Navajo Technical Documentation

Author: Matthijs Philip
Company: Dexels BV
Version: 0.5
Last Modification: March 17, 2003

Table of contents

1 Introduction

Short no-nonsense, general introduction to the Navajo environment …

… without stupid Acronyms and buzzwords please …

1.1 Main features and methodology

The driving technologies behind the Navajo environment are XML and Java. XML is used at the interface level; Java is used both for the Navajo core engine and at the level of user defined connector logic. A generic, XML based composition language, called NavaScript is used to define which parts of the business process flow can be captured in separate processes implemented as (web) services. These services can be assigned to be used by authenticated and authorized client-applications (called users) only. Services are accessed using protocol listeners.

A service composition uses several application connectors for accessing existing applications and for transforming service XML content. Furthermore, a service can be augmented with business rules, defined in a user profile. User definitions, services definitions, business process flows and user profiles are stored in the Navajo repository.

1.2 Navajo document flow

A standard Navajo service distinguishes between the following phases:

Authorization/authentication

The required parameters for a specific service are being sent from the client to the Navajo server, originating from an XML input form (a so called NavaForm). The identity of the client is verified and the client's authorization relating to the service is checked. Users, services, conditions and parameters are being stored in a separate repository.

Validation and document augmentation

The content of the NavaForm is validated using client and service specific business rules. Client specific business parameters can be added to the document, using similar NavaScript expressions as the ones used to construct the form in the first place.

Business Object Mapping

The parameter values in the NavaScript document are translated to Java Business Objects by the Navajo server, which actually process the required service. A NavaScript document is generated as a response. This response document is sent back to the client and can serve as the input document of a different Navajo service.

1.3 NavaScript and NavaForm

NavaScript is a generic scripting language that is used to define services. The output of a NavaScript is always a NavaForm XML document, the input usually is another NavaForm TML document.

1.4 Navajo expression language

Conditional and arithmetic expressions can be used to create specific rules which support validation of NavaForm input, output and data-flow. Conditions can be extended by custom made functions, usually written in Java, which implements business specific validation rules.

1.5 Reusable business objects: NavaBeans

Functional business logic that can be captured in a distinct object that produces predictable output based on given input parameters can be implemented by using so called NavaBeans. The attributes of these objects can be read and written to, using the NavaScript expression language. In the objects themselves arbitrary functionality can be implemented, including mathematical functions, calls to databases (e.g. Oracle, MySQL, SQL-server, Sybase), other applications (e.g. SAP, Baan, Siebel) or even other Navajo services. See chapter 4.

...

2 NavaForm

2.1 Introduction and example

To define services in a generic way, a specific structured XML definition for input and output parameters has been created. A NavaForm document that is being sent by the Navajo server can holds:

- A document body containing the structured service parameters
- A method block in which services can be defined that process the returned data.

NavaForm documents are self-describing and can contain both input and output parameters. NavaForm documents can be "filled in" and sent back to one of the services specified in the NavaForm method block.

Let's start with a simple service which searches for a city based on a zip code and address number. A Navajo service typically consists of two sub-services; in our case:

- InitZipCodeLookup
- ProcessZipCodeLookup

The first sub-service, InitZipCodeLookup is a service that requires no input. When client calls this service and has the proper credentials, he or she receives the following NavaForm document:

```
<tml>
   <message name="Input">
      <property name="ZipCode" type="string" direction="in" length="6"
        value=""/>
      <property name="AddressNumber" type="integer" direction="in"
        length="5" value=""/>
   </message>
   <methods>
      <method name="ProcessZipCodeLookup">
         <required name="Input"/>
      </method>
   </methods>
</tml>
```

This NavaForm document is relatively straightforward. It's starts with an opening **tml** tag that serves as the parent for the other elements. A so called **message** is defined that contains two parameters called **properties**. Properties always contain a *name*, a *type* and a *direction*. When a property is an input property (direction="in") a *length* attribute should be provided and a *value* attribute should be filled in.

In order for the actual data to be processed, a **methods** block is defined which contains one or more **method** elements that serve as subsequent services. Per method it is possible to define which parameters are required by specifying their top-level message(s), using the **required** element. In our example only one subsequent service, ProcessZipCodeLookup with one required message (Input) is defined. After filling in a zip code and an address number, the client sends the NavaForm document to the Navajo server.

The Navajo server receives the NavaForm document and authorizes and authenticates the client and the service. The following response is sent back:

```
<tml>
  <message name="Output">
    <property name="City" type="string" direction="out" value="Amsterdam"/>
  </message>
</tml>
```

We see here that the response only contains an output property (direction="out") and doesn't have a methods block. This means that there are no follow-up services for ProcessZipCodeLookup.

2.2 NavaForm elements and attributes

NavaForm documents make use of the elements depicted in table 2.1

message	name*, count, condition, comment
property	name*, type*, direction*, length, value, cardinality, description, comment
option	name*, value*, selected, condition, comment
methods	
method	name*, condition, description, comment
required	message*, comment
comment	value*

Table 2.1 - NavaForm elements and their attributes

Message

Message elements are defined by the user to uniquely identify and categorize the data that a service generates. Messages usually contain a collection of properties, maps to NavaBeans, local parameters and possibly even other nested messages.

A message always has a name assigned to it. The count attribute can be used to tell how many times a given message must be processed by the interpreter. Just as in the map, field, param, option and expression elements (some of which we well discuss later), the condition attribute may contain conditional logic which defines whether an element should be parsed or not. All elements, methods and comment elements can contain a comment attribute. Comments are not interpreted but are only used for the convenience of the developer. The HTML pages that NavaDoc generates do display the comment though.

Property

Property elements appear as the children of a message or a map element. Properties can be seen as the basic data entities that are the result of the processing of the NavaScript documents. Properties always have a name, type and direction. The type can be either boolean, integer, float, points, string, date or selection. The direction is used to distinguish between outgoing and incoming properties, where incoming properties are usually used in subsequent scripts and are seen as changeable entities by the end user. Outgoing properties are usually for display only and would appear as read only fields.

The length and value of properties are used to restrict possibilities or add a default values. Cardinality can be either "1" or "+" and should only used

when the property is of type selection. Obviously a "+" reflects multiple selections and "1" only one valid option (think of either multiple checkboxes versus radio buttons or a single valued dropdown).

Option

Option elements always appear as the children of a property of type selection. Options should always have a name and a value. Additionally the selected attribute can be set to either true or false.

Methods

The methods element is used to group a collection of method elements.

Method

The method element contains information on what NavaScript should be the target of the current one. In many ways methods reflect the buttons that one would find at the bottom of a form. Apart from the required name attribute of a method, which should be equal to the name of the target NavaScript (apart from the .xsl extension), a method may have a description as a more readable label for the user. A method can have one or more “required” elements, depicting which messages of the current script are required as input for the next NavaScript.

Required

The required element appears as a child to a method element and delineates which messages are to be sent to the NavaScript that appears in the name of the parent method. If omitted, all messages of the current NavaScript are being sent. The required element only has a message attribute, which is obligatory.

Comment

Finally the comment element offers the developer the possibility to add some comment to the NavaScript file, in order to make things more comprehensible. Comment should be placed under the value attribute.

3 NavaScript

This section discusses how NavaScript is used to implement basic services, using an XML based scripting language. We will start of with a simple example to give the reader an impression of what he is dealing with. NavaScript files generate NavaForm documents to return to the client.

In paragraph 3.2, we will give an overview of all the NavaScript elements and attributes after which the reader will be ready for a more elaborate example.

3.1 Simple NavaScript example

This is an example that shows how NavaScript can be used to convert euros to dollars or vice versa. The example consists of two files, one NavaForm file that expects input from the user, and a NavaScript file that actually converts the input amount the way the user told it to.

The first script that we will call InitConversion.xsl will look something like this:

```
<tml>
  <message name="ConversionData">
    <property name="InputAmount" type="float" direction="in" length="6"/>
    <property name="ConversionDirection" type="selection" direction="in"
      cardinality="1">
      <option name="DollarsToEuros" value="D"/>
      <option name="EurosToDollars" value="E"/>
    </property>
  </message>
  <methods>
    <method name="ProcessConversion">
      <required message="ConversionData"/>
    </method>
  </methods>
</tml>
```

This script consists of one message with two properties that are shown to the user. Using an XSLT-style sheet, this XML-file could be converted to a basic HTML form which can be displayed in a standard web browser. The properties reflect input fields of such a form and the method will be displayed as a submit button which submits the data to the next service. The name of this service is similar to the name of the method and will therefore be ProcessConversion.xsl

The NavaScript file that processes the conversion and displays the result to the user in a NavaForm will look like this:

```
<tsl>
  <message name="ResultData">
    <property name="OutputAmount" type="float" direction="out">
      <expression value="[/ConversionData/InputAmount] * 0.95"
        condition="[/ConversionData/ConversionDirection:value] == 'D'"/>
      <expression value="[/ConversionData/InputAmount] / 0.95"
        condition="[/ConversionData/ConversionDirection:value] == 'E'"/>
    </property>
  </message>
</tsl>
```

What we see here is a file that, based on the selected ConversionDirection either multiplies or divides the InputAmount by a given rate (in our case a hard coded 0.95). The result is displayed in a ResultData message with just one outgoing property called OutputAmount. To set the value of this property, we can access the parameter values of the incoming NavaForm with the [/MessageName/PropertyName] construct: the value of the incoming amount will be [/ConversionData/InputAmount]. To get the value of a selection property, add the suffix ":value" – so we can find the value of the conversion direction with: [/ConversionData/ConversionDirection:value]. Only the first expression which has its condition evaluated to true is parsed, so either the input amount is multiplied or divided by our rate. Since the possibilities of our selection property are restricted to just two, no additional condition is necessary (in other cases we could add an empty condition to a third expression, which will at least evaluate to true).

3.2 NavaScript elements and attributes

Like all XML documents NavaScript files consist of possible nested groupings of elements which can have multiple attributes. An overview of the NavaScript elements and their attributes (* means obligatory) are depicted in table 3.1. Each of the elements is described in more detail below (for the message, property, option and comment elements, we refer to section 2.2)

Element	Attributes
tsl	author, id, repository, notes
message	name*, count, condition, comment
property	name*, type*, direction*, length, value, cardinality, description, comment
option	name*, value*, selected, condition, comment
map	object, ref, filter, condition, comment
field	name*, condition, comment
param	name*, condition, comment
expression	value*, condition, comment
comment	value*

Table 3.1 - NavaScript elements and their attributes

Tsl

Similar to a standard xml-tag, all NavaScript files start using a tsl element. For versioning reasons an author, id, repository and some notes may be added.

Map

Maps are used to link a NavaBean to a NavaScript. Maps can appear at any place in a script and can be both parent and child to a message element. A map either has an object or a ref (reference) attribute. In the first case object refers to a NavaBean and is usually a Java-style class identifier. The object attributes of the NavaBean itself can then be accessed directly within the context of the map. In the second case, with the ref attribute being filled, the context of a map can be used to refer to either a mappable object that is an attribute of the currents object context, or to a message or messages from an incoming NavaScript file.

When using the ref attribute, the filter attribute can be used as a condition to what should be parsed and what should not.

Field

The field parameter is used to access the object attributes of the NavaBean context. A field always has a name attribute which refers to the name of the objects in the NavaBean. Additionally a field should have an expression child, which sets or retrieves the value of one of the NavaBeans attributes.

Param

Param elements are used to set local variables or parameters within the context of the NavaScript in which they are present. Params can be used to store information, or reuse information from one map in another. As with fields the actual value of the parameter is set in an expression child element. Section 3.4 shows an example of how the param element can be used.

Expression

Expressions always appear as the child of either a field or a param element and as such should contain a value attribute.

3.3 More elaborate NavaScript example

This paragraph gives an example of two simple NavaScript files that query a user from a database for a given username. The first NavaForm, which we would call InitSearchUser, could look something like this:

```
<tml>
  <message name="UserSearch">
    <property name="SearchName" type="string" direction="in" length="15"/>
  </message>
  <methods>
    <method name="ProcessSearchUser">
      <required message="UserSearch"/>
    </method>
  </methods>
</tml>
```

The second file which we called ProcessSearchUser, uses the SearchName to look for a first name and last name in the database.

```
<tsl>
  <map object="com.dexels.navajo.adapter.SQLMap">
    <field name="query">
      <expression value="'SELECT firstname, lastname FROM Users WHERE
        username = ?'"/>
    </field>
    <field name="parameter">
      <expression value="[/UserSearch/UserName]"/>
    </field>
      <message name="Users">
        <map ref="resultSet">
          <property direction="out" name="FirstName" type="string">
            <expression value="$columnValue('firstname')"/>
          </property>
          <property direction="out" name="LastName" type="string">
            <expression value="$columnValue('lastname')"/>
          </property>
        </map>
      </message>
```

```
  </map>
</tsl>
```

In this file the first map refers to a Java adapter / NavaBean in which we defined our database parameters. The field elements query and parameter are used to send the query and the username to this adapter. The question marks in the query string refer sequentially to the fields named parameter below. The username parameter is filled with [/UserSearch/UserName] which refers to the UserName property of the UserSearch incoming message (i.e. our first NavaScript). The result(s) of the query are put in a Users message, which for its properties (FirstName and LastName) is mapped to a default object called resultSet (this is an attribute of the com.dexels.navajo.adapter.SQLMap object). To match the column data to the value of the properties we used $columnValue('column_name'). The dollar sign means that we are getting or retrieving data from the map or NavaBean. No such sign (as in the query and parameter fields, means we are sending or setting data in the NavaBean).

3.4 Local parameters

In NavaScript it is also possible to store local parameters to be used throughout a script. The **param** tag can set a variable with a given name and a value which henceforth can be accessed within the context of a single script file, just like a normal property with a "__parms__" prefix.

This small example shows how a parameter is set based on a query and then inserted into a message.

```
<tsl>
   <map object="com.dexels.navajo.adapter.SQLMap">
      <field name="query" value="SELECT PaymentMethod FROM Members WHERE
       username = ?'"/>
      <field name="parameter">
         <expression value="[/Members/MemberIdentifier]"/>
      </field>
      <param name="PaymentMethod">
         <expression value="$columnValue('PaymentMethod')"/>
      </param>
   </map>
   <message name="Result">
      <property name="MemberPaymentMethod" type="string" direction="out">
         <expression value="[__/parms__/PaymentMethod]"/>
      </property>
   </message>
</tsl>
```

Usually parameters are used between different maps and messages to store for instance a transaction context, selected values for certain selection properties etc.

In the next chapter we will describe the Navajo expression language that can be used for various conditional and arithmetic operations.

4 Expression Language

On various places within the Navajo framework arithmetic terms can be used to ensure conditional logic or express values. These expressions can be found at many different places:

- In the parameter table of the Navajo database, both conditional and value expressions can be added, usually refered to as business parameters
- In the condition table expressions can be added to validate incoming NavaScript documents, these reflect business rules
- Within the NavaScript files themselves conditional and value expressions are used to ensure that correct data is being set

4.1 Sample expression

A typical example of a conditional expression would be:

```
[/MyMsg/SomeProperty] * 8 <= ([/OtherMsg/AnotherProperty] - 12) / 5.5
```

Just as seen in the example NavaScript files, the text between the straight brackets ("[" "]") serves as a reference to a NavaScript property. This conditional expression could be used in the validation of this document:

```
<tsl>
  <message name="MyMsg">
    <property name="SomeProperty" type="integer" direction="in"
      value="8"/>
  </message>
  <message name="OtherMsg">
    <property name="AnotherProperty" type="float" direction="in"
      value="100"/>
  </message>
</tsl>
```

In this case the expression would evaluate to false since 8*8 (=64) is not smaller than (100 -12) / 5.5 (=16)

4.2 Operators

The operators that Navajo uses can be categorized as follows:

Binary arithmetic operators

/	(divide)
*	(multiply)
+	(add)
-	(substract)
%	(modulo)

Binary comparison operators

<	(smaller than)
<=	(smaller or equal to)
>	(bigger than)
>=	(bigger or equal to)
==	(equals)
!=	(not equal)

Binary logical operators

AND (logical and operator). Result is either true of false
OR (logical or operator). Result is either true of false

Unary logical operators

! (logical not operator). Result is either true of false
? (exists). Used to test the existence of a property, result is either true of false

Sartre operators

FORALL() Result is either true of false
EXISTS() Result is either true of false

These two so called Sartre operators can be used to check an entire list of operands. FORALL() evaluates to true if all elements of the list satisfy the given expression or evaluator. EXISTS() evaluates to true if at least 1 element of the list evaluates to true. The first of the two parameters of these functions refers to the message or submessages that need to be searched. The second parameter can be any one or more conditions which are to be met. For instance when we have a NavaScript file with multiple instances of message ClubMember in which each ClubMember has an Age and a Gender, we can check if there is at least one female member older than 20 with the following expression:

```
EXISTS('/ClubMember/', `[Gender] == 'female' AND [Age] > 20`)
```

Similarly whe could use the FORALL() function combined with a user-defined CheckRelatieCode() function to check if all ClubMemberShips who's [Update] checkbox (boolean) is set to true, have a valid ClubIdentifier as such:

```
FORALL('/ClubMembership/ClubMemberships', `[Update] == false OR
(CheckRelatieCode([ClubIdentifier]) == true AND [Update] == true)`)
```

Note that the expression is surrounded by the "`" character (the grave accent, usually under the tilde ("~") button) to allow us to use normal apostrophes ("'") in our expressions.

Appendix A gives an overview of Navajo's datatypes and how operators can be used to combine them.

5 NavaBeans and Functions

In this chapter we discuss how NavaBeans and user-defined functions can be used to map Java code to the script in the NavaScript files.

5.1 NavaBeans

Anywhere in the NavaScripts mappable objects or NavaBeans can be defined which are able to read and set the attributes of Java objects. An example of a mappable object (interface) called MyMap is given in the following Java code:

```
public class MyMap implements Mappable {

  public String userName;
  public int userAge;

  public void load() {
  }

  public void store() {
    userName.toUpperCase();
  }

  public void kill() {
    System.out.println("\nException occured in MyMap()");
  }

  public String getUserName() {
    return userName;
  }

  public void setUserName(String s) {
    this.userName = s;
  }

  public int getUserAge() {
    return userAge;
  }

  public void setUserAge(int a) {
    this.userAge = a;
  }
}
```

Every map implements interface Mappabe and consists of at least a load(), store() and kill() method. For every property of the NavaBean there should be a setXXX() and a getXXX(method). Everytime that a NavaScript encounters a <map object="..."> tag, the load() method of that map is called. In the load we could for instance initialize database parameters, check additional security etcetera. After the entire map is parsed and the closing tag is found, naturally the store() method is called. In our case the store() method transforms the userName to upper case. The kill() method serves to catch any exceptions that might trigger.

The attributes in the map can be referenced to in the NavaScript code with the same names as the variables in the NavaBean. Using the "$" sign in front of the name, retrieves an attribute, using just the name sets it.

Using this map the following expressions could be formulated:

```
$userName == 'DEXELS'
```

```
$userAge + 12 >= 35
```

Maps can also be nested, so in the case, in which we define a parentMap which uses a variable of type MyMap called myMap, we could access the userName and userAge with `$myMap.userName` and `$myMap.userAge`. An overview of current available maps and their usage is depicted in Appendix B.

5.2 Functions

Another way to link Java to the NavaScript files is to use function that can be accessed directly from the scripts. Apart from the build-in functions, which are described in detail in Appendix C, users can easily define their own specific ones that implement the Dexels function interface.

A very simple example of a user-defined function would be the ToUpper() function that parses a string and converts it to upper case. The code of this function looks like this:

```
public class ToUpper extends FunctionInterface {

    public ToUpper() {}

    public Object evaluate() throws Exception {
        String s = (String) this.getOperands().get(0);
        return s.toUpperCase();
    }

    public String usage() {
        return "";
    }

    public String remarks() {
        return "";
    }
}
```

As you can see the function implements the FunctionInterface which has an evaluate(), usage() and remarks() method. The usage() and remarks() methods are used for the convenience of the developer, and can for instance be dumped to the standard output. The evaluate() method does all the actual work and parses, converts, or basicaly changes the operands.

An example of how to use a function in a NavaScript file would be:

```
<field name="parameter">
  <expression condition="" value="ToUpper([/MyMsg/MyProperty])"/>
</field>
```

As long as the datatype of [/MyMsg/MyProperty] matches possible input for the function all should function correctly.

Functions can be as complicated as you would want and can have as many parameters as required. Another example of a little bit more complicated function could be a TimeToString() function that parses a date object and converts the time to a string. The code of such a function would look like this:

```
public class TimeToString extends FunctionInterface {
```

```
  public TimeToString() {
  }

  public Object evaluate() throws Exception {
    java.util.Date d = (java.util.Date) getOperand(0);
    java.util.Calendar c = java.util.Calendar.getInstance();
    c.setTime(d);
    String hours = c.get(java.util.Calendar.HOUR_OF_DAY)+"";
    String minutes =   c.get(java.util.Calendar.MINUTE)+"";
    return hours + ":" + minutes;
  }

  public String usage() {
    return "TimeToString(Date d)";
  }

  public String remarks() {
    return "Converts a time in a date object to a string (HH:MM)";
  }
}
```

Method evaluate() returns any of the following Java datatypes:

- java.lang.Boolean
- java.lang.Integer
- java.lang.Double
- java.lang.String
- java.util.Date
- java.util.ArrayList

Method getOperands() returns an object of type java.util.ArrayList, which contains all the operands.

Appendix A – Datatypes

NavaScript supports boolean, integer, float, string, date and list data types. Within expressions it's possible to combine different datatypes. This table displays which combinations of datatypes are possible, which arithmetic operators can be used and which datatype is the result of the combination.

Argument 1	Argument 2	Operators	Result	Java Type
Integer	Integer	(*, /, +, -, %)	Integer	java.lang.Integer
Integer	Float	(*, /, +, -)	Float	java.lang.Double
Integer	String	(+)	String	java.lang.String
Float	Float	(*, /, +, -)	Float	java.lang.Double
Float	String	(+)	String	java.lang.String
Date	String	(+)	String	java.lang.String
Date	Date (pattern)	(+, -)	Date	java.util.Date
String	String	(+)	String	java.lang.String
List	Integer	(*, +, -)	List	java.util.ArrayList
List	Float	(*, +, -)	List	java.util.ArrayList
List	List	(+, -)	List	java.util.ArrayList

Table A.1 – Navajo Datatypes

Datatype **integer** has the following pattern:

("-")?[1-9][0-9]* examples are 3850, 24 and -6503

Datatype **float** has the following pattern:

("-")?(0"."[0-9]+)|([1-9]*"."[0.9]+) examples are 0.372, 1267.34 and -4.5

Datatype **string** has the following pattern:

"'"([A-z]|[0-9]|" "|":"|";"|"-"|"_")+"'" examples are 'this is a string' or 'D3x37s : r0x0r'

Datatype boolean has the following pattern:
"true" | "false" the ony possible values are true and false

A.1 Datatype date and date patterns

Dates in Navajo always are always of the following pattern: **yyyy-mm-dd**. Dates support aritmetical additions if the other operand is a so called date pattern, a special datatype used only as an operand and never as a result of an operator (operands correspond to arguments, operators do the actual processing). Using a date pattern, an offset can be assigned to the year, month or day fields of a date datatype respectively.

The expression `1900-01-01 + 81#3#1` will evaluate to `1981-04-02`

It's important to know that date datatypes **can not** be directly set in an expression, but should always appear as a reference to a NavaScript property of type date.

```
<tsl>
  <message name="MyMsg">
    <property name="MyDate" type="date" direction="in" value="2001-10-16"/>
  </message>
</tsl>
```

For example our last expression should be:

```
[/MyMsg/MyDate] + 1#3#10 = 2003-01-26
```

The year, month and day fields of a date datatype can also be accessed individually using the .year, .month and .day postfix operators

For instance `[/MyMsg/MyDate]`.year = 2001

The binary comparison operators (<, <=, >, >=, ==, !=) can also be used to compare two properties of type date or a date and a integer. In this last case the "age" of the date is automatically calculated to create a meaningful comparison. Additionaly they can be used in combination with the build-in global variable TODAY or the Date() function.

These examples will clarify date comparisons (using our MyDate property)

```
[/MyMsg/MyDate] > 2
```

Evaluates to true if the current date is more than 2 years away from 2001-10-16

```
[/MyMsg/MyDate] <= TODAY
```

Currently evaluates to true since we are passed 2001 by now.

```
[/MyMsg/MyDate] == Date('1950-10-25')
```

The Date() function parses the string to a date, which is than checked for equality with our date property. Of course it evaluates to false.

A.2 List datatypes

To at least support selection properties with multiple selected options, Navajo supports a list datatype which can be of the following syntax:

{'fox', 'dog', 'wolf'} or {12,65,83,289,3} as a list of strings and integers respectively

A valid expression would be:

{13,5,82} + 10 which results in {23,15,92}

Using this example:

```
<message name="Club">
  <property name="ClubFunctions" type="selection" direction="in"
    cardinality="+">
    <option name="Consul" value="CONS" selected="0"/>
    <option name="Administrator" value="ADMIN" selected="1"/>
    <option name="Referee" value="REF" selected="0"/>
    <option name="Trainer" value="TRAIN" selected="1"/>
  </property>
</message>
```

The following expressions:

```
[/Club/ClubFunctions:name]
[/Club/ClubFunctions:value]
```

Evaluate to:

{'Administrator', 'Trainer'} and {'ADMIN', 'TRAIN'}

The evaluative function Contains() which expects two arguments: a list and either a string, integer or float can be used to check the presence of an

entity in a list. An example that could be used to check whether a person is a trainer would be:

```
Contains([/Club/ClubFunctions:value], 'TRAIN') == true
```

We could also create a new list in the function itself:

```
Contains({[/Club/ClubFunctions:value], 'ASSISTANT'}, 'ASSISTANT')
```

This would of course evaluate to true since we check the presence of an entity that we added to the list.

Another useful function is Sum() which adds up the elements in a list, if they are of type integer or float.

Appendix B – NavaBeans

This appendix gives an overview of the most common NavaBeans or mappable objects that are currently implemented in the Navajo Framework. The most common NavaBeans are described below

B.1 SQLMap

The SQLMap NavaBean can be used to map a datasource to a NavaScript file. It uses the following parameters of which we will describe the most common ones.

Input parameters

- autoCommit(boolean)
- datasource(string)
- deleteDatasource(string)
- doUpdate(boolean)
- endIndex(string, int)
- parameter(object)
- query(string)
- reload(string)
- resultSetIndex(int)
- rowCount(int)
- startIndex(string, int)
- transactionContext(int)
- transactionIsolationLevel(int)
- update(string)

Output parameters

parameter name	return value
- columnName(int)	string
- currentElements(string)	int
- endIndex(string)	int
- remainingElements(string)	int
- resultSet()	resultSetMap
- rowCount()	int
- startIndex(string)	int
- totalElements(string)	int
- transactionContext	string

Field resultSet of type resultSetMap has the following output parameter:

- columnValue(string)	string

Field **query** is used to specify a query. A query can contain certain parameters that can be displayed with a question mark (?) – analogous to the prepared statement syntax within the JDBC API.

Field **parameter** can be mapped exactly as many times as there are parameters defined in the query. The sequence of these parameters must of course match the sequence of the query.

The returned recordset is accessible via the **resultSet** map that contains all the columnValues of each individual record. To retrieve the value of a

given entry of this map, supply columnValue with the name of the database column matching the one in the query. For instance: $columnValue('username') – note the dollar sign used to retrieve information from a NavaBean.

Check section 2.2 for a NavaScript example of how to use a SQLMap.

B.2 SPMap

In many ways similar to the SQLMap, the SPMap can be used to execute stored procedures. The SPMap uses the same parameters as the SQLMap (in Java terms, the SPMap extends the SQLMap). It uses two other parameters though, one to set the outputParameterType and one as the actual value of the SP's outputParameter itself.

To call a stored procedure which sets data instead of retrieving it, use the field named **update** in stead of the field named **query**. When using an **update** field always include a **doUpdate** field which expects boolean to trigger the actual call of the SP.

When the SP returns a value, the **outputParameterType** must be used to set the data type which the SP is returning. The types that can be used here are INTEGER (SMALLINT, TINYINT), VARCHAR, CHAR, FLOAT, DOUBLE, DATE, TIMESTAMP and BIT.

The value of the SP's output can be retrieved with the $outputParameter(1) command.

This script shows an example of how NavaScript calls a SP.

```
<tsl>
   <map object="com.dexels.navajo.adapter.SPMap">
      <field name="update">
         <expression condition="" value="'{? = Call
          clubmember_DEL(?, ?)}'"/>
      </field>
      <field name="outputParameterType">
         <expression condition="" value="'INTEGER'"/>
      </field>
      <field name="parameter">
         <expression condition="" value="[/ClubMember/MemberIdentifier]"/>
      </field>
      <field name="parameter">
         <expression condition="" value="[/ClubMember/ClubIdentifier]"/>
      </field>
      <field name="doUpdate">
         <expression condition="" value="true"/>
      </field>
      <param name="Status">
         <expression condition="" value="$outputParameter(1)"/>
      </param>
   </map>
   <message condition="" count="1" name="Result">
      <property direction="out" name="Status">
         <expression condition="" value="[/__parms__/Status]"/>
      </property>
   </message>
</tsl>
```

B.3 NavajoMap

The NavajoMap can be used to map NavaScript files within other NavaScript files. This may come in handy when two files share a lot of characteristics. A NavajoMap commonly expects the following parameters/fields:

Input

- server(string)
- username(string)
- password(string)
- propertyName(string)
- stringProperty(string)
- integerProperty(int)
- dateProperty(date)
- booleanProperty(boolean)
- doSend(boolean)

Output

- exists(string) boolean
- stringProperty(string) string

The **server** field specifies the URL of the server – servlet that has to be called.

A sequence of **propertyName** and (**stringProperty**, **integerProperty**, **dateProperty** or **booleanProperty**) can be used to refer to the names of properties in the mapped NavaScript file and to set their respective values (using the correct data type). The example below will clarify their use.

The **doSend** field is used to actually call the script and sets the name of the NavaScript file, the **username** and **password** fields can be used to set authentication.

NavajoMap Example

This partial NavaScript file shows how another NavaScript file can be mapped:

```
<map object="com.dexels.navajo.adapter.NavajoMap">
   <field name="server">
      <expression condition="" value="'localhost/club-app/servlet/Postman'"/>
   </field>
   <field name="propertyName">
      <expression condition="" value="'/Input/ZipCode'"/>
   </field>
   <field name="stringProperty">
      <expression condition="" value="[/__parms__/ZipCode]"/>
   </field>
   <field name="propertyName">
      <expression condition="" value="'/Input/AddressNumber'"/>
   </field>
   <field name="integerProperty">
      <expression condition="" value="[/__parms__/AddressNumber]"/>
   </field>
   <field name="doSend">
      <expression condition="" value="'ProcessZipCodeSearch'"/>
   </field>
   <property direction="out" name="StreetName">
```

```
      <expression condition="$exists('/Output/StreetName') == true"
        value="$stringProperty('/Output/StreetName')"/>
      <expression value=""/>
    </property>
    <property direction="out" name="City">
      <expression condition="$exists('/Output/City') == true"
        value="$stringProperty('/Output/City')"/>
      <expression value="''"/>
    </property>
</map>
```

This example uses the functionality of an already existing NavaScript file called ProcessZipCodeSearch which requires two properties called [/Input/ZipCode] and [/Input/AddressNumber], the first a string, the second an integer to return a streetname and a city (in two properties called [/Output/StreetName] and [/Output/City]). Based on two parameters, the zipcode and the addressnumber, which are set somewhere earlier in our script (not present in this example), the map calls the ProcessZipCodeSearch, after which all the properties that are in this file can be accessed. The conditional **exists**(string) function can be used to check if a required property is actually there. To return the value of a property in a remote NavaScript, the **stringProperty**(string) function is used. Note the use of the square brackets which are omitted when referencing remote properties.

B.4 ProxyMap

A very simple map that can be used to send Navajo content to another Navajo environment/server is the ProxyMap. It's only fields, all input are:

- server
- username
- password
- method

As one would expect the **server** specifies the URL of the Navajo server and the username and **password** fields are used for authentication. The **method** field refers to the NavaScript that has to be called on the server that is being called.

Appendix C – Built-in functions

An overview of all current built-in Navajo functions, their operands and a short description is depicted in table C.1.

Function	In	Out	Description
Age	Date	Integer	Returns the amount of years passed since the input date.
CheckRange	List & integer	Boolean	Returns false when any of the list's operands >= the integer value or == 0
Contains	List & (string, integer || float)	Boolean	Returns true when the list contains an instance of the string, integer or float
Date	String	Date	Formats a date based on a YYYY-MM-DD string
DateAdd	Date, integer, string	Date	Adds the integers value to the supplied field string of the given date. Field can be YEAR, MONTH, DAY or WEEK
DateField	Date, string	Integer	Returns a numerical representation of the given field string. Field can be WEEK, YEAR or DAY
ElfProef	Integer	Boolean	Checks whether a given integer conforms to the so-called ElfProef check Used for bankaccountnumber validity in Holland.
EqualsIgnoreCase	String, string	Boolean	Checks for equality of two strings ignoring capitalization
FormatDate	Date, string	Date	Reformats a date based on the given date-formatter string.
FormatStringList	List, string	String	Formats a string of the list's items, seperated by the given string
Max	Two integers or doubles	Integer or double	Returns the maximum number
Min	Two integers or doubles	Integer or double	Returns the minimum number
NextMonth	Integer	Date	Returns a date of the next month
ParameterList	Integer	String	Returns a string of comma separate ? values for use in SQL queries
Round	Double, integer	Double	Rounds the double's value with an amount of digits after the point equal to the integer's value
Size	List	Integer	This function returns the size of a lists arguments
StringField	String, String, Integer	String	This function returns a specified string field given an initial string, the seperator string that is used and the index of the required field.
Sum	List	Integer	Adds the values of the numbers in the list
ToDouble	String	Double	Converts the string to a double
ToInteger	String	Integer	Converts the string to an integer
ToString	Integer or double	String	Convert the number to a string
ToUpper	String	String	Convert the string to uppercase

Table C.1 – Built-in functions