropriate instances of those classes are handed to the application via the MainLoopContext that was handed to the factory that is used to instantiate proxies and stubs.

An *application developer* **MAY** provide additional implementations of all these classes.

During instantiation (i.e. during the call to *CommonAPI::Runtime::createFactory*) a binding MAY already issue calls to the callbacks that are registered with the *CommonAPI::MainLoopContext*. Therefore, it is mandatory to do any registration of required callbacks BEFORE doing so.

To create a factory that supports Mainloop integration, a factory has to be created this way:

```
std::shared_ptr<CommonAPI::Runtime> runtime = CommonAPI::Runtime::load();
std::shared_ptr<CommonAPI::MainLoopContext> context =
runtime->getNewMainLoopContext();
  //Do any registration of callbacks for the actual Mainloop here!
std::shared_ptr<CommonAPI::Factory> factory = runtime->createFactory(context);
```

Configuring CommonAPI

Change the behavior of interfaces

There are basically two possibilities to realize a specific behavior of your interface or to realize new features:

- As described above the middleware implementation can be changed without changing the API for the applications. Changes in the configuration in the middleware or platform can be realized by changing the deployment specification and the deployment settings in the *.depl files.
- For attributes (see specification below) there is the possibility to define and implement so-called extensions. This allows the developer to extend the standard framework with own implementations in a predefined and specified way. One example is to implement a cache for attributes on proxy side.

Configuration Files

Each CommonAPI configuration file will define additional parameters for specific categories.
Which categories and which parameters for each of those categories are available will be detailed below.

All parameters for all categories are optional. For each omitted parameter a reasonable default will be set. Because of this, it is not mandatory to provide a config file unless you want to alter any of the configurable default values.

CommonAPI config files can be defined locally per binary, globally per binary or globally for all binaries.
If more than one config file is defined for a given binary (e.g. one locally and one globally) and a given category is defined in several of these config files, for each parameter that may be provided for this category the value found in the most specific config file will take precedence. If a category is defined several times within the same config file, the first occurrence of each parameter will take precedence.

All categories and all parameters are separated from each other by one or more newline characters.

CommonAPI Config files have to be named this way:

> *Binary local: "<FqnOfBinary>.conf", e.g. "/usr/bin/myBinary.conf" if the binary is "/usr/bin/myBinary"*
> *Binary global: "/etc/CommonApi/<NameOfBinary>.conf", e.g. "/etc/CommonAPI/myBinary.conf"*
> *Global: "/etc/CommonAPI/CommonAPI.conf"*

Available categories

Well known names of specific middleware bindings

Allows to set parameters that influence the loading procedure of specific middleware bindings.
The syntax is:

> *{binding:<well known binding name>}*
> *libpath=<Fully qualified name of the library of the binding>*
> *alias=<One or more desired aliases for the binding, separated by ":">*
> *genpath=<One or more fully qualified names to libraries containing additional (generated) code for this binding, separated by ":">*
> *default*

- **libpath**: Provides a fully qualified name that replaces the search path when trying to dynamically load the identified binding. The library found at libpath will take precedence over all other dynamically discoverable libraries for this binding.
  - If a library for the specified middleware binding is linked to the binary already, this parameter will have no effect.
  - *Not* finding an appropriate library at libpath is considered to be an error! In this case, no further attempts to resolve the library will be made, and the load function will return an empty *std::shared_ptr<CommonAPI::Runtime>*. If an explicit error state is desired, one of the overloaded *Runtime::load()* functions may be called to pass in an instance of *Runtime::LoadState* as argument.

- **alias**: In order to load a specific middleware binding, one normally has to know the well known name of the middleware (e.g. "DBus" for the D-Bus middleware binding) and pass this name as parameter when calling CommonAPI::Runtime::load("<name>"). _*alias* maps the well known name for this purpose to one or more arbitrary aliases, thereby decoupling the loading of a specific middleware binding from its specific name.
    - You MAY specify this parameter more than once for a binding. The effect will be the same as if you had one alias parameter specifying the exact same names separated by ":".
    - If the same alias is specified more than once, only the first occurrence of the alias will be considered.
    - As CommonAPI itself does not know about which well known middleware names there are, it is possible to specify the well known name of an actual binding as an alias for any other middleware binding. In this case, the actual middleware binding will not be accessible any longer, unless you specify another unique alias for it.

- **genpath**: Specifies one or more paths at which a generic library containing additional (e.g. generated middleware and interface code for the middleware binding is to be found. This additional code will be injected when the specific middleware considers it to be the right time to do so.
    - You MAY specify this parameter more than once for a binding. The effect will be the same as if you had one genpath parameter specifying the exact same values separated by ":".
    - If *No* such parameter is defined, the standard search paths "/usr/lib" and "/usr/local/lib" plus any additional paths defined in the environment variable COMMONAPI_BINDING_PATH (see below) will be searched for any libraries that match the name pattern "lib<wellKnownMiddlewareName>Gen-<arbitraryName>.so[.major.minor.revision]". All matching libraries will be loaded.
    - *Not* finding an appropriate library at any single one of the defined genpaths may result in undefined behavior.

Note for developers
The *genpath* parameter will be parsed by the CommonAPI framework and stored in the singleton class *CommonAPI::Configuration*. Actually loading the libraries and following the rules described here however is task of the specific middleware binding. You might want to use the convenience methods provided in <CommonAPI/utils.h> for this purpose. By taking control of the actual proceedings, you may introduce additional mechanisms of discovering and loading such libraries, and you may defer the loading of these libraries until you deem it to be the right time to do so.

- **default**: Specifies the library for this binding as the default that is to be loaded if no parameter is given to *CommonAPI::Runtime::load()*.
    - *Not* finding an appropriate library for a configured default binding at neither specified nor the default paths is considered to be an error! In this case, no further attempts to resolve the library will be made, and the load function will return an empty *std::shared_ptr<CommonAPI::Runtime>*. If an explicit error state is desired, one of the overloaded *Runtime::load()* functions may be called to pass in an instance of *Runtime::LoadState* as argument.

Environment Variables

- **COMMONAPI_BINDING_PATH**: By default, the standard paths "/usr/lib" and "/usr/local/lib" will be searched for binding libraries that are loaded dynamically (i.e. at runtime without linking them to the binary beforehand). All paths defined in this environment variable will take precedence over those two default paths. Separator between several paths is ":".

Known Limitations

- Contract: Franca IDL *Contracts* are not supported.
- gcc: Currently, compilation requires C++11 features first introduced with gcc version 4.6. All necessary language features are provided since gcc version 4.4. though.

Attachments

Name	Größe	Ersteller	Erstellungsdatum	Kommentar
JPEG-Datei Bild1.jpg	22 kB	Juergen Gehring	Nov 15, 2012 16:09	
JPEG-Datei Bild2.jpg	32 kB	Juergen Gehring	Nov 15, 2012 16:23	
JPEG-Datei Bild3.jpg	40 kB	Juergen Gehring	Nov 15, 2012 16:25	
JPEG-Datei Bild4.jpg	38 kB	Juergen Gehring	Nov 15, 2012 16:27	
JPEG-Datei Bild5.jpg	27 kB	Juergen Gehring	Nov 15, 2012 17:32	
Datei test-derived-types.fidl	1 kB	Manfred Bathelt	Nov 12, 2012 13:05	FIDL containing enums, structures arrays and maps
Datei test-interface-proxy.fidl	1 kB	Manfred Bathelt	Nov 12, 2012 13:47	FIDL containing example interface specification
Datei test-predefined-types.fidl	0,5 kB	Manfred Bathelt	Nov 12, 2012 13:05	FIDL containing all FRANCA basic data types