Wolframe documentation

Table of Contents

Developing Wolframe Server Applications

Wolframe Application Development Manual

Developing Wolframe Server Applications: Wolframe Application Development Manual

Publication date April 27, 2014 version 0.0.4
Copyright © 2010 - 2014 Project Wolframe

Table of Contents

List of Figures

List of Tables

Foreword

The Wolframe project was started in 2010. The goal was to create a platform for fully customizable business applications that can be hosted in modern system environments.

This manual introduces the architecture of Wolframe and explains how to build client/server applications with it. After reading this you should be able to create an application on your own.

Chapter 1. Introduction

First we describe the overall architecture and the data flow in a Wolframe application.

1.1. Architecture

Wolframe is a 3-tier application server.

Figure 1.1. Overview
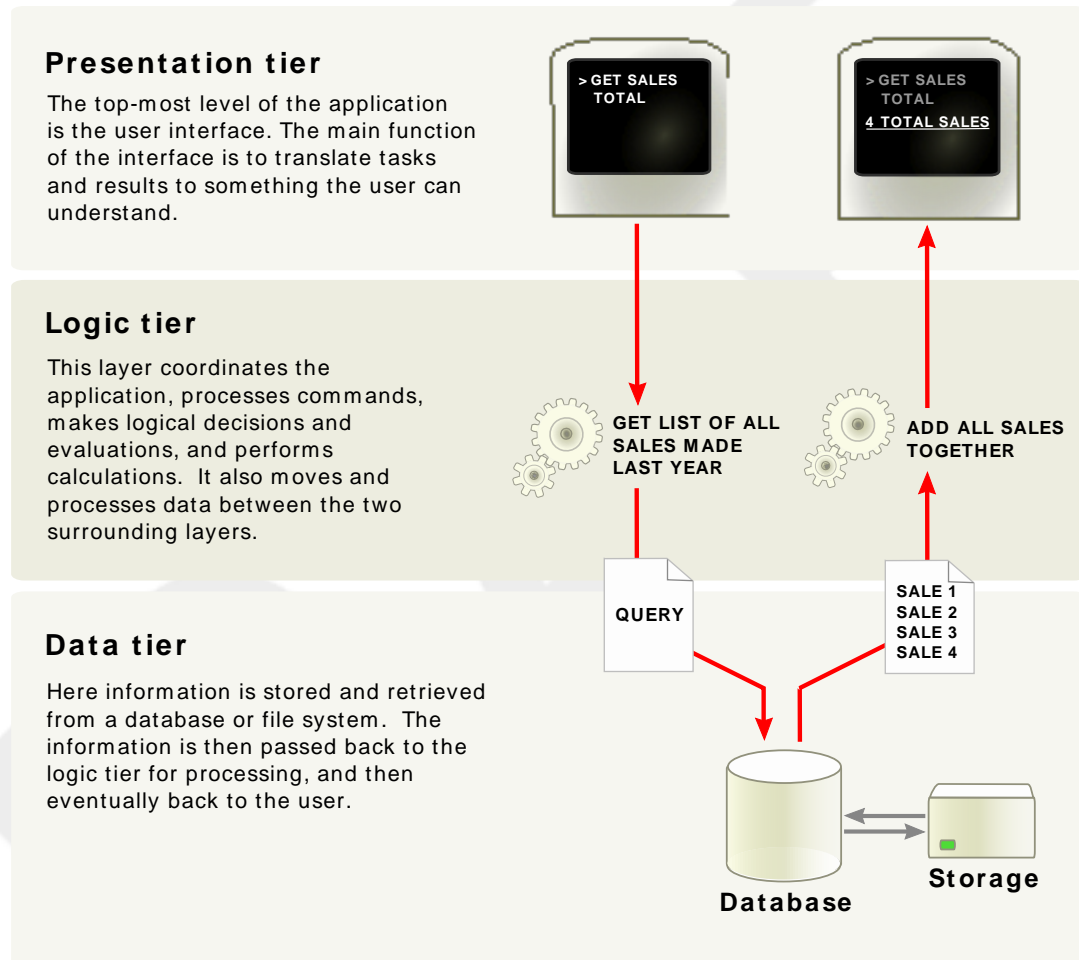
1.1.1. Presentation tier

The presentation tier of Wolframe is implemented as a thin client. It maps the presentation of the application from the request answers it gets from the server. Also the data describing this mapping is loaded from the server when connecting to it. So the whole application is driven by the server. Special use cases are designers of user interfaces that upload the presentation data for other users to the server.

1.1.2. Logic tier

The logic tier of Wolframe describes the transformation of input of the presentation tier to a set of instructions for the data tier. The input to the logic tier consists of a command name plus a structured

content also referred to as document. The logic tier returns a single document to the presentation tier. The logic tier supports scripting languages to define the input/output mapping between the layers. Wolframe introduces three concepts as building blocks of the logic tier:

- *Filters*: Filters are transforming serialized input data (XML,JSON,CSV,etc.) to a unified serialization of hierarchically structured data and to serialize any form of processed data for output. Filters are implemented as loadable modules (e.g. XML filter based on libxml2, JSON filter based on cJSON) or as scripts based on a filter module (XSLT filter script for rewriting input or output)

- *Forms*: Forms are data structures defined in a data definition language (DDL). Forms are used to validate and normalize input (XML validation, token normalization, structure definition). The recommended definition of a command in the logic tier has a form to validate its input and a form to validate its output before returning it to the caller.

- *Functions*: Functions delegate processing to the data tier (transactions) or they are simple data transformations or they serve as interface to integrate with other environments (e.g. .NET). Functions have a unique name and are called with a structure as argument and a structure as result. Functions can call other functions for delegation, e.g. a transaction definition can call a .NET function for preprocessing its input or a .NET function can call a Python function to do parts of the processing.

You find a detailed description of the Logic tier and how to use it in this book.

1.1.3. Data tier

The data tier of Wolframe defines the functions for calling a transaction. The main transaction function gets a complete description of the transaction to execute as input and returns all results of the transaction as output. The logic tier builds the result data structure out of this result. The main transaction function is stateless and an abstraction of the transactional context. (The transaction context does not exist outside this function. Explained differently: Two functions do not refer to the same transaction).

1.2. Application Server Requests

Wolframe application server requests consist of a named command and a structured content (document) as argument.

The following illustration shows the processing of one client request to the server. A call of the Wolframe logic tier gets to a command handler that calls functions given by the provider to perform the transaction requested.

Figure 1.2. Overview
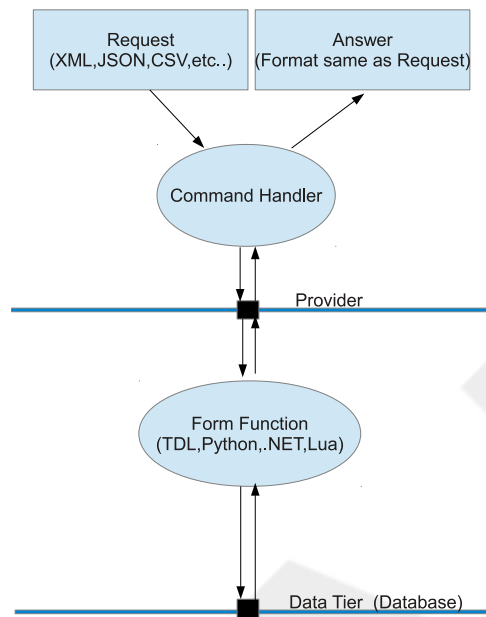
In the following chapter will describe now the Wolframe standard command handler and how it is configured. Then we will show how to write programs that declare the functions executing the requests and how you link them to your application.

Chapter 2. Command Handler

This chapter introduces the standard command handler of the logic tier.

2.1. The Standard Command Handler

2.1.1. Introduction

The Wolframe standard command handler is called *directmap* and named so in the configuration because it only declares a redirection of the commands to functions based on the document type and the command identifier specified by the client in the request.

The declarations of the Wolframe Standard Command Handler (*directmap*) are specified in a program source file with the extension '.dmap' that is declared in the configuration.

2.1.2. Example Configuration

The following configuration declares a program `example.tdl` written in the transaction definition language (TDL) to contain the function declarations for the provider that can be called by the command handler. It declares the database with name pgdb to be used as the database for transactions. It loads a description `example.dmap` that will declare the mappings of commands to the filters used and functions called. It specifies the filter with name libxml2 to be used for documents of format XML and the filter with name cjson to be used for documents of format JSON, if not specified else in `example.dmap`.

```
  ; Simple Data Processing Configuration Example
Processor
{
    ; Programs to load:
    program example.tdl              ; a program with functions (in TDL)
    database pgdb                    ; references transaction database

    ; Command handlers to load:
    cmdhandler
    {
        directmap                    ; the standard command handler
        {
            program example.dmap     ; description of command mappings

            filter XML=libxml2       ; std filter for XML document format
            filter JSON=cjson        ; std filter for JSON document format
        }
    }
}
```

2.1.3. Example Command Description

The following source example could be one of the `example.dmap` in the configuration example introduced above. It defines two commands. The first one links a command "insert" with document type "Customer" as content to a transaction function "doInsertCustomer". The content is validated

automatically against a form named "Customer" if not explicitly defined else. The command has no result except that it succeeds or fails. The second example command links a command "get" with a document type "Employee" to a function "doSelectEmployee". The input is not validated and the transaction output is validated and mapped through the form "Employee".

```
COMMAND insert Customer CALL doInsertCustomer;
COMMAND get Employee SKIP CALL doSelectEmployee( xml ) RETURN Employee;
```

2.1.4. Command Description Language

A command map description file like our example shown consists of instructions started with `COMMAND` and terminated by semicolon ';'. The first argument after `COMMAND` is the name of the command followed by the name of the document type of the input document. The name of the command is optional. If not specified the first argument after `COMMAND` names the input document type.

Keywords

Conflicts with keywords and names are solved by using strings instead of identifiers for names defined or alternatively by enclosing names of list of names in '(' ')' brackets. The command description language has the following keywords:

COMMAND
CALL
RETURN
SKIP
STANDALONE
FILTER
INPUT
OUTPUT

Simple Document Map

The following example shows the simplest possible declaration. It states that documents with the document type "Document" are forwarded to a function with the same name "Document".

```
COMMAND Document;
```

Command with Action Prefix

The next example adds a action name to the declaration. The implicit name of the function called is insertDocument:

```
COMMAND insert Document;
```

Explicit Function Name Declaration

For declaring the function called explicitly like for example a function `doInsertDocument` we need to declare it with `CALL <functionname>`:

```
COMMAND insert Document CALL doInsertDocument;
```

Returned Document Declaration

The document type returned is specified with `RETURN <doctype>`:

```
COMMAND process Document RETURN Document;
```

or with explicit naming of a function called

```
COMMAND process Document CALL doProcessDocument RETURN Document;
```

Skipping the Document Validation

If you want to skip the input document validation, either because you are dealing with legacy software where a strict definition of a schema is not possible or because the function called has strict typing and validates the input on its own (.NET,C++), then you can add a declaration `SKIP`:

```
COMMAND process Document SKIP CALL doProcessDocument RETURN Document;
```

The same you can specify for the output with a `SKIP` following the `RETURN` of the output declaration:

```
COMMAND process Document CALL doProcessDocument RETURN SKIP Document;
```

A second optional parameter of the `RETURN SKIP` declaration specifies the root element of the returned data object. In this case the form definition does not have to exist in a DDL form definition. The output can be built without a form definition defined in a DDL. The following example shows such a definition with 'list' as root element defined. Such a command definition makes sense for strongly typed languages like .NET or native C++ where data validation can be delagated completely to the strongly typed structure definition of the called function.

```
COMMAND process Document CALL doProcessDocument RETURN SKIP Document list;
```

Return a Standalone Document

If we want to return a document as standalone (standalone="yes" in the header in case of XML) without validation (validation depends on the document type) the we have to declare this with RETURN STANDALONE instead of RETURN SKIP. The argument of RETURN STANDALONE is the document root element used in document format that need a root element to be declared (XML and JSON). The following example shows such a declaration:

```
COMMAND process Document CALL doProcessDocument RETURN STANDALONE list;
```

Explicit Filter Definitions for a Command

For most processing it's enough to declare the standard filters in the configuration of the command handler. But in certain cases we want to declare a filter explicitly for a command, for example to preprocess a certain document type with an XSLT filter. Explicitly declared filters always refer to a document format and documents of other formats have to be converted first or they cannot be preprocessed. The conversions mechanisms we will explain in detail later. Explicit filter declarations are done with

- FILTER <name> or

- FILTER INPUT <inputfiltername> or

- FILTER OUTPUT <outputfiltername> or

- FILTER INPUT <inputfiltername> OUTPUT <outputfiltername>

Here is an example:

```
COMMAND process Document FILTER INPUT myXsltInputFilter
    CALL doProcessDocument RETURN Document;
```

Using Brackets

For better readability you can use optional '(' ')' brackets on the arguments. This way you can distinguish better between keywords and arguments and also avoid conflicts with keywords:

```
COMMAND ( process Document )
    FILTER INPUT ( myXsltInputFilter ) CALL ( doProcessDocument )
    RETURN ( Document );
```

Overview

Each command declaration has as already explained the form

- `COMMAND <doctype> [OPTIONS] ;` or

- `COMMAND <action> <doctype> [OPTIONS] ;`

The following table shows an overview of the elements that can be used in the `[OPTIONS]` part of the command:

Table 2.1. Options

Keywords	Arguments	Description
CALL	Function Name	Names the function to be called for processing the request
RETURN	Document Type	Specifies the type of the document returned and forces validation of the output
RETURN SKIP	Document Type	Specifies the type of the document returned but skips validation of the output
SKIP	(no arguments)	Specifies the input document validation to be skipped
FILTER INPUT	Filter Name	Specifies that the filter <Name> should be used as input filter
FILTER OUTPUT	Filter Name	Specifies that the filter <Name> should be used as output filter
FILTER	Filter Name	Specifies that the filter <Name> should be used both as input and output filter

Chapter 3. Functions

This chapter describes how functions are linked to the logic tier. It gives an overview on the language bindings available for Wolframe.

For defining database transactions Wolframe introduces a language called TDL (Transaction Definition Language). TDL embeddes the language of the underlaying database (SQL) in a language that defines how sets of elements of input and output are addressed.

This chapter also describes how data types are defined that can be used in data definion languages (DDL) for form desciptions. Forms and their definition will be introduced in a different chapter.

After reading this chapter you should be able to write functions of the Wolframe logic tier on your own.

Be aware that you have to configure a programming language of the logic tier in Wolframe before using it. Each chapter introducing a programming language will have a section that describes how the server configuration of Wolframe has to be extended for its availability.

3.1. Transactions in TDL

3.1.1. Introduction

For the description of transactions Wolframe provides the transaction definition language (TDL) introduced here. Wolframe transactions in TDL are defined as functions in a transactional context. This means that whatever is executed in a transaction function belongs by default to a database transaction with an automatic commit on function completion if not explicitly defined otherwise by the caller. Errors lead to an automatic abort of the database transaction.

A TDL transaction function takes a structure as input and returns a structure as output. The Wolframe database interface defines a transaction as object where the input is passed to as a structure and the output is fetched from it as a structure.

TDL is a language to describe the building of transaction input and the building of the result structure from the database output. It defines a transaction as a sequence of instructions on multiple data. An instruction is either described as a single embedded database command in the language of the underlying database, a name of a function declared in the database (e.g. a PLSQL function) or a TDL subroutine call working on multiple data.

Working on multiple data means that the instruction is executed for every item of an input set. This set can consist of the set of results of a previous instruction or a selection of the input of the transaction function. A "for each" selector defines the input set as part of the command.

Each instruction result can be declared as being part of the transaction result structure. The language has no control structures and is therefore not a general purpose programming language. It just offers some mapping of the input to commands and from the command results back to the output.

To convert input data the transaction definition language defines a preprocessing section where globally defined Wolframe functions can be called for the selected input. To build an output structure that cannot be modeled with a language without control structures and recursion, TDL provides the possibility to define a function as filter for postprocessing of the result of the transaction function.

The TDL is case insensitive. For clearness and better readability TDL keywords are written in uppercase here.

3.1.2. Configuration

Each TDL program source referenced has to be declared in the `Processor` section of the configuration with `program <sourcefile>`.

3.1.3. Language Description

A TDL program consists of subroutine declarations and exported transaction function declarations. Subroutines have the same structure as transaction function blocks but without pre- and postprocessing and authorization method declarations.

Subroutines

A subroutine declaration starts with the Keyword `SUBROUTINE` followed by the subroutine name and optionally some parameter names in brackets ('(' ')') separated by comma. The declared subroutine name identifies the function in the scope of this sourcefile after this subroutine declaration. The name is not exported and the subroutine not available for other TDL modules. The body of the function contains the following parts:

- `DATABASE <database name list>`

  This optional definition is restriction the definition and availability of the function to a set of databases. The databases are listed by name separated by comma (','). The names are the database id's defined in your server configuration. If the database declaration is omitted then the transaction function is avaiable for any database. This declaration allows you to run your application with configurations using different databases but sharing a common code base.

- `BEGIN <...instructions...> END`

  The main processing block starts with `BEGIN` and ends with `END`. It contains all the commands executed when calling this subroutine from another subroutine or a transaction function.

The following pseudocode example shows the parts of a subroutine declaration:

```
SUBROUTINE <name> ( <parameter name list> )
DATABASE <list of database names>
BEGIN
 ...<instructions>...
END
```

The DATABASE declaration is optional.

Transaction Function Declarations

A transaction function declaration starts with the Keyword `TRANSACTION` followed by the name of the transaction function. This name identifies the function globally. The body of the function contains the following parts:

- `AUTHORIZE ( <auth-function>, <auth-resource> )`

  This optional definition is dealing with authorization and access rights.

- `DATABASE <database name list>`

  This optional definition is restriction the definition and availability of the function to a set of databases. The databases are listed by name separated by comma (','). The names are the database id's defined in your server configuration. If the database declaration is omitted then the transaction function is avaiable for any database. This declaration allows you to run your application with configurations using different databases but sharing a common code base.

- `RESULT FILTER <post-filter-name>`

  This optional declaration defines a function applied as post filter to the transaction function. The idea is that you might want to return a structure as result that cannot be built by TDL. For example

a recursive structure like a tree. The result filter function is called with the structure printed by the main processing block (BEGIN .. END) and the result of the filter function is returned to the caller instead.

- `PREPROC <...instructions...> ENDPROC`

This optional block contains instructions on the transaction function input. The result of these preprocessing instructions are put into the input structure, so that they can be referred to in the main code definition block of the transaction. We can call any global normalization or form function in the preprocessing block to enrich or transform the input to process.

- `BEGIN <...instructions...> END`

The main processing block starts with `BEGIN` and ends with `END`. It contains all the database instructions needed for completing this transaction.

- `AUDIT [CRITICAL] <funcname...> WITH BEGIN <...instructions...> END`

This optional block specifies a function that is executed at the end of a transaction. The input of the function is the structure built from the output of the instructions block. If CRITICAL is specified then the transaction fails (rollback) if the audit function fails. Otherwise there is just the error of the audit function logged, but the transaction is completed (commit). You can specify several audit functions. The variables in the instructions block refer to the scope of the main processing block. So you can reference everything that is referencable after the last instruction of the main processing block.

- `AUDIT [CRITICAL] <funcname...> ( <...parameter...> )`

If the input structure of the audit function is just one parameter list this alternative syntax for an audit function declaration can be used. You simply specify the audit function call after the AUDIT or optionally after the CRITICAL keyword.

The following pseudo code snippet shows the explained building blocks in transaction functions together:

```
TRANSACTION <name>
AUTHORIZE ( <auth-function>, <auth-resource> )
DATABASE <list of database names>
RESULT INTO <result-block-name> FILTER <post-filter-name>
PREPROC
 ...<preprocessing instructions>...
ENDPROC
BEGIN
 ...<instructions>...
END
AUDIT CRITICAL <funcname> ( ...<parameter>... )
```

The lines with AUTHORIZE,DATABASE,RESULT INTO and FILTER are optional. So is the preprocessing block PREPROC..ENDPROC. A simpler transaction function looks like the following:

```
TRANSACTION <name>
BEGIN
 ...<instructions>...
END
```

Main Processing Instructions

Main processing instructions defined in the main execution block of a subroutine or transaction function consist of three parts in the following order terminated by a semicolon ';' (the order of the INTO and FOREACH expression can be switched):

- `INTO <result substructure name>`

This optional directive specifies if and where the results of the database commands should be put into as part of the function output. In subroutines this substructure is relative to the current substructure addressed in the callers context. For example a subroutine with an "INTO myres" directive called by a subroutine with an "INTO output" directive will write its result into a substructure with path "output/myres".

- `FOREACH <selector>`

This optional directive defines the set of elements on which the instruction is executed one by one. Specifying a set of two elements will cause the function to be called twice. An empty set as selection will cause the instruction to be ignored. Without quantifier the database command or subroutine call of the instruction will be always be executed once.

The argument of the FOREACH expression is either a reference to the result of a previous instruction or a path selecting a set of input elements.

Results of previous instructions are referenced either with the keyword RESULT referring to the result set of the previous command or with a variable naming a result set declared with this name before.

Input elements are selected by path relative to the path currently selected, starting from the input root element when entering a transaction function. The current path selected and the base element of any relative path calculated in this scope changes when a subroutine is called in a FOREACH selection context. For example calling a subroutine in a 'FOREACH person' context will cause relative paths in this subroutine to be sub elements of 'person'.

- `DO <command>`

Commands in an instruction are either embedded database commands, named database functions (e.g. PLSQL functions) or subroutine calls. Command arguments are either constants or relative paths from the selector path in the FOREACH selection or referring to elements in the result of a previous command. If an argument is a relative path from the selector context, its reference has to be unique in the context of the element selected by the selector. If an argument references a previous command result it must either be unique or dependent an the FOREACH argument. Results that are sets with more than one element can only be referenced if they are bound to the FOREACH quantifier.

Main Processing Example

The following example illustrate how the FOREACH,INTO,DO expressions in the main processing block work together:

```
TRANSACTION insertCustomerAddresses
BEGIN
    DO SELECT id FROM Customer
        WHERE name = $(customer/name);
    FOREACH /customer/address
        DO INSERT INTO Address (id,address)
        VALUES ($RESULT.id, $(address));
END
```

Preprocessing Instructions

Preprocessing instructions defined in the PREPROC execution block of a transaction function consist similar to the instructions in the main execution block of three parts in the following order terminated by a semicolon ';' (the order of the INTO and FOREACH expression can be switched and has no meaning, e.g. FOREACH..INTO == INTO..FOREACH):

- `INTO <result substructure name>`

  This optional directive specifies if and where the results of the preprocessing commands should be put into as part of the input to be processed by the main processing instructions. The relative paths of the destination structure are calculated relative to a FOREACH selection element.

- `FOREACH <selector>`

  This optional directive defines the set of elements on which the instruction is executed one by one. The preprocessing command is executed once for each element in the selected set and it will not be executed at all if the selected set is empty.

- `DO <command>`

  Commands in an instruction are function calls to globally defined form functions or normalization functions. Command arguments are constants or relative paths from the selector path in the FOREACH selection. They are uniquely referencing elements in the context of a selected element.

Preprocessing Example

The following example illustrate how the "FOREACH, INTO, DO" expressions in the main processing block work together:

```
TRANSACTION insertPersonTerms
PREPROC
    FOREACH //address/* INTO normalized
        DO normalizeStructureElements(.);
    FOREACH //id INTO normalized
        DO normalizeNumber(.);
ENDPROC
BEGIN
    DO UNIQUE SELECT id FROM Person
        WHERE name = $(person/name);
    FOREACH //normalized DO
        INSERT INTO SearchTerm (id, value)
        VALUES ($RESULT.id, $(.));
END
```

Selector Path

An element of the input or a set of input elements can be selected by a path. A path is a sequence of one of the following elements separated by slashes:

- `Identifier`

  An identifier uniquely selects a sub element of the current position in the tree.

- *

  An asterisk selects any sub element of the current position in the tree.

- ..

  Two dots in a row select the parent element of the current position in the tree.

- .

  One dots selects the current element in the tree. This operator can also be useful as part of a path to force the expression to be interpreted as path if it could also be interpreted as a keyword of the TDL language (for example `./RESULT`).

A slash at the beginning of a path selects the root element of the transaction function input tree. Two subsequent slashes express that the following node is (transitively) any descendant of the current node in the tree.

Paths can appear as argument of a FOREACH selector where they specify the set of elements on which the attached command is executed on. Or they can appear as reference to an argument in a command expression where they specify uniquely one element that is passed as argument to the command when it is executed.

When used in embedded database statements, selector paths are referenced with `$(<path expression>)`. When used as database function or subroutine call arguments path expressions can be used in plain without '$' and '(' ')' markers. These markers are just used to identify substitution entities.

Path Expression Examples

The following list shows different ways of addressing an element by path:

- /

  Root element

- /organization

  Root element with name "organization"

- /organization/address/city

  Element "city" of root "organization" descendant "address"

- .//id

  Any descendant element with name "id" of the current element

- //person/id

  Child with name "id" of any descendant "person" of the root element

- //id

  Any descendant element with name "id" of the root element

- /address/*

  Any direct descendant of the root element "address"

- .

  Currently selected element

Path Usage Example

This example shows the usage of path expression in the preprocessing and the main processing part of a transaction function:

```
TRANSACTION selectPerson
PREPROC
    FOREACH /person/name
        INTO normalized DO normalizeName( . );
    FOREACH /person
        INTO citycode DO getCityCode( city );
ENDPROC
BEGIN
    FOREACH person
        DO INSERT INTO Person (Name,NormalizedName,CityCode)
            VALUES ($(name),$(name/normalized),$(citycode));
END
```

Referencing Database Results

Database results of the previous instruction are referenced with a '$RESULT.' followed by the column identifier or column number. Column numbers start always from 1, independent from the database! So be aware that even if the database counts column from 0 you have to use 1 for the first column.

As already explained before, database result sets of cardinality bigger than one cannot be addressed if not bound to a FOREACH selection. In statements potentially addressing more than one result element you have to add a FOREACH RESULT quantifier.

For addressing results of instructions preceding the previous instruction, you have to name them (see next section). The name of the result can then be used as FOREACH argument to select the elements of a set to be used as base for the command arguments of the instruction. Without binding instruction commands with a FOREACH quantifier the named results of an instruction can be referenced as $<name>.<columnref>, for example as $person.id for the column with name 'id' of the result named as 'person'.

The 'RESULT.' prefix in references to the previous instruction result is a default and can be omitted in instructions that are not explicitly bound to any other result than the last one. So the following two instructions are equivalent:

```
DO SELECT name FROM Company
    WHERE id = $RESULT.id
DO SELECT name FROM Company
    WHERE id = $id
```

and so are the following two instructions:

```
FOREACH RESULT
    DO SELECT name FROM Company
        WHERE id = $RESULT.id
FOREACH RESULT
    DO SELECT name FROM Company
```

```
                        WHERE id = $id
```

The result name prefix of any named result can also be omitted if the instruction is bound to a
FOREACH selector naming the result. So the following two statements in the context of an existing
database result named "ATTRIBUTES" are equivalent:

```
    FOREACH ATTRIBUTES
        DO SELECT name FROM Company
             WHERE id = $ATTRIBUTES.id
    FOREACH ATTRIBUTES
        DO SELECT name FROM Company
             WHERE id = $id
```

Naming Database Results

Database results can be hold and made referenceable by name with the declaration KEEP AS
<resultname> following immediately the instruction with the result to be referenced. The identifier
<resultname> references the result in a variable reference or a FOREACH selector expression.

Named Result Example

This example illustrates how a result is declared by name and referenced:

```
TRANSACTION selectDevices
BEGIN
    DO SELECT id FROM DevIdMap
        WHERE name = $(device/name);
    KEEP AS dev;
    FOREACH dev
        DO SELECT key,name,registration
            FROM Devices WHERE sid=$id;
END
```

Referencing Subroutine Parameters

Subroutine Parameters are addressed like results but with the prefix PARAM. instead of RESULT. or
a named result prefix. "PARAM." is reserved for parameters. The first instruction without FOREACH
quantifier can reference the parameters without prefix by name.

```
SUBROUTINE selectDevice( id)
BEGIN
    INTO device
        DO SELECT name FROM DevIdMap
            WHERE id = $PARAM.id;
END

TRANSACTION selectDevices
BEGIN
    DO selectDevice( id );
END
```

Constraints on Database Results

Database commands returning results can have constraints to catch certain errors that would not be recognized at all or too late otherwise. For example a transaction having a result of a previous command as argument would not be executed if the result of the previous command is empty. Nevertheless the overall transaction would succeed because no database error occurring during execution of the commands defined for the transaction.

Constraints on database results are expressed as keywords following the DO keyword of an instruction in the main processing section. If a constraint on database results is violated the whole transaction fails and a rollback occurrs.

The following list explains the result constraints available:

- NONEMPTY

  Declares that the database result for each element of the input must not be empty.

- UNIQUE

  Declares that the database result for each element of the input must be unique, if it exists. Result sets with more than one element are refused but empty sets are accepted. If you want to declare each result to have to exist, you have to put the double constraint "UNIQUE NONEMPTY" or "NONEMPTY UNIQUE".

Example with Result Constraints

This example illustrates how to add result constraint for database commands returning results:

```
TRANSACTION selectCustomerAddress
BEGIN
    DO NONEMPTY UNIQUE SELECT id FROM Customer
        WHERE name = $(customer/name);
    INTO address
        DO NONEMPTY SELECT street,city,country
            FROM Address WHERE id = $id;
END
```

Rewriting Error Messages for the Client

Sometimes internal error messages are confusing and are not helpful to the user that does not have a deeper knowledge about the database internals. For a set of error types it is possible to add a message to be shown to the user if an error of a certain class happens. The instruction ON ERROR <errorclass> HINT <message>; following a database instruction catches the errors of class <errorclass> and add the string <message> to the error message show to the user.

We can have many subsequent ON ERROR definitions in a row if the error classes to be caught are various.

Database Error HINT Example

The following example shows the usage HINTs in error cases. It catches errors that are constraint violations (error class CONSTRAINT) and extends the error message with a hint that will be shown to the client as error message:

```
TRANSACTION insertCustomer
BEGIN
    DO INSERT INTO Customer (name) VALUES ($(name));
    ON ERROR CONSTRAINT
        HINT "Customers must have a unique name.";
END
```

On the client side the following message will be shown:

```
unique constaint violation in transaction 'insertCustomer'
-- Customers must have a unique name.
```

Substructures in the Result

We already learned how to define substructures of the transaction function result with the RESULT INTO directive of a TRANSACTION. But we can also define a scope in the result structure for sub blocks. A sub-block in the result is declared with

```
INTO <resulttag>
BEGIN
 ...<instruction list>...
END
```

All the results of the instruction list that get into the final result will be attached to the substructure with name <resulttag>. The nesting of result blocks can be arbitrary and the path of the elements in the result follows the scope of the sub-blocks.

Explicit Definition of Elements in the Result

The result of a transaction consists normally of database command results that are mapped into the result with the attached INTO directive. For printing variable values or constant values you can in certain SQL databases use a select constant statement without specifying a table. Unfortunately select of constants might not be supported in your database of choice. Besides that explicit printing seems to be much more readable. The statement INTO <resulttag> PRINT <value>; prints a value that can be a constant, variable or an input or result reference into the substructure named <resulttag>. The following artificial example illustrates this.

```
TRANSACTION doPrintX
RESULT INTO person
BEGIN
    INTO name PRINT 'jussi';
    INTO id PRINT '1';
END
```

Database Specific Code

TDL allows the support of different transaction databases with one code base. For example one for testing and demonstration and one for the productive system. We can tag transactions,subroutines

or whole TDL sources as beeing valid for one or a list of databases with the command `DATABASE` followed by a comma separated list of database names as declared in the configuration. The following example declares the transaction function 'getCustomer' to be valid only for the databases DB1 and DBtest.

```
TRANSACTION getCustomer
DATABASE DB1,DBtest
BEGIN
    INTO customer
        DO SELECT * FROM CustomerData
            WHERE ID=$(id);
END
```

The following example does the same but declares the valid databases for the whole TDL file. In this case the database declaration has to appear as first declaration in the file.

```
DATABASE DB1,DBtest

TRANSACTION getCustomer
BEGIN
    INTO customer DO SELECT *
        FROM CustomerData WHERE ID=$(id);
END
```

Subroutine Templates

To reuse code with different context, for example for doing the same procedure on different tables, subroutine templates can be defined in TDL. Subroutine templates become useful when we want to make items instantiable that are not allowed to be dependent on variable arguments. Most SQL implementations for example forbid tables to be dependent on variable arguments. To reuse code on different tables you can define subroutine templates with the involved table names as template argument. The following example defines a transaction using the template subroutine insertIntoTree on a table passed as template argument. The subroutine template arguments are substituting the identifiers in embedded database statements by the passed identifier. Only whole identifiers and not substrings of identifiers and no string contents are substituted.

```
TEMPLATE <TreeTable>
SUBROUTINE insertIntoTree( parentID)
BEGIN
    DO NONEMPTY UNIQUE SELECT rgt FROM TreeTable
        WHERE ID = $PARAM.parentID;
    DO UPDATE TreeTable
        SET rgt = rgt + 2 WHERE rgt >= $1;
    DO UPDATE TreeTable
        SET lft = lft + 2 WHERE lft > $1;
    DO INSERT INTO TreeTable (parentID, lft, rgt)
        VALUES ( $PARAM.parentID, $1, $1+1);
```

```
        DO NONEMPTY UNIQUE SELECT ID AS "ID" from TreeTable
            WHERE lft = $1;
END

TRANSACTION addTag
BEGIN
    DO insertIntoTree<TagTable>( $(parentID) )
    DO UPDATE TagTable
        SET name=$(name),description=$(description)
            WHERE ID=$RESULT.id;
END
```

Includes

TDL has the possibility to include files for reusing subroutines or subroutine templates in different modules. The keyword INCLUDE followed by the name of the relative path of the TDL file without the extension .tdl includes the declarations of the included file. The declarations in the included file are treated as they would have been made in the including file instead. The following example swhows the use of include. We assume that the subroutine template insertIntoTree of the example before is defined in a separate include file treeOperations.tdl located in the same folder as the TDL program.

```
INCLUDE treeOperations

TRANSACTION addTag
BEGIN
    DO insertIntoTree<TagTable>( $(parentID) )
    DO UPDATE TagTable
        SET name=$(name),description=$(description)
            WHERE ID=$RESULT.id;
END
```

Auditing

TDL defines hooks to add function calls for auditing transactions. An audit call is a form function call with a structure build from transaction input and some database results. An auditing function call can be marked as critical, so that the final commit is dependent not only on the transaction success but also on the success of the auditing function call. The following two examples show equivalent calls of audit. One with the function call syntax for calls with a flat structure (only atomic parameters) as parameter and one with the parameter build from a result structure of a BEGIN..END block executed. The later one can be used for audit function calls with a more complex parameter structure.

Audit example with function call syntax

```
TRANSACTION doInsertUser
BEGIN
 DO INSERT INTO Users (name) values ($(name));
 DO SELECT id FROM Users WHERE name = $(name);
```

```
END
AUDIT CRITICAL auditUserInsert( $RESULT.id, $(name) )
```

Audit example with parameter as structure

```
TRANSACTION doInsertUser
BEGIN
 DO INSERT INTO Users (name) values ($(name));
 DO SELECT id FROM Users WHERE name = $(name);
END
AUDIT CRITICAL auditUserInsert WITH
BEGIN
 INTO id PRINT $RESULT.id;
 INTO name PRINT $(name);
END
```

3.2. Functions in .NET

3.2.1. Introduction

You can write functions for the logic tier of Wolframe in languages based on .NET (http://www.microsoft.com/net) like for example C# and VB.NET. Because .NET based libraries can only be called by Wolframe as a compiled and not as an interpreted language, you have to build a .NET assembly out of a group of function implementations before using it. There are further restrictions on a .NET implementation. We will discuss all of them, so that you should be able to write and configure .NET assemblies for using in Wolframe on your own after reading this chapter.

3.2.2. Configuration

For enabling .NET you have to declare the loading of the module 'mod_command_dotnet' in the main section of the server configuration file.

```
module mod_command_dotnet
```

For the configuration of the .NET assemblies to be loaded, see section 'Configure .NET Modules'.

3.2.3. Function Interface

Function Context

In .NET the building blocks for functions called by Wolframe are classes and method calls. The way of defining callable items for Wolframe is restricted either due to the current state of the Wolframe COM/.NET interoperability implementation or due to general or version dependent restrictions of .NET objects exposed via COM/.NET interop. We list here the restrictions:

- The methods exported as functions for Wolframe must not be defined in a nested class. They should be defined in a top level class without namespace. This is a restriction imposed by the current development state of Wolframe.

- The class must be derived from an interface with all methods exported declared.

- The methods must not be static because COM/.NET interop, as far as we know, cannot cope with static method calls. Even if the methods nature is static, they have to be defined as ordinary method calls.

Function Signature

Functions callable from Wolframe take an arbitrary number of arguments as input and return a structure (`struct`) as output. The named input parameters referencing atomic elements or complex structures are forming the input structure of the Wolframe function. A Wolframe function called with a structure containing the elements "A" and "B" is implemented in .NET as function taking two arguments with the name "A" and "B". Both "A" and "B" can represent either atomic elements or arbitrary complex structures. .NET functions that need to call global Wolframe functions, for example to perform database transactions, need to declare a `ProcProvider` interface from Wolframe namespace as additional parameter. We will describe the `ProcProvider` interface in a separate section of this chapter.

Example

The following simple example without provider context is declared without marshalling and introspection tags. It can therefore not be called by `Wolframe`. We explain later how to make it callable. The example just illustrates the structure of the exported object with its interface (example C#):

```
using System;
using System.Runtime.InteropServices;

public struct Address
{
    public string street;
    public string country;
};

public interface FunctionInterface
{
    Address GetAddress( string street, string country);
}

public class Functions : FunctionInterface
{
    public Address GetAddress( string street, string country)
    {
        Address rt = new Address();
        rt.street = street;
        rt.country = country;
        return rt;
    }
}
```

3.2.4. Prepare .NET Assemblies

Wolframe itself is not a .NET application. Therefore it has to call .NET functions via COM/.NET interop interface of a hosted CLR (Common Language Runtime). To make functions written in .NET callable by Wolframe, the following steps have to be performed:

Make Assemblies COM Visible

First the asseblies with the functions exported to Wolframe have to be build COM visible. To make the .NET functions called from Wolframe COM visible, you have to tick "Properties/Assembly Information" the switch "Make assembly COM visible". Furthermore every object and method that is part of the exported API (also objects used as parameters) has to be tagged in the source as COM visible with `[ComVisible(true)]`.

Tag Exported Objects with a Guid

Each object that is part of the exported API has to be tagged with a global unique identifier (Guid) in order to be adressable. Modules with .NET functions will have to be globally registered and the objects need to be identified by the Guid because that's the only way to make the record info structure visible for Wolframe. The record info structure is needed to serialize/deserialize .NET objects from another interpreter context that is not registered for .NET. There are many ways to create a Guid and tag an object like this: `[Guid("390E047F-36FD-4F23-8CE8-3A4C24B33AD3")]`.

Add Marshalling Tags to Values

For marshalling function calls correctly, Wolframe needs tags for every parameter and member of a sub structure of a parameter of methods exported as functions. The following table lists the supported types and their marshalling tags:

Table 3.1. Marshalling Tags

.NET Type	Marshalling Tag
I2	`[MarshalAs(UnmanagedType.I2)]`
I4	`[MarshalAs(UnmanagedType.I4)]`
I8	`[MarshalAs(UnmanagedType.I8)]`
UI2	`[MarshalAs(UnmanagedType.UI2)]`
UI4	`[MarshalAs(UnmanagedType.UI4)]`
UI8	`[MarshalAs(UnmanagedType.UI8)]`
R4	`[MarshalAs(UnmanagedType.R4)]`
R8	`[MarshalAs(UnmanagedType.R8)]`
BOOL	`[MarshalAs(UnmanagedType.BOOL)]`
string	`[MarshalAs(UnmanagedType.BStr)]`
RECORD	no tag needed
array of structures	`[MarshalAs(UnmanagedType.SafeArray, SafeArraySubType = VarEnum.VT_RECORD)]`
array of strings	`[MarshalAs(UnmanagedType.SafeArray, SafeArraySubType = VarEnum.VT_BSTR)]`
array of XX (XX=I2,I4,I8,..)	`[MarshalAs(UnmanagedType.SafeArray, SafeArraySubType = VarEnum.VT_XX)]`

Decimal floating point and numeric types (DECIMAL) are not yet supported, but will soon be available.

Example with COM Introspection Tags

The following C# module definition repeats the example introduced above with the correct tagging for COM visibility and introspection:

```csharp
using System;
using System.Runtime.InteropServices;

[ComVisible(true)]
[Guid("390E047F-36FD-4F23-8CE8-3A4C24B33AD3")]
public struct Address
{
    [MarshalAs(UnmanagedType.BStr)] public string street;
    [MarshalAs(UnmanagedType.BStr)] public string country;
};

[ComVisible(true)]
public interface FunctionInterface
{
    [ComVisible(true)]  Address GetAddress( [MarshalAs(UnmanagedType.BStr)] str
}

[ComVisible(true)]
[ClassInterface(ClassInterfaceType.None)]
public class Functions : FunctionInterface
{
    public Address GetAddress([MarshalAs(UnmanagedType.BStr)] string street, [Ma
    {
        Address rt = new Address();
        rt.street = street;
        rt.country = country;
        return rt;
    }
}
```

Create a Type Library

For making the API introspectable by Wolframe, we have to create a TLB (Type Library) file from the assembly (DLL) after build. The type library has to be recreated every time the module interface (API) changes. The type library is created with the program `tlbexp`. All created type library (.tlb) file that will be loaded with the same runtime environment have to be copied into the same directory. They will be referenced for introspection in the configuration. The configuration of .NET will be explained later.

Register the Type Library

The type library created with `tlbexp` has also to be registered. For this you call the program `regtlibv12` with your type library file (.tlb file) as argument. The type libary fegistration has to be repeated when the the module interface (API) changes.

Register the Assembly in the GAC

Wolframe does not accept local assemblies. In order to be addressable over the type library interface assemblies need to be put into the global assembly cache (GAC). Unfortunately this has to be

repeated every time the assembly binary changes. There is no way around. For the registration in the GAC we have to call the program `gacutil /if <assemblypath>` with the assembly path <assemblypath> as argument. The command `gacutil` has to be called from administrator command line. Before calling `gacutil`, assemblies have to be strongly signed. We refer here to the MSDN documentation for how to sign an application.

Register the Types in the Assembly

We have to register the types declared in the assembly to enable Wolframe to create these types. An example could be a provider function returning a structure that is called from a Wolframe .NET function. The structure returned here has to be build in an unmanaged context. In order to be valid in the managed context, the type has to be registered. For the registration of the types in an assembly we have to call the program `regasm <assemblypath>` with the assembly path <assemblypath> as argument. The command `regasm` has to be called from administrator command line.

3.2.5. Calling Wolframe Functions

Wolframe functions in .NET calling globally defined Wolframe functions need to declare the processor provider interface as an additional parameter. The processor provider interface is defined as follows (example C#):

```
namespace Wolframe
{
    public interface ProcProvider
    {
        object call(
        [In] string funcname,
        [In] object argument,
        [In] Guid resulttype);
    }
}
```

To use it we have to include the reference to the assembly `WolframeProcessorProvider.DLL`.

The interface defined there has a method `call` taking 3 arguments: The name of the function to call, the object to pass as argument and the `Guid` of the object type to return. The returned object will be created with help of the registered `Guid` and can be casted to the type with this `Guid`.

The following example shows the usage of a `Wolframe.ProcProvider` call. The method `GetUserObject` is declared as Wolframe function requiring the processor provider context as additional argument and taking one object of type `User` as argument named `usr`. The example function implementation redirects the call to the global Wolframe function named `GetAddress` returning an object of type `Address` (example C#):

```
public Address GetUserAddress(
            Wolframe.ProcProvider provider,
            User usr
) {
    Address rt = (Address)provider.call(
                    "GetAddress", usr,
                    typeof(Address).GUID);
    return rt;
}
```

The objects involved in this example need no more tagging because the provider context and also structures (`struct`) need no additional mashalling tags.

3.2.6. Configure .NET Assemblies

.NET modules are grouped together in a configuration block that specifies the configuration of the Microsoft Common Language Runtime (CLR) used for .NET interop calls. The configuration block has the header `runtimeEnv dotNET` and configures the version of the runtime loaded (clrversion) and the path where the typelibraries (.tlb) files can be found (typelibpath).

With the `assembly` definitions you declare the registered assemblies to load.

```
RuntimeEnv dotNet
{
    clrversion      "v4.0.30319"
    typelibpath     programs/typelibrary
    assembly        "Functions, Version=1.0.0.30, Culture=neutral, PublicKeyToken=
    assembly        "Utilities, Version=1.0.0.27, Culture=neutral, PublicKeyToken=
}
```

Table 3.2. Attributes of assembly declarations

Name	Description
<no identifier>	The first element of the assembly definition does not have an attribute identifier. The value is the name of the assembly (and also of the type library)
Version	4 element (Major.Minor.Build.Revision) version number of the assembly. This value is defined in the assembly info file of the assembly project.
Culture	For Wolframe applications until now always "neutral". Functionality is in Wolframe not yet culture dependent on the server side.
PublicKeyToken	Public key token values for signed assemblies. See next section how to set it.
processorArchitecture	Meaning not explained here. Has on ordinary Windows .NET plattforms usually the value "MSIL". Read the MSDN documentation to dig deeper.

Get the PublicKeyToken

We already found out that Wolframe .NET modules have to be strongly signed. Each strongly signed assembly has such a public key token that has to be used as attribute when referencing the assembly.

We can get the `PublicKeyToken` of the assembly by calling `sn -T <assemblypath>` from the command line (cmd) with <assemblypath> as the path of the assembly. The printed value is the public key to insert as attribute value of `PublicKeyToken` in the Wolframe configuration for each .NET assembly.

3.2.7. Validation Issues

Languages of .NET called via the CLR are strongly typed languages. This means that the input of a function and the output is already validated to be of a strictly defined structure. So a validation by passing the input through a form might not be needed anymore. Validation with .NET data structures is weaker than for example XML validation with forms defined in a schema language. But only if distinguishing XML attributes from content elements is an issue. See in the documentation of the standard command handler how validation can be skipped with the attribute `SKIP`.

3.3. Functions in Python

3.3.1. Current Development State

You can write functions for the logic tier of Wolframe in the Python programming language (http:// www.python.org).

The implementation of Python calls is not yet available. But Wolframe will provide Python functions soon.

3.4. Functions in Lua

3.4.1. Introduction

You can write functions for the logic tier of Wolframe with Lua. Lua is a scripting language designed, implemented, and maintained at PUC-Rio in Brazil by Roberto Ierusalimschy, Waldemar Celes and Luiz Henrique de Figueiredo (see http://www.lua.org/authors.html). A description of Lua is not provided here. For an introduction into programming with Lua see http://www.lua.org. The official manual which is also available as book is very good. Wolframe introduces some Lua interfaces to access input and output and to execute functions.

3.4.2. Configuration

For enabling Lua you have to declare the loading of the module 'mod_command_lua' in the main section of the server configuration file.

```
module mod_command_lua
```

Each Lua script referenced has to be declared in the `Processor` section of the configuration with `program <sourcefile>`. The script is recognized as Lua script by the file extension ".lua". Files without this extension cannot be loaded as Lua scripts.

3.4.3. Declaring Functions

For Lua we do not have to declare anything in addition to the Lua script. If you configure a Lua script as program, all global functions declared in this script are declared as global form functions. For avoiding name conflicts you should declare private functions of the script as `local`.

3.4.4. Wolframe Provider Library

Wolframe lets you access objects of the global context through a library called `provider` offering the following functions:

Table 3.3. Method

Name	Parameter	Returns
form	Name of the form	An instance of the form
type	Type name and initializer list	A constructor function to create a value instance of this type
formfunction	Name of the function	Form function defined in a Wolframe program or module
document	Content string of the document to process	Returns an object of type "document" that allows the processing of the contents passed as argument. See description of type "document"

3.4.5. Using Atomic Data Types

Wolframe lets us extend the type system consisting of Lua basic data types with our own. We can create atomic data types defined in a module or in a DDL datatype definition program (.wnmp file). For this you call the type method of the provider with the type name as first argument plus the type initializer argument list as additional parameters. The function returns a constructor function that can be called with the initialization value as argument to get a value instance of this type. The name of the type can refer to one of the following:

Table 3.4. List of Atomic Data Types

Class	Initializer Arguments	Description
Custom Data Type	Custom Type Parameters	A custom data type defined in a module with arithmetic operators and methods
Normalization Function	Initializer Arguments	A type defined as normalization function in a module
DDL Data Type	(no arguments)	A normalizer defined as sequence of normalization functions in a .wnmp source file
Data Type 'bignumber'	(no arguments)	Arbitrary precision number type
Data Type 'datetime'	(no arguments)	Data type representing date and time down to a granularity of microseconds

Data Type 'datetime'

The data type 'datetime' is used as interface for date time values.

Table 3.5. Methods of 'datetime'

Method Name	Arguments	Description
<constructor>	year, month, day, hour, minute, second ,millisecond, microsecond	Creates a date and time value with a granularity down to microseconds
<constructor>	year, month, day, hour, minute, second ,millisecond	Creates a date and time value with a granularity down to milliseconds
<constructor>	year, month, day, hour, minute, second	Creates a date and time value

Method Name	Arguments	Description
<constructor>	year, month, day	Creates a date value
year	(no arguments)	Return the value of the year
month	(no arguments)	Return the value of the month (1..12)
day	(no arguments)	Return the value of the day in the month (1..31)
hour	(no arguments)	Return the value of the hour in the day (0..23)
minute	(no arguments)	Return the value of the minute (0..59)
second	(no arguments)	Return the value of the second (0..63 : 59 + leap seconds)
millisecond	(no arguments)	Return the value of the millisecond (0..1023)
microsecond	(no arguments)	Return the value of the microsecond (0..1023)
__tostring	(no arguments)	Return the date as string in the format YYYYMMDD, YYYYMMDDhhmmss, YYYYMMDDhhmmssll or YYYYMMDDhhmmssllcc, depending on constructor used to create the date and time value.

Data Type 'bignumber'

The data type 'bignumber' is used to reference fixed point BCD numbers with a precision of 32767 digits between -1E32767 and +1E32767.

Table 3.6. Methods of 'datetime'

Method Name	Arguments	Description
<constructor>	number value as string	Creates a bignumber from its string representation
<constructor>	number value	Creates a bignumber from a lua number value (double precision floating point number)
precision	(no arguments)	Return the number of significant digits in the number
scale	(no arguments)	Return the number of fractional digits (may be negative, may be bigger than precision)
digits	(no arguments)	Return the significant digits in the number
tonumber	(no arguments)	Return the number as lua number value (double precision floating point number) with possible lost of accurancy
__tostring	(no arguments)	Return the big number value as string (not normalized).

3.4.6. Serialization Iterators

Some objects introduced in the following sections are accessible through iterators. These are called serialization iterators and are Lua function closures. You should not mix them with the standard Lua iterators though the semantic is similar. Serialization iterators do not return nodes of the tree as subtree objects but only the node data in the order of a pre-order traversal. You can recursively iterate on the tree and build the object during traversal if you want. The returned elements of the serialization iterators are tuples with the following meaning:

Table 3.7. Serialization Iterator Elements

Tuple First Element	Tuple Second Element	Description
NIL/false	string/number	Open (tag is second element)
NIL/false	NIL/false	Close
Any non NIL/false	string/number	Attribute assignment (value is first, tag is second element)
string/number	NIL/false	Content value (value is first element)

3.4.7. Iterator Library

Wolframe lets you access serialization iterators through a library called `iterator` offering the following functions:

Table 3.8. Method

Name	Parameter	Returns
scope	serialization iterator (*)	An iterator restricted on the subnodes of the last visited node (**)

(*) See section "Serialization Iterator"

(**) If iterator.scope is called, all elements of the returned iterator has to be visited in order to continue iteration with the origin iterator on which iterator.scope was called.

3.4.8. Global Objects

Besides the provider library Wolframe defines the following objects global in the script execution context:

Name   Description
logger   object with methods for logging or debugging

3.4.9. Using Forms

The provider function `provider.form( )` with the name of the form as string as parameter returns an empty instance of a form. It takes the name of the form as string argument. If you for example have a form configured called "employee" and you want to create an employee object from a Lua table, you call

```
bcf = provider.form( "employee" )
```

```
bcf:fill( {surname='Hans', name='Muster', company='Wolframe'} )
```

The first line creates the data form object. The second line fills the data into the data form object.

The form method `fill` takes a second optional parameter. Passing "strict" as second parameter enforces a strict validation of the input against the form, meaning that attributes are checked to be attributes (when using XML serialization) and non optional elements are checked to be initialized. Passing "complete" as second parameter forces non optional elements to be checked for initialization but does not distinguish between attributes and content values. "relaxed" is the default and checks only the existence of filled-in values in the form.

Given the following validation form in simple form DDL syntax (see chapter "Forms"):

```
FORM Employee
{
    employee
    {
        ID !@int                ; Internal customer id (mandatory)
        name !string            ; Name of the customer (mandatory)
        company string          ; Company he is working for (optional)
    }
}
```

the call of `fill` in the following piece of code will raise an error because some elements of the form ('ID' and 'name') are missing in the input:

```
bc = provider.form( "employee" ):fill( {company='Wolframe'}, "strict" )
```

To access the data in a form there are two form methods available. `get()` returns a serialization iterator on the form data. There is also a method `value()` that returns the form data as Lua data structure (a Lua table or atomic value).

3.4.10. Form Functions

For calling transactions or built-in functions loaded as modules the Lua layer defines the concept of functions. The provider function `provider.formfunction` with the name of the function as argument returns a Lua function. This function takes a table or a serialization iterator as argument and returns a data form structure. The data in the returned form data structure can be accessed with `get()` that returns a serialization iterator on the content and `value()` that returns a Lua table or atomic value.

If you for example have a transaction called "insertEmployee" defined in a transaction description program file declared in the configuration called "insertEmployee" and you want to call it with the 'employee' object defined above as input, you do

```
f = provider.formfunction( "insertEmployee")
res = f ( {surname='Hans', name='Muster', company='Wolframe'} )
```

```
t = res:value()
output:print( t[ "id" ] )
```

The first line creates the function called "insertEmployee" as Lua function. The second calls the transaction, the third creates a Lua table out of the result and the fourth selects and prints the "id" element in the table.

3.4.11. List of Lua Objects

This is a list of all objects and functions declared by Wolframe:

Table 3.9. Data forms declared by DDL

Method Name	Arguments	Returns	Description
get		serialization iterator (*)	Returns a serialization iterator on the form elements
value		Lua table	Returns the contents of the data form as Lua table or atomic value
__tostring		string	String representation of form for debugging
name		string	Returns the global name of the form.
fill	Lua table or serialization iterator (*), optional validation mode (**)	the filled form (for concatenation)	Validates input and fills the input data into the form.

(*) See section "Serialization Iterator"

(**) "strict" (full validation), "complete" (only check for all non optional elements initialization) or "relaxed" (no validation except matching of input to elements)

Table 3.10. Data forms returned by functions

Method Name	Returns	Description
get	serialization iterator (*)	Returns a serialization iterator on the form elements
value	Lua table or atomic value	Returns the contents of the data form as Lua table or atomic value
__tostring	string	String representation of form for debugging

(*) See section "Serialization Iterator"

Table 3.11. Document

Method Name	Arguments	Description
docformat	-	Returns the format of the document {'XML','JSON',etc..}

Method Name	Arguments	Description
as	filter and/or document type table	Attaches a filter to the document to be used for processing
doctype	-	Returns the document type of the content. For retrieving the document type you have first to define a filter.
value	-	Returns the contents of the document as Lua table or atomic value
form	-	Returns the contents of the document as filled form instace
get	-	Returns a serialization iterator (*) on the form elements

(*) See section "Serialization Iterator"

Table 3.12. Logger functions

Method Name	Arguments	Description
logger.printc	arbitrary list of arguments	Print arguments to standard console output
logger.print	loglevel (string) plus arbitrary list of arguments	log argument list with defined log level

Table 3.13. Global functions

Function Name	Arguments	Description
provider.form	name of form (string)	Returns an empty data form object of the given type
provider.formfunction	name of function (string)	Returns a lua function to execute the Wolframe function specified by name
provider.type	name of data type (string)	Returns a constructor function for the data type given by name. The name specifies either a custom data type or a normalization function as used in forms or one of the additional userdata types 'datetime' or 'bignumber'.
provider.document	Content string of the document to process	Returns an object of type "document" that allows the processing of the contents passed as argument. See description of type "document"
scope	serialization iterator (*)	Returns a serialization iterator to iterate till the end of the current tag (**). Consumes the iterated scope from the argument iterator.

(*) See section "Serialization Iterator"

(**) The serialization iterator of a defined scope must be consumed completely before consuming anything of the parent iterator. Otherwise it may lead to unexpected results because they share some part of the iterator state.

3.5. Functions in Native C++

3.5.1. Introduction

You can write functions for the logic tier of Wolframe with C++. Because native C++ is by nature a compiled and not an interpreted language, you have to build a module out of your function implementation.

3.5.2. Prerequisites

For native C++ you need a C++ build system with compiler and linker or an integrated development environment for C++.

3.5.3. Declaring Functions

Form functions declared in C++ have two arguments. The output structure to fill is passed by reference as first and the input structure passed is by value. The input structure copy should not be modified by the callee. This means in C++ that it is passed as const reference. The function returns an `int` that is `0` on success and any other value indicating an error code. The function may also throw a runtime error exception in case of an error. The following example shows a function declaration. The function declaration is not complete because the input output structures need to be declared with some additional attributes needed for introspection. We will explain this in the following section.

Example Function Declaration

The function takes a structure as input and writes the result into an output structure. In this example input and output type are the same, but this is not required. It's just the same here for simplicity.

The elements of the function declaration are put into a structure with four elements. The `typedef` for the InputType and OutputType structures is required, because the input and output types should be recogniceable without complicated type introspection templates. (Template based introspection might cause spurious and hard to understand error messages when building the module).

The function `name` returns the name of the function that identifies the function in the Wolframe global scope.

The `exec` function declared as static function with this signature refers to the function implementation.

```
// ... PUT THE INCLUDES FOR THE "Customer" STRUCTURE DECLARATION HERE !

struct ProcessCustomer
{
    typedef Customer InputType;
    typedef Customer OutputType;
    static const char* name() {return "process_customer";}

    static int exec( const proc::ProcessorProvider* provider, InputType& res, c
};
```

3.5.4. Input/Output Data Structures

For defining input and output parameter structures in C++ you have to define the structure and its serialization description. The serialization description is a static function `getStructDescription` without arguments returning a const structure that describes what element names to bind to which structure elements.

The following example shows a form function parameter structure defined in C++.

Header File

Declares the structure and the serialization description of the structure. Structures may contain structures with their own serialization description.

```
#include "serialize/struct/structDescriptionBase.hpp"
#include <string>

namespace _Wolframe {
namespace example {

struct Customer
{
    int ID;                          // Internal customer id
    std::string name;                // Name of the customer
    std::string canonical_Name;      // Customer name in canonical form
    std::string country;             // Country
    std::string locality;            // Locality

    static const serialize::StructDescriptionBase* getStructDescription();
};

}}//namespace
```

Source File

Declares 'ID' as attribute and name, canonical_Name, country, locality as tags. The '--' operator marks the end of attributes section and the start of content section.

```
#include "serialize/struct/structDescription.hpp"

using namespace _Wolframe;

namespace {
struct CustomerDescription :public serialize::StructDescription<Customer>
{
    CustomerDescription()
    {
```

```
            (*this)
            ("ID", &Customer::ID)
            --
            ("name", &Customer::name)
            ("canonical_Name", &Customer::canonical_Name)
            ("country", &Customer::country)
            ("locality", &Customer::locality)
            ;
    }
};

const serialize::StructDescriptionBase* Customer::getStructDescription()
{
    static CustomerDescription rt;
    return &rt;
}
```

3.5.5. Writing the Module

Now we have all pieces together to build a loadable Wolframe module with our example C++ function. The following example shows what you have to declare in the main module source file.

Module Declaration

The module declaration needs to include `appdevel.hpp` and of course all headers with the function and data structure declarations needed. The module starts with the header macro CPP_APPLICATION_FORM_FUNCTION_MODULE with a short description of the module. What follows are the function declarations declared with the macro CPP_APPLICATION_FORM_FUNCTION. This macro has the following arguments in this order:

Name	Description
NAME	identifier of the function
FUNCTION	implementation of the function
OUTPUT	output structure of the function
INPUT	input structure of the function

The declaration list is closed with the parameterless footer macro CPP_APPLICATION_FORM_FUNCTION_MODULE_END. The following example shows an example module declaration:

```
#include "appDevel.hpp"
// ... PUT THE INCLUDES FOR THE "ProcessCustomer" FUNCTION DECLARATION HERE !

#include "customersFunction.hpp"

using namespace _Wolframe;

WF_MODULE_BEGIN( "ProcessCustomerFunction", "process customer function")
WF_FORM_FUNCTION("process_customer",ProcessCustomer::exec,Customer,Customer)
WF_MODULE_END
```

3.5.6. Building the Module

For building the module we have to include all modules introduced here and to link at against the wolframe serialization library (wolframe_serialize) and the wolframe core library (wolframe).

3.5.7. Using the Module

The module built can be loaded as the other modules by declaring it in the wolframe LoadModules section of the configuration. Simply list it there with `module   <yourModuleName>` with <yourModuleName> being the name or path to your module.

3.5.8. Validation Issues

C++ is a strongly typed language. This means that the input of a function and the output is already validated to be of a strictly defined structure. So a validation by passing the input through a form might not be needed anymore. The constructs used to describe structures of Wolframe in native C++ are even capable of describing attributes like used in XML (section 'Input/Output Data Structures' above). See in the documentation of the standard command handler how validation can be skipped with the attribute `SKIP`.

Chapter 4. Forms

Forms are data structures used to validate input and output data and to do some basic normalization in order to make data accessible in a uniform way. Forms are defined in a data definition language (DDL) and translated by a compiler at startup. Those compilers are defined as loadable modules.

This chapter describes how form data schemas are linked to the logic tier. It introduces a data description language (DDL) called `simpleform` that allows you to specify data schemas with the validation and normalization of atomic types. It also describes the Wolframe module concept for form descriptions that allows you to add a compiler for your existing data schemas.

After reading this chapter you should be able to write data forms of Wolframe of the logic tier in the `simpleform` data description language on your own. You should also know how a new data description language (DDL) could be added.

Be aware that you have to configure a data description language type (DDL compiler) of the logic tier in Wolframe before using it. Each chapter introducing a data form description language will have a section that describes how the server configuration of Wolframe has to be extended for its availability.

4.1. Form Data Definition Languages

4.1.1. Introduction

Form data structures can be defined in a DDL (Data Definition Language). It depends very much on the application what DDL is best to use. Users may already have their data definitions defined in a certain way. The form DDL can be defined in the way you want. Wolframe offers a plugin mechanism for DDL compilers and provides examples of such compilers. You configure the DDL sources to load and the compiler to use.

With the DDL form description we get a deserialization of some content into a structure and a serialization for the output. We get also a validation and normalization procedure of the content by assigning types to atomic form elements that validate and normalize the data elements. Most of the business transactions should be doable as input form description, output form description and a transaction that maps input to output without control flow aware programming.

All types of data forms introduced here are equivalent in use for all programs.

4.1.2. Forms in Simpleform DDL

As example of a form DDL we provide the *simpleform* DDL. The format is based on the "INFO"-format introduced by Marcin Kalicinski for the boost property tree library. We used this library to show an example that is easy to understand and small enough. The format uses key value pairs separated with spaces for atomic elements and curly brackets '{' '}' to describe structures. The key represents the name of the element and the value represents the type of the element. The type is defined by a typename and some operators that describe additional properties.

Each form declaration starts with a keyword 'FORM' or 'STRUCT'. The difference between 'FORM' and 'STRUCT' is that the later is used for declarations that are only referenced inside the same file as sub structure reference, while 'FORM' declares a structure to be exported as global form declaration. This header is followed by the structure declaration inside curly brackets '{' '}'.

There is only one predefined data type known in *simpleform* DDL: "string". All other data types are defined as sequence of *normalizer* functions in a normalize definition file. The *normalizer* functions assigned to a type validate the value and transform it to its normalized form. We explain in the next section how data types are defined.

The following element attributes are known in *simpleform* DDL:

Table 4.1. element attributes in simpleform

Attribute	Location	Description
@	prefix of data type	Expresses that the element is an attribute and not a content element of the structure. This has only influence on the XML or similar representation of the form content
?	prefix of data type	Expresses that the element is optional also in strict validation
^	prefix of form name	Expresses that the element is optional and refers to a structure defined in the same module that is expanded only if the element is present. With this construct it is possible to define recursive structures like trees.
!	prefix of data type	Expresses that the element is always mandatory (also in non strict validation)
[]	suffix of data type	Expresses that the element is an array of this type
[]	without data type	Expresses this element is an array of structures and that the structure defined describes the prototype (initialization) element of the array.
='..'	end of data type declaration	Expresses that '..' is the default initialization value of this element.

Using a single underscore as typename ('_') means that element is embedded into the structure without being referenceable by name. In case of an atomic value it means that value represents the content value of the structure. In case of a substructure it means that the structure inherits the embedded elements of the substructure.

The following example shows a form defined in *simpleform* DDL.

```
FORM Customer
{
    customer
    {
        ID !@int                 ; Internal customer id (mandatory)
        name string              ; Name of the customer
        canonical_Name string    ; Customer name in canonical form
        country string           ; Country
        locality ?string         ; Locality (optional)
    }
}
```

4.2. Datatypes in DDLs

4.2.1. Introduction

The basic elements to build atomic data types in Wolframe are normalization functions. Basic normalization functions are written in C++ and loadable as modules.

As we already mentioned are atomic elements in forms typed. With each type a function is associated to validate and normalize the atomic element of that type. There is only one predefined type called 'string'. strings are neither validated nor transformed for processing in any way. The others are defined in files with the extension `.wnmp` that are referenced as programs in the configuration.

A `.wnmp` file contains assignments of a type name to sequences of basic normalization function calls where the first takes the initial input. A normalization function call can either be a normalizer function or a custom data type defined in a module or a method of the predecessing custom data type in the sequence of the normalization function calls. The output of a function in the sequence gets the input of the next one and the final output for the last one. Each normalization step validates the input as atomic type (arithmetic,string,etc.) and transforms it to another atomic type.

4.2.2. Example

The example defines 3 numeric types including trimming of the input string for mode tolerant parsing and a string type that is converted to lowercase as normalization.

```
int=trim,integer(5);
uint=trim,unsigned;
currency=trim,fixedpoint( 13, 2);
name=trim,lcname;
```

4.2.3. Language Description

Type Assignments

Each type declaration in a `.wnmp` file starts with an identifier followed by an assignment operator '='. The left side identifier specifies the name of the type. This type name can be used in a DDL as name instead of the built-in type `string`. A token of this type is validated and normalized with the comma separated sequence of function references on the right side of the assignment. A function reference consists of an identifier plus an optional comma separated list of constant arguments in brackets ('(' and ')'). The interpretation of the arguments depend on the function type. An integer type for example could have the maximum number of digits of the integer type.

4.2.4. Configuration

For declaring and using a .wnmp file in our example above, we have to load the module 'mod_normalize_string' and the module 'mod_normalize_number'. For this we add the following two lines to the LoadModules section of our Wolframe configuration:

```
    module mod_normalize_number
    module mod_normalize_string
```

We also have to add the declaration of the program "example.wnmp" (listing example above) to the Processor section of the configuration.

```
program example.wnmp
```

Chapter 5. Filters

Filters describe the transformation of serialized data to a unified serialization of hierarchical structured data and back. The application does not care about data formats as long as there exists a filter providing the unified form of serialization.

This chapter describes how filters for different data formats are linked to the logic tier. For each data format supported by Wolframe one or more filter type is introduced.

After reading this chapter you should be able to handle different document formats and encodings in the logic tier of Wolframe. You will know how to add programs for scriptable filters like XSLT.

Be aware that you have to configure a data filter of the logic tier in Wolframe before using it. Each chapter introducing a filter type will have a section that describes how the server configuration of Wolframe has to be extended for its availability.

5.1. XML Filter

5.1.1. Introduction

You can use XML for data filters in the logic tier of Wolframe. There are the following variants of XML filters available:

- libxml2 (http://www.xmlsoft.org) or

- textwolf (http://www.textwolf.net)

5.1.2. Character Set Encodings

The libxml2 and the textwolf filter support at least the following character set encodings. For character set encodings that are not in the list, please ask the Wolframe team.

- UTF-8 or

- UTF-16LE or

- UTF-16 (UTF-16BE) or

- UTF-32LE (UCS-4LE) but only with textwolf or

- UTF-32 (UTF-32BE or UCS-4BE) or

- ISO 8859 (code pages '1' to '9')

5.1.3. Configuration

For using an XML filter based libxml2, you have to load the module 'mod_filter_libxml2'. For this you add the following line to the LoadModules section of your Wolframe configuration:

```
module mod_filter_libxml2
```

For using an XML filter based textwolf, you have to load the module 'mod_filter_textwolf'. For this you add the following line to the LoadModules section of your Wolframe configuration:

```
        module mod_filter_textwolf
```

5.2. JSON Filter

5.2.1. Introduction

You can use JSON for data filters in the logic tier of Wolframe. The standard JSON filter of Wolframe is called cjson and based on the library cJSON (http://sourceforge.net/projects/cjson) from Dave Gamble.

5.2.2. Character Set Encodings

Without explicitly specified, the cjson filter support the following character set encodings. For character set encodings that are not in the list, please ask the Wolframe team.

- UTF-8 or

- UTF-16LE or

- UTF-16 (UTF-16BE) or

- UTF-32LE (UCS-4LE) or

- UTF-32 (UTF-32BE or UCS-4BE) or

5.2.3. Configuration

For using the JSON filter based cJSON, you have to load the module 'mod_filter_cjson'. For this you add the following line to the LoadModules section of your Wolframe configuration:

```
        module mod_filter_cjson
```

5.3. XSLT Filter

5.3.1. Introduction

You can use XSLT for data filters in the logic tier of Wolframe. The XSLT filter of Wolframe for is based on libxml2 (http://www.xmlsoft.org).

5.3.2. Character Set Encodings

Without explicitly specified, the XSLT filter support the following character set encodings. For character set encodings that are not in the list, please ask the Wolframe team.

- UTF-8 or

- UTF-16LE or

- UTF-16 (UTF-16BE) or

- UTF-32LE (UCS-4LE) or

- UTF-32 (UTF-32BE or UCS-4BE) or

5.3.3. Configuration

For using an XSLT filter based libxml2, you have to load the module 'mod_filter_libxml2'. For this you add the following line to the LoadModules section of your Wolframe configuration:

```
module mod_filter_libxml2
```

You also have to add the program of the XSLT filter into the Processor section of the configuration. The name of the filter is the filename of the XSLT filter program without path and extension. In our example the filter would be named invoice_ISOxxxx:

```
program invoice_ISOxxxx.xslt
```

Chapter 6. Testing and Defect Handling

In this chapter we learn how parts of a Wolframe application can be verified to work correctly. The basis for testing and debugging a Wolframe application is the command line tool wolfilter.

6.1. Using wolfilter

The command line program wolfilter allows you to call any Wolframe function or filter or mapping into a form structure on command line.

There are two possibilities to declare the items involved in the test. Either you pass the configuration with the option '--config' and the name of the command to execute or you declare the items one by one with program options. These two approaches are not mixable. Either you use '--config' or pass the parameters one by one. A try to mix both of them in one call is refused by wolfilter.

The following examples assume the input file name to be in.xml or in.json and the output file to be named out.xml or out.json respectively.

6.1.1. Testing a Filter

The following example shows the mapping through a libxml2 filter. Filters are tested by passing a dash '-' command to execute.

```
cat in.xml | wolfilter -f libxml2 -m mod_filter_libxml2 - > out.xml
```

The following example shows the processing of the input through an xslt filter and mapping the output through a token filter that shows the tokenization of the input by the input filter.

```
cat in.xml | wolfilter -i myfilter -o token\
 -m mod_filter_libxml2\
 -m mod_filter_token\
 -p myfilter.xslt - > out.xml
```

6.1.2. Testing a Form

The following example shows the mapping through a form defined with simpleform DDL. Mapping through forms is tested by passing the name of the form as command to execute.

```
cat in.xml | wolfilter -f libxml2 \
 -m mod_filter_libxml2\
 -p myform.sfrm MyForm > out.xml
```

we assume here that the form to use is defined in myform.sfrm and called MyForm.

6.1.3. Testing a Function

The following example shows the execution of a function written in Lua. A JSON filter is used for input and output.

```
cat in.xml | wolfilter -f cjson \
 -m mod_filter_cjson -m mod_command_lua \
 -m mod_command_directmap
 -p myfunc.lua MyFunc > out.json
```

we assume here that the exported function to call defined in myfunc.lua is called MyFunc.

Glossary

This is the glossary for the Wolframe Application Building Manual. Although it covers most of the terms used in the Wolframe world, some terms might be skipped if they are rarely used in this context. These terms are explained in the Application Building Manual.

External glossary

Data Definition Language   A domain specific language for describing data structures

Wolframe glossary

Connection Handler   Interface for the networking to one client/server connection during its whole lifetime.

Command Handler   Interface for delegating processing of client protocol commands in a hierarchical way. A command handler is created by a connection handler or another command handler. During command execution the input/output of the connection is entirely handled by the command handler. Command Handlers are used to build the communication protocol processing as hierarchical state machine.

Network Input   Data passed from the networking framework to the connection handler through its channel for processing.

Network Output   Data passed back through the channel to the networking framework from the connection handler as output of processing.

Program   A set of named units of description of processing or data in a source file. The source file is loaded and interpreted at server startup. It exports some unique symbols to address single units in a program library. All units in this library are usable in any context of the logic tier.

Transaction   A transaction is a call of a database defined in a transaction program. A transaction either fails completely or succeeds as whole. Auditing is seen as part of the transaction. Transactions have an object as input and return an object or an error as result. Authorization tags that are checked against the user privileges of the connection can be attached to transactions.

Processor program   A program that is executed by a command handler. It works with data input, data output and transaction execution as the only interfaces to the system. Exceptions, signals and events are handled by the processor in the background without showing them to the program. A program in this context is sequential.

Lua   Lua (www.lua.org) is a scripting language. It is used in Wolframe as one language for writing programs. It distinguishes itself by being lightweight. It has to be lightweight because the language context has to be recreated with every start of a clients command execution for security reasons.

Filter   Filters are attached to network input and output to read and write input in a well defined format. Filters let you process input and print output in an iterative way. Filters are loaded by the system at startup and have a unique name.

Form   A form is a hierarchical description of typed data. Forms are used to create objects from a serialization and to validate input. Forms are defined in programs written in a DDL (Data Definition Language) or as declared as part of a build-in function API.

Direct Map Command Handler	The direct map command handler, also refered to as the standard comand handle is a command handler describing how server requts are executed. It describes requests by a form for the input, a form for the output, and a function called. The input of a direct map function is the object deserialization of the input defined in the input form. The result of the function is returned as serialization of the transaction result defined in the output form.
Channel	The flow for a single connection. Not all objects have channels (e.g. databases).
Group	A set of objects of the same type seen as one single object for the objects that use it.
Unit	An element of a group. A group is a set of units.
Provider	An entity providing objects of a kind. Some providers are factories, but not all of them (e.g. the database provider)
End Of Data (EoD)	Marks the end of the data to be processed by a processor. End of data is marked with CR LF dot ('.') CR LF or LF dot LF. For passing lines with a dot ('.') at the start of a content line, the client has to escape an LF dot in the content with LF dot dot. This escaping applies also to the result returned to the client. So client has to unescape LF dot sequences by replacing them by a LF.
End Of Message (EoM)	Marks the end of a network input message. End of message is handled by the processor without showing up to the processor program (yield execution).
Application Reference Path	The application server defines this file path where all relative paths defined in the configuration refer to.
Yield execution	The processor yields execution and gives control back to the connection handler, when it cannot continue execution. The connection handler has to resume with an action returned to the networking framework, that fetches the resources needed by the processor to continue. (see also [http://en.wikipedia.org/wiki/Coroutine Wikipedia yield in coroutine]). Lua is able to yield execution.

A

AAAA	Acronym for Authentication, Authorization, Accounting and Auditing See Also Authentication, Authorization, Accounting, Auditing.
Authentication	Some reasonable definition here. BTW, this just an example. See Also AAAA.
Authorization	Some reasonable definition here. BTW, this just an example. See Also AAAA.
Accounting	Some reasonable definition here. BTW, this just an example. See Also AAAA.
Auditing	Some reasonable definition here. BTW, this just an example. See Also AAAA.

S

SSL	Secure Sockets Layer
	Cryptographic protocols which provide secure communications on the Internet. SSL is a predecessor to TLS

See Also TLS.

T

TLS   Transport Layer Security

                                 Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer
                                 (SSL), are cryptographic protocols that provide communication security over
                                 the Internet. TLS and SSL encrypt the segments of network connections
                                 above the Transport Layer, using asymmetric cryptography for key exchange,
                                 symmetric encryption for privacy, and message authentication codes for
                                 message integrity. [Wikipedia]
                                 See Also SSL.

Index

Appendix A. Language Grammars

A.1. Grammars

.1. EBNF of the Transaction Definition Language

Tokens

[1]   Identifier **::=** (['A'-'Z'] | ['a'-'z'] | ['0'-'9'] | '_') +
[2]   FunctionId **::=** (['A'-'Z'] | ['a'-'z'] | ['0'-'9'] | '_' | '.') +
[3]   Number **::=** (['0'-'9']) +
[4]   String **::=** (['\''']([^'\''])+['\'''']+) | (['\'"']([^'\'"'])+['\'"']+)
[5]   IdList **::=** (Identifier) (| ',' IdList)
[6]   Selection **::=** (('/' | '//' |) (Identifier | '..' | '.') (Selection |))
[7] EmbeddedArg **::=** (String | '$' ((Identifier '.' |) Number) | ((Identifier
                        '.' |) Number) | '(' Selection ')'))
[8]   FuncArg **::=** (String | Selection | EmbeddedArg)
[9]   PreprcArg **::=** (String | Selection)

Statements

[10]   Destination **::=** 'INTO' Identifier
[11]   Foreach **::=** 'FOREACH' Selection
[12] EmbeddedOp **::=** SQLStatement
[13]   FuncArgLst **::=** (FuncArg ',' FuncArgLst | FuncArg)
[14]   NamedOp **::=** Identifier '(' (FuncArgLst |) ')'
[15]   Action **::=** 'DO' (EmbeddedOp | NamedOp)
[16]   Statement **::=** ((Destination |) (Foreach |) Action ';') | ('KEEP'
                        'AS' Identifier ';')

Pre-Processing Commands

[17] PreprcArgLst **::=** (PreprcArg ',' PreprcArgLst | PreprcArg)
[18]   PreprcOp **::=** FunctionId '(' (PreprcArgLst |) ')'
[19]   PreprcCall **::=** 'DO' (PreprcOp)
[20]   PreprcStm **::=** (Destination |) (Foreach |) PreprcCall ';'

Transactions

[21] PreprcStmLst **::=** (PreprcStm | PreprcStm PreprcStmLst)
[22]   PreprcBody **::=** 'PREPROC' PreprcStmLst 'ENDPROC'
[23] StatementLst **::=** (Statement | Statement StatementLst)
[24]   OpBody **::=** 'BEGIN' StatementLst 'END'
[25]   Subroutine **::=** 'SUBROUTINE' Identifier ['DATABASE' IdList]
                        ['RESULT' 'INTO' Identifier] OpBody
[26]   Transaction **::=** 'TRANSACTION' Identifier ['AUTHORIZE' '('
                        Identifier ',' Identifier ')'] [PreprcBody]
                        ['DATABASE' IdList] ['RESULT' 'INTO' Identifier
                        ] OpBody
[27]   Program **::=** (Transaction | Subroutine) (Program |)

.2. EBNF of the Normalizer Function Declarations

Tokens

[28]   Identifier **::=** (['A'-'Z'] | ['a'-'z'] | ['0'-'9'] | '_') +
[29]   Number **::=** (['0'-'9']) +

Declarations

[30]   Namespace **: :** =Identifier ':'
[31]   Argument **: :** = (Identifier | Number)
[32]ArgumentList **: :** = ((Argument ',' ArgumentList) | Argument)
[33]   Call **: :** = ((Namespace Identifier) | Identifier) ['('
                   ArgumentList ')']
[34]   CallSeq **: :** = ((Call ',' CallSeq) | Call)
[35]   Typedef **: :** = Identifier '=' CallSeq ';'
[36]   Program **: :** = ((Typedef Program) | Typedef)

Appendix B. GNU General Public License version 3

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. http://fsf.org/

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and

dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

a. The work must carry prominent notices stating that you modified it, and giving a relevant date.

b. The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

c. You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d. If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c. Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d. Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e. Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a. Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b. Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c. Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d. Limiting the use for publicity purposes of names of licensors or authors of the material; or

e. Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f. Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY
APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT
HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT
WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE
OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU
ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR
CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES,
INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES
ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT
NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES
SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE
WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN
ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal
effect according to their terms, reviewing courts shall apply local law that most closely approximates
an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption
of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the
best way to achieve this is to make it free software which everyone can redistribute and change under
these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each
source file to most effectively state the exclusion of warranty; and each file should have at least the
"copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year name of author

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.


You should have received a copy of the GNU General Public License
along with this program.  If not, see http://www.gnu.org/licenses/.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
program Copyright (C) year name of author
This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see http://www.gnu.org/licenses/.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read http://www.gnu.org/philosophy/why-not-lgpl.html.

Wolframe Clients

Clients to access Wolframe

Wolframe Clients: Clients to access Wolframe

Publication date April 27, 2014 version 0.0.4
Copyright © 2010 - 2014 Project Wolframe

Table of Contents

List of Tables

Chapter 1. Introduction

This part of the manual describes how the user interface part (presentation tier, also called client) of Wolframe applications can be built.

A Wolframe client can be of various kinds. They all communicate with the server over a text based protocol in a plain or encrypted session. All methods used are based on open standards.

We will introduce two examples of clients: The Wolframe standard client and a web client communicating via a web server with the Wolframe application server.

After reading this chapter you should be able to create a Wolframe client based of one of these two examples on your own.

Chapter 2. The Wolframe Standard Client

This chapter describes the standard Wolframe client called wolfclient and how a user interface is built.

2.1. Architecture

The Wolframe standard client wolfclient is a thin client which executes XML requests via the Wolframe protocol and presents XML answers. It is written in Qt and is cross-platform. Qt is currently available on http://doc.qt.digia.com/qt/index.html. User interfaces for wolfclient are defined as a set of forms using standard Qt widgets and are if ever possible defined using the Qt Interface Designer (see http://qt-project.org/doc/qt-4.8/designer-manual.html).

2.2. Artifacts

The wolfclient renders user interface forms dynamically, this means no code generation or compilation is involved when creating user interfaces for Wolframe.

2.2.1. UI forms

The UI files follow the schema 'qt-ui-4.7.xsd', as documented in http://qt-project.org/doc/qt-4.8/designer-ui-file-format.html. The UI files have the extension *.ui*

UI files are created and edited with the Qt designer.

2.2.2. UI form translations

The wolfclient uses the Qt translation format, version 2.0 for form translations as described in *http://qt-project.org/doc/qt-4.8/linguist-ts-file-format.htmll* [http://qt-project.org/doc/qt-4.8/linguist-ts-file-format.html]. Those are the files with extension *.ts*.

The translation files can get merged and generated with the *lupdate* tool, then translated with the *Qt Linguist*.

The Qt client needs the files in compiled form as files with the extension *.qm*. The *lupdate* tool is taking care of that.

Read more on translations in http://qt-project.org/doc/qt-4.8/linguist-manual.html.

2.2.3. Resources

Binary resource files contain images for the user interface.

Binary resource files (extension *.rss*) are compiled from a XML file (extension *.qrc*) with the *rcc* resource compiler.

2.3. Programming the Interface

Programming means we annotate the XML of the UI form files with some extra properties. They control the following things:

- Which events in the current form replace it with a new form, e. g. clicking the *Edit* button loads the form called *edit_item*.

- When and how requests to the Wolframe server should be sent and how the results should be interpreted when adding data to the widgets, e.g. executing a *save item request* with all the data in the text fields of the form added to the request XML.

2.3.1. Mapping XML Data

Starting Position

For mapping data structures from the user interface elements to the data description needed to fulfill an interface for a server request we need some kind of translation. An implicit mapping would only be able to describe very trivial data mappings. After drawing the user interface this translation has to be defined. On the other hand the requests answer returned by the server has to be mapped to be shown in the user interface elements view. Here applies the same: Some kind of translation is needed to map a server data structure to the user interface elements.

First Example

Let's have a look at a `QLineEdit` element of a form and a possible XML representation of the data used for a request.


```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE customer SYSTEM 'Customer'>
<customer>
  <name>John Smith</name>
  <address>Blue Police Box</address>
</customer>
```

For an insert or update request that transmits all data of the form to the server we have to fill the name field and the address field into the request data structure XML. The translation is defined as dynamic property "action" or "action." plus a suffix for the action identifier if needed. We will explain this naming of actions later. The value of the property is describing the request and could look as follows:
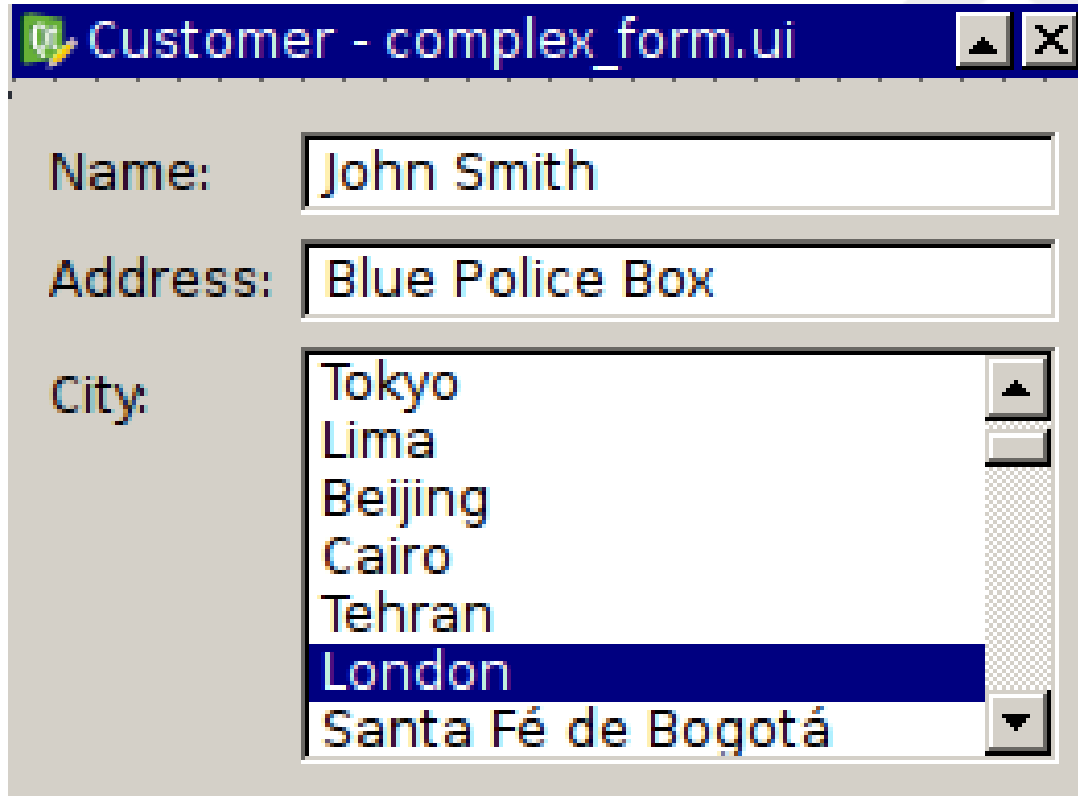
```
update: Customer customer {name{{main.name}}; address{{main.address}}}
```

For the initial filling of the form with data we submit a request that just sends an id to the server. The answer that is returned by the server has then to be translated to fill the name field and the address field of the form. The translation is defined as dynamic property "answer" or "answer." plus a suffix for the action identifier. A detailed description of the language in the request and answer property value that describes requests and answers will presented in the next chapter. We provide here just an example:

```
Customer customer {name{{main.name}}; address{{main.address}}}
```

Another Example

Some elements are more complicated than that. They present the user a list of options or items the user to pick from, e.g. a list of cities.

When the form is saved, the currently selected element is written into the resulting XML:

```
<customer>
  <name>John Smith</name>
  <address>Blue Police Box</address>
  <city>6</city>
</customer>
```

In this case the widget with the city list can load its own domain data as a separate XML request:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE city SYSTEM 'CityListRequest'>
<cities/>
```

and the corresponding domain load request answer definition in the dynamic property "answer" could look like this:

```
CityList cities {city[] {id={main.city.id}; {main.city.value}}}
```

The answer contains all possible values in the domain, in our case a list of all cities and their internal id.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE "cities" SYSTEM "CityList">
<cities>
<city id='1'>Tokyo</city>
<city id='2'>Lima</city>
<city id='3'>Beijing</city>
<city id='4'>Cairo</city>
<city id='5'>Tehran</city>
<city id='6'>London</city>
</cities>
```

2.3.2. Switching UI forms

A UI form contains a set of widgets, the dynamic property `form` contains the name of a widget (without extension *.ui*) to load.

For linking a push `QPushButton` click in the Qt designer to the switching of the form you have to attach a dynamic property named `form` of type `string` to the corresponding widget of type `QPushButton`:
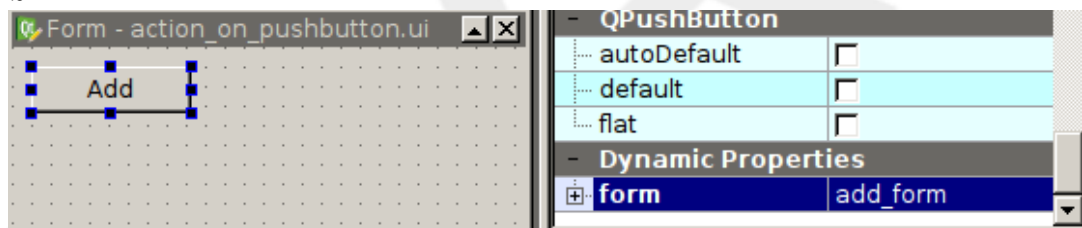
Before loading the next form the client terminates all current requests, for instance a save request of the form data. In case of an error in an action any defined switching of the form is cancelled.

2.3.3. States and Behaviour

Reserved Private Dynamic Properties

The _w_ prefix is used for internal widget properties not of interest for the user.

Reserved Public Dynamic Properties

The dynamic properties introduced here are edited by the user to stear application behaviour:

Stearing of Widget Behaviour

The following properties are reserved for states steering the behavior of the user interface:

Table 2.1. Properties

Name	Description
initialFocus	Boolean value for one widget in a form that should get the initial keyboard focus.

User Interface Flow

The following properties stear the user interface elements flow:

Table 2.2. Properties

Name	Description
form	Defines a form to be opened on click (push button). If the widget has an action defined, then the action is executed before and the form is opened when the action succeeds and not opened when it fails.
form:IDENTIFIER	Defines a form related to a context menu entry with identifier IDENTIFIER. If the context menu entry has also an action defined, then the action is executed before. The form is opened only if the action succeeds.

Additional Interface Elements

The following properties define additional interface elements:

Table 2.3. Properties

Name	Description
contextmenu	Defines a context menu with a comma separated list of identifiers of actions defined as propery value. Two following commas without menu entry identifier are used to define a separator.
contextmenu:NAME	Defines the (translatable) text of a context menu entry. NAME refers to a non empty name in the list of context menu entries.

Defining Server Request/Answer

The following properties are used for the communication with the server:

Table 2.4. Properties

Name	Description
action	Defines a server request. This can either be a load action request for a widget that is not a push button or an action request without answer than OK/ ERROR for a push button
action:IDENTIFIER	Defines an action request either related to a context menu entry (when clicked) or related to a dataslot declaration of this widget named with IDENTIFIER.
dropmove	Defines a action request that is issued on a drop request moving an object inside a widget or between widgets of the same type (same object name). The request is an action request without other answer than success or failure. Refresh after the action completed is triggered via a datasignal

Name	Description
	'datasignal:drop' defined in the drop widget and a 'datasignal:drag' defined in the drag widget.
dropmove:OBJECTNAME	Defines a server request that is issued on a drop request moving an object from a widget with object name OBJECTNAME. The request is an action request without other answer than success or failure. Refresh after the action completed is triggered via a datasignal 'datasignal:drop' defined in the drop widget and a 'datasignal:drag' defined in the drag widget.
dropcopy	Defines a action request that is issued on a drop request copying an object inside a widget or between widgets of the same type (same object name). The kind of request and the signaling after completion is the same for a 'dropmove' action.
dropcopy:OBJECTNAME	Defines a server request that is issued on a drop request copying an object from a widget with object name OBJECTNAME. The kind of request and the signaling after completion is the same for a 'dropmove:OBJECTNAME' action.
answer	Defines the format of the action request answer linked to the widget activation (for example a click on a push button).
answer:IDENTIFIER	Defines the format of the request answer of the action defined as 'action:IDENTIFIER'

Variables and Symbolic Links

Table 2.5. Properties

Name	Description
global:IDENTIFIER	Defines an assignment from a global variable IDENTIFIER at initialization and writing the global variable when closing the widget.
assign:PROP	Defines an assingment of property PROP to the property defined as value "assign:PROP" on data load and refresh
link:IDENTIFIER	Defines a symbolic link to another widget. Defining the property "link:<name>" = <widgetid>: defines <name> to be a reference to the widget with the widgetid set to <widgetid>. Links are used to read data from other widgets on load and refresh.
widgetid	Unique identifier of the widget used for identifying it when resolving symbolic links or an address of a request aswer. When not explicitely defined it is implicitely defined as unique identifier on widget creation. Unique means unique during one run of one client. It's a simple counter plus the name of the widget.
synonym:NAME	Defines a renaming of the identifier NAME to the identifier in the property value. Be careful when

Name	Description
	using synonyms. They are the last construct you should consider to use in the client.

Widget States Depending on Data

Table 2.6. Properties

Name	Description
state:IDENFITIER	Defines a state of the widget dependent on a condition. IDENTIFIER is one of 'enabled', 'disabled', 'hidden', 'visible'. The state condition is defined the property value. The value can be a property reference in '{' '}' brackets. The condition is true when the property is defined. A condition can also be a boolean expression of the form <prop> <op> <value>, where <prop> is a property reference in '{' '}' brackets, <op> an operator and <value> a constant value Valid operators are: '==' (string),'!=' (string),'<=' (integer), '<' (integer) ,'>=' (integer), '>' (integer) For 'action' definitions the state 'state:enabled' is dependent on the properties referenced in the 'action' value.

Additional Signals and Slots

Table 2.7. Properties

Name	Description
datasignal:IDENTIFIER	Defines a signal of type IDENTIFIER (clicked, doubleclicked, destroyed, signaled, loaded, drag, drop) with the slot name and destination address defined as property value of "datasignal:IDENTIFIER" Datasignal destinations can be defined as follows: As widgetid, as slot identifier (declared with 'dataslot'), as widget path. A preceding identifier followed by '@' specifies what to do with the widget of the target slot. If you specify 'close' there in a form top level widget then the form is closed. Every other identifier causes a reload of the widget.
dataslot	Defines a comma separated list of slots for the signal of with the property value as slot identifer and optionally followed by a widget id in '(..)' brackets that sepcifies a sender from where the signal is accepted.

Drag and Drop

Drag and Drop events are defined with the properties 'dropmove' and 'dropcopy' that define the action requests issued on a drop event. See description of the properties in "Defining Server Request/ Answer". For using drag and drop the property 'acceptDrops' has to be enabled and the Widget has to be capable to do drag and drop. Drag and drop is currently only possible for the Qt standard list widgets, tree widgets and table widgets or for user defined widgets that delegate the mouse events

accordingly. We do not describe here how user defined widgets can implement this mechanism of drag and drop.

What happens when an object is dragged from one object and dropped at another object is a request sent to the server. To address the elements involved in drag and drop some variables are set before issueing the request. These Variables can therefore be used in the request to specify the operation to implement the drag and drop. One of these variables is a widget link 'dragobj' that points the origin widget of the drag. With {dragobj.selected} we can address the item or set of items selected with the drag. The other variable is 'dropid' that selects the value or id of the target widget of the drop. What this value means is dependent on the widget class.

Besides the 'dropmove' and 'dropcopy' there are the datasignal properties 'datasignal:drag' and 'datasignal:drop' that can be used to specify the needed widget refresh signals that have to be performed after the drag and drop operation.

2.3.4. Widget properties as dynamic property values

Dynamic properties can reference properties of widgets like for example `property = {variable expression}`.

The expression can reference addressable widgets and their properties. Every Qt class has its very own set of properties it understands.

2.4. Programming Server Requests/Answers

2.4.1. Adressing Widget Data

Widget data elements elements are addressed by using the relative path of the element from the widget where the request or answer was specified. The relative path is a sequence of widget object names separated by dots ('.'). Only atomic element references are specified in request/answer structure.

Biggest Common Ancestor Path

The grouping of elements into structures is done by the biggest common ancestor path of all atomic element references in a structure. It is assumed that this biggest common ancestor is addressing the structure. If for example a structure has the atomic widget element references "home.user.name" and "home.user.id" then we assume that "home.user" is adressing the structure containing "name" and "id" in the widget data.

Addressing Atomic Elements

Table 2.8. Basic Elements of Request/Answer

Description	Syntax
Constant (server request only)	string with single (') or double (") quotes or numeric integral constant
Mandatory attribute	name={variablepath}
Mandatory content value	name{{variablepath}}
Optional attribute	name={variablepath:?}
Optional content value	name{{variablepath:?}}
Optional attribute with default	name={variablepath:default}
Optional content with default value	name{{variablepath:default}}
Ignored attribute	name={?}

Description	Syntax
Ignored content value	name{{?}}
Ignored sub structure	name{?}

Special Path Elements

Variable references can address other widgets than sub widgets of the current widget.

Addressing the Form Widget

The reserved path element 'main' addresses the form widget root.

Widget Links

A dynamic property with the prefix 'link:' followed by an identifier as name declares the widget with the widget id as dynamic property value of the link definition to be referencable by name. The referencing name is the identifier after the prefix 'link:'. So if we for example define a dynamic property 'link:myform' with a widget id as value, then we can use the variable 'myform' in a widget path to address the widget.

The mechanism of widget links is mainly used for implementing form/sub-form relationships. A form opens a subform and passes its widget id to it with the form parameter 'widgetid=..'. A link is defined to the sub-form with the widget id passed to it. The subform signals some action to the parent that can address the data entered in the subform via this link.

2.4.2. Data Structures

Structure elements are separated by semicolon ';' and put into '{' brackets '}' with the name of the structure in front.

Example

The following example shows an address as structure:

```
address{
  tag=1;
  surname{{person.surname}};
  prename{{person.prename}};
  street{{address.street}}
}
```

2.4.3. Arrays

Arrays are marked with opened and closed square brackets '[' ']' without specifying dimension (arbitrary size or empty when missing).

Description

Table 2.9. Types of Arrays

Description	Syntax
Arbitrary Size Array of Content Values	name[]{{variablepath}}
Arbitrary Size Array of Structures	name[]{structure definition}

Example

The following example shows an array of addresses:

```
address[]{
surname{{address.surname}};
prename{{address.prename}};
street{{address.street}}
}
```

The widget element paths used to address the widget elements have to have a common ancestor path. In our example this would be 'address'. The common ancestor path is determining how elements are grouped together in the widget. It tells what belongs together to the same array element in the widget. Without common common ancestor path it would be impossible to determine what is forming a structure in the widget data. It distinguishes the case of having an array of adresses and the case of having an array of surnames, and array of prenames and an array of streets. The later makes not much sense here. With the common prefix we state how entities are grouped together to structures in the representation in the widget.

2.4.4. Indirection and Recursion

Indirection allows to define recursive structures. Indirection means that an element is specified as reference that is expanded when the element appears in the data structure to map. The grouping element of the indirection elements is the common ancestor of all non indirection elements in the structure containing the indirection.

Description

Table 2.10. Types of Indirections

Description	Syntax (name equals ancestor name)	Syntax (name differs from ancestor name)
Single Element Indirection	^ancestor	^item:ancestor
Multiple Indirection	^ancestor[]	^item:ancestor[]

Example (Arbitrary Tree)

Example representing a tree with arbitrary number of children per node:

```
item{
  id={treewidget.id};
  name{{treewidget.name}};
  ^item[]
}
```

Example (Binary Tree)

Example representing a binary tree:

```
item{
  id={treewidget.id};
  name{{treewidget.name}};
  ^left:item;
  ^right:item
}
```

2.5. Eliminating Interface Defects

Functional defects in the user interface like for example syntax errors in the definitions of the request answer can be eliminated by inspecting the error messages reported by the wolfclient in developer mode and fixing the interface accordingly.

2.5.1. Switch the Developer Mode On

In order to inspect the internals of your client program, we have first to switch on "Developer Mode" in the "Developer" context of the "Preferences Dialog". The following picture emphasizes the check box you have to enable (highlighted green).
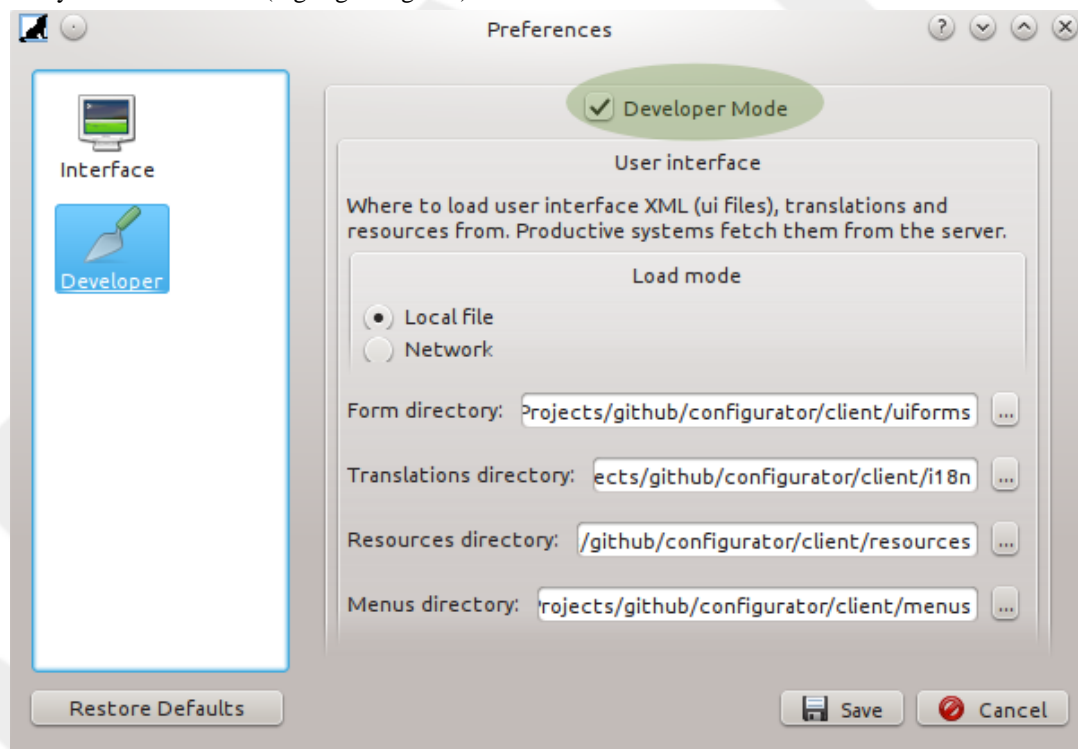
2.5.2. Inspect Errors and Warnings and Debug Messages Reported

To inspect internal messages reported by the wolfclient in developer mode we have to open the debug window. The debug window is opened by clicking on the bug icon in the main tool bar or via the developer context menu. The following picture shows an example debug output. Each action we do from now on with the debug window opened can be followed on the level of messages it emits.

We can see the messages in the message list when clicking on the "Refresh" button. The navigation allows us to restrict our focus on messages on a node in the object tree by clicking on it. Clicking

on the root node shows all messages in the recent history. The history starts with the last main node created before opening the debug window. All message restrictions show the messages in order of their emission. We can restrict also on the severity of messages in the severity level selection (the select box set to "Debug" as default left of the "Refresh" button).

The "Clear" button allows us to empty the recent history without closing the debug window.
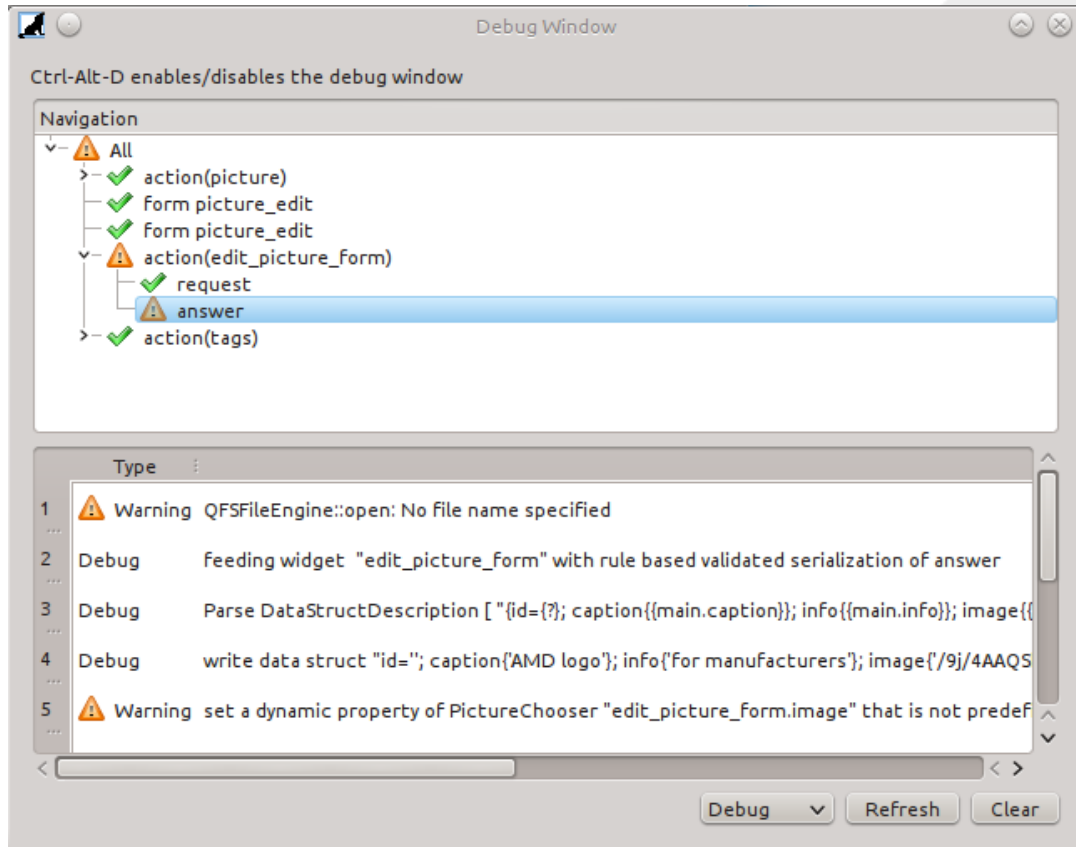
Index

Wolframe Server Extension Modules

Write your own modules in C++

Wolframe Server Extension Modules: Write your own modules in C++

Publication date April 27, 2014 version 0.0.4
Copyright © 2010 - 2014 Project Wolframe

Commercial Usage.   Licensees holding valid Project Wolframe Commercial licenses may use this file in accordance with the Project Wolframe Commercial License Agreement provided with the Software or, alternatively, in accordance with the terms contained in a written agreement between the licensee and Project Wolframe.

GNU General Public License Usage.   Alternatively, you can redistribute this file and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

Wolframe is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with Wolframe. If not, see http://www.gnu.org/licenses/

If you have questions regarding the use of this file, please contact Project Wolframe.

Table of Contents

List of Tables

Foreword

This manual introduces the extension modules of Wolframe and explains how to build them. After reading this you should be able to write Wolframe extension modules on your own.

Chapter 1. Introduction

First we introduce the basic C++ data structures you have to understand in order to develop your own Wolframe modules. Later we will introduce the different module types and the building blocks used to build them.

Chapter 2. Basic Data Types

In this chapter we give a survey of the basic data types used in the Wolframe module interfaces.

2.1. Variant Type

The variant data type describes an atomic value of any scalar or string type. It is the basic type for interfaces to all language bindings for writing Wolframe applications. The type `Variant` is defined in `types/variant.hpp` and has the following interface:

```
namespace _Wolframe {
namespace types {

class Variant
{
public:
    //Different value types a variant can have:
    enum Type
    {
        Custom,       //< data type defined by a custom data type module
        Timestamp,    //< date and time value with a precision down to microseconα
        BigNumber,    //< big BCD fixed point number in the range of 1E-32767 to 1
        Double,       //< IEEE 754 double precision floating point number
        Int,          //< 64 bit signed integer value
        UInt,         //< 64 bit unsigned integer value
        Bool,         //< boolean value
        String        //< 0-terminated UTF-8 string
    };
    //Current type enum or type name of this:
    Type type() const;
    const char* typeName() const;

    //Null constructor:
    Variant();

    //Copy constructors:
    Variant( bool o);
    Variant( double o);
    Variant( float o);
    Variant( int o);
    Variant( unsigned int o);
    Variant( Data::Int o);
    Variant( Data::UInt o);
    Variant( const char* o);
    Variant( const char* o, std::size_t n);
    Variant( const std::string& o);
    Variant( const Variant& o);
    Variant( const types::CustomDataType* typ,
             const types::CustomDataInitializer* dsc=0);
    Variant( const types::CustomDataValue& o);
    Variant( const types::DateTime& o);
    Variant( const types::BigNumber& o);

    //Assignment operators:
    Variant& operator=( const Variant& o);
```

```cpp
Variant& operator=( bool o);
Variant& operator=( double o);
Variant& operator=( float o);
Variant& operator=( int o);
Variant& operator=( unsigned int o);
Variant& operator=( Data::Int o);
Variant& operator=( Data::UInt o);
Variant& operator=( const char* o);
Variant& operator=( const std::string& o);
Variant& operator=( const types::CustomDataValue& o);
Variant& operator=( const char* o);
Variant& operator=( const types::DateTime& o);
Variant& operator=( const types::BigNumber& o);

//Initializer as constant (borrowed value reference):
void initConstant( const char* o, std::size_t l);
void initConstant( const std::string& o);
void initConstant( const char* o);

//Comparison operators:
bool operator==( const Variant& o) const;
bool operator!=( const Variant& o) const;
bool operator>( const Variant& o) const;
bool operator>=( const Variant& o) const;
bool operator<=( const Variant& o) const;
bool operator<( const Variant& o) const;

//Getter functions with value conversion if needed:
std::string tostring() const;
std::wstring towstring() const;
double todouble() const;
bool tobool() const;
Data::Int toint() const;
Data::UInt touint() const;
Data::Timestamp totimestamp() const;

//Base pointer in case of a string (throws if not string):
char* charptr() const;
//Size in case of a string (throws if not string):
std::size_t charsize() const;
///\brief Get the pointer to the custom data object (throws for non custom
const CustomDataValue* customref() const;
///\brief Get the pointer to the custom data object (throws for non custom
CustomDataValue* customref();
///\brief Get the pointer to the big number object (throws for non big numbe
const types::BigNumber* bignumref() const;
///\brief Get the pointer to the big number object (throws for non big numbe
types::BigNumber* bignumref();

///\brief Getter with value conversion
Data::Timestamp totimestamp() const;

///\brief Test if this value is atomic (not a structure or an indirection)
bool atomic() const;
//Evaluate if defined (not Null):
bool defined() const;
//Reset to Null:
void clear();
```

```
    //Convert type:
    void convert( Type type_);
    //Move assignment from value o (o gets Null):
    void move( Variant& o);
    ///\brief Assigning o to this including a conversion to a defined type
    void assign( Type type_, const Variant& o);
};

}} //namespace
```

Certain interfaces like filters use the type `VariantConst` that is the same as a variant but does not hold ownership on the value it references. `VariantConst` is defined to avoid unnecessary string copies mainly in filters. It inherits the properties of the type `Variant` and adds or overwrites some methods. `VariantConst` has to be used carefully because we have to ensure on our own that the referenced value exists as long as the `VariantConst` variable exists. The mechanisms of C++ do not support you here. You have to know what you do. The type `VariantConst` is also defined in `types/variant.hpp` and has the following interface:

```
namespace _Wolframe {
namespace types {

struct VariantConst :public Variant
{
    //Null constructor:
    VariantConst();
    //Copy constructors:
    VariantConst( const Variant& o);
    VariantConst( const VariantConst& o);
    VariantConst( bool o);
    VariantConst( double o);
    VariantConst( float o);
    VariantConst( int o);
    VariantConst( unsigned int o);
    VariantConst( Data::Int o);
    VariantConst( Data::UInt o);
    VariantConst( const char* o);
    VariantConst( const char* o, std::size_t n);
    VariantConst( const std::string& o);
    VariantConst( const types::CustomDataValue& o);
    VariantConst( const types::BigNumber& o);
    VariantConst( const types::DateTime& o);

    //Assignment operators:
    VariantConst& operator=( const Variant& o);
    VariantConst& operator=( const VariantConst& o);
    VariantConst& operator=( bool o);
    VariantConst& operator=( double o);
    VariantConst& operator=( float o);
    VariantConst& operator=( int o);
    VariantConst& operator=( unsigned int o);
    VariantConst& operator=( Data::Int o);
    VariantConst& operator=( Data::UInt o);
```

```
        VariantConst& operator=( const char* o);
        VariantConst& operator=( const std::string& o);
        VariantConst& operator=( const types::CustomDataValue& o);
        VariantConst& operator=( const types::BigNumber& o);
        VariantConst& operator=( const types::DateTime& o);
        VariantConst& operator=( const char* o);
        VariantConst& operator=( const std::string& o);
    };
    }} //namespace
```

```
        VariantConst& operator=( const char* o);
        VariantConst& operator=( const std::string& o);
```

Chapter 3. Module Declaration

In this chapter we introduce how modules are declared for extending a Wolframe application with our own functions and objects.

3.1. Module Declaration Frame

A module has to include "appdevel/moduleFrameMacros.hpp" or simply "appDevel.hpp" and declare a module header and a module trailer with the macros `WF_MODULE_BEGIN` and `WF_MODULE_END`. A single module source file built as Wolframe application extension module must contain only one `WF_MODULE_BEGIN` .... `WF_MODULE_END` declaration. But a module declaration can contain an arbitrary number of objects that not conflicting in anything they define (names, etc.) The following example shows an empty module without any exported objects, thus the simplest module we can declare.

3.1.1. Empty Module Declaration Example

```
#include "appDevel.hpp"
WF_MODULE_BEGIN( "empty", "an example module not exporting anything")
WF_MODULE_END
```

3.1.2. Module Declaration Macros

The macro WF_MODULE_BEGIN has two parameters

Table 3.1. Parameters of WF_MODULE_BEGIN

NAME	DESCRIPTION
Identifier of the module (*)	Description sentence of the module for user info when inspecting a module.

(*) Currently not used but will be when a namespace concept will be implemented.

The end declaration WF_MODULE_END closes the module object and defines the module entry point structure.

3.2. Building a Module

For building a module you need to reference the Wolframe core library (-lwolframe) and eventually some of the extension libraries (-lwolframe_serialize, -lwolframe_langbind, -lwolframe_database) that's all. You will find example makefiles in the examples of the project. But you are free to use your own build mechanism.

3.3. Exported Objects of a Module

In this section we explain how modules are filled with functionality. We can define an arbitrary number of objects in a module as long as they do not conflict (e.g. have name clashes etc.)

3.3.1. Define Normalization Functions (Normalizers)

In this chapter we introduce how to declare a normalizer function in a module for defining your own DLL form data types. First we introduce the data structures you have to know to implement normalizer

functions and then we will show the module building blocks to declare a normalizer function in a module.

Normalizer Interface

A normalize function is defined as interface in order to be able to define it as object with data. This is because normalizer functions can be parametrized. For example to express the normalize function domain. The following listing shows the interface definition:

```
namespace _Wolframe {
namespace types {

struct NormalizeFunction
{
    virtual ~NormalizeFunction(){}
    virtual const char* name() const=0;
    virtual Variant execute( const Variant& i) const=0;
};
}}
```

The object is created by a function type (here with the example function name `CreateNormalizeFunction`) with the following interface

```
_Wolframe::types::NormalizeFunction* CreateNormalizeFunction(
        _Wolframe::types::NormalizeResourceHandle* reshnd,
        const std::vector<types::Variant>& arg);
```

The resource handle parameter (`reshnd`) is the module singleton object instance that is declared as class in the module building blocks (see following section). The argument (`arg`) is a list of variant type arguments that parametrize the function. What the function gets as arguments are the comma separated list of parameters in '(' brackets ')' when the function is referenced in a `.wnmp` file (type normalization declaration file, see section "Data Types in DDLs" in the chapter "Forms" of the "Application Development Manual") or constructed with the provider.type method in a script.

Building Blocks

When you include "appdevel/normalizeModuleMacros.hpp" or simply "appDevel.hpp" you get the building blocks declared to build a normalizer function in a module. These building blocks will be exmplained in this section.

Declaring a resource singleton object

Some normalizer functions share resource object declared only once as a singleton in this module. Such a resource class is defined as a class derived from `types::NormalizeResourceHandle` with an empty constructor. When we have declared this resource signleton class we can include it in the module before any normalizer referencing it as

```
WF_NORMALIZER_RESOURCE( ResourceClass )
```

with `ResourceClass` identifying the module singleton resource class and object.

Declaring a normalizer not using any resource

The following declaration shows a declaration of a simple normalizer function.

```
WF_NORMALIZER_FUNCTION(name,constructor)
```

where name is the identifier string of the function in the system and constructor a function with the signature of the `CreateNormalizeFunction` shown in the section 'Normalize Interface' above.

Declaring a normalizer using a resource

The following declaration shows a declaration of a normalizer function using a resource module singleton object defined as class 'ResourceClass' and declared with the `WF_NORMALIZER_RESOURCE` macro (section 'Declaring a resource singleton object').

```
WF_NORMALIZER_WITH_RESOURCE(name,constructor,ResourceClass)
```

The parameter `name` and `constructor` are defined as in the `WF_NORMALIZER_FUNCTION` macro.

Examples

Example without resources

As first example we show a module that implements 2 normalization functions `Int` and `Float` without a global resource class. `Int` converts a value to an 64 bit integer or throws an exception, if this is not possible. `Float` converts a value to a double presicion floating point number or throws an exception, if this is not possible.

```cpp
#include "appDevel.hpp"

using namespace _Wolframe;

class NormalizeInt
    :public types::NormalizeFunction
{
public:
    NormalizeInt(){}
    virtual ~NormalizeInt(){}
    virtual const char* name() const
            {return "int";}
    virtual types::Variant execute( const types::Variant& i) const
            {return types::Variant( i.toint());}
    virtual types::NormalizeFunction* copy() const
            {return new NormalizeInt(*this);}

    static types::NormalizeFunction* create(
            types::NormalizeResourceHandle*,
            const std::vector<types::Variant>&)
    {
        return new NormalizeInt();
    }
```

```
    };

    class NormalizeFloat
        :public types::NormalizeFunction
    {
    public:
        NormalizeFloat(){}
        virtual ~NormalizeFloat(){}
        virtual const char* name() const
                {return "float";}
        virtual types::Variant execute( const types::Variant& i) const
                {return types::Variant( i.todouble());}
        virtual types::NormalizeFunction* copy() const
                {return new NormalizeFloat(*this);}

        static types::NormalizeFunction* create(
                types::NormalizeResourceHandle*,
                const std::vector<types::Variant>&)
        {
            return new NormalizeFloat();
        }
    };

    WF_MODULE_BEGIN(
        "example1",
        "normalizer module without resources")

        WF_NORMALIZER( "int", NormalizeInt::create)
        WF_NORMALIZER( "float", NormalizeFloat::create)

    WF_MODULE_END
```

Example with resources

The second example show one of the functions in the example above (Int) but declares to use resources. The resource object is not really used, but you see in the example how it gets bound to the function that uses it.

```
#include "appDevel.hpp"

using namespace _Wolframe;

class ConversionResources
    :public types::NormalizeResourceHandle
{
public:
    ConversionResources()
    {}
    virtual ~ConversionResources()
    {}
};

class NormalizeInt
```

```
                    :public types::NormalizeFunction
{
public:
    explicit NormalizeInt( const ConversionResources* res_)
        :res(res_){}
    virtual ~NormalizeInt()
        {}
    virtual const char* name() const
        {return "int";}
    virtual types::Variant execute( const types::Variant& i) const
        {return types::Variant( i.toint());}
    virtual types::NormalizeFunction* copy() const
        {return new NormalizeInt(*this);}

    static types::NormalizeFunction* create(
        types::NormalizeResourceHandle* reshnd,
        const std::vector<types::Variant>&)
    {
        ConversionResources* res
            = dynamic_cast<ConversionResources*>(reshnd);
        return new NormalizeInt( res);
    }
private:
    const ConversionResources* res;
};

WF_MODULE_BEGIN(
    "example2",
    "normalizer module with resources")

    WF_NORMALIZER_RESOURCE( ConversionResources)
    WF_NORMALIZER_WITH_RESOURCE(
        "Int", NormalizeInt::create, ConversionResources)
WF_MODULE_END
```

3.3.2. Define Custom Data Types

In this chapter we introduce how to declare a custom data type in a module. Custom data types can be used in scripting language bindings and as normalizers referenced in a `.wnmp` file (type normalization declaration file, see section "Data Types in DDLs" in the chapter "Forms" of the "Application Development Manual") First we introduce the data structures you have to know to implement a custom data type and then we will show the module building block to declare a custom data type in a module.

Custom Data Type Interface

A custom data type definition involves 3 classes: CustomDataType,CustomDataValue and CustomDataInitializer. The CustomDataInitializer class is optional and only needed when value construction has to be parametrized. If an initializer is involved then it is created and passed as argument to the method constructing the custom data type value (class CustomDataValue). The class CustomDataType defines the custom data type and all its methods defined. The class CustomDataValue defines a value instance of this type. The class CustomDataInitializer, if specified, defines an object describing the parametrization of the value construction. An example of an initializer

could be the format of a date or the precision in a fixed point number. The following listings show these interfaces:

CustomDataType Structure

The class to build the custom data type definition structure composed of methods added with `CustomDataType::define( .. )`. From this class we do not derive. We incrementally add method by method by calling `CustomDataType::define( .. )` in the type constructor function.

```cpp
namespace _Wolframe {
namespace types {

// Custom Data Type Definition
class CustomDataType
{
public:
    typedef unsigned int ID;
    enum UnaryOperatorType {Increment,Decrement,Negation};
    enum BinaryOperatorType {Add,Subtract,Multiply,Divide,Power,Concat};
    enum ConversionOperatorType {ToString,ToInt,ToUInt,ToDouble,ToTimestamp};
    enum DimensionOperatorType {Length};

    typedef types::Variant (*ConversionOperator)(
        const CustomDataValue& operand);
    typedef types::Variant (*UnaryOperator)(
        const CustomDataValue& operand);
    typedef types::Variant (*BinaryOperator)(
        const CustomDataValue& operand, const Variant& arg);
    typedef std::size_t (*DimensionOperator)(
        const CustomDataValue& arg);
    typedef types::Variant (*CustomDataValueMethod)(
        const CustomDataValue& val,
        const std::vector<types::Variant>& arg);
    typedef CustomDataValue* (*CustomDataValueConstructor)(
        const CustomDataInitializer* initializer);
    typedef CustomDataInitializer* (*CreateCustomDataInitializer)(
        const std::vector<types::Variant>& arg);

public:
    CustomDataType()
        :m_id(0)
    {
        std::memset( &m_vmt, 0, sizeof( m_vmt));
    }

    CustomDataType( const std::string& name_,
            CustomDataValueConstructor constructor_,
            CreateCustomDataInitializer initializerconstructor_=0);

    void define( UnaryOperatorType type, UnaryOperator op);
    void define( BinaryOperatorType type, BinaryOperator op);
    void define( ConversionOperatorType type, ConversionOperator op);
    void define( DimensionOperatorType type, DimensionOperator op);
    void define( const char* methodname, CustomDataValueMethod method);
};
```

```
typedef CustomDataType* (*CreateCustomDataType)( const std::string& name);

}}//namespace
```

CustomDataInitializer Interface

The custom data inizializer definition. From this class we have to derive our own initializer definions.

```
namespace _Wolframe {
namespace types {

// Initializer for a custom data value
class CustomDataInitializer
{
public:
    CustomDataInitializer();
    virtual ~CustomDataInitializer();
};

}}//namespace
```

Class CustomDataValue

The custom data type value instance definition. From this class we have to derive our own custom value definions.

```
namespace _Wolframe {
namespace types {

// Custom data value interface
class CustomDataValue
{
public:
    CustomDataValue();
    CustomDataValue( const CustomDataValue& o);
    virtual ~CustomDataValue();

    const CustomDataType* type() const;
    const CustomDataInitializer* initializer() const;

    virtual int compare( const CustomDataValue& o) const=0;
    virtual std::string tostring() const=0;
    virtual void assign( const Variant& o)=0;
    virtual CustomDataValue* copy() const=0;

    // try to convert the value to one of the basic
    // variant types and return true on success:
    virtual bool getBaseTypeValue( Variant&) const;
```

```
};

}}//namespace
```

Building Blocks

When you include "appdevel/customDatatypeModuleMacros.hpp" or simply "appDevel.hpp" you get the building block declared to build a custom data type in a module.

Declaring a custom data type

The following declaration shows a declaration of a simple custom data type.

```
WF_CUSTOM_DATATYPE(name,constructor)
```

where name is the identifier string of the function in the system and constructor a function with the following signature:

```
typedef CustomDataType* (*CreateCustomDataType)( const std::string& name);
```

3.3.3. Define Filters

In this chapter we introduce how to declare a filter type in a module. Filters are used to deserialize input and to serialize output.

Filter element types

Filters provide a uniform interface to content as sequence of elements. The elements have one of the following types.

Table 3.2. Filter element types

Identifier	Description
OpenTag	Open a substructure (element value is the name of the structure) as the current scope.
CloseTag	Close the current substructure scope or marks the end of the document if there is no substructure scope open left (top level close).
Attribute	Declare an attribute (element value is the name of the attribute)
Value	Declare a value. If the previous element was an attribute then the value specifies the content value of the attribute. Otherwise the value specifies the content value (only one allowed) of the current substructure scope.

Filter element values

Filter values are chunks of the input and are interpreted depending on the filter element type.

Filter Interface

A filter definition is a structure with 2 substructure references: An input filter (InputFilter) and an output filter (OutputFilter). You have to include "filter/filter.hpp" to declare a filter.

Input Filter Structure

From this interface you have to derive to get an input filter class.

```cpp
namespace _Wolframe {
namespace langbind {

// Input filter interface
class InputFilter
{
public:
    // State of the input filter
    enum State
    {
        Open,         // normal input processing
        EndOfMessage, // end of message reached (yield)
        Error         // an error occurred
    };

    // Default constructor
    explicit InputFilter( const char* name_);

    // Copy constructor
    ///\param[in] o input filter to copy
    InputFilter( const InputFilter& o);

    // Destructor
    virtual ~InputFilter();

    // Get a self copy
    virtual InputFilter* copy() const=0;

    // Get an instance copy of this in its initial state
    virtual InputFilter* initcopy() const=0;

    // Declare the next input chunk to the filter
    virtual void putInput(
        const void* ptr,
        std::size_t size, bool end)=0;

    // Get the rest of the input chunk left
    // unparsed yet (defaults to nothing left)
    virtual void getRest(
        const void*& ptr,
        std::size_t& size, bool& end);

    // Get a named member value of the filter
    virtual bool getValue(
        const char* id, std::string& val) const;

    // Get next element
```

```
        virtual bool getNext(
            ElementType& type,
            const void*& element, std::size_t& elementsize)=0;

        // Get type of the document
        virtual bool getDocType( types::DocType& doctype);

        // Evaluate if the document metadata are avalable
        // and set state for fetching them if needed
        virtual bool getMetadata();

        // Get the current state
        State state() const;

        // Set input filter state with error message
        void setState( State s, const char* msg=0);
    };

    // Shared input filter reference
    typedef types::CountedReference<InputFilter> InputFilterR;


}}//namespace
#endif
```

Output Filter Structure

From this interface you have to derive to get an output filter class.

```
namespace _Wolframe {
namespace langbind {

// Output filter
class OutputFilter
    :public FilterBase
{
public:
    // State of the input filter
    enum State
    {
        Open,           //< normal input processing
        EndOfBuffer,    //< end of buffer reached
        Error           //< have to stop with an error
    };

    // Default constructor
    OutputFilter(
        const char* name_,
        const ContentFilterAttributes* attr_=0);

    // Copy constructor
    OutputFilter( const OutputFilter& o);
```

```
        // Destructor
        virtual ~OutputFilter(){}

        // Get a self copy
        virtual OutputFilter* copy() const=0;

        // Print the follow element to the buffer
        virtual bool print(
            ElementType type,
            const void* element,
            std::size_t elementsize)=0;

        // Set type of the document.
        virtual void setDocType( const types::DocType&);

        // Get the current state
        State state() const;

        // Set output filter state with error message
        void setState( State s, const char* msg=0);

    protected:
        std::size_t write( const void* dt, std::size_t dtsize);
    };

    ///\typedef OutputFilterR
    // Shared output filter reference
    typedef types::CountedReference<OutputFilter> OutputFilterR;

}}//namespace
#endif
```

Filter Structure

The structure 'filter' you have to create and instantiate with an input filter and an output filter reference. There is a filter type defined with a virtual constructor to instantiate the filter. From this class you have to derive.

```
namespace _Wolframe {
namespace langbind {

class Filter
{
public:
    Filter( const InputFilterR& i_, const OutputFilterR& o_)
        :m_inputfilter(i_),m_outputfilter(o_)
    {
        m_outputfilter->setAttributes( m_inputfilter.get());
    }
};
```

```
typedef std::pair<std::string,std::string> FilterArgument;

class FilterType
{
public:
    virtual ~FilterType(){}
    virtual Filter* create( const std::vector<FilterArgument>& arg) const=0;
};

}}//namespace
```

Building Blocks

When you include "appdevel/filterModuleMacros.hpp" or simply "appDevel.hpp" you get the building block declared to build a filter in a module.

Declaring a filter

The following declaration shows a declaration of a simple custom data type.

```
WF_FILTER_TYPE(name,constructor)
```

where name is the identifier string of the function in the system and constructor a function with the following signature:

```
typedef FilterType* (*CreateFilterType)();
```

Glossary

This is the glossary for the Wolframe Extensions Development Manual.

Wolframe glossary

Normalization Function

A Normalization Function is a function taking an atomic value as input and returning an atomic value as output. It validates the input and throws an exception if the validation failes. It transforms the value into a normalized form.

Variant Type

A variant type represents an atomic value with its type. The value can appear as an integral or floating or fixed point number or as a boolean or as a string. The variant types helps to interface with interpreted non strongly typed or value typed languages. The name "variant" for this type has been chosen because it is used in many other systems (Microsoft COM/.NET, Qt, boost) as name for this kind of a union type.

Index