d.ecs forms server scripting: Administrator

Table of Contents

1. Developer documentation d.ecs forms server scripting 3
    1.1. Introduction 3
        1.1.1. About the d.ecs forms server scripting 3
        1.1.2. Prerequisites 3
        1.1.3. Structure of the scripting-interface 3
        1.1.4. Return values of script-methods in data sources 5
        1.1.5. Data types 5
        1.1.6. Communication with databases 5
    1.2. Description of the server-side scripting-methods 6
        1.2.1. Sub-object context 6
        1.2.2. Sub-object params 12
        1.2.3. Sub-object data 13
        1.2.4. Sub-object misc 17
        1.2.5. Sub-object log 20
        1.2.6. Sub-object engine 22
        1.2.7. Sub-object instance 44
        1.2.8. Sub-object convert 62
        1.2.9. Data structures 65
        1.2.10. Global methods 68
        1.2.11. Groovy-modules 68
    1.3. Deprecated methods 69
    1.4. CRUD 69
    1.5. Webservice (SOAP) 71
        1.5.1. Webservice (SOAP) 71
    1.6. Appendix 71
        1.6.1. List of all supported d.3 API functions for direct access 71
    1.7. Additional information sources and imprint 73

1. Developer documentation d.ecs forms server scripting

1.1. Introduction

1.1.1. About the d.ecs forms server scripting

Server-scripts can either be used as **data sources** or as **form-actions**.

Data source

A **data source** is intended for the data connection to third-party systems and only exists in the d.ecs forms object model. It allows to import data from external systems into d.ecs forms and export them again. Data sources are independent from the form. They cannot access form contents. They can only retrieve data based on some parameters and return them. Data sources have the following properties:

- The have dedicated import parameters (`ds.params`) and return values (`ds.data`).
- The return value of a data source is cached. In the further processing of a form instance, only the cached values are used. Every new call of the data source works cumulative on the data of the previous call. If you do not want this, the current record must be cleared. (`ds.data.clear()`)
- The method-names are predefined (`loadData`, `saveData`)

Form-action

The second field of operation for server-based scripts are **actions**. They have two main differences to **data sources**.

- Actions are located in the form-context and can read and write its values.
- Actions must be explicitly called from within the form and are not cached.

The server-side script language in d.ecs forms is Groovy.

Basically, Groovy is an extension of Java extending it with more dynamics and a more compact syntax. You can, however, also develop in Java.

> Attention:
>
> All scripts created as well as external libraries used must be compiled for use with a Java Runtime Environment version 8.

1.1.2. Prerequisites

This manual describes the d.ecs forms-server scripting and is exclusively targeted at administrators and technical users which are to develop or maintain the logic for forms.

To understand the following information, you must have made experiences in the programming language Groovy or Java. Additional knowledge of a development environment is recommended.

1.1.3. Structure of the scripting-interface

Before discussing the available scripting methods in detail, it is first necessary to examine the general structure of the scripting interface, respectively, the underlying model in greater detail.

The scripting interface is is addressed via a method. This method always has one single parameter or object. In data source (in the global object model) this parameter is usually called `ds` and in scripts to be called from within forms it would be `form`. This object allows you to perform all interactions. All parameters are read and the return values are passed. Depending on the type of the script (action or data source), the form-context can be accessed. Global auxiliary functions for the access to a database or the d.3 system are also applied via one of these objects or subobjects.

Example data source

Example:

```
def loadData(def ds) {
   def a = ds.params.getValue( "a" );
   def tmp = "You entered: ${a}";
   ds.data.setValue( "result", tmp );
   return null;
}
```

Example action:

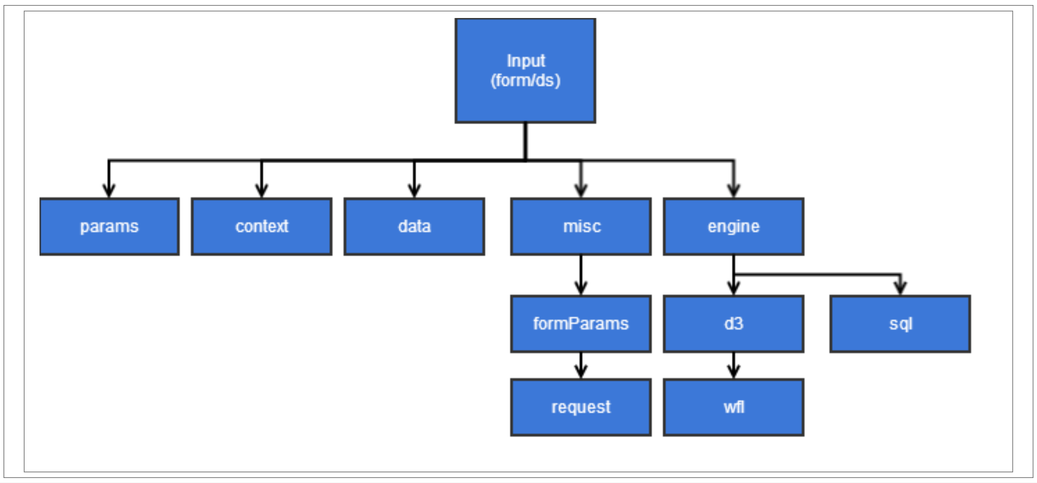
Example:

```
def doSomething( def form ) {
   def a = form.context.getValue( "MyObject.MyProperty" );
   def tmp = "You entered: ${a}";
   form.context.setValue( "MyOtherObject.MyProperty", tmp );
   return null;
}
```

Note

The sections below use the parameter `form` in the interface descriptions (according to the use in the form-contexts). In data sources in the d.ecs forms object model you must equivalently use the parameter `ds`. The parameter `ds` is only used in the descriptions here, if the used methods are exclusively intended for data sources in the d.ecs forms object model.

In this process, the Input object is divided into further sub-objects, which provide their own methods in turn. This refined subdivision is designed to support an encapsulation and aggregation of similar methods. The following sub-objects are provided: `params, context, data, misc, engine, log.`



	The **Params object** provides the parameters to be passed to the script directly.
	The **Context-object** allows to access the form-contexts.
	The **Data-object** provides the data, which is directly returned from the script. Alternatively, the return-values can be passed back with `return`.
	The **Misc-object** provides miscellaneous auxiliary functions which cannot be grouped under the sub-objects above.
	The **Engine-object** provides global auxiliary functions (d.3, SQL, ...).

Significant events (information messages, errors) can logged via the **Log object**. All messages are stored in the d.ecs forms server log.

The available methods will be described in detail in the later chapters of the manual.

Example:

Always work with the fully-qualified function signature, please. Do not assign the sub-objects to a new variable. Moreover the access must not be performed dynamically using "`eval`".

FALSE:

```
def data = form.data;

def value = data.getValue( "customer.name" );
```

CORRECT:

```
def value = form.data.getValue( "customer.name" );
```

1.1.4. Return values of script-methods in data sources

Script-methods can return their result in two different ways.

Note

It is recommended always to do it using `ds.data.setValue(…)`.

For easier syntax, however, they can be returned as a "return" value. If only one value has been defined as a return value in a data source, then this value can be returned directly. Else, a name-value pair (map) must be created.

The following syntax variants are equivalent:

```
ds.data.setValue( "name", value );
```

and

```
return ["name" : value]
```

1.1.5. Data types

The following data types are supported by d.ecs forms:

d.ecs forms data type	recommended Groovy data type	Example:
Boolean value	Boolean	`form.context.setValue( "Object.booleanValue", true );`
Date	java.sql.Date	`form.context.setValue( "Objekt.Datum", new java.sql.Date( new java.util.Date( ).getTime( ) ) );`
Integer	Integer	`form.context.setValue( "Object.Integer", 1 );`
Number	Number	`form.context.setValue( "Object.Number", 1.1 );`
Text	String	`form.context.setValue( "Object.Text", "This is a text" );`

Attention:

d.ecs forms currently supports only the pure date format (w/o timestamp). Classes such as `java.util.Date` always contain a timestamp. Thus, it is urgently recommended always to create a date with the type `java.sql.Date` as these objects only contain a date. Other values are currently tolerated by d.ecs forms in this version but are acknowledged with a warning in the log. To prevent type conflicts in future versions of d.ecs forms, these scenarios should be avoided.

1.1.6. Communication with databases

In case that you are communicating with database systems in your Groovy scripts, which are not equivalent to the DBMS of the d.ecs forms system then you must specify separate JDBC driver archives for this effect. The respective JAR files must be stored in the d.ecs forms installation under **`<d3presenactionserver>/forms/lib/custom`**. Having saved it, you must restart the d.ecs form server.

1.2. Description of the server-side scripting-methods

The available methods will be described in detail in the later chapters of the manual.

Note

Please also note that the client-scripting is supplemented by a client-side scripting. Detailed information on this subject can be found in the respective documentation.

1.2.1. Sub-object context

Attention:

The methods of the sub-object 'Context' may only be used with the d.ecs forms object model to access the object located within the respective object model. It is not possible to access objects from an external form-context.

setValue(...)

The function `setValue(...)` allows to change the values of a property in the form-context.

Signature:		

`setValue( path, value )`
 	
`setValue( path, index, value )`		

Import parameters		
`path`	String	The name of the property in the form-context
`index`	Integer	Index of the value in the list.
`value`	Object	For `setValue(name,value)`, the respective value is written to the respective property of the form-context. The type of the value must match that of the configured type in the data model. (`String, Number, Date, …`)
		For `setValue(name, index, value)`, the value is updated at the respective position of the list.

```
/*

Use-case Description:

In the following scenario we have a small animal farm. The animal types
are stored in the data property "farm.animals". In this property we have
the values "cat", "dog" and "horse". Now the "horse" died and we have a
new animal "bigfoot". So we can easily replace the animal type.

*/

def deadAnimal = "horse";
def newAnimal  = "bigfoot";

def animals = form.context.getValues( "farm.animals" );

if( animals && animals.size() > 0 ){

   /* Get the position of the dead animal */
   def index = -1;
   def max = animals.size();
   for( def i=0; i < max; i++ ){
      if( animal.get(i).equals( deadAnimal ) ){
         index = i;
         break;
      }
   }

   if( index != -1 ) {
      /* We can replace the dead animal */
      form.context.setValue( "farm.animals", index, newAnimal );
   } else {
      /* The dead animal was not stored; add new animal */
      form.context.setValue( "farm.animals", animals.length, newAnimal );
   }

}
```

```
/*

Use-case Description:

Now we want to manage a list with customers and their addresses. In the
user interface we have a table with the following columns: CustomerId,
Name, Street, Postal Code, City. Our data property is named "Customers".

Our table looks like this:

CustomerId | Name         | Street           | PostalCode  | City

1            Mrs. Miller    Am Davos 3         48712         Gescher
2            Mr. Hendricks  Hofstrasse 2       48712         Gescher
3            Mrs. Bean      Am Ententeich 3    48712         Gescher

We want to make two modifications:

(1) Update the street of Mr. Hendricks
(2) Add a new customer (Mr. Bill)

*/

/* First, update the street from Mr. Hendricks */
form.context.setValue( "Customers.Street", 1, "Schildarp 6" );

/* Now, we want to add a new customer */

def rowCount = form.context.getValues( "Customers.Id" ).size();

/* Note that the row count is the correct index for the new value. This
   is because our list starts with the index 0. */
form.context.setValue( "Customers.Id", rowCount, "4" );
form.context.setValue( "Customers.Name", rowCount, "Mr. Bill" );
form.context.setValue( "Customers.Street", rowCount, "Am Davos 4" );
form.context.setValue( "Customers.PostalCode", rowCount, "48712" );
form.context.setValue( "Customers.City", rowCount, "Gescher" );
```

Attention:

If you want to process large amounts of data with multiple values with an index, then using this method can lead to performance issues. In this case, it is recommended to compile the values in a collection first and then write them to the data model with setValues(..) in a batch.

setValues(...)

The function `setValue(...)` allows to change the values of a property in the form-context.

Signature:		
`setValues( path, values )`		
Import parameters		
`path`	String	The name of the property in the form-context
`values`	List<Object>	The values as a list

```
/*

Use-case Description:

In the following scenario we have a small animal farm. The animal types
are stored in the data property "farm.animals". In this property we have
the values "cat", "dog" and "horse". Due to a horrible natural disaster all
animals died. So we have to "buy" new animals.

*/

def newAnimals = ["bigfoot", "yeti", "donkey"];
form.context.setValues( "farm.animals", newAnimals );
```

addValue(...)

The function `addValue(...)` adds a value to a property in the form-context. Existing values are moved back by one position starting from the specified index.

Signature:		

`addValue( path, value )`
`addValue( path, index, value )`		

Import parameters		
`path`	String	The name of the property in the form-context
`index`	Integer	Index at which the value is to be added. If not specified, the value is appended to the end of the list.
`value`	Object	The new value

```
/*

Use-case Description:

In the following scenario we have a small animal farm. The animal types
are stored in the data property "farm.animals". In this property we have
the values "cat", "dog" and "horse". Now we have a new animal "bigfoot".
So we can easily add the new animal at the end of the list.

*/

def newAnimal  = "bigfoot";
form.context.addValue( "farm.animals", newAnimal);
```

```
// If we want to add the new animal at the beginning of the list, we can do
the following
form.context.addValue( "farm.animals", 0, newAnimal);
```

getValue(...)

The function `setValue(...)` allows to change the values of a property in the form-context.

Signature:		

`Object getValue( path )`
`Object getValue( path, index )`		

Import parameters		
`path`	String	The name of the property in the form-context
`index`	Integer	Index of the value in the list.
Return value		
		For `getValue( name )`, the respective value of the parameter is returned in the according data type (`String, Number, Date,` …).
		For `getValue(name, index )`, the value is returned at the respective position in the list.

```
/*

Use-case Description:

In the following scenario we have a small animal farm. The animal types
are stored in the data property "farm.animals". In this property we have
the values "cat", "dog" and "horse". Now we want to get the first and
the third animal type.

*/

/* Get the first animal type */
def firstAnimalType = form.context.getValue( "farm.animals" ); /* "cat" */

/* or */
firstAnimalType = form.context.getValue( "farm.animals", 0 );  /* "cat" */

/* Get the third animal type */
def thirdAnimalType = form.context.getValue( "farm.animals", 2 ); /*
"horse" */
```

getValues(...)

The function `getValues(...)` allows to change the values of a property in the form-context.

Signature:		
`List<Object> getValues( path )`		
Import parameters		
`path`	String	The name of the property in the form-context
Return value		
	List<Object>	List of the values

```
/*

Use-case Description:
```

```
In the following scenario we have a small animal farm. The animal types are
stored in the data property "farm.animals". In this property we have the
values "cat", "dog" and "horse". We want to get all animal types an show
them in a popup.

*/

/* Get all animal types on the farm */
def animals = form.context.getValues( "farm.animals" );

if( animals && animals.size() > 0 ){

   def popupString = "";

   for( def animal in animals){
      popupString = popupString + animal + ",";
   }

   def msg = "On the farm live the following animals: " +
popupString.subString( 0, popupString.length() - 1 );
   form.notification.info( msg );
}
```

deleteValue(...)

deleteValue(...)

The function `deleteValue(...)` allows to delete the values of a property in the form-context.

Signature:		
`deleteValue( path )`		
`deleteValue( path )`		
Import parameters		
`path`	String	The name of the property in the form-context
`index`	Integer	Index of the value in the list.

```
/*

Use-case Description:

In the following scenario we have a small animal farm. The animal types
are stored in the data property "farm.animals". In this property we have
the values "cat", "dog" and "horse". Due to a horrible natural disaster
all horses died.

*/

form.context.deleteValue( "farm.animals", 2 );
```

deleteValues(...)

The function `deleteValues(...)` completely clears all values of a property in the form-context.

Signature:		
`deleteValues( path )`		
Import parameters		
`path`	String	The name of the property in the form-context

```
/*

Use-case Description:

In the following scenario we have a small animal farm. The animal types
are stored in the data property "farm.animals". In this property we have
the values "cat", "dog" and "horse". Due to a horrible natural disaster
all animals died.

*/

form.context.deleteValues( "farm.animals" );
```

getObjectProperties(...)

The function `getObjectProperties(...)` allows you to determine the metadata of all properties for a specific object (or all objects).

Signature:		

`Map<String, PropertyMetaData> getObjectProperties( )`
 	
`Map<String, PropertyMetaData> getObjectProperties( objectName )`		

Import parameters		
objectName	String	Name of the object (optional).
Return value		
	Map<String, PropertyMetaData>	List of determined properties. The key is in the form `Object.Property` (e.g. `"Customer.Nr"`)

Attention:

In the object model, only the call of `ds.context.getObjectProperties( objectName )` with specific object names is supported. A call without object name (even a call within server-actions) only considers the objects, which exist directly in the respective form-context or have been imported from the object model.

```
/*

Use-case Description:

We want to detect all properties of the object "Customer" in order to list
them.

*/

def props = form.context.getObjectProperties( "Customer" );

if( props ){

   props.each{ name, metadata ->
      form.log.info( "Property ${name} found. Multivalue: $
{metadata.isMultiValue()}" );
```

```
   }
}
```

PropertyMetaData

getName()

The method determines the property name

Signature:	
`getName()`	
Return value	
String	The name of the property in the notation "Object.Property"

isMultiValue()

The method determines, if the property is a multi-value property

Signature:	
isMultiValue()	
Return value	
boolean	`true`: Multi-value `false`: Single-value

1.2.2. Sub-object params

params.getValue(...)

The function `getValue(...)` allows to read the value of a direct script-parameter.

Signature:		

`Object getValue( name )`
`Object getValue( name, index )`		

Import parameters		
`name`	String	Name of the parameter. This was assigned either assigned within a data source (**data-object**) or directly passed in the **client-script**. When for example using it in combination with an auto-complete function, the parameter `in` is set implicitly, which is here equivalent to the current entry in the **auto-complete-field**.
`index`	Integer	Index of the value in the list.
Return value		
		For `getValue( name )`, the respective value of the parameter is returned in the according data type (`String, Number, Date,` ...).
		For `getValue(name, index )`, the value is returned at the respective position in the list.
Field of application		
The function serves to handover parameters in the global data objects and the server actions.		

Note

The parameter names `type` and `method` can currently not be used.

```
def loadData( def form ) {
   def a = form.params.getValue( "a" );
   // ...
   return null;
}
```

params.getValues(...)

The function `getValues(...)` allows to read the values of a direct script-parameter.

Signature:		
`List<Object> getValues(name)`		
Import parameters		
`name`	String	Name of the parameter. This was assigned either assigned within a data source (**data-object**) or directly passed in the **client-script**.
Return value		
	`List<Object>`	List of the values
Field of application		
The function serves to handover parameters in the global data objects and the server actions.		

Note

The parameter names `type` and `method` can currently not be used.

```
def loadData( def form ) {
   def a = form.params.getValues( "a" );
   // ...
   return null;
}
```

1.2.3. Sub-object data

data.clear()

The function `clear()` deletes all existing values of a data source. This should usually always be called at the beginning of a loadData-method. If you did not call it, you would basically return the result of the previous call.

Signature:
`clear()`
Field of application
If `clear()` is not called, loading data in portions (e.g. paging) can be implemented and accelerated on the server.

```
def loadData( def ds ) {
   ds.data.clear();
   def param1 = ds.params.getValue( "parameter1" );

   ds.data.setValue( "result1", "Param1 was '${param1}'" );
   return null;
}
```

data.addValue(...)

The function `addValue(...)` adds a value to a column of the result table. Existing values are moved back by one position starting from the specified index.

Signature:		

`addValue( path, value )`
 	
`addValue( path, index, value )`		

Import parameters		
`path`	String	The name of the column
`index`	Integer	Index at which the value is to be added. If not specified, the value is appended to the end of the list.
`value`	Object	The new value

```
def loadData(def ds) {
   def param1 = ds.params.getValue( "parameter1" );

   ds.data.addValue( "result1", "Param1 was '${param1}'" );
   ds.data.addValue( "result2", 2 );
   return null;
}
```

data.setValue(...)

The function `setValue(...)` passes a return value back to d.ecs forms.

Signature:		

`setValue( name,value )`
 	
`setValue( name, index, value )`		

Import parameters		
`name`	String	The name of the return value. The name must have been previously configured within a data source together with the data type.
`index`	Integer	The position in the list, if it is a list.
`value`	Object	The value. The data type of the value (`String`, `Number`, `Date`, ...) must match that of the configured type in the data model.
Field of application		
Is mainly used to pass the values to the data model. But is also used with an auto-complete function. There, the `id` and the `caption` (both in a list) are returned via `setValue`.		

```
def loadData(def ds) {
   def param1 = ds.params.getValue( "parameter1" );

   ds.data.setValue( "result1", "Param1 was '${param1}'" );
   ds.data.setValue( "result2", 2 );
   return null;
}
```

Attention:

If you want to process large amounts of data with multiple values with an index, then using this method can lead to performance issues. In this case, it is recommended to compile the values in a collection first and then write them to the data model with setValues(..) in a batch.

data.setValues(...)

The function `setValues(...)` passes the return values back to d.ecs forms.

Signature:		
`setValues( name, value )`		
Import parameters		
`name`	String	The name of the return value. The name must have been previously configured within a data source together with the data type.
`values`	List<Object>	The values. The data type of the value (`String`, `Number`, `Date`, ...) must match that of the configured type in the data model.
Field of application		
Is mainly used to pass the values to the **data model**. But is also used with an **auto-complete function**. There, the `id` and the `caption` (both in a list) are returned via `setValues`.		

```
def loadData(def ds) {
   def param1 = ds.params.getValue( "parameter1" );

   ds.data.setValues( "resultList", ["a","be","ce"] );
   return null;
}
```

data.addValue(...)

The `addValue(...)` function adds a value to the return value that is returned to d.ecs forms. Existing values are moved back by one position starting from the specified index.

Signature:		

`addValue( name, value )`
 	
`addValue( name, index, value )`		

Import parameters		
`name`	String	The name of the return value. The name must have been previously configured within a data source together with the data type.
`index`	Integer	Index at which the value is to be added. If not specified, the value is appended to the end of the list.
`value`	Object	The new value The data type of the value (`String`, `Number`, `Date`, ...) must match that of the configured type in the data model.
Field of application		
Is mainly used to pass the values to the data model. But is also used with an auto-complete function. There, the `id` and the `caption` (both in a list) are returned via `addValue`.		

```
def loadData(def ds) {
   ds.data.setValue( "result", 0, "value0");
   // ds.data.getValues( "result") already contains a value "value0" at
position 0
   // Now we add two additional values. One at the beginning and one at the
end of the list
   ds.data.addValue( "result", 0, "beginning");
   ds.data.addValue( "result", "end");

   return null;
}
```

data.getValue(...)

The function `getValue(...)` reads a value from the cache of the data sources, which was previously written to this data source on loading it.

Signature:		

`Object getValue( name )`
 	
`Object getValue( name, index )`		

Import parameters		
`name`	String	The name of the return value. The name must have been previously configured within a data source together with the data type.
`index`	Integer	The position in the list, if it is a list.
`value`	Object	The value.
Return value		
		For `getValue( name )`, the respective value of the parameter is returned in the according data type (`String, Number, Date, ...`).
		For `getValue(name, index )`, the value is returned at the respective position in the list.

Signature:		
		The data type of the value (`String`, `Number`, `Date`, ...) must match that of the configured type in the data model.
Field of application		
The function can currently only be used within data sources.		

```
def saveData(def ds) {
   def r1 = ds.data.getValue( "result1" );
   def r2 = ds.data.getValue( "result2" );

   // write data to other systems
   // ...

   return null;
}
```

data.getValues(...)

The function `getValues(...)` reads all values from the cache of the data sources, which were previously written to this data source on loading it.

Signature:		
`Object getValues( name )`		
Import parameters		
`name`	String	The name of the return value. The name must have been previously configured within a data source together with the data type.
Return value		
`List<Object>`		List of the values
Field of application		
The function can currently only be used within data sources.		

```
def saveData( def ds ) {
   def rl = ds.data.getValues( "resultList" ); // Liste

   // write data to other systems
   // ...

   return null;
}
```

data.deleteValue(...)

deleteValue(...)

The function `deleteValue(...)` allows to delete the values of a property in the result table.

Signature:		
`deleteValue( path )`		
`deleteValue( path )`		
Import parameters		
`path`	String	The name of the column
`index`	Integer	Index of the value in the list.

```
def loadData(def ds) {
   // First: Remove first value of columnA for some reason
   ds.data.deleteValue( "columnA", 0 );
   // Then: Continue with loading new values
   ds.data.addValue( "columnA", "someValue" );

   return null;
}
```

data.deleteValues(...)

The function `deleteValues(...)` allows to delete all values of a property in the result table.

Signature:		
`deleteValues( path )`		
Import parameters		
`path`	String	The name of the column

```
 def loadData(def ds) {
   // First: Remove all values of columnA for some reason
   ds.data.deleteValues( "columnA" );

   // Then: Continue with loading new values
   ds.data.addValue( "columnA", "someValue" );

   return null;
}
```

data.getTable()

The function `ds.data.getTable()` allows you to get a tabular view of all existing values of the data in a data source.

This function should only be used, if all data contained in ds.data have the same length.

Signature:	
`getTable()`	
Return value	
`Table`	The content of all properties as a table.
Field of application	
If you want or need to access data based on columns instead of rows.	

```
def saveData(def ds) {
        for( row in ds.data.getTable()) {
                def rowNumber = row.getRowNumber()
                def name = row.getValue( "name") // identical to
ds.data.getValue("name",rowNumber)
                def fistName = row.getValue( "firstName") // firstName
corresponds to name
                //...
        }
   return null;
}
```

1.2.4. Sub-object misc

The sub-object **misc** contains further sub-objects:

- formParams

Sub-object formParams

getInstanceId(...)

The function `getInstanceId()` allows you to read the current form-instance-ID.

Signature:	
`String getInstanceId( )`	
Return value	
`String`	Form-instance-ID

```
def doSomething( def form ) {
   def formInstId = form.misc.formParams.getInstanceId();
   // ...
   return null;
}
```

request.getValue(...)

`request.getValue...)` allows to read a form parameter. Usually this is a request parameter that has been used to call the form.

Signature:		
`String getValue( parameterName )`		
Import parameters		
`parameterName`	String	Name of the parameter.
Return value		
	String	Value of the parameter as a string.
Field of application		
Reading any form parameter.		

```
/*

Use-case Description:

Each time we open a new form, an url parameter will be attached.

In our script code we want to access this parameter.

*/

/* The url of the form is "http://[...]&source=DB1[...]" */

def source = form.misc.formParams.request.getValue( "source" ); /* "DB1" */
```

Note

This method always returns values of the type `String`. To get typed values, you must use a typed object property from the data model which is reading the request parameter. Please refer to the d.ecs forms manual for this effect.

request.getValues(...)

`request.getValues(...)` allows to read a form parameter (with multiple values). Usually this is a HTTP-request parameter that has been used to call the form.

Signature:
`List<String> getValues( parameterName )`
Import parameters

Signature:		
`parameterName`	String	Name of the parameter.
Return value		
	List<String>	The values of the parameter as a string.
Field of application		
Reading any form parameter.		

```
/*

Use-case Description:

Each time we open a new form, an url parameter will be attached.

In our script code we want to access this parameter.

Maybe there are multiple values for one parameter.

*/

/* The url of the form is "http://[...]&source=DB1[...]&source=DB2[...]" */

def sources = form.misc.formParams.request.getValues( "source" ); /*
["DB1", "DB2"] */
```

Note

This method always returns values of the type `List<String>`. To get typed values, you must use a typed object property from the data model which is reading the request parameter. Please refer to the d.ecs forms manual for this effect.

i18n.getValue(...)

`i18n.getValue(...)` allows you to load a translation form the system.

Signature:		
`String getValue( key )`		
Import parameters		
key	String	The key of the translation
Return value		
	String	The translation
Field of application		
Reading a translation		

```
def doSomething( def form ) {
   def translation = form.misc.i18n.getValue( "key" );
   // ...
   return null;
}
```

login.getUsername(...)

The function `login.getUsername()` allows you to read the name of the user currently logged in.

Note

This does not necessarily have to be a d.3 user name. With a login of the form "MyDomain\User", this is also the return value of the function.

When using d.ecs identity provider, this behavior may differ depending on the configuration of the LDAP user provider.

Signature:	
`String getUsername( )`	
Return value	
`String`	Name of the currently logged in user

```
/*

Use-case Description:

In one workflow step a user has to give the green light to do something. We
want to log his decision and his username. That is why we need the current
user name.

*/

def usr = form.misc.login.getUsername();

/* Now we save the user name in a disabled input field ("input") */
form.context.setValue( "user", usr );
```

1.2.5. Sub-object log

debug(...)

The function `debug(...)` writes an entry with the loglevel "Debug" to the d.ecs forms server log.

Debug is not logged in the default setting. This must be enabled manually.

Signature:		
`debug( msg )`		
Import parameters		
`msg`	String	Message text to be written to log
Field of application		
Debug should only be used to identify problems.		

```
/*

Use-case Description:

We have a complex script and want to follow the script processing line by
line. So we use debug messages which will only be shown in debug mode.

*/

/* [...] */
form.log.debug( "I am a debug message" );
/* [...] */
```

info(...)

The function `info(...)` writes an entry with the log level “`Info`“ to the d.ecs forms server log.

Signature:		
`info( msg )`		
Import parameters		
`msg`	String	Message text to be written to log

Signature:
Field of application
Info should be used to log general information. You should not log any parameters for a potential troubleshooting here.

```
/*

Use-case Description:

We have some events which should be logged for information (e. g.
successful save).

*/

/* [...] */
form.log.info( "I am an info message" );
/* [...] */
```

warn(...)

The function `warn(...)` writes an entry with the loglevel `Warn` to the d.ecs forms server log.

Signature:		
`warn( msg )`		
Import parameters		
`msg`	String	Message text to be written to log
Field of application		
Warn should be used if an issue arises.		

```
/*

Use-case Description:

We have some events which are perhaps not ok. As a consequence we do not
want to throw an error but we want to warn.

*/

form.log.warn( "I am a warning" );
```

error(...)

The function `error(...)` writes an entry with the loglevel `Error`" to the d.ecs forms server log.

Signature:		
`error( msg )`		
Import parameters		
`msg`	String	Message text to be written to log
Field of application		
Error should be used after an issue has occurred		

```
/*

Use-case Description:

Something went wrong and we want to throw an error. Notice that this
function only writes an error into the log.
```

```
*/

form.log.error( "I am an error message" );
```

1.2.6. Sub-object engine

Sub-object sql

execute(...)

Executes any SQL-statement.

Signature:		
`execute( alias, query, parameter )`		
Import parameters		
`alias`	String	The name of the configured database connection.
`query`	String	The SQL-statement.
`parameter`	List<Object> *alternatively:* Map	A list of the parameters of the SQL-statement. A map of the parameters of the SQL-statement.
Field of application		
The function executes any SQL-statement, but does not provide any options for the interpretation of the result. It is thus only useful for writing operations. The data processing should take place with another method (such as executeAndStore or executeAndGet).		

```
def deleteKunde( def form ) {
   def nr = form.params.getValue( "nr" );
   def query = "DELETE FROM Kunden WHERE nr=?";
   form.engine.sql.execute( "dbAlias", query, [nr] );
   return null;
}

def deleteKunde( def form ) {
   def query = "DELETE FROM Kunden WHERE nr=:nr";
   form.engine.sql.execute( "dbAlias", query, [nr : form.params.getValue(
"nr" )] );
   return null;
}
```

executeAndStore(...)

Typically executes an SQL `Select`-statement and adopts the result of the SQL-statement directly in the cache of the data source.

The previous result of the data source is deleted in this process. It does not need to be cleared with `ds.data.clear()`.

The column names of the database table are automatically mapped to the names of the return values of the data source.

Note
In an auto-complete function, it is recommended to work with the SQL-keyword `as` to rename the return values for the auto-complete etc. to `id` and `caption`. Especially in an Oracle-database, the name `id` and `caption` must be written in quotation marks (see example below) to enforce the lower-case spelling.

Signature:		
`executeAndStore( alias, query, parameter )`		
Import parameters		
`alias`	String	The name of the configured database connection.
`query`	String	The SQL-statement.
`parameter`	List<Object>	A list of the parameters of the SQL-statement.
	alternatively:	A map of the parameters of the SQL-statement.
	Map	
Field of application		
The function executes any SQL statement and adopts the result of the SQL-statement directly in the cache of the data source.		

```
def loadData( def form ) {
   def name = form.params.getValue( "in" );

   def query = """\
      SELECT Knr as "id", Nachname as "caption"
      FROM Benutzer
      WHERE Nachname=?
      """;

   def params = [name];
   form.engine.sql.executeAndStore( "dbAlias", query, params );
   return null;
}

def loadData( def form ) {
   def query = """\
      SELECT Knr as "id", Nachname as "caption"
      FROM Benutzer
      WHERE Nachname=:name
      """;

   def params = [name : form.params.getValue( "in" )];
   form.engine.sql.executeAndStore( "dbAlias", query, params );
   return null;
}
```

executeAndGet(...)

The method `executeAndGet(...)` executes an SQL-statement and allows to manually interpret the result.

Signature:		
`executeAndGet( alias, query, parameter )`		
Import parameters		
`alias`	String	The name of the configured database connection.
`query`	String	The SQL-statement.
`parameter`	List<Object>	A list of the parameters of the SQL-statement.
	alternatively:	A map of the parameters of the SQL-statement.
	Map	
Return value		

Signature:		
	List<GroovyRowResult>	A list in which every list entry is equivalent to one row of the result. The individual values of this row have to be addressed directly (differing from other cases), i.e. not using `row.getValue("name")` but `row.name` (see example).
Field of application		
Use this method for SELECT-statements, if you do not want to adopt their result into the data source automatically and edit it before, instead.		

```
def loadData( def form ) {
   def name = form.params.getValue( "in" );
   def query = """
      SELECT KNr, Nachname, Vorname, Strasse, Plz, Ort
      FROM Benutzer WHERE Nachname=?
   """;

   def params = [name];
   def rows = form.engine.sql.executeAndGet( "dbAlias", query, params );

   form.data.clear();

   def ids = [], captions = []
   for( row in rows ) {
      ids.add(row.KNr)
      captions.add(row.Nachname + ", " + row.Vorname)
   }

   form.data.setValues( "id", ids);
   form.data.setValues( "caption", captions);

   return null;
}


def loadData( def form ) {
   def query = """
      SELECT KNr, Nachname, Vorname, Strasse, Plz, Ort
      FROM Benutzer WHERE Nachname=:name
   """;

   def params = [name : form.params.getValue( "in" )];
   def rows = form.engine.sql.executeAndGet( "dbAlias", query, params );

   form.data.clear();

   def ids = [], captions = []
   for( row in rows ) {
      ids.add(row.KNr)
      captions.add(row.Nachname + ", " + row.Vorname)
   }

   form.data.setValues( "id", ids);
   form.data.setValues( "caption", captions);

   return null;
}
```

executeInsert(...)

The method `executeInsert(...)` typically executes an SQL `Insert`-statement. In this process, the statement may return created IDs.

Signature:		
`executeInsert( alias, query, parameter )`		
Import parameters		
`alias`	String	The name of the configured database connection.
`query`	String	The SQL-statement.
`parameter`	List<Object> *alternatively:* Map	A list of the parameters of the SQL-statement. A map of the parameters of the SQL-statement.
Return value		
	List<List<Object>>	If the statement creates values (`Auto-Increment`, etc.), then these can be read here.
Field of application		
Executes an SQL `Insert`-statement. In this process, the statement may return created IDs.		

```
def doCreate( def form ) {
        def name = form.params.getValue( "name" );
        def firstname = form.params.getValue( "firstname" );
        def city = form.params.getValue( "city" );
        def query = "INSERT INTO USERS VALUES(?,?,?)";
        def params = [name,vorname,city];
        def keys = form.engine.sql.executeInsert( "dbAlias", query, params
);
        def id = keys[0][0];
        return null;
}


def doCreate( def form ) {
        def paramName = form.params.getValue( "name" );
        def paramFirstname = form.params.getValue( "firstname" );
        def paramCity = form.params.getValue( "city" );
        def query = "INSERT INTO USERS VALUES(:name,:firstname,:city)";
        def params = [name : paramName , vorname : paramFirstname , city :
paramCity ];
        def keys = form.engine.sql.executeInsert( "dbAlias", query, params
);
        def id = keys[0][0];
        return null;
}
```

executeUpdate(...)

The method `executeUpdate(...)` typically executes an SQL `Update`-statement and returns the number of rows changed by the update.

Signature:		
`executeUpdate( alias, query, parameter )`		
Import parameters		
`alias`	String	The name of the configured database connection.
`query`	String	The SQL-statement.

Signature:		
`parameter`	List<Object>	A list of the parameters of the SQL-statement.
	alternatively:	A map of the parameters of the SQL-statement.
	Map	
Return value		
	Integer	Returns the number of updated rows.
Field of application		
Executes an SQL `Update`-statement and returns the number of rows changed by the update.		

```
def doUpdate( def form ) {
    def nr = form.params.getValue( "nr" );
        def name = form.params.getValue( "name" );
        def query = "UPDATE Customers SET name=? WHERE nr=?";
        def updateRows = form.engine.sql.executeUpdate( "dbAlias", query,
[name,nr] );
        def ok = updateRows > 0;
        return null;
}

def doUpdate( def form ) {
    def paramNr = form.params.getValue( "nr" );
        def paramName = form.params.getValue( "name" );
        def query = "UPDATE Customers SET name=:name WHERE nr=:nr";
        def updateRows = form.engine.sql.executeUpdate( "dbAlias", query,
[name : paramName, nr : paramNr] );
        def ok = updateRows > 0;
        return null;
}
```

createSQLConnection(...)

The method `createSQLConnection(...)` provides a Groovy-SQL-object based on the passed data-base connection.

Signature:		
`createSQLConnection( alias )`		
Import parameters		
`alias`	String	The name of the configured database connection.
Return value		
	groovy.sql.Sql	Groovy-SQL-object
Field of application		
It is recommended, to work directly on the Groovy-SQL-object, if no native d.ecs forms functions exist in the section `form.engine.sql` for the desired use case.		

Attention:

Please note that created SQL-objects must be closed after use using the `close()`-method. This is **NOT** done by the d.ecs forms system.

```
def doUpdate( def form ) {
   def sql = form.engine.sql.createSQLConnection( "dbAlias" );
   try{
      def result = sql.execute( "SELECT * FROM Customers;" );
   } finally {
      sql.close();
   }
```

```
    return null;
}
```

Sub-object d3

The **server-side scripts** allow to access the d.3 API.

There are two approaches

1. direct access
2. access via simplified methods

The **direct access** supports the d.3 API functions listed in the appendix.

Attention:

The **direct access** thus allows you to access all public functions of the d.3 API listed in the appendix of this documentation. The documentation of those d.3 API-functions can be downloaded from the d.velop service portal, whereby this requires a non-disclosure agreement to be signed. Additionally, it is recommended to attend a training for the d.3 administration. Please contact our Technology Partner Management of the d.velop AG (d.velop.competencenetwork@d-velop.de).

For some standard tasks **simplified methods** are provided. These encapsulate the partially tedious handling with parameters.

Note

If you now operate a script there and the form is called from within the d.3 workflow, the login-information (d.3 system and d.3 user) are automatically adopted from the d.3 client. Specifying a d.3 alias-object is then only required, if the d.3 API calls are to be explicitly executed under another than the logged in user.

Direct access

In the server-side Groovy-scripts, the d.3 API can be directly accessed. For this effect, a central object must be created via which the access is performed.

Access from form context

```
def d3api = form.engine.d3.createD3Api( );
```

The object is configured in the same way as generic d3fc functions are called.

Example:

SearchDocument

```
def d3api = form.engine.d3.createD3Api( );
d3api.setConnectionAlias( "d3-Connection" ); // Optional
d3api.setFunctionName( "SearchDocument" );
d3api.setImportParams( ["doc_type_short" : "TEST", "doc_field1" : "value of doc_field1"] );
d3api.execute( );

if( d3api.returnCode != 0 ) {
   throw new Exception( "An error occured. Message: " + d3api.returnMessage );
}

def searchResult = d3api.getExportTable( );
for( document in searchResult )
   def docId = document.getValue( "doc_id" );
   // ...
}
```

GetDocumentList

```
def d3api = action.engine.d3.createD3Api();
d3api.setFunctionName( "GetDocumentList");
d3api.setImportParams( [
        "AttributeName[1]"   : "doc_status",      "AttributeValue[1]" : "Fr",
        "AttributeName[2]"   : "doc_type_short", "AttributeValue[2]" : "D",
        "OutColumnName[1]"   : "doc_id",
        "SortName[1]"        : "doc_id",          "SortDirection[1]"  : "DESC"
])
d3api.execute();

if( d3api.returnCode != 0 ) {
   throw new Exception( "An error occured. Message: " + d3api.returnMessage );
}

def searchResult = d3api.getExportTable( );
for( document in searchResult )
   def docId = document.getValue( "doc_id" );
   // ...
}
```

SetUserVariablesInWorkPath

```
def d3api = form.engine.d3.createD3Api( );
d3api.setConnectionAlias( "d3-Connection" ); // Optional
d3api.setFunctionName( "SetUserVariablesInWorkPath" );
d3api.setImportParams( ["doc_id" : "P000000001"] );

def importTable = d3api.getImportTable();
importTable.setValue( "var_name", 0, "Variable 1");
importTable.setValue( "var_value", 0, "Value of Variable 1");
importTable.setValue( "var_name", 1, "Variable 2");
importTable.setValue( "var_value", 1, "Value of Variable 2");

d3api.execute( );

if( d3api.returnCode != 0 ) {
   throw new Exception( "An error occured. Message: " + d3api.returnMessage );
}
```

Simplified access

Apart from the direct access to the d.3 API, d.ecs forms also allows a simplified access. These are methods encapsulating more complex d.3 API calls. This allows you to execute d.3 API functions on the d.3 server without deeper knowledge of the functions.

These methods directly execute the d.3 function on the d.3 server and return an exception with the message of the d.3 server depending on the return code of the d.3 server. The following versions are currently supported:

Method	d.3 API-function
callGetDocumentTypeTitleAll	GetDocumentTypeTitleAll
GetMultipleAttributes	GetMultipleAttributes
callGetValidValuesForAttribute	GetValidValuesForAttribute
callImportDocument_TextDocument	ImportDocument (with a text document)
callReceiveNote	ReceiveNote
callSearchDocument	SearchDocument
callSendNote	SendNote
callUpdateAttributes	UpdateAttributes
callPutDocumentIntoWorkPath(...)	PutDocumentIntoWorkPath

For details please on the d.3 API functions please consult the documentation of the d.3 API.

callGetDocumentTypeTitleAll

The method determines all properties of a document type.

Signature:		
`callGetDocumentTypeTitleAll( docTypeShort )`		
Import parameters		
`docTypeShort`	String	Short name of the document type whose attributes are to be determined
Return value		
	Table	The expoort table with reference to the d.3 API function GetDocumentTypeTitleAll
Field of application		
The function is available in server-side Groovy-scripts in the form-context in the d.ecs forms object model.		

```
def d3api = ds.engine.d3.createD3Api( );
def exportTable = d3api.callGetDocumentTypeTitleAll( "TEST" );

// If the expected export table has several rows...
for( docType in exportTable ){
   def docTypeLong = docType.getValue( "doc_type_long" );
}

// ...or if only one row is expected
if( exportTable.size() > 0 ){
   def docTypeLong = exportTable.getValue( "doc_type_long", 0);
}
```

GetMultipleAttributes

The method determines the attributes with multiple instantiations for a given document.

Signature:		
`callGetMultipleAttributes( docId)`		
Import parameters		
`docId`	String	The document-ID for whose document the multi-value fields are to be determined
Return value		
	Table	A table equivalent to a 60-field table. Accordingly, for each multivalue-field, one column exists in the table.
	Exception	If return code does not equal 0
Field of application		
The function is available in server-side Groovy-scripts in the form-context in the d.ecs forms object model.		

```
def d3api = ds.engine.d3.createD3Api( );
def exportTable = d3api.callGetMultipleAttributes( "P000000001" );
```

```
for( row in exportTable ){
   def row_number = row.getRowNumber();
   def value60 = row.getValue( "doc_field_60" );
   def value61 = row.getValue( "doc_field_61" );
}
```

callGetValidValuesForAttribute

The method determines the valid value range for one or all document type-specific attribute(s) for a given document type if there are restrictions specified.

Signature:		
`callGetValidValuesForAttribute( importParams)`		
Import parameters		
`importParams`	Map<String, Object>	The import-parameters with reference to the d.3 API function GetValidValuesForAttribute
Return value		
	Table	A table, in which each doc-field is represented by an individual column.
	Exception	If return code does not equal 0
Field of application		
The function is available in server-side Groovy-scripts in the form-context in the d.ecs forms object model.		

```
def d3api = ds.engine.d3.createD3Api( );
def exportTable = d3api.callGetValidValuesForAttribute( [
"document_type_short" : "TEST", "doc_field[2]" : "Value of doc_field[2]"]
);

// Access doc field values
def values_doc_field_3 = exportTable.getValues( "doc_field_3" );
values_doc_field_3.each{ value ->

   // Assumption: doc_field_3 has 'Date' values
   // Format Date to String
   def date_as_string = value.format( "dd.mm.yyyy" );
   // Format String back to Date
   def value = Date.parse( "dd.mm.yyyy", date_as_string );
}

def values_doc_field_4 = exportTable.getValues( "doc_field_4" );
values_doc_field_4.each{ value ->
   // Do something else with the value
}
```

callImportDocument_TextDocument

The method stores a text document in the **d.3** archive.

Signature:		
`callImportDocument_TextDocument( importParams, textDocument)`		
Import parameters		
`importParams`	Map<String, Object>	The import-parameters with reference to the d.3 API function ImportDocument
`textDocument`	String	The content of the document
Return value		
	String	The document-ID of the stored document
	Exception	If return code does not equal 0

Signature:
Field of application
The function is available in server-side Groovy-scripts in the form-context in the d.ecs forms object model.

```
def d3api = ds.engine.d3.createD3Api( );
def importParams = [ "doc_type_short" : "TEST", "doc_status" : "Fr",
"doc_field[1]" : "Value of doc_field[1]", "doc_field[2]" : "Value of
doc_field[2]"];
def content = "This is the document´s content";
def docId = d3api.callImportDocument_TextDocument( importParams, content );
```

callPutDocumentIntoWorkPath(...)

The method starts a d.3 workflow and thus allows to specify workflow variables.

Signature:		
`callPutDocumentIntoWorkPath(docId, wflId, wflVars)`		
Import parameters		
`docId`	String	The d.3 document ID
`wflId`	String	The d.3 workflow ID (process ID)
`wflVars`	Map<String, Object>	The d.3 workflow-variables (or <null>)
Return value		
	void	
	Exception	If return code does not equal 0
Field of application		
The function is available in server-side Groovy-scripts in the form-context in the object model		

```
//###########################################
// Start workflow (d.3 7.2.2 or above)
//###########################################

def formInstanceId = "wfl/${docId}/${UUID.randomUUID()}"
def wflId = "5djh6n426els6a62b4dn5obnf"
def wflVars = ["formInstanceId" : formInstanceId, "wf_var1" : "wf_val1"]

def startWorkflowCall = form.engine.d3.createD3Api()
startWorkflowCall.setConnectionAlias( "d3-Connection")
startWorkflowCall.callPutDocumentIntoWorkPath(docId, wflId, wflVars)
```

callReceiveNote

The method determines the contents of a note for a document.

Signature:		
`callReceiveNote( docId )`		
Import parameters		
`docId`	String	The document-ID of the document
Return value		
	String	The content of the note file
	Exception	If return code does not equal 0 and does not equal 320
Field of application		
The function is available in server-side Groovy-scripts in the form-context in the d.ecs forms object model.		

```
def d3api = ds.engine.d3.createD3Api( );
def note = d3api.callReceiveNote( "P000000001" );
```

callSearchDocument

Based on specified search criteria, this function determines the attributes of the matching documents.

Signature:		
`callSearchDocument( searchParams )`		
Import parameters		
`searchParams`	Map<String, Object>	The import-parameters with reference to the d.3 API function SearchDocument
Return value		
	Table	The expoort table with reference to the d.3 API function SearchDocument
	Exception	If return code does not equal 0 and does not equal 10
Field of application		
The function is available in server-side Groovy-scripts in the form-context in the d.ecs forms object model.		

```
def d3api = ds.engine.d3.createD3Api( );
def searchParams = ["doc_field[1]" : "value of doc_field[1]"]
def searchResult = d3api.callSearchDocument( searchParams );

for( document in searchResult ){
   def docId = document.getValue( "doc_id" );
}
```

callSendNote

Appends the note for a document with a new entry or creates the note, if it does not already exist.

Signature:		
`callSendNote( docId, note )`		
Import parameters		
`docId`	String	The document-ID of the document
`note`	String	Content of the note file to be appended / created
Field of application		
The function is available in server-side Groovy-scripts in the form-context in the d.ecs forms object model.		

```
def d3api = ds.engine.d3.createD3Api( );
d3api.callSendNote( "P000000001", "Notiz" );
```

callUpdateAttributes

Updates the properties of a document.

Signature:		
`callUpdateAttributes( docId, attributes)`		
Import parameters		
`docId`	String	The document-ID of the document to be updated
`attributes`	Map<String, Object>	The import-parameters with reference to the d.3 API function UpdateAttributes
Return value		
	void	
	Exception	If return code does not equal 0
Field of application		
The function is available in server-side Groovy-scripts in the form-context in the d.ecs forms object model.		

```
def d3api = ds.engine.d3.createD3Api( );
def attributes = ["doc_field[1]" : "New value of doc_field[1]",
"doc_field[2]" : "New value of doc_field[2]"]
d3api.callUpdateAttributes( "P000000001", attributes );
```

getDocumentAsStream(...)

This allows you to download a document from d.3. This method expects an OutputStream and writes the d.3 document into it.

Signature:		
`getDocumentAsStream( params , os )`		
Import parameters		
params	Map<String, String>	The parameters used to select the desired document.
os	OutputStream	An OutputStream to which the document is written.
Field of application		
The function is available in server-side Groovy-scripts in the form-context in the d.ecs forms object model.		

The following options can be specified for the parameters:

Name	Meaning	Mandatory parameters
docId	Specifies the document-ID of the document to be opened.	Yes
status	Specifies the status of the document (Be=Processing, Pr=Verification, Fr=Release, Ar=Archive).	No
archiveIndex	For more than one archive-version, you can specify the desired one here. If not specified, the latest Archive-version is selected.	No
dependentDocument	Selects a dependent document instead of the document file (e.g. "t1").	No

Note

The d.3 document is provided immediately on calling the method and the output stream is closed. Calling the d.3 API .execute() is then not necessary.

```
import java.io.*;

def doSomething( def form ){

   def d3api = form.engine.d3.createD3Api();
   d3api.getDocumentAsStream([
      "docId" : "P123456789",
      "status" : "Ar",
      "archiveIndex" : "1",
      "dependentDocument" : "t1"
   ], new FileOutputStream(form.engine.io.getTempFile("outputfile.tif")));

   return null;
}
```

getDocId()

This method determines the d.3 document-ID of the document on which the current form-instance was based.

Signature:	
`String getDocId( )`	
Return value	
String	Currentd.3 document ID.

setDocId()

This method sets the d.3 document-ID of the document on which the current form-instance was based.

Attention:

The passed d.3 document ID must not be `null` or empty. Deselecting a d.3 document later is not possible with this method.

Signature:		
`setDocId( docId )`		
Import parameters		
docId	String	d.3 document ID on which the form-context is to be based

```
/*

Use-case Description:

We import a document and want to use the corresponding doc id from now.

*/

def initContext( def form ){

   // Import the document and get the new docId
   // ...

   def docId = "T000000005";
   form.engine.d3.setDocId( docId );
}
```

readD3DocumentData()

This method updates all properties reading their value from a d.3 document. A d.3 document ID is required to execute this method. This must either be passed initially with a form-URL or later via `form.engine.d3.setDocId()`.

Attention:

Possible values existing locally are thus overwritten.

Signature:
`readD3DocumentData( )`

```
/*

Use-case Description:

We have set a new d.3 document id for this form context and want to read
all referenced values now.

*/

def initContext( def form ){
   def docId = "T000000005";
   form.engine.d3.setDocId( docId );

   form.engine.d3.readD3DocumentData( );
}
```

Sub-object login

getRepositoryId()

This method determines the ID of the d.3 repository on which the current login to the form-call was based.

Signature:	
`String getRepositoryId( )`	
Return value	
String	Current d.3 repository ID.

getUserNameDms()

This method determines the d.3 user name truncated to 10 bytes on which the current login to the form-call was based.

Signature:	
`String getUserNameDms()`	
Return value	
String	Current user name (10 Bytes)

getUserNameDmsLong()

This method determines the d.3 user name with a maximum of 30 characters on which the current login to the form-call was based.

Signature:	
`String getUserNameDmsLong()`	
Return value	
String	Currently d.3 user name (30 characters)

d.3 login via the d.ecs identity provider

From **d.ecs forms** version 3.1, you can authenticate and authorize form accesses via the **d.ecs identity provider** 2.0 (IDP2).

In order to use the **d.3** API in a form athenticated via IDP2, it is required to open the form with the additional repository-ID of the **d.3** systems to be used. The **d.es forms designer** provides a respective option. For this effect, read the chapter Open form in browser in the administration documentation for d.ecs forms.

When using a d.3 API-function, a special login session is negotiated between the **d.ecs forms** server and the **d.3 server** using the current IDP2-session. This requires a d.3 server version 8.1 or higher.

Note

Recommendation:

When using the d.3 API within a form authenticated via the d.ecs identity provider, the first access to the d.3 server should be perfromed as early as possible (e.g. when loading the form).

This prevents a potential issue with expired IDP-tokens as the IDP2-token has a limited lifetime.

Else, if you encounter an issue when generating a d3-API-session, the form must be reloaded afterwards to establish a valid IDP2-session. All unsaved form content is lost in this process.

Sub-object d3fc

In the **server-side scripts**, the d3fc protocol allows you to execute any d3fc function. The d3fc-function in contrast to the d.3 API-functions allow to call any function.

The target server does not necessarily have to be a d.3 Server. Other d3fc-based servers can be addressed. However, it is also possible here to call any customized API function on the d.3 server (from version 8). For more information, refer to chapter "Groovy API-function" in the documentation "d.3 server scripting (groovy)" of the d.3 server.

Generally, only `string` is supported as a data type.

In the server-side Groovy-scripts, the d3fc can be directly accessed. For this effect, a central object must be created via which the access is performed.

Access from form context

```
def d3fc = form.engine.d3fc.createGenericD3FC()
```

This object is the basis for all communication with the d3fc-server. The methods of the d3fc object allow you to configure the call and execute and interpret it. The methods of the *configuration* define the type of the call and the required parameters. They all have to be set before the *execution* of the *execute* method. Only after the *execute*, the methods of the *Interpretation* can be used to process the call result.

Use case

From d.3 version 8.0 the d.3 server provides the option to configure your own d.3 function calls. The example below shows how to call a user-specific d.3 function from d.ecs forms. The example shows how you can work with import and export parameters and with an import and export table. The user-specific d.3 function is also implemented in Groovy but has to be registerd on the d.3 server instead of in d.ecs forms.

Example

```
/*
 * Use-case Description:
 * We want to invoke the custom d.3 function "GetGreetingsForUsers" witch
is implemented in the d.3 server.
 */

/*
 * d.ecs forms server script:
 */
def getGreetingsForUserFromD3Server( def form ) {

   def d3fc = form.engine.d3fc.createGenericD3FC()
   d3fc.setFunctionName("GetGreetingsForUsers")
   d3fc.setImportParams(["greettext": "Hello"])
   d3fc.setImportTable(["name": ["Otto","Lisa"]])
   d3fc.execute()

   def greets = d3fc.getExportTable().getValues("greet")

   form.context.setValues( "UI.greets", greets );

   return null;
}


/*
 * d.3 server script:
 */
import com.dvelop.d3.server.*;
import com.dvelop.d3.server.core.*;

public class GetGreetingsForUsers extends D3ApiCall {

    public int execute(D3Interface d3) {
        def ip = d3.remote.getImportParams()
        def greet = ip.get("greettext");

        def importTable = d3.remote.getImportTable("name")
        def exportTable = [];

        for(row in importTable) {
            def name = row.get("name");
            exportTable.add(["greet":"$greet $name"])
        }

        d3.remote.setExportParams(["count" : exportTable.size()])
        d3.remote.setExportTable(exportTable)
        return 0
    }
}
```

setConnectionAlias(...)

This method sets the connection-alias (from the d.ecs forms object model) to be used for the execution of the d.3 function.

If no connection-alias is set, the d.3 connection from the d.3 authentication is used, if available.

`setConnectionAlias( name )`		
Parameters		
`name`	`String`	Name of the connection-alias

Example

```
def d3fc = form.engine.d3fc.createGenericD3FC()
d3fc.setConnectionAlias( "MyCustomServer")
//...
```

setFunctionName(...) (mandatory)

This method sets the name of the d.3 function to be used with reference to the d.3 API documentation.

Signature:		
`setFunctionName( name )`		
Parameters		
`name`	`String`	Name of the function

Example

```
def d3fc = form.engine.d3fc.createGenericD3FC()
d3fc.setFunctionName( "DoSomething")
//...
```

setImportParams(...)

This method sets the import parameters of the function to be used.

Signature		
`setImportParams( importParams )`		
Parameters		
`importParams`	`Map<String, String>`	A map with all required import parameters.

Example

```
def d3fc = form.engine.d3fc.createGenericD3FC()
//...
d3fc.setImportParams( ["param1" : "value1", "param2" : "value2"])
//...
d3fc.execute()
//...
```

setImportTable(...)

This method defines an import table of the function to be used.

Signature		
`setImportTable( importTable )`		
Parameters		
`importTable`	`Map<String, List<String>>`	A map with all required columns of the list.

Example

```
def d3fc = form.engine.d3fc.createGenericD3FC()
//...
d3fc.setImportTable( ["col1" : ["col1-row1", "col1-row2"], "col2" : ["col2-
row1", "col2-row2"]])
//...
d3fc.execute()
//...
```

setImportTableAsText(...)

This method sets a strings as payload to be appended to the d3fc protocol instead of the table.

Signature		
`setImportTableAsText( text, encoding)`		
Parameters		
`text`	`String`	The text
`encoding`	`String`	The encoding according to Java-standard (e.g. UTF8)

Example

```
def d3fc = form.engine.d3fc.createGenericD3FC()
//...
d3fc.setImportTableAsText( "This is a Text","UTF8")
//...
d3fc.execute()
//...
```

setImportTableAsFile(...)

This method sets the specified file as payload to be appended to the d3fc protocol.

Signature
`setImportTableAsFile( file)`

Parameters		
`file`	`File`	A file as java.io.File

Example

```
def d3fc = form.engine.d3fc.createGenericD3FC()
//...
d3fc.setImportTableAsFile( new File("myFile.txt"))
//...
d3fc.execute()
//...
```

setImportTableAsStream(...)

This method sets the specified stream as payload to be appended to the d3fc protocol. It is important to specify the length of the stream in Bytes.

Signature		
`setImportTableAsStream( stream, len)`		
Parameters		
`stream`	`InputStream`	A java.io.InputStream or compatible input stream
len	long	the length of the stream

Example

```
def d3fc = form.engine.d3fc.createGenericD3FC()
//...
def fileToImport = new File("myFile.txt")
d3fc.setImportTableAsStream( new FileInputStream(fileToImport),
fileToImport.length())
//...
d3fc.execute()
//...
```

getImportTable( )

This method provides a table that can be used to represent the import table with reference to the d.3 API documentation.

Signature:	
`getImportTable()`	
Return value	
Table	Object to create a table structure with reference to the d.3 API documentation

Example

```
def d3fc = form.engine.d3fc.createGenericD3FC()
//...
def it = d3fc.getImportTable()
it.setValue("col1",0,"val1")
//...
d3fc.execute()
//...
```

setExportParams(...)

Some d3fc-server implementations such as the d.3 server require the export parameters to be prede-fined. The method setExportParams definines the export-parameters to be returned after the execution.

The reference to the resulting export-parameters must be fetched with the method getExportParams() after execute.

Signature		
`setExportParams( exportParams )`		
Parameters		
`exportParams`	`Map<String, String>`	A map with all required export parameters of this function.

Example

```
def d3fc = form.engine.d3fc.createGenericD3FC()
//...
d3fc.setExportParams( ["number":""])
d3fc.execute()

def number = d3fc.getExportParams().get( "number")
//...
```

setExportTableAsBytes(...)

This method must be used, if this function call is to expect a byte array. The reference to the resulting byte aray must be fetched with the method getExportParams() after execute().

Signature		
`setExportTableAsBytes( asBytes )`		
Parameters		
`asBytes`	`boolean`	true or false. If true, then the method getExportTableAsBytes() oder getExportTableAsText(encoding) can be used after the call to access the data

Example

```
def d3fc = form.engine.d3fc.createGenericD3FC()
d3fc.setFunctionName( "DoSomething")
//...
d3fc.setExportTableAsBytes( true)
d3fc.execute()
byte[] bytes = d3fc.getExportTableAsBytes();
// or
String text = d3fc.getExportTableAsText("UTF8");
```

execute( )

This method executes a previously configured function.

Signature:
`execute( )`

getReturnCode( )

This method provides the return code of the executed function.

Signature:	
`getReturnCode()`	
Return value	
Integer	Return code of the executed function

getReturnMessage( )

This method provides the return message of the executed function.

Signature:	
`getReturnMessage()`	
Return value	
String	Return message of the executed function

getExportParams( )

These method returns the resulting export-parameters of the executed function.

Signature:	
`getExportParams()`	
Return value	
Map<String, String>	Map with the export parameters.

Example

```
def d3fc = form.engine.d3fc.createGenericD3FC()
d3fc.setFunctionName( "DoSomething")
//...
d3fc.execute()

def param1= d3fc.getExportParams().get( "param1")
//...
```

getExportTable( )

This method returns the resulting export-table of the executed function.

Signature:	
`getExportTable()`	
Return value	
Table	Table structure equivalent to the export table with reference to the d.3 API documentation

Example

```
def d3fc = form.engine.d3fc.createGenericD3FC()
d3fc.setFunctionName( "DoSomething")
//...
d3fc.execute()

def exportTable = d3fc.getExportTable()
form.log.info( "Found ${exportTable.size()} hits")

for( row in exportTable) {
    def rowValueInCol1 = row.getValue( "col1")
    //...
}
```

getExportTableAsBytes( )

This method returns the resulting export-data of the executed function as a byte-array. This can only be used, if the `AsBytes` was set (true) before executing `setExportTable`.

Signature:
`getExportTableAsBytes()`

Signature:	
Return value	
bytes[]	Export data as Byte array

getExportTableAsText(...)

This method returns the export-data of the executed function which have been defined as a byte-array and converts them to string.

Signature:	
`getExportTableAsText(encoding)`	
Parameters	
String	Encoding (to Java-Standard) for the byte-to-string conversion (e.g. "UTF8" or "ISO-8859-1")
Return value	
String	Export data as string

Subobjekt io

getSecureTempFile()

The function `getSecureTempFile()` allows you to access encrypted files in the temporary directory.

Unencrypted files can also be used with it.

Signature:	
`getSecureTempFile(filename)`	
Parameters	
`filename`	Name of the file in the temporary folder.
Return value	
`SecureTemp-File`	An object for further use with the file name from filename within the temporary folder
Exception	
	IOException, if issues arose with the file system or if less than 500 Megabytes of free disk diskspace were available on the installation drive

```
/*
Use-case Description:
We want to access a file that has been uploaded by the user.
The success script of the upload element calls this action with the
parameter "filename".
*/

def readUploadedFile( def form ){

   def filename = form.params.getValue( "filename" );

   def tempFile = form.engine.io.getSecureTempFile( filename );

   def inputStream = tempFile.getInputStream();
   inputStream.withStream {
      // do something with the file
   }

   return null;
}
```

SecureTempFile

A `getSecureTempFile()` object allows you to access encrypted and unencrypted files in the temporary directory. The function getSecureTempFile() allows to maintain an instance of such an object.

The object provides methods allowing to access the file.

getInputStream(...)

This method allows you to open an InputStream to the file. The InputStream returns the decrypted content of the file.

`getInputStream( )`	
Return value	
java.io.InputStream	A stream providing the content of the file

getLength(...)

This method returns the length of the unencrypted content of the file.

`getLength(...)`	
Return value	
long	The number of bytes of the unencrypted file content.

getTempFile()

The function `getTempFile(filename)` allows you to work with a local temporary folder on the server which can be used in the current form session. At this point, no file exists and must be created by a function call. Thus, the folder is created empty and is fully deleted on logging out of the session.

Note

Files that have been uploaded with the Upload element are stored in the temporary directory in encrypted form.

To be able to read these files you must use the function getSecureTempFile(). An ordinary FileInputStream would only read the encrypted data.

Signature:	
`getTempFile(filename)`	
Parameter	
`filename`	Name of the file in the temporary folder.
Return value	
`File`	An instance of `java.io.File` with the file name from filename within the temporary folder
Exception	
	IOException, if issues arose with the file system or if less than 500 Megabytes of free disk diskspace were available on the installation drive

```
/*
Use-case Description:
We want to create a PDF of this Formular and provide
a download url to that PDF for the current user.
*/

def createPDFAndDownloadLink( def form ){

   //reserve a temp file
   def tempfile = form.engine.io.getTempFile("currentform.pdf")
   // create the PDF
   def converter = form.convert.toPDF()
   converter.setOutputStream( new FileOutputStream( tempfile ))
   converter.execute()

   //create a download link for this generated PDF
   def url = form.engine.io.getUrlForTempFile("currentform.pdf")

   //Provide the URL to the formular via transient property
   form.context.setValue( "TransientData.url", url)

   return null
}
```

getUrlForTempFile()

The function `getUrlForTempFile(filename)` creates a URL to a file created with the method `get-TempFile(filename)`. This URL is only valid within the currently running user session.

> Note
>
> To provide thie download to the current form user later, the JavaScript function `form.misc.http.download(url)` can be used.

Signature:	
`getUrlForTempFile(filename)`	
Parameter	
`filename`	Name of the file previously created with `getTempFile(filename)`
Return value	
`String`	The URL as string

```
/*
 * Use-case Description:
 * We want to provide a download for a d.3 document
 */
def createLinkForD3Document( def form ){
   //reserve a temp file within the temp folder
   def tempfile = form.engine.io.getTempFile("d3doc.txt")
   //write the d.3 document to the temp file
   def d3api = form.engine.d3.createD3Api()
   d3api.getDocumentAsStream(["docId" : "T000000564", "status" : "Fr" ],
new FileOutputStream(tempfile))

   //create a download link for the temp file
   def url = form.engine.io.getUrlForTempFile("d3doc.txt");

   //Provide the URL to the user interface via transient property
   form.context.setValue( "TransientData.previewURL", url);

   return null;
}
```

1.2.7. Sub-object instance

save()

The function `save(...)` saves the actual content of the data-property in the third-party systems AND, if an instance-ID exists, in the internal persistence layer of d.ecs forms.

Signature:
`save()`

```
/*

Use-case Description:

We want to do some business logic and save the external data sources and
the internal data.

*/

def save( def form ){
   // Do some business logic

   // Then save all external data sources and the internal data
   form.instance.save();
}
```

saveInstance()

The function `saveInstance()` saves the form-instance. A possibly selected sub-instance is also saved with it.

Signature:
`saveInstance()`

```
/*

Use-case Description:

We want to do some business logic and save only the internal data.

*/

def save( def form ){
   // Do some business logic

   // Then save the internal data
   form.instance.saveInstance();
}
```

closeInstance(...)

The function `closeInstance(...)` closes the instance of the form. The values of the instance are archived in the process. This instance and the instance-ID can afterwards no longer be used in the forms.

A possibly selected sub-instance is also saved with it. However, then no other open subinstances must exist. Optionally, you can specify, after how many days the instance is to be fully deleted.

Signature:		
`closeInstance() closeInstance( daysToLive )`		
Import parameters		
`daysToLive`	Integer	The parameter specifies how long the instance is to remain archived. After daysToLive days, the instance is fully deleted. When entering the value ‘0’, the instance is deleted at the next possible time. This occurs after a maximum of 6 hours or after a restart of the d.ecs forms server.

```
/*

Use-case Description:

We want to do some business logic and close the current instance.

*/

def close( def form ){
   // Do some business logic

   // Close the current instance.
   form.instance.closeInstance();

   // Close the current instance and delete it after 3 days
   // form.instance.closeInstance(3);

}
```

setSubInstanceId(...)

The function `setSubInstanceId(...)` allows you to specify the ID of the subinstance. If this subinstance already exsists, i.e. has been saved before, its content is relaoded.

Signature:		
`setSubInstanceId( subInstanceId)`		
Import parameters		
`subInstanceId`	String	Specifies the ID of the subinstance. Subinstances are children of an instance and are only valid within the currently selected (main) instance. The subInstanzId must not be null. (See removeSubInstanceId())

```
/*

Use-case Description:

We want to use one subinstance for each user to parallely collect
information from each user.
We have to do this in the Server Initialization Script

*/

def initContext( def form ){
   def subInstanceId = form.misc.login.getUsername();
   form.instance.setSubInstanceId(subInstanceId);
}
```

Attention:

Setting a subinstance only affects the current form session and is not persistant. The setting must be repeated before each loading of a form.

Attention:

In a transaction, setting or changing the subinstance is not possible.

removeSubInstanceId()

The function `removeSubInstanceId()` allows you to remove a subInstanceId configured with `setSubInstanceId( subInstanceId)`. Afterwards you are working only in the main instance again.

Signature:
`removeSubInstanceId()`

```
/*

Use-case Description:

We previously set a subInstance and now need to work with the main instance
exclusively in the further process.
*/

def subInstanceValue = form.context.getValue( "Object.property"); // value
in current subinstance only
form.instance.removeSubInstanceId();

// now move the subInstanceValue to the main instance
form.context.setValue( "Object.property", subInstanceValue); // value now
globally available
```

saveSubInstance()

The function `saveSubInstance()` saves only the subinstance of the form.

Signature:
`saveSubInstance()`

```
/*

Use-case Description:

We want to do some business logic and save only the internal data of the
current selected subinstance.

*/

def save( def form ){
   // Do some business logic

   // Then save the internal data of the subinstance
   form.instance.saveSubInstance();
}
```

closeSubInstance(...)

The function `closeInstance(...)` closes the specified subinstance. The values of the subinstance are archived in the process. This subinstance should no longer be used afterwards. You cannot create another subinstance with this ID within the current main instance any more, either.

Signature:		
`closeSubInstance(subInstanceId)`		
Parameters		
`subInstanceId`	String	Subinstance ID to be closed

```
/*

Use-case Description:
```

```
We want to do some business logic and close the given subinstance.

*/

def close( def form ){
   // Do some business logic

   // Close the current instance.
   form.instance.closeSubInstance( "someSubInstanceId");
}
```

getSubInstanceId()

The function `getSubInstanceId()` allows you to determine a subInstanceId configured with `setSubInstanceId( subInstanceId)`.

Signature:		
`String getSubInstanceId()`		
Return value		
`subInstanceId`	String	The ID of the subinstance or null if none has been set before. Subinstances are children of an instance and are only valid within the currently selected (main) instance.

```
/*

Use-case Description:

We need to know the value of the currently used subInstanceId.
*/

def subInstanceId = form.instance.getSubInstanceId();
```

getSubContext(...)

The function `getSubContext(subInstanceId)` allows you to read the values from another subinstance. The returned Groovy-object possesses all reading methods as the previously known context-object (form.context).

Signature:		
`SubContextInstance getSubContext( subInstanceId)`		
Import parameters		
`subInstanceId`	String	Specifies the ID of the subinstance. The subInstanceId must not be null.
Return value		
	SubContextInstance	Reading access to all values of the subinstance.

```
/*
Use-case Description:

We want to read a value from a foreign subinstance.
*/

def subContext = form.instance.getSubContext("mySubInstanceId");

def subInstanceValue = subContext.getValue("Object.property");
```

SubContextInstance

A subcontext is not used during the parallel processing of instances. It references the values of an existing subinstance already stored in the database and provides reading access to it. Writing access is not possible. The storage location must first be configured in the d.ecs forms designer under the option "subinstance". Only then it is a separate subinstance.

getValue(...)

The method determines the first or the indexed value of a property (in the subinstance)

Signature:		

`getValue( propertyName )`
`getValue(propertyName, index )`		

Import parameters		
`propertyName`	String	Name of the property
index	Integer	Index
Return value		
	Object (according to data type)	Value of the property

getValues(...)

The method determines all values of a property (in the subinstance)

Signature:		
`getValues( propertyName)`		
Import parameters		
`propertyName`	String	Name of the property
Return value		
	List<Object> (according to the data type)	Values of the property

getAllSubInstanceIds()

The function `getAllSubInstanceIds()` allows you to determine all IDs of the subinstances saved for this instance.

Signature:		
`List<String> getAllSubInstanceIds()`		
Return value		
`subInstanceIds`	List<String>	All IDs of the subinstance as a list of strings.

```
/*

Use-case Description:

We need to know all currently existing (saved) subInstanceIds of this
instance.
*/

def subInstanceIds = form.instance.getAllSubInstanceIds( );
for( subInstanceId in subInstanceIds) {
        // process each subInstance
}
```

getRemoteContext()

The function `getRemoteContext()` allows you to determine the current instance values from the database. This is a merely reading access. If instance has never been saved before and thus does not exist in the database, the method returns `NULL`.

Signature:	
`RemoteContext getRemoteContext( )`	
Return value	
RemoteContext	The current instance values in the database, or `NULL`, if the instance does not yet exist in the database.

```
/*

Use-case Description:

We want to get the value of Object.Property as it currently exists in the
database.

*/

def remoteContext = form.instance.getRemoteContext();

// First check for null value
if( remoteContext){
   def value = remoteContext.getValue( "Object.Property");
}
```

RemoteContext

A `RemoteContext`-object allows you to determine the values of properties as the currently exist in the in the database.

To create a `RemoteContext` object, you use the method `form.instance.getRemoteContext()`.

getValue(...)

The function `getValue(...)` allows to read the value of a property in the remote-context.

Signature:		
`Object getValue(` path)		
`Object getValue(` path, index)		
Import parameters		
`path`	String	The name of the property in the remote-context
`index`	Integer	Index of the value in the list.
Return value		
		For `getValue( name )`, the respective value of the parameter is returned in the according data type (`String, Number, Date,` ...).
		For `setValue(name, index )`, the value is updated at the respective position of the list.

getValues(...)

The function `getValues(...)` allows you to read a value of a property in the remote-context.

Signature:		
`List<Object> getValues(` path)		
Import parameters		
`path`	String	The name of the property in the remote-context
Return value		
	List<Object>	List of the values

getRemoteChanges(...)

The function `getRemoteChanges(...)` allows you to dtermine all value updates parrallelly performed in the instance database by another user. This is a comparison between current data in the instance-database and the revision of the instance-database at the time of loading (or the last saving or merging) in the current form session.

Signature:	
`RemoteChanges getRemoteChanges()`	
Return value	
RemoteChanges	Object representing all value updates of the database against the local data.

Attention:

Value updates parrallelly performed in the instance database by another user only refer to the main instance. Subinstances are not considered as these are by design not to be used by several users simultaneously.

Example:

For examples on using this API, please refer to the templates (server-actions) on adopting remote-data in the d.ecs forms designer.

RemoteChanges

An object of the type `RemoteChanges` represents the value changes in the instance-database against the data model of the current form session. The updates can be adding, updating or removing a value.

getChangesInProperties(...)

This method filters the determined value changes per property respectively. **No** tabular context is established between the properties of an object.

Attention:

Properties not saved in the instance are not considered here.

Signature:		

`RemotePropertyChanges getChangesInProperties( )`
 	
`RemotePropertyChanges getChangesInProperties( propertiesOrObjects )`		

Import parameters		
propertiesOrObjects	String[]	Array of properties or objects for which the changes are to be determined. This method is recommended for performance reasons, if you only want to determine the changes of a few properties in a large data model.
Return value		
	RemotePropertyChanges	Object representing all value updates of the database against the local data.

RemotePropertyChanges

An object of the type `RemotePropertyChanges` represents the value changes in the instance-database against the data model of the current form session. These changes differentiate between adding a new value or updating an existing value or deleting it.

getCreations()

The method determines all changes in which a new value has been added.

Signature:	
`Collection<RemoteValueChange> getCreations()`	
Return value	
Collection<RemoteValueChange>	Collection of all respective changes. You can apply all Groovy-standard-methods for collections to this collection.

getUpdates()

The method determines all changes in which a new value has been edited.

Signature:	
`Collection<RemoteValueChange> getUpdates()`	
Return value	
Collection<RemoteValueChange>	Collection of all respective changes. You can apply all Groovy-standard-methods for collections to this collection.

getDeletions()

The method determines all changes in which an existing value has been deleted.

Signature:	
`Collection<RemoteValueChange> getDeletions()`	
Return value	
Collection<RemoteValueChange>	Collection of all respective changes. You can apply all Groovy-standard-methods for collections to this collection.

RemoteValueChange

An object of the type `RemoteValueChange` represents the value change in the instance-database against the data model of the current form session. The updates can be adding, updating or removing a value.

getObject()

The method determines the name of the object to which the property from `getProperty()` belongs.

Signature:	
`String getObject()`	
Return value	
String	Name of the object

getProperty()

The method determines the property name whose value is represented by the object.

Signature:	
`String getProperty()`	
Return value	
String	The name of the property

getRemoteIndex()

The method determines the index of the value within the value list in the database.

Signature:	
`Integer getRemoteIndex()`	
Return value	
Integer	The index in the database

getRemoteValue()

The method determines the actual value in the database.

Signature:		
`Object getRemoteValue()`		
Return value		
	Object (according to data type)	The value in the database

getLocalIndex()

The method determines the index of the value within the current local value list.

Signature:		
`Integer getLocalIndex()`		
Return value		
	Integer	The current local index

getChangesInRows(...)

This method filters the determined value changes per line respectively. A tabular context is established between the multi-value-properties of an object.

Attention:

Properties of the type 'Single value' and properties not saved in the instance are not considered here.

Signature:		

`RemoteRowChanges getChangesInRows( )`
 	
`RemoteRowChanges getChangesInRows( objects )`		

Import parameters		
objects	String[]	Array of objects for which the changes are to be determined. This method is recommended for performance reasons, if you only want to determine the changes of a few properties in a large data model.
Return value		
	RemoteRowChanges	Object representing all row updates of the database against the local data.

Attention:

`getChangesInRows(...)` only considers properties of the type `Multi-value`.

RemoteRowChanges

An object of the type RemoteRowChanges represents a group of value changes in the instance-database against the data model of the current form session. Each update-object represents one row withing the defined object. These changes differentiate between adding a new value or updating an existing value or deleting it.

getRowCreations()

The method determines all changes in which a new row has been added.

Signature:		
`Collection<RemoteRowValueChange> getRowCreations()`		
Return value		
	Collection<RemoteRowValueChange>	Collection of all respective changes. You can apply all Groovy-standard-methods for collections to this collection.

getRowUpdates()

The method determines all changes in which at least one existing value in an existing row has been edited.

Signature:	
`Collection<RemoteRowValueChange> getRowUpdates()`	
Return value	
Collection<RemoteRowValueChange>	Collection of all respective changes. You can apply all Groovy-standard-methods for collections to this collection.

getRowDeletions()

The method determines all changes in which an existing row has been deleted.

Signature:	
`Collection<RemoteRowValueChange> getRowDeletions()`	
Return value	
Collection<RemoteRowValueChange>	Collection of all respective changes. You can apply all Groovy-standard-methods for collections to this collection.

RemoteRowValueChange

An object of the type `RemoteRowValueChange` represents the change of one row of the define object in the instance-database against the data model of the current form session. The updates can be adding, updating or removing a row.

getObject()

The method determines the name of the object.

Signature:	
`String getObject()`	
Return value	
String	Name of the object

getProperties()

The method determines the name of all properties of the type Multi-value in an object

Signature:	
`List<String> getProperties()`	
Return value	
List<String>	List of the names

getChangedProperties()

The method determines the name of all properties of the type Multi-value affected by the change in an object

Signature:	
`List<String> getChangedProperties()`	
Return value	
List<String>	List of the names

getRemoteIndex()

The method determines the index of the row within the object in the database.

Signature:
`Integer getRemoteIndex()`

Signature:	
Return value	
Integer	The index in the database

getRemoteValue(String property)

The method determines the actual value of a specific property (column) within the row in the database.

Signature:		
`Object getRemoteValue( String property)`		
Import parameters		
`property`	String	The name of the property (column)
Return value		
	Object (according to data type)	The value in the database

getChangeInProperty(String property)

The method determines the value change of a specific property (cell) within the row in the database.

Signature:	
`RemoteValueChange getChangeInProperty( String property)`	
Return value	
RemoteValueChange	The value change of a specific property (cell) within the row.

getLocalIndex()

The method determines the index of the value within the current local value list.

Signature:	
`Integer getLocalIndex()`	
Return value	
Integer	The current local index

markAsMerged(...)

This methods marks all value changes determined in this object as synchronized. This has the following effects

- Calling `getRemoteChanges(...)` against the database state again on which this object is based no longer yields any result
- Saving the current instance no longer leads to any more conflicts, if the state in the database after the call of `getRemoteChanges(...)` has not been updated by another user again.

Signature:
markAsMerged()

beginRemoteTransaction()

The function `beginRemoteTransaction()` starts a database-controlled transaction. It is ensured that only one transaction can be opened at a time. Other transactions on the same form-instance are stopped in the meantime.

Attention:

Opening such a transaction requires an instance-ID.

Signature:		
`RemoteTransaction beginRemoteTransaction()` `RemoteTransaction beginRemoteTransaction(` timeoutSeconds)		
Import parameters		
`timeoutSeconds`	Integer	Time in seconds, after which an attempt to start the transaction is cancelled. This is relevant if the instance is blocked by another transaction (too long). Attention: Please note that specifying a timeout is an implementation of the respective database management system (DBMS). As a result, different behaviors may occur here depending on the DBMS used. Thus, please check the behavior desired for your use case in combination with your DBMS.
Return value		
	RemoteTransaction	Object with some methods to be used within the interactive transaction:

```
/*
Use-case Description:

Save instance and subinstance within a transaction. If the transaction is
blocked more than 5 seconds, throw an exception
*/

try{
        def transaction = form.instance.beginRemoteTransaction( 5 );

        // Do something on the transaction-Instance

        // now commit everything
        transaction.commit();
} catch( Exception e){
        form.fail( 4711, e.getMessage());
}
```

RemoteTransaction

Transactions in **d.ecs forms** allow you to initially apply changes to the data model temporarily in order to either adopt or discard them afterwards. A `RemoteTransaction`, moreover, opens a blocking transaction in the instance database. This ensures that only one transaction can be opened at a time. Other transactions on the same form-instance are stopped in the meantime. Blocking only applies between the `beginRemoteTransaction()` and the `commit()` line. An exception is thrown, if changes to the instance have been applied before during the current form session (e.g. by another user). If this is not handled, this leads to the return code -3 in the client-script.

To create a `RemoteTransaction` object, you can use the method form.instance.beginRemoteTransaction().

Attention:

As long as the `transaction.commit()` method is called on this `RemoteTransaction`, no final changes are applied to the database.

Example

```
def transaction = form.instance.beginRemoteTransaction( );

// do something within the transaction

// e.g. change values and save them
transaction.context.setValue( "Object.Property", "value");
transaction.instance.saveInstance();
// very important line:
transaction.commit( );
```

commit()

Must always be called to complete a transaction and to actually apply all updates. If this method is not called within a script a rollback is performed.

Signature:
`commit()`

Attention:

Committing an action also closes it. If you want to continue working with a transaction afterwards, it must be reopened with `form.instance.beginRemoteTransaction()`.

Committing an action moreover has the effect that all updates to the instance applied within the transaction are adopted. Respective changes that have meanwhile been applied **outside** the transaction are thus overwritten. For this reason, it is urgently recommended to apply changes to the instance with an active transaction only with the methods `transaction.context.*` and `transaction.instance.*`.

Sub-object context

Provides an interface to the local data model equivalent to `form.context`. However, this interface behaves transactionally in the form so that the changes applied here only take effect once the transaction is closed with a `commit()`.

Sub-object instance

Provides an interface to the local data model equivalent to `form.instance`. This is limited to the following methods:

- `acceptCreation()`
- `acceptUpdate()`
- `acceptDeletion()`
- `acceptRowCreation()`
- `acceptRowUpdate()`
- `acceptRowDeletion()`
- `saveInstance()`
- closeInstance()
- saveSubInstance()
- closeSubInstance()
- getRemoteChanges()
- getRemoteRowChanges()
- getRemoteContext()
- hasLocalChanges()
- `hasLocalRowChanges()`

Note

Calling this method within `transaction.instance` (equivalently to `transaction.context`) has the effect that the changes applied there are only adopted on calling `commit()`.

acceptCreation(...)

The function `acceptCreation(...)` allows you to adopt a new value added to the instance database by another user in the local data.

Signature:		
`acceptCreation( index, remoteChange)`		
`acceptCreation( remoteChange)`		
Import parameters		
`index`	Integer	Target-index in the local data model

Signature:		
remoteChange	RemoteValueChange	Update-object representing the value update of the database against the local data.

```
/*

Use-case Description:

We want to get all values, which have been created by another user in
parallel, and accept them in our own data model.
The remote index of each value should be accepted, too.

*/

def mergeValues( def form ){
   def remoteChanges = form.instance.getRemoteChanges();
   remoteChanges.getCreations( ).each{ change ->
      form.instance.acceptCreation( change );
   }
}
```

acceptUpdate(...)

The function `acceptUpdate(...)` allows you to adopt a change to a value applied to the instance database by another user in the local data.

Signature:		
`acceptUpdate( remoteChange)`		
Import parameters		
remoteChange	RemoteValueChange	Update-object representing the value update of the database against the local data.

```
/*

Use-case Description:

We want to get all values, which have been updated by another user in
parallel, and accept them in our own data model.

*/

def mergeValues( def form ){
   def remoteChanges = form.instance.getRemoteChanges();
   remoteChanges.getUpdates( ).each{ change ->
      form.instance.acceptUpdate( change );
   }
}
```

acceptDeletion(...)

The function `acceptDeletion(...)` allows you to adopt the deletion of a value performed in the instance database by another user in the local data.

Signature:		
`acceptDeletion( remoteChange)`		
Import parameters		
remoteChange	RemoteValueChange	Update-object representing the value update of the database against the local data.

```
/*

Use-case Description:

We want to remove all values, which have been removed by another user in
parallel, in our own data model.

*/

def mergeValues( def form ){
   def remoteChanges = form.instance.getRemoteChanges();
   remoteChanges.getDeletions( ).each{ change ->
      form.instance.acceptDeletion( change );
   }
}
```

acceptRowCreation(...)

The function `acceptRowCreation(...)` allows you to adopt a new row added to the instance database by another user in the local data.

Signature:		
`acceptRowCreation( index, remoteRowValueChange)`		
`acceptRowCreation( remoteRowValueChange)`		
Import parameters		
`index`	Integer	Target-index in the local data model
RemoteRowValueChange	RemoteRowValueChange	Update-object representing the value update of the database against the local data.

```
/*

Use-case Description:

We want to get all rows, which have been created by another user in
parallel, and accept them in our own data model.
The remote index of each value should be accepted, too.

*/

def mergeValues( def form ){
   def remoteChanges = form.instance.getRemoteRowChanges();
   remoteChanges.getRowCreations( ).each{ change ->
      form.instance.acceptRowCreation( change );
   }
}
```

acceptRowUpdate(...)

The function `acceptRowUpdate(...)` allows you to adopt a change applied to at least one value of a row in the instance database by another user in the local data.

Signature:
`acceptRowUpdate( remoteRowValueChange)`
`acceptRowUpdate( remoteRowValueChange, includeUnchangedRemoteColumns)`
Import parameters

Signature:		
RemoteRowValueChange	RemoteRowValueChange	Update-object representing the value update of the database against the local data.
includeUnchangedRemoteColumns	Boolean	If true, then the values of the columns that have NOT be parallelly changed are also adopted. (Default: false)

Attention:

If `includeUnchangedRemoteColumns` is passed with '`false`' (default), then only the values into the local data model are transferred that have actually been changed in the database. Local changes in the other columns are not overwritten.

```
/*

Use-case Description:

We want to get all rows, which have been updated by another user in
parallel, and accept them in our own data model.

*/

def mergeValues( def form ){
   def remoteChanges = form.instance.getRemoteRowChanges();
   remoteChanges.getRowUpdates( ).each{ change ->
      form.instance.acceptRowUpdate( change );
   }
}
```

acceptRowDeletion(...)

The function `acceptRowDeletion(...)` allows you to adopt the deletion of a row performed in the instance database by another user in the local data.

Signature:		
`acceptRowDeletion( remoteRowValueChange)`		
Import parameters		
RemoteRowValueChange	RemoteRowValueChange	Update-object representing the value update of the database against the local data.

```
/*

Use-case Description:

We want to remove all rows, which have been removed by another user in
parallel, in our own data model.

*/

def mergeValues( def form ){
   def remoteChanges = form.instance.getRemoteRowChanges();
   remoteChanges.getRowDeletions( ).each{ change ->
      form.instance.acceptRowDeletion( change );
   }
}
```

hasLocalChanges(...)

The function `hasLocalChanges(...)` allows you to check, if a property or a value has been changed locally since the time of last saving (or last loading). The function can be used like this:

- Determine local changes conflicting with the value changes in the database
- Determine local changes within an object
- Determine local changes within a property

Signature:		
`boolean hasLocalChanges( remoteChange)` `boolean hasLocalChanges( objectOrPropertyName)`		
Import parameters		
remoteChange	RemoteValueChange	Update-object representing the value update of the database against the local data.
objectOrPropertyName	String	Name of an object or a property
Return value		
	boolean	`true`, if the value or property features local changes.

```
/*

Use-case Description:

We want to remove all values, which have been removed by another user in
parallel, in our own data model ONLY IF the value was not changed locally

*/

def mergeValues( def form ){
   def remoteChanges = form.instance.getRemoteChanges();
   remoteChanges.getDeletions( ).each{ change ->
          if( !form.instance.hasLocalChanges( change )){
         form.instance.acceptDeletion( change );
          }
   }
}
```

hasLocalRowChanges(...)

The function `hasLocalChanges(...)` allows you to check, if a row has been changed locally since the time of last saving (or last loading). It is irrellevant in which column or row the local change has taken place. The function can be used like this:

- Determine local changes conflicting with the row changes in the database

Signature:		
`boolean hasLocalRowChanges( remoteChange)`		
Import parameters		
remoteChange	RemoteRowValueChange	Update-object representing the row update of the database against the local data.
Return value		
	boolean	`true`, if the value or property features local changes.

Attention:

If the row does not exist, `fals` is returned.

```
/*

Use-case Description:
```

```
We want to remove all rows, which have been removed by another user in
parallel, in our own data model ONLY IF the row was not changed locally

*/

def mergeValues( def form ){
   def remoteChanges = form.instance.getRemoteRowChanges();
   remoteChanges.getRowDeletions( ).each{ change ->
          if( !form.instance.hasLocalRowChanges( change )){
         form.instance.acceptRowDeletion( change );
          }
   }
}
```

1.2.8. Sub-object convert

toPDF(...)

The function `toPDF()` allows you to convert a form to a PDF-document.

Signature:	
ConverterInstance toPDF()	
Return value	
`ConverterInstance`	This object allows to configure the conversion further.

```
/*

Use-case Description:

We want to convert the current form to a PDF document.

*/

def converter = form.convert.toPDF();
 // Do something with the converter object.
converter.setOutputStream(new FileOutputStream("D:\\document.pdf"))
converter.execute();
```

ConverterInstance

doNotPrint(...)

The function `doNotPrint()` allows you to hide selected elements from the PDF to be created.

Signature:		
`void doNotPrint( elementId ) void doNotPrint( elementsArray )` `void doNotPrint( elementsList )`		
Import parameters		
elementId	String	The name of a form-element to be hidden.
`elementsArray`	String[]	An array with the names of several form-elements to be hidden.
`elementsList`	List<String>	A list with the names of several form-elements to be hidden.

Attention:

The function does not support table columns. If the name of a table column is passed, this causes an exception.

```
/*

Use-case Description:

Our form contains many elements that we want to hide based on
conditional logic.

*/

def converter = form.convert.toPDF();
if ( myCondition ) {
        converter.doNotPrint( "someElement" );
}
```

execute(...)

The function `execute()`performs the PDF-conversion with the selected parameters.

Signature:
`execute( )`

```
/*

Use-case Description:
We want to convert our form to PDF and save the result to disk.

*/

def converter = form.convert.toPDF();
converter.setOutputStream(new FileOutputStream( "C:\\test.pdf" ))
converter.execute();
```

setFormId(...)

The function `setFormId()` allows you to select another form for the PDF-conversion.

Signature:		
void setFormId(formId)		
Import parameters		
`formId`	String	The unique ID of the form to be used for the conversion.

```
/*

Use-case Description:

Because our form contains large amounts of conditional logic, it doesn't
look good
when converted to PDF. Instead, we created a secondary form which uses the
same
context which is only used as a PDF template.

*/

def converter = form.convert.toPDF();
```

```
converter.setFormId( "myFormId" );
```

setFormVersion(...)

The function `setFormVersion()` allows you to select another form build for the PDF-conversion.

Signature:		
`void setFormVersion( formVersion )`		
Import parameters		
`formVersion`	Integer	The build number of the desired form.

```
/*

Use-case Description:
We want to select a specific build of the form that we want to
convert to a PDF.

*/

def converter = form.convert.toPDF();
converter.setFormVersion( 2 );
```

setLocale(...)

The function `setLocale()` allows you to select the language for the PDF-conversion.

Signature:		
`void setLocale( locale )`		
Import parameters		
locale	String	A string with the language code of the desired language.

Note

If you are using the parameter `locale` as string, please note the IETF BCP 47 format. Example:

"de" for German.

"de-AT" for German (Austria)

```
/*

Use-case Description:
We want to select a specific locale to convert our form to PDF.

*/
import java.util.Locale;


def converter = form.convert.toPDF();
converter.setLocale( "de-AT" );
```

setOutputStream(...)

An output stream must be specified via the function `setOutputStream()`. The generated PDF-document is written to it on calling execute().

Signature:		
`void setOutputStream(os)`		
Import parameters		
`os`	OutputStream	A java.io.OutputStream

```
/*

Use-case Description:
We want to select a specific build of the form that we want to
convert to a PDF.

*/

def converter = form.convert.toPDF();
converter.setOutputStream(new FileOutputStream( "d:\\test.pdf"))
```

setSubInstanceId(...)

The function `setSubInstanceId()` sets another subinstance as the basis for the PDF-conversion.

Signature:		
`void setSubInstanceId( subInstanceId )`		
Import parameters		
`subInstanceId`	String	The name of the desired subinstance.

```
/*

Use-case Description:
We want to select a specific subinstance that we want to
convert to a PDF.

*/

def converter = form.convert.toPDF();
converter.setSubInstanceId( "mySubInstance" );
```

1.2.9. Data structures

Table

This data structure represents tables with rows and columns. A table can be iterated, i.e. it can be used in the respective Groovy Loop constructs (for-each, each, find etc.). This is iterated via Table row.

A table is, for example, used to implement the import and export-tables in d.3 API-functions or in data sources (ds.data.getTable()).

getValue(...)

The method determines the value of a table row

Signature:		
`getValue( colName, index )`		
Import parameters		
`colName`	String	The name of the column
`index`	Integer	Row index

Signature:		
Return value		
	Object (according to data type)	The value of the respective row

getValues(...)

The method determines all values of a column

Signature:		
`getValues( colName )`		
Import parameters		
`colName`	String	The name of the column
Return value		
	List<Object> (according to the data type)	The values of the respective column

size( )

The method determines the maximum number of rows

Signature:		
`size( )`		
Return value		
	Integer	The maximum number of rows

iterator( )

The method creates an iterator allowing you to iterate via the table rows.

Signature:		
`iterator( )`		
Return value		
	Iterator<Table row>	An iterator across all rows

setValue(...)

The method sets the value of a cell

Signature:		
`setValue( colName, index, value )`		
Import parameters		
`colName`	String	The name of the column
`index`	Integer	Row index
`value`	Object (according to data type)	The new value

addValue(...)

The method appends a new value at the end of a table. Existing values are moved back by one position starting from the specified index.

Signature:		

`addValue( colName, value )`
`addValue( colName, index, value )`		

Import parameters		
`colName`	String	The name of the column in the table

Signature:		
`index`	Integer	Index at which the value is to be added. If not specified, the value is appended to the end of the list.
`value`	Object (according to data type)	The new value

setValues(...)

The method sets all values of a column

Signature:		
`setValues( colName, values )`		
Import parameters		
`colName`	String	The name of the column
`values`	List<Object> (according to the data type)	The new values

deleteValue(...)

The method clears the value of a cell

Signature:		
`deleteValue( colName, index )`		
Import parameters		
`colName`	String	The name of the column
`index`	Integer	Row index

deleteValues(...)

The method clears all values of a column

Signature:		
`deleteValues( colName )`		
Import parameters		
`colName`	String	The name of the column

Table row

This data structure represents the row of a Table. This can be determined explicitly via the iterator()-function or implicitly using Loop constructs (for, each, find etc.).

An object of the type table row is compatible to a `Map<String,Object` whereby its methods `containsValue()`, `remove()` and `clear()` are not implemented. Else, such an object can be used like a map with its methods `get()` or `put()`.

getValue(...)

The method determines the value of a table row

Signature:		

`getValue( colName )`
`get( colName )`		

Import parameters		
`colName`	String	The name of the column
Return value		
	Object (according to data type)	The value of the respective row

setValue(...)

The method sets the value of a cell

Signature:		

`setValue( colName, value )`
`put( colName, value )`		

Import parameters		
`colName`	String	The name of the column
value	Object (according to data type)	The new value

getRowNumber( )

The method returns row number of the current table row

Signature:		
`getRowNumber( )`		
Return value		
	Integer	The row number

1.2.10. Global methods

fail(...)

With `fail(...)` you can stop the execution of a script in case of an error or forward a message to the client.

Signature:		
`void fail(errorCode, errorMessage)`		
Import parameters		
`errorCode`	int	A positive error code.
`errorMessage`	String	Descriptive error text
Field of application		
Throwing an individually defined exception in case of an error. The client can interpret the error code and text in the onFailure-function. Calling this function stops the execution of the script.		

Attention:

All changes applied to the form with `context.setValue(...)` or `context.setValues(...)` before calling `fail(...)` are maintained.

```
def doSomething( def form ) {
    form.fail( 1, "This text will be sent to the client" );
    // Code below here will not be executed
}
```

1.2.11. Groovy-modules

modules(...)

The function `modules(...)` allows you to access a Groovy-module referenced in the form-context or object model.

Signature:		
`Module modules( moduleName )`		
Import parameters		
`moduleName`	String	The name of the module to be loaded.
Return value		

Signature:	
Modules	The requested module.

This function allows you to access global modules.

Attention:

Make sure that the desired module is included in the form-context or in the object model.

If a module name is passed that does not exist in the form-context, an error occurs and the form is closed.

```
/* Use-case Description: We want to execute the function "myFunction" which is defined in the
module "MyModule" */ form.modules("MyModule").myFunction);
```

1.3. Deprecated methods

Sub-object	Method	Deprecated with version	Notes on possible substitution
Within the d.3 API object (`form.engine.d3.createD3Api()`)	`d3api.setImportBytes()`	3.0.0	`d3api.setImportTableAsStream()`
Within the d.3 API object (`form.engine.d3.createD3Api()`)	`d3api.getExportBytes()`	3.0.0	`d3api.getExportTableAsBytes()`
`form.engine.d3.wfl`	`getDocId()`	3.0.0	`form.engine.d3.getDocId()`

1.4. CRUD

The d.ecs forms engine can analyse the values of a data source and divide them into the three sections:

- **Create**: Values added within d.ecs forms
- **Update:** Values changed within d.ecs forms
- **Delete**: Values deleted within d.ecs forms

This is done with reference to comparison operations to the last load- or save process. The section Create combines everything that was added to the form. Update only includes changes, and so on.

To use this mechanism the following three methods must be implemented in the server-side (Groovy) script instead of the saveData-method:

- processCreate
- processUpdate
- processDelete

Apart from this, these methods behave just as the saveData-method. The difference is only the pre-filtering into the three sections cretae, update and delete. These methods are only called, if a respective Create, Update or Delete has been performed since creating the form instance or since the latest saving of the form (`form.instance.save()`).

The following sample script requires a data source (object model) with the properties name, firstname, address, zip, city und nr (each of the type multiple values), an according database table and a matching database alias (myDB).

CRUD data source

```
def loadData(def ds) {
        def query="SELECT name,vorname,adresse,plz,ort,nr FROM forms.KUNDEN"
        ds.engine.sql.executeAndStore( "myDB", query)
        ds.log.info("Loaded "+ds.data.getTable().size()+" Customers")
        return null
```

```
}

def processUpdate(def ds) {
        def query="""UPDATE forms.KUNDEN
                                SET name=?, vorname=?, adresse=?, plz=?,
ort=?
                                WHERE nr=?"""

        for (row in ds.data.getTable()) {
                def nr=row.getValue("nr")
                if (nr==null)
                        throw new Exception("Missing nr in $row for
processUpdate")

                def name=row.getValue("name")
                def vorname=row.getValue("vorname")
                def adresse=row.getValue("adresse")
                def plz=row.getValue("plz")
                def ort=row.getValue("ort")
                def params = [name,vorname,adresse,plz,ort,nr]

                ds.log.info("Updating Customer "+params)
                ds.engine.sql.executeUpdate("myDB",query, params)
        }
        return null
}

def processCreate(def ds) {
        def query="INSERT INTO forms.KUNDEN VALUES(?,?,?,?,?,?)"

        for (row in ds.data.getTable()) {
                def nr=row.getValue("nr")
                if (nr==null)
                        throw new Exception("Missing nr in $row for
processCreate")

                def name=row.getValue("name")
                def vorname=row.getValue("vorname")
                def adresse=row.getValue("adresse")
                def plz=row.getValue("plz")
                def ort=row.getValue("ort")
                def params = [name,vorname,adresse,plz,ort,nr]

                ds.log.info("Creating Customer "+params)
                ds.engine.sql.executeInsert("myDB",query,params)
        }
        return null
}

def processDelete(def ds) {
        def query="DELETE FROM forms.KUNDEN WHERE nr=?"
        for (row in ds.data.getTable()) {
                def nr = row.getValue("nr")
                if (nr==null)
                        throw new Exception("Missing nr in $row for
processDelete")

                ds.log.info("Deleting Customer "+nr)
                ds.engine.sql.execute("myDB",query,[nr])
        }
        return null
}
```

1.5. Webservice (SOAP)

1.5.1. Webservice (SOAP)

You can address webservices on the server-side.

Attention:

You may require a Java library for this effect (JAR file). This must be added to the system (from d.ecs forms 3.0) as a library and referenced in the corresponding form context or object model. More information on creating and using JAR files as libraries can be found in the d.ecs forms administration manual.

JAX-WS

It is recommended to use the JAX-WS as this is included in Java by default. For this effect, you must create a Java library outside of d.ecs forms (see documentation JAX_WS) which is implementing the webservice call and encapsulates it in an easy interface. The complexity of the webservice thus remains hidden. Another advantage is that this library can be tested and used individually.

1.6. Appendix

1.6.1. List of all supported d.3 API functions for direct access

This is a list of d.3-functions to be used in chapter Direct access

Note

You can find information on how to obtain the documentation for the API functions in chapter Sub-object d3.

A

AcknowledgeReceivedHoldFile

AcquireLockToken

AddDocumentsToFavorites

B

BlockDocument

C

ChangeHoldFileRecipient

ChangePasswordforUser

CheckExistenceOfNewHoldFiles

CheckInOut

CloneDocument

D

DeleteDocument

DeleteDocumentSystemValues

G

GetActivityStream

GetAttributesAlterationHistory

GetColorCodeHistory

GetCompanyNames

GetCurrentTimestamp

GetD3ServerConfiguration

GetD3Set

GetD3SetFilter

GetDocumentList

GetDocumentListBySystemValue

GetDocumentListFacetDescription

GetDocumentSystemValue

GetDocumentSystemValues

GetDocumentTypeLongUser

GetDocumentTypeShortUser

GetDocumentTypeTitleAll

GetDocumentTypeTitleUser

GetDocumentTypesForUserGroup

GetDocumentVersionHistory

GetExtendedProperties

GetFacetForDocumentList

GetHoldFileInfo

GetHoldFileOverview

GetHoldFilesReceived

GetHoldFilesSent

GetStructureDocument

GetSubUsers

GetSupUsers

GetSystemAttributes

GetUser

GetUserGroup

GetUserGroupsLongText

GetUserInGroup

GetUserInRoll

GetUserVerifierForGroup

GetValidValuesForAttribute

GetValueForConfigVariables

I

ImportDocument

ImportNewVersionDocument

InsertDependentDocument

L

LinkDocuments

P

ProceedToNextStepWorkPath

PutDocumentIntoWorkPath

R

ReadWorklist

ReceiveNote

ReleaseDocument

ReleaseLockToken

RemoveDocumentsFromFavorites

ReserveNextDocID

S

SearchDocument

SearchDocumentInIDList

SearchHoldFileDocument

SendDependentDocument

SendHoldFile

SendNote

SendTemporaryFile

SetDocHistoryData

SetDocumentSystemValues

SetHoldFileRead

T

TransferDocument

U

GetMultiAttributesAlterHistory	UnlinkDocuments
GetMultipleAttributes	UpdateAttributes
GetPSUrl	**V**
GetParentStructureDocument	ValidateAttributes
GetPhysicalFileInformation	ValidatePasswordForUser
GetRecentLinkedFolderByUser	VerifyDocument
GetRecentModifiedFilesbyUser	
GetSignatureInformation	

1.7. Additional information sources and imprint

If you want to deepen your knowledge of d.velop software, visit the d.velop academy digital learning platform at https://dvelopacademy.keelearning.de/.

Our E-learning modules let you develop a more in-depth knowledge and specialist expertise at your own speed. A huge number of E-learning modules are free for you to access without registering beforehand.

Visit our Knowledge Base on the d.velop service portal. In the Knowledge Base, you can find all our latest solutions, answers to frequently asked questions and how-to topics for specific tasks. You can find the Knowledge Base at the following address: https://kb.d-velop.de/

Find the central imprint at https://www.d-velop.com/imprint.