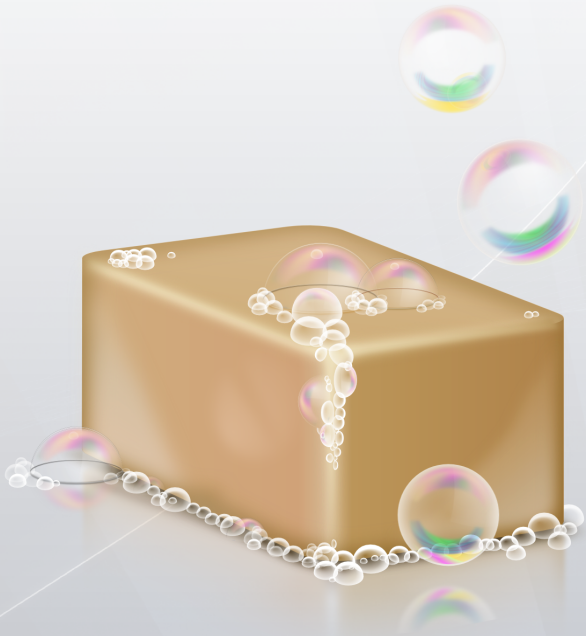

Programmers Manual

KD SOAP

KDAB

www.kdab.com

Table of Contents

Chapter 1. Introduction

Welcome to the KD SOAP Programmer's Manual. KD SOAP is Klarälvdalens Datakonsult AB's web service access package for Qt applications. This manual will get you started creating your own web service clients or web service providers. It covers the fundamentals of coding with KD SOAP and provides plenty of tips for advanced programmers.

- Depending on your version of KD SOAP, you will find a unique `INSTALL.txt` file containing instructions on how to install KD SOAP on your platform. Each instruction set also includes a step-by-step description of how to build KD SOAP directly from the source code.
- KD SOAP also comes with an extensive "API Reference" Manual (generated from the source code itself). It is available both as a PDF file and as browsable HTML pages.

The "API Reference" is an excellent resource for topics not covered in the Programmer's Manual. Both the Programmer's Manual and the API Reference are designed to be used in conjunction with each other. If you have a question not covered in the following chapters, check the API reference for a solution (or Appendix A, *Q&A Section* at the end of this manual).

- What is KD SOAP?

  KD SOAP is a tool for creating client applications for web services which export their service API as SOAP objects. As of KD SOAP 1.1, it can also be used to implement such services. A web service is a program that provides a machine-accessible interface to its functionality via HTTP. One way of handling this kind of remote method calls is the SOAP standard, which describes method calls, their parameters and their return values as XML documents.

  The library provides an abstraction layer for both the actual transport as well as the construction of data objects and method calls. The latter relieves application developers from manually writing XML marshalling and demarshalling code, allowing them to build arbitrary complex data structures using simple C++ classes. The transport abstraction for the client use case provides both synchronous as well as Qt signal/slot based asynchronous remote method call and response handling. The server use case supports both single-threaded as well as multithreaded operations.

  The kdwsdl2cpp code generator delivered alongside the library provides additional means for increasing developer productivity by either generating proxy API for either a target web service (client use case) or a server stub (server use case), based on a formal WSDL service description. Using classes generated by this approach adds build time type checking and in-process like object behavior, i.e. using C++ data types as parameters and return values of each of the web service's methods.

- What are the uses of KD SOAP?

  KD SOAP is used by a variety of programs for a variety of different purposes. For example, one application uses KD SOAP to access data from a CRM system such as SugarCRM or Salesforce. For other examples, visit our web site at `http://www.kdab.com/kd-soap/`

Before Using KD SOAP

You should be familiar with writing Qt applications and have a working knowledge of C++ before using KD SOAP. When you are in doubt about how a Qt class mentioned in this Programmer's Manual works, please check the Qt reference documentation or a good book about Qt. Also, to browse KD SOAP API Reference, start with the file `doc/html/index.html` or `http://docs.kdab.com/kdsoap/`.

The Structure of This Manual

Where do we start?

The first part of this manual begins with an introduction to the KD SOAP 1 client API, then goes through the basic steps and methods for the user to create their own SOAP client. Each subsequent chapter covers more advanced material like different variations of asynchronous calls or providing additional data to the transport protocol layer.

The second part that begins with Chapter 5, *KD SOAP 1.1 Server API Introduction*, demonstrates in a similar fashion how the server API introduced in KD SOAP 1.1 can be used to implement a SOAP web service.

Chapter 6, *Using WSDL to Generate Client or Server API*, finally describes a less generic, but easier and safer-to-use generated code approach.

We provide you with many more example programs than are shown in this manual. We recommend that you try them out and run them. Have a look at the code and experiment with the various settings, both by adjusting them via the user interface and by trying out your own code modifications.

What's next?

In the next chapter we introduce the KD SOAP 1 API.

Chapter 2. KD SOAP 1 Client API Introduction

KD SOAP builds on technologies introduced with Qt 4, most importantly `QNetworkAccessManager`. This ensures that HTTP requests done by KD SOAP will support

- following the operating system's proxy settings,
- processing of cookies,
- and doing HTTP authentication using the following methods: Basic, NTML version 1 (version 2 is not supported by Qt yet) and Digest-MD5.

Overview

Code Sample

```
const QString endPoint =
  QLatin1String("http://www.27seconds.com/Holidays/US/Dates/USHolidayDates.asmx");
const QString messageNamespace =
  QLatin1String("http://www.27seconds.com/Holidays/US/Dates/");

KDSoapClientInterface client(endPoint, messageNamespace);

KDSoapMessage message;
message.addArgument(QLatin1String("year"), 2010);

KDSoapMessage response = client.call(QLatin1String("GetValentinesDay"), message);
```

The code example shows the three basic steps for calling a web service method.

- Setup of the proxy object used for interfacing with the web service.
- Construction of the method call's parameters.
- Performing the remote call.

The response, i.e. either the call's return value or an error structure, is delivered in the same abstract form in which the call parameters were constructed. Therefore, simple use cases like this sample only require use of two classes: `KDSoapClientInterface` and `KDSoapMessage`.

Concepts

For now, to get an overview about the KD SOAP 1 API and its features, you need to understand the following basic concepts:

- All interactions with web services happen through instances of `KDSoapClientInterface`. One such instance is needed for each web service with which an application wants to interact, i.e. it is not possible to switch an instance to a different service at runtime.

  `KDSoapClientInterface` provides functionality to make synchronous and asynchronous method calls. The first can be convenient in non-interactive applications or when used in threads, the latter is also suitable for use in a GUI application's main thread.

- Web service method calls require the name of the method and a single parameter object of type `KDSoapMessage`. Optional data such as SOAP request headers, e.g. for passing a session identifier alongside the request, can be passed with each method call or set permanently on the client interface instance.

  The single parameter object can contain an arbitrary number of named parameter values, each potentially again being a list. Simple types such as numbers or strings are representable by a single parameter value; complex types, e.g. a structure with multiple members, can be represented as a list of said members.

```
KDSoapMessage message;

// add a simple type argument
message.addArgument(QLatin1String("text"), "some text");

// add a structured type argument
QRect rect(0, 0, 100, 200);
KDSoapValueList rectArgument;
rectArgument.addArgument(QLatin1String("x"), rect.x());
rectArgument.addArgument(QLatin1String("y"), rect.y());
rectArgument.addArgument(QLatin1String("width"), rect.width());
rectArgument.addArgument(QLatin1String("height"), rect.height());

message.addArgument(QLatin1String("rect"), rectArgument);
```

Note

When manually constructing message call arguments, it is the responsibility of the developer to provide KD SOAP with marshalling hints, if necessary. These hints are either defined as enums of the respective KD SOAP class or through explicit type specification.

- `KDSoapClientInterface::SoapVersion`
- `KDSoapMessage::Use`
- `KDSoapValue::setType()`
- `KDSoapValueList::setArrayType()`

See Chapter 6, *Using WSDL to Generate Client or Server API*, for an

approach on how to automate that.

- The call result is also of type `KDSoapMessage`, allowing structured data to be returned, similar to how a C++ method can return a class type.

  SOAP calls can result in an error being returned rather than a method return value, e.g. the web service might not be reachable due to network problems, the method might be unknown to the web service, etc. In such cases the returned `KDSoapMessage` is marked as a fault message, see `KDSoapMessage::isFault()`.

Chapter 3. Synchronous vs. Asynchronous Calls

As shown in the code sample, calling a web service method is mapped by KD SOAP to calling a method on a `KDSoapClientInterface` instance. This kind of abstraction makes day-to-day work with web services convenient; however, it is important to consider the involved processing steps when deciding where and how the application makes use of this simplification layer.

The involved steps in the client to web service direction are:

- Creation of an XML document for describing the method to call and its parameters.
- Sending of the XML document via a network connection using HTTP.
- Processing of the XML document on the web service's host.

The same applies for the reverse direction with the only difference being the semantics of the document content (method return value or error, instead of method call).

While the first and last step require some time for XML processing, they are (depending on complexity of the involved parameter or return value data types) negligible compared to the middle one, due to it being subject to network latency and bandwidth limitations which, in turn, are often unpredictable.

Synchronous Calls

Calls performed by invoking the `KDSoapClientInterface`'s `call()` method are synchronous calls. Synchronous means that the calling thread will not return from the method before it has completed the full roundtrip processing. Due to its similarity with an in-process method call it is the easier to use this variant supported by KD SOAP. However, it should only be used when its blocking nature is of little concern, e.g. in a worker thread or in a non-interactive application like a command line tool.

Asynchronous Calls

Calls performed by invoking the `KDSoapClientInterface::asyncCall()` method are asynchronous calls. Asynchronous means that the calling thread will only process step one (XML generation) and create an HTTP transfer request. It will not wait for the transfer's execution or any other response, but instead will return an instance of `KDSoapPendingCall`. This object then serves as a handle to determine, at any later point, whether the whole processing chain has been completed and the resulting return value.

Performing the call from code sample in an asynchronous way:

```
...
// mPendingCall is a member variable so it lives beyond
// the scope of the method creating it and so is mClient
mPendingCall = mClient->asyncCall(QLatin1String("GetValentinesDay"), message);

...

if (mPendingCall.isFinished()) {
  KDSoapMessage response = mPendingCall.returnMessage();
}
```

Of course, checking for completion at some arbitrary time might not be very useful. Instead the application will most likely want to be notified about the call's completion. This is supported by `KDSoapPendingCallWatcher`.

```
...

KDSoapPendingCall pendingCall =
  mClient->asyncCall(QLatin1String("GetValentinesDay"), message);

// create a watcher object that will signal the call's completion
KDSoapPendingCallWatcher* watcher =
  new KDSoapPendingCallWatcher(pendingCall, this);
connect(watcher, SIGNAL(finished(KDSoapPendingCallWatcher*)),
        this, SLOT(pendingCallFinished(KDSoapPendingCallWatcher*)));
...

void MyClass::pendingCallFinished(KDSoapPendingCallWatcher* pendingCall)
{
  KDSoapMessage response = pendingCall->returnMessage();
}
```

Both variants support having more than one call pending, i.e. calling another web service method while a previous call is still being processed. However, there might be restrictions on the web service's side, e.g. not allowing more than one call per client at any given time to limit resource usage. In general, it is also undetermined whether the results arrive in the same or a different order than in which the requests were sent. A safer solution for multiple concurrent calls is to use multiple instances of KDSoapClientInterface, which results in multiple independent connections to the web service.

Sometimes, it might not be necessary to know whether a call succeeded or its result. In such cases, use `KDSoapClientInterface::callNoReply()` instead.

Chapter 4. SOAP Headers

SOAP specifies a mechanism to transmit data alongside the actual SOAP document, so-called SOAP headers. The term headers indicates that these values will be part of the HTTP headers section which makes them accessible for components involved in the transfer which do understand HTTP but might not have SOAP capabilities, e.g. the web server or a load balancer in front of it.

KD SOAP supports headers on a per method call basis as well as keeping them as local state across calls.

SOAP Headers per Client Interface

One of the most common use cases for SOAP headers is providing session information for the recipient of the SOAP transmission. This is mainly necessary because HTTP is a stateless interfacing scheme, where each call arrives through a new network connection. In order to associate such independent connections to the same client, the first connection establishes some sort of session tracking, usually in the form of a session identifier string. This identifier then has to be provided with each call.

While a header containing such an identifier could be provided at each method call, it will usually be more convenient to let KD SOAP take care of that by setting the header as a kind of local state on the `KDSoapClientInterface` instance:

```
KDSoapMessage message;
message.addArgument(QLatin1String("user"), userName);
message.addArgument(QLatin1String("password"), password);

KDSoapMessage response = client.call(QLatin1String("Login"), message);
const QString sessionId = response.arguments()[0].value().toString();

KDSoapMessage header;
header.addArgument(QLatin1String("SessionId"), sessionId);

client.setHeader(QLatin1String("SessionHeader"), header);

...

response = client.call(QLatin1String("GetUserDetails"), KDSoapMessage());
```

The example above assumes a fictitious web service which requires a login call to establish a user's authentification and then requires the returned session token to be used for any subsequent call. Setting the identifier as a header allows KD SOAP to take care of sending it alongside any call invoked through the same `KDSoapClientInterface` instance.

Please note that the actual name of the header argument in the case of this example, `"SessionId"`, is part of the web service's interface description, while the name passed to `KDSoapClientInterface::setHeader()` is purely used for identifying the header between KD SOAP and the application. Thus, it can between chosen by the user.

SOAP Headers for Single Calls

Sometimes it might not be possible to use a header across calls, e.g. the value of the header may change with each call or be specific to the method being called. In cases like this, it is necessary to provide the headers when invoking the `KDSoapClientInterface::call()` (or one of its asynchronous variants):

```
KDSoapMessage header;
header.addArgument(QLatin1String("SessionId"), sessionId);

KDSoapHeaders headers;
headers << header;

response = client.call(QLatin1String("GetUserDetails"),
                       KDSoapMessage(), QString(), headers);
```

SOAP Action

As already hinted in the example above, there is another per-call parameter: SOAP Actions. It is a mandatory SOAP header, but allowed to be empty. Its purpose, as suggested by the SOAP specification, is to express the intent of the call. In most cases, this can be left empty and KD SOAP will generate a valid value based on the method name.

As an example of where it is necessary to actually set this, see Google's search API:

```
KDSoapMessage message;

...

const soapAction = QLatin1String("urn:GoogleSearchAction");
KDSoapMessage response =
  client.call(QLatin1String("doGoogleSearch"), message, soapAction);
```

Chapter 5. KD SOAP 1.1 Server API Introduction

Implementations of SOAP web services require two problems to be solved: low-level HTTP connection handling and high-level SOAP call handling. KD SOAP's solution follows a layered approach, with the low-level part already being fully implemented and the high-level part being delegated to a user-provided implementation of an interface.

Connection Handling

The KD SOAP server component contains all necessary facilities to handle SOAP calls, including classes for receiving and processing HTTP connections via TCP/IP. In other words, a program deploying KD SOAP to provide its SOAP service does not require any additional web server of any sort, i.e. stand-alone SOAP web service capability.

Listening for and accepting of TCP/IP connections is handled by `KDSoapServer`. Its socket level connection handling supports normal, unencrypted connections as well as SSL encrypted connections. Encryption capability is one of the class' features that can be turned on and off through respective flags:

```
// assuming mSoapServer is a member of type KDSoapServer

// enable SSL encryption support
KDSoapServer::Features features = mSoapServer.features();
features |= KDSoapServer::Ssl;
mSoapServer.setFeatures( features );
```

Since `KDSoapServer` is derived from `QTcpServer`, it can be treated as such when it comes to listening for TCP/IP connection, e.g. which interface(s) and port to listen on, whether to wait blockingly for connections or using the current thread's event loop for non-blocking operations.

On top of that `KDSoapServer` supports two strategies for handling the individual client connections:

- Single-Threaded: processing all client connections and respectively their SOAP calls within the same thread that is doing the connection listening.
- Multi-Threaded: processing client connections and respectively their SOAP calls within different worker threads, managed by a thread pool provided by `KDThreadPool`.

By default, `KDSoapServer` will operate in single-threaded manner. Enabling multi-threading is, however, very simple:

```
// assuming mSoapServer is a member of type KDSoapServer

mSoapServer.setThreadPool( new KDThreadPool( this ) );
```

A thread pool instance can even be shared between server instances:

```
// assuming mSoapServer1 and mSoapServer2 are members of type KDSoapServer

KDThreadPool *threadPool = new KDThreadPool( this );

mSoapServer1.setThreadPool( threadPool );
mSoapServer2.setThreadPool( threadPool );
```

SOAP Call Handling

After `KDSoapServer` has determined that an incoming connection is indeed a SOAP call it is responsible for, it needs to delegate the actual SOAP method call to code provided by the web service's developer. It does that by calling a factory method and checking if the created object has the interface it expects to have so it can the call that interface's methods to hand over the call's data to the application specific implementation.

These delegates need to be of type `QObject` and implement the `KDSoapObjectInterface` interface. Taking the example, if the holiday service used throughout this manual, it could look like this:

```
class HolidayService : public QObject, public KDSoapServerObjectInterface
{
    Q_OBJECT
    Q_INTERFACE(KDSoapServerObjectInterface)

public:
    void processRequest( const KDSoapMessage &request,
                         KDSoapMessage &response,
                         const QByteArray &soapAction );
};
```

As mentioned earlier, `KDSoapServer` needs to be able to create instances of this class through a call to its factory method `createServerObject()`:

```
class HolidayServer : public KDSoapServer
{
    Q_OBJECT

public:
    QObject* createServerObject()
    {
        return new HolidayService();
    }
};
```

Note

When operating `KDSoapServer` in a multi-threaded fashion this method will be called by multiple threads.

The `processRequest()` method of the actual call handler class is called for each successfully decodable SOAP request. The call's data is passed in through the *`request`* parameter with additional information being available through the *`soapAction`* parameter and through `KDSoapObjectInterface::requestHeaders()`.

The implementor is expected to fill the *`response`* object with the call's result value(s). Additional response headers can be set through `KDSoapObjectInterface::setResponseHeaders()`.

Note

In the event of an error that needs to be reported back to the caller, e.g. wrong number, types or values of call arguments, use `KDSoapObjectInterface::setFault()` instead.

Example implementation for the HolidayService's `GetValentinesDay` method:

```
void HolidayService::processRequest( const KDSoapMessage &request,
                                     KDSoapMessage &response,
                                     const QByteArray &soapAction )
{
    const QString method = request.name();
    if ( method == QLatin1String( "GetValentinesDay" ||
         soapAction == "http://www.27seconds.com/Holidays/US/Dates/GetValentinesDay" )
    {
        // we expect the year parameter in "request"
        const KDSoapValueList args = request.childValues();

        // add check for args.count() == 1 otherwise fault

        const KDSoapValue value = args.at( 0 );

        // add check if value.name() == "year" otherwise fault

        const int year = value.value().value<int>();

        // create Valentine's day date for given year
        const QDate date( year, 2, 14 );
        const QDateTime dateTime( date );
        const KDDateTime result( dateTime );

        // create response content
        KDSoapValue resultValue( QLatin1String( "GetValentinesDayResponse" ),
                                 QVariant()
                               );
        KDSoapValueList & resultArgs = resultValue.childValues();
        resultArgs.append( KDSoapValue( QLatin1String( "GetValentinesDayResult" ),
                                        result.toDateString(),
                                        KDSoapNamespaceManager::xmlSchema2001(),
                                        QLatin1String("dateTime")
                                      )
                         );

        // fill return object
        response = resultValue;
    }
    else // unhandled method, let base class deal with that
        KDSoapServerObjectInterface::processRequest( request, response, soapAction );
}
```

Chapter 6. Using WSDL to Generate Client or Server API

Introduction

Most SOAP based web services are formally described in an XML-based markup language called web service description language (WSDL). Such WSDL documents describe web service methods, their parameters and return values, as well as possible errors, in a formalized way that makes them viable as input for code generation tools.

Compared to the generic approach described in the earlier chapters of this manual, using a generated client or server API introduces several additional advantages:

- Named methods: instead of referring to web service methods by names as string parameters to a generic `call()` method, a client interface generated from a WSDL document will have C++ methods for each of the web service's methods. Instead of handling all method calls in a generic `processRequest()` method, a generated server stub will have pure virtual C++ methods for each of the web service's methods, which can then just be implemented by overriding them in a class derived from the generated stub.

- Type-specific classes: instead of building a single message object containing the method's parameters as lists of named values, methods of a generated client or server interface take C++ types for each of their parameters. If such a parameter is a type described in the WSDL document, a matching class with suitably-named methods and again specific types will have been generated as well.

- Build time checks: As methods and parameter and return types are no longer generic, typos in method or parameter names, missing parameters and some forms of invalid parameter contents can now be caught by the C++ compiler, instead of resulting in a SOAP error at runtime during method invocation.

KD SOAP has full support for this powerful approach through its kdwsdl2cpp code generator code generation tool. The following sections document how to use it and the client or server API it generates.

Using kdwsdl2cpp code generator

Before using kdwsdl2cpp code generator, it is necessary to obtain the WSDL document describing the target web service and have it available as a local file. It is usually available as a download on a website associated with the web service or via similar online distribution methods.

Code generation using kdwsdl2cpp code generator requires two steps: one to generate

the header file and one to generate the source file. For example, when processing the `holidays.wsdl` file (which can be found in `examples/holidays_wsdl`), the following two commands need to be executed for the client-side use case:

```
kdwsdl2cpp -o wsdl_holidays.h holidays.wsdl
```

This creates a header file `wsdl_holidays.h` (choice of file name is up to the user), which will then contain the declarations of all data classes described in the WSDL file, as well as the class representing the web service's interface.

```
kdwsdl2cpp -o wsdl_holidays.cpp -impl wsdl_holidays.h holidays.wsdl
```

This creates a source file `wsdl_holidays.cpp` (choice of file name is again up to the user) using the header file `wsdl_holidays.h` as the include for class declarations.

The server-side use case is almost equivalent. The only difference is that an additional *`-server`* option has to be passed to kdwsdl2cpp code generator at both invocations:

```
kdwsdl2cpp -server -o wsdl_holidays.h holidays.wsdl
```

```
kdwsdl2cpp -server -o wsdl_holidays.cpp -impl wsdl_holidays.h holidays.wsdl
```

Using the Generated Client Code

The generated client-side code for the example call used throughout this manual looks like this:

```
class USHolidayDates : public QObject
{
    Q_OBJECT

public:
    USHolidayDates( QObject* parent = 0 );
    ~USHolidayDates();

    TNS__GetValentinesDayResponse
      getValentinesDay( const TNS__GetValentinesDay& parameters );

    void asyncGetValentinesDay( const TNS__GetValentinesDay& parameters );

    void getValentinesDayDone( const TNS__GetValentinesDayResponse& parameters );

    void getValentinesDayError( const KDSoapMessage& fault );
};
```

This is an excerpt of the generated web service interface class. It is a subclass of `QObject`, so it can make use of signals for delivering asynchronous results. Both synchronous as well as asynchronous calls take an instance of the generated class `TNS__GetValentinesDay` as their only parameter. The synchronous variant returns an instance of class `TNS__GetValentinesDayResponse` once it has completed the call, while the asynchronous `asyncGetValentinesDay()` returns without value. Its results

are delivered through one of the two signals: `getValentinesDayDone()`, which carries an instance of the same response class returned by the synchronous `getValentinesDay()`, or `getValentinesDayError()` which carries a fault message (see Section , “Concepts”).

```
class TNS__GetValentinesDay
{
public:
    void setYear( int year );
    int year() const;
};
```

This is an excerpt of the generated request parameter class, using the native integer type for the year parameter.

```
class TNS__GetValentinesDayResponse
{
public:
    void setGetValentinesDayResult( const QDateTime& getValentinesDayResult );
    QDateTime getValentinesDayResult() const;
};
```

This is an excerpt of the generated return value class, using Qt's `QDateTime` to represent the resulting date.

Deploying these classes to perform the example call:

```
USHolidayDates client;

TNS__GetValentinesDay request;
resquest.setYear( 2010 );

TNS__GetValentinesDayResponse response = client.getValentinesDay( request );
const QDateTime valentinesDay = response.getValentinesDayResult();
```

Using the Generated Server Code

The generated server-side code for the example call looks like this:

```
class USHolidayDatesServerBase : public QObject, public KDSoapServerObjectInterface
{
    Q_OBJECT

    Q_INTERFACES(KDSoapServerObjectInterface)

public:
    virtual TNS__GetValentinesDayResponse
        getValentinesDay( const TNS__GetValentinesDay& parameters ) = 0;
};
```

This is an excerpt of the generated web service stub class. It is a subclass of `QObject`, so it can make use of Qt's mechanism for handling C++ interfaces. It is also a subclass

of `KDSoapServerObjectInterface`, which provides the interface between the connection handling code and the SOAP method processing code. The interface's `processRequest()` method has been provided by the code generator, thus leaving only the SOAP callable methods to implement:

```
class USHolidayDatesServer : public USHolidayDatesServerBase
{
    Q_OBJECT

public:
    virtual TNS__GetValentinesDayResponse
        getValentinesDay( const TNS__GetValentinesDay& parameters )
    {
        const int year = parameters.year();

        // Valentine's day is trivial since it is always February 14th of the year
        const QDate date( year, 2, 14 );
        const QDateTime dateTime( date );
        const KDDateTime resultValue( dateTime );

        // create and fill response object
        TNS__GetValentinesDayResponse response;
        response.setGetValentinesDayResult( resultValue );

        return response;
    }
};
```

Additional to deriving from the generated class and implementing all pure virtuals, we need an instance of the connection handling class based on `KDSoapServer`, as described in the chapter on server implementation without code generation. Its `createServerObject()` method can simply return new instances of the class derived from the generated stub:

```
class HolidaysServer : public KDSoapServer
{
    Q_OBJECT

public:
    virtual QObject* createServerObject()
    {
        return new USHolidayDatesServer;
    }
};
```

Using kdwsdl2cpp code generator with QMake

While running kdwsdl2cpp code generator manually and adding the resulting source and header file to a project's QMake `.pro` file works as expected, having QMake take care of the code generation makes it easier to adopt changes in the WSDL file.

KD SOAP supports this by making WSDL file processing similar to how QMake treats Qt Designer's `.ui` files. The necessary steps are:

- Include the `kdsoap.pri` QMake include file in the project's `.pro` file. It is part of the KD SOAP package and should be copied to the user's project directory.

- Add the WSDL files to the `KDWSDL` variable. For the example above, it would look like this:

```
KDWSDL = holidays.wsdl
```

  By default this will generate client proxy classes. Adding server stub classes can be achieved by specifying the necessary option like this:

```
KDWSDL_OPTIONS = -server
```

  Generated files will be prefixed with `wsdl_` and have the proper extension, depending on file type. I.e. the above line will result in `wsdl_holidays.h` and `wsdl_holidays.cpp`.

- Set the environment variable `KDSOAPDIR` to the base path of your KD SOAP build or install directory or provide it as a value to the `qmake` run.

Appendix A. Q&A Section

Building and installing KD SOAP

Q:.
How can I build and install KD SOAP from source?

A:.
The procedure to follow for building and installing KD SOAP is described in file `Install.src`. Please refer to that file for details.

Contacting KD SOAP Support

Q:.
How can I get help (or report issues, resp.) on KD SOAP?

A:.
To report issues/problems or ask for help on KD SOAP, please send your mail with a description of your problem/question/wishes to the support address kdsoap-support@kdab.com. Please include a description of your setup: CPU type, operating system with release number, compiler (version) used, any changes you made on libraries that are linked, etc. In other words, include every detail that might help us set up a comparable test environment in our labs.

In most cases, it will make sense to include a small sample program showing the problem you are describing. We will then reproduce the issue on our machines and either fix your sample code or adjust our own code (in case your reported issue might turn out to result from sub-optimal implementation in KD SOAP).

Note

Providing us with a compilable sample program file will help us find a good solution to the reported problem, as we will be using the same code that you have been trying to use yourself.

Often the easiest way to create such a sample program could be to look at one of our example programs, e.g. `examples/holidays_wsdl/holidays.cpp` and do something similar with your WSDL file.

If the web service is not available to us, e.g. if the software is not available as at least a trial version and you (or your customer) are not allowed to create a test account for us on your installation, we'll probably require logs of the SOAP communication as well.

KD SOAP has built-in functionality to print the SOAP communication to the application's standard output. To enable this, either set the environment variable `KDSOAP_DEBUG` to `1` or put the following code into the test application:

```
qputenv( "KDSOAP_DEBUG", "1" );
```