d.velop

d.ecs forms server scripting: Administrieren

Inhaltsverzeichnis

1. Entwicklerdokumentation d.ecs forms server scripting

1.1. Einleitung

1.1.1. Über das d.ecs forms server scripting

Server-Skripte können entweder als **Datenquelle** oder als **Formular-Aktionen** verwendet werden.

Datenquelle

Eine **Datenquelle** ist für die Datenanbindung an Fremdsysteme gedacht und existiert ausschließlich im d.ecs forms Objektmodell. Damit lassen sich Daten aus anderen Systemen in d.ecs forms importieren und auch wieder exportieren. Datenquellen sind Formular-unabhängig. Sie können nicht auf Formular-Inhalte zugreifen. Sie können lediglich anhand einiger Parameter Daten beschaffen und diese zurückgeben. Datenquellen haben folgende Eigenschaften:

- Sie besitzen dedizierte Eingabe-Parameter (`ds.params`) und Rückgabe-Werte (`ds.data`).
- Der Rückgabe-Wert der Datenquelle wird zwischengespeichert (Cache). Im weiteren Verlauf einer Formular-Instanz werden nur noch die zwischengespeicherten Werte verwendet. Jeder erneute Aufruf einer Datenquelle arbeitet additiv auf den Daten des vorherigen Aufrufs. Möchte man dies nicht, muss der aktuelle Datensatz geleert werden. (`ds.data.clear()`)
- Die Methoden-Namen sind vorgegeben (`loadData`, `saveData`)

Formular-Aktion

Der zweite Einsatzzweck von serverseitigen Skripten sind **Aktionen**. Hier gibt es zwei wesentliche Unterschiede zu **Datenquellen**.

- Aktionen befinden sich im Formular-Kontext und können schreibend und lesend auf dessen Werte zugreifen.
- Aktionen müssen explizit aus dem Formular heraus aufgerufen werden und werden nicht im Cache zwischengespeichert.

Die serverseitig in d.ecs forms verwendete Skriptsprache ist Groovy.

Prinzipiell handelt es sich bei Groovy um eine Erweiterung von Java, die mehr Dynamik und eine kompaktere Syntax ermöglicht. Es kann aber auch Java programmiert werden.

> Achtung
>
> Alle erstellten Skripte so wie verwendete externer Bibliotheken müssen für die Verwendung mit einer Java Runtime Environment in der Version 8 kompiliert sein.

1.1.2. Voraussetzungen

Dieses Handbuch erläutert das d.ecs forms Server scripting und ist ausschließlich für Administratoren und technische Anwender gedacht, die die Logik für Formulare entwickeln oder warten sollen.

Um die folgenden Informationen zu verstehen, benötigen Sie Erfahrungen in der Programmiersprache Groovy oder Java; Kenntnisse von Entwicklungsumgebungen sind von Vorteil.

1.1.3. Aufbau der Skripting-Schnittstelle

Bevor auf die zur Verfügung gestellten Skripting-Methoden im Detail eingegangen wird, wird zunächst der allgemeine Aufbau der Skripting-Schnittstelle bzw. das zu Grunde liegende Modell näher erläutert.

Angesteuert wird die Skripting-Schnittstelle über eine Methode. Diese Methode hat immer ein zentrales Input-Objekt. In Datenquellen (im globalen Objektmodell) heißt dieser Parameter üblicherweise "`ds`" und in Skripten, die aus einem Formular heraus aufgerufen wurden, "`form`". Über dieses Objekt werden alle Interaktionen vorgenommen. Es werden sowohl Parameter ausgelesen als auch Rückgabe-Werte übergeben. Je nach Typ des Skriptes kann auf den Formular-Kontext zugegriffen werden. Auch globale Hilfsfunktionen zum Zugriff auf eine Datenbank oder das d.3 System werden über dieses Objekt bzw. eines der Subobjekte vorgenommen.

Beispiel Datenquelle

Beispiel

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

Beispiel Aktion:

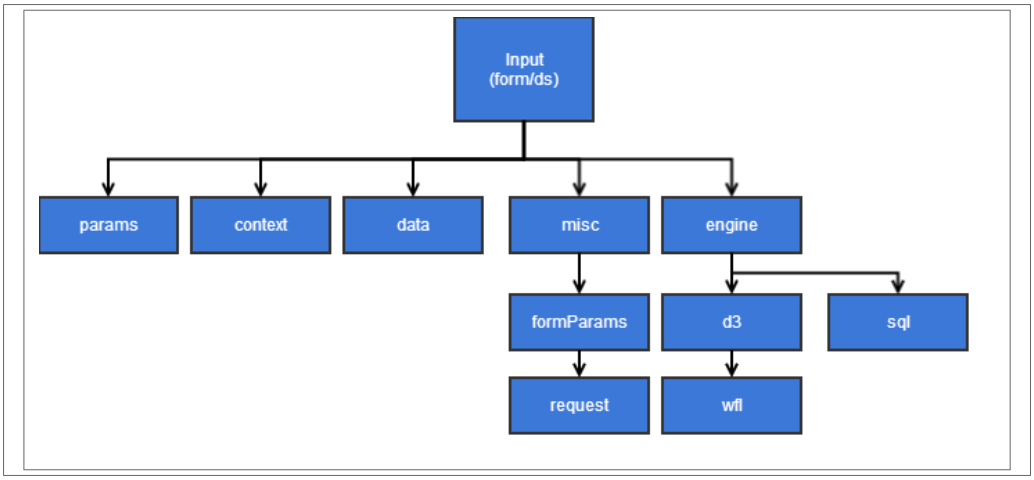
Beispiel

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

Hinweis

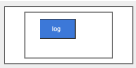
Im Folgenden wird in den Schnittstellen-Beschreibungen der Parameter `form` (entsprechend der Verwendung in Formular-Kontexten) verwendet. In Datenquellen im d.ecs forms Objektmodell muss dafür analog der Parameter `ds` verwendet werden. Der Parameter `ds` wird in den Beschreibungen hier nur dann verwendet, wenn die verwendeten Methoden ausschließlich für Datenquellen im d.ecs forms Objektmodell gedacht sind.

Dabei untergliedert sich das Input-Objekt in weitere Subobjekte, welche jeweils eigene Methoden zur Verfügung stellen. Diese verfeinerte Untergliederung soll eine Kapselung und Zusammenfassung ähnlicher Methoden unterstützen. Angeboten werden dabei die folgenden Subobjekte: `params, context, data, misc, engine, log`.



Das **Params-Objekt** stellt die Parameter bereit, die direkt an das Skript übergeben wurden.

Das **Context-Objekt** erlaubt den Zugriff auf die Daten (Kontext) des Formulars.

	Das **Data-Objekt** stellt Daten bereit, die das Skript direkt wieder zurück gibt. Alternativ können die Rückgabe-Werte mit `return` zurückgegeben werden.
	Das **Misc-Objekt** stellt diverse Hilfsfunktionen bereit, die sich den o.g. Subobjekten nicht zuordnen lassen.
	Das **Engine-Objekt** stellt globale Hilfsfunktionen (d.3, SQL, ...) bereit.
	Wichtige Ereignisse (Informationsmeldungen, Fehler) können über das **Log-Objekt** protokolliert werden. Alle Meldungen werden im d.ecs forms server Log gespeichert.

Im weiteren Verlauf der Dokumentation erfolgt eine ausführliche Beschreibung der einzelnen zur Verfügung gestellten Methoden.

Beispiel

Bitte arbeiten Sie immer mit der voll qualifizierten Funktionssignatur. Weisen Sie die Subobjekte keinesfalls einer neuen Variable zu. Ferner darf der Zugriff nicht dynamisch über "`eval`" erfolgen.

FALSCH:

```
def data = form.data;

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

RICHTIG:

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

1.1.4. Rückgabewerte von Skript-Methoden in Datenquellen

Skript-Methoden können ihr Ergebnis auf zwei unterschiedliche Arten zurückgeben.

Hinweis

Es wird empfohlen, dies immer über `ds.data.setValue(...)`zu tun.

Sie können jedoch zur einfacheren Schreibweise als "return"-Wert zurückgegeben werden. Wenn in einer Datenquelle nur ein einziger Wert als Rückgabe-Wert definiert wurde, dann kann dieser Wert direkt zurückgegeben werden. Ansonsten muss ein Name-Wert-Paar (Map) erstellt werden.

Folgende Schreibweisen sind gleichwertig:

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

und

```
return ["name" : wert]
```

1.1.5. Datentypen

Folgende Datentypen werden von d.ecs forms unterstützt:

d.ecs forms Datentyp	empfohlener Groovy Datentyp	Beispiel
Wahrheitswert	Boolean	`form.context.setValue( "Objekt.Wahrheitswert", true );`
Datum	java.sql.Date	`form.context.setValue( "Objekt.Datum", new java.sql.Date( new java.util.Date( ).getTime( ) ) );`
Ganzzahl	Integer	`form.context.setValue( "Objekt.Ganzzahl", 1 );`
Nummer	Number	`form.context.setValue( "Objekt.Nummer", 1.1 );`
Text	String	`form.context.setValue( "Objekt.Text", "Dies ist ein Text" );`

Achtung

d.ecs forms unterstützt aktuell ausschließlich das reine Datumsformat (ohne Zeitstempel). Klassen wie `java.util.Date` beinhalten immer auch einen Zeitstempel. Es wird daher ausdrücklich empfohlen, ein Datum immer durch den Typen `java.sql.Date` zu erzeugen, da diese Objekte immer ausschließlich ein Datum enthalten. Andere Datentypen werden von d.ecs forms in der aktuellen Version zwar toleriert, jedoch mit einer Warnung im Log quittiert. Um auch in zukünftigen Versionen von d.ecs forms keine Typ-Konflikte zu erhalten, sollten solche Konstellationen stets vermieden werden.

1.1.6. Kommunikation mit Datenbanken

Sollten Sie in Ihren Groovy-Skripten mit Datenbanksystemen kommunizieren, die nicht dem DBMS des d.ecs forms Systems entsprechen, so müssen hierfür separate JDBC-Treiber-Archive hinterlegt werden. Die entsprechenden JAR-Dateien müssen in der d.ecs forms Installation unter **`<d3presentationserver>/forms/lib/custom`** hinterlegt werden. Nach Ablage ist ein Neustart des d.ecs forms servers notwendig.

1.2. Erläuterung der server-seitigen Skripting-Methoden

Im weiteren Verlauf der Dokumentation erfolgt eine ausführliche Beschreibung der einzelnen zur Verfügung gestellten Methoden.

Hinweis

Bitte beachten Sie auch, dass neben dem Server-Skripting ein clientseitiges Skripting angeboten wird. Nähere Informationen hierzu entnehmen Sie bitte der entsprechenden Dokumentation.

1.2.1. Subobjekt context

Achtung

Die Methoden des Subobjekts 'context' dürfen innerhalb des d.ecs forms Objektmodells nur für den Zugriff auf die Objekte verwendet werden, die sich innerhalb jenes Objektmodells befinden. Zugriffe auf Objekte aus externen Formular-Kontexten ist nicht möglich.

setValue(...)

Über die Funktion `setValue(...)` wird der Wert einer Eigenschaft im Formular-Kontext geändert.

Signatur		
`setValue( path, value )`		
`setValue( path, index, value )`		
Importparameter		
`path`	String	Der Name der Eigenschaft im Formularkontext
`index`	Integer	Index des Wertes in der Liste.
`value`	Object	Bei `setValue(name,value)` wird der entsprechende Wert in die Eigenschaft im Formular-Kontext geschrieben. Der Typ des Wertes muss dem konfigurierten Typ im Datenmodell entsprechen. (`String, Number, Date, …`)
		Bei `setValue(name, index, value)` wird der Wert an der jeweiligen Position innerhalb der Liste verändert.

```
/*

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" );
```

> Achtung
>
> Soll im Rahmen der Verarbeitung großer Datenmengen eine Vielzahl von Werten mit Index gesetzt werden, so kann das Verwenden dieser Methode zu Performance-Problemen führen. In diesem Fall wird empfohlen, die Werte zunächst in einer Collection zu sammeln und diese dann gebündelt per setValues(..) in das Datenmodell zu schreiben.

setValues(...)

Über die Funktion `setValues(...)` werden die Werte einer Eigenschaft im Formular-Kontext geändert.

Signatur		
`setValues( path, values )`		
Importparameter		
`path`	String	Der Name der Eigenschaft im Formular-Kontext
`values`	List<Object>	Die Werte als Liste

```
/*

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(...)

Über die Funktion `addValue(...)` wird einer Eigenschaft im Formular-Kontext ein Wert hinzugefügt. Bestehende Werte werden ab dem angegebenen Index um eine Stelle nach hinten verschoben.

Signatur		
`addValue( path, value )`		
`addValue( path, index, value )`		
Importparameter		
`path`	String	Der Name der Eigenschaft im Formularkontext
`index`	Integer	Index, an dem der Wert eingefügt werden soll. Wenn nicht angegeben, wird der Wert am Ende der Liste angehangen.
`value`	Object	Der neue Wert

```
/*

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(...)

Über die Funktion `getValue(...)` wird der Wert einer Eigenschaft im Formular-Kontext ausgelesen.

Signatur		
`Object getValue( path )`		
`Object getValue( path, index )`		
Importparameter		
`path`	String	Der Name der Eigenschaft im Formular-Kontext
`index`	Integer	Index des Wertes in der Liste
Rückgabewert		
		Bei`getValue( name )` wird der entsprechende Wert des Parameters im entsprechenden Datentyp zurückgegeben (`String, Number, Date, ...`).
		Bei`getValue( name, index )` wird der Wert an der jeweiligen Position innerhalb der Liste zurückgegeben.

```
/*

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(...)

Über die Funktion `getValues(...)` wird der Wert einer Eigenschaft im Formular-Kontext als Liste ausgelesen.

Signatur		
`List<Object> getValues( path )`		
Importparameter		
`path`	String	Der Name der Eigenschaft im Formular-Kontext
Rückgabewert		
	List<Object>	Liste der Werte

```
/*

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(...)

Über die Funktion `deleteValue(...)` wird der Wert einer Eigenschaft im Formular-Kontext gelöscht.

Signatur		
`deleteValue( path )`		
`deleteValue( path, index )`		
Importparameter		
`path`	String	Der Name der Eigenschaft im Formular-Kontext
`index`	Integer	Index des Wertes in der Liste

```
/*

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(...)

Über die Funktion `deleteValues(...)` werden alle Werte einer Eigenschaft im Formular-Kontext gelöscht.

Signatur		
`deleteValues( path )`		
Importparameter		
`path`	String	Der Name der Eigenschaft im Formular-Kontext

```
/*

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(...)

Über die Funktion `getObjectProperties(...)` können die Metadaten aller Eigenschaften zu einem bestimmten Objekt (oder allen Objekten) ermittelt werden.

Signatur		
`Map<String, PropertyMetaData> getObjectProperties( )`		
`Map<String, PropertyMetaData> getObjectProperties( objectName )`		
Importparameter		
objectName	String	Der Name des Objektes (optional)
Rückgabewert		
	Map<String, PropertyMetaData>	Liste der ermittelten Eigenschaften. Der Key ist in der Form `Object.Property (z.B. "Kunde.Nr")`

> **Achtung**
>
> Im Objektmodell wird nur der Aufruf von `ds.context.getObjectProperties( objectName )` mit konkretem Objektnamen unterstützt. Ein Aufruf ohne Objektnamen berücksichtigt (auch beim Aufruf innerhalb von Server-Aktionen) nur die Objekte, die direkt im entsprechenden Formular-Kontext existieren oder aus dem Objektmodell importiert wurden.

```
/*
```

```
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()

Die Methode ermittelt den Namen der Eigenschaft

Signatur	
`getName()`	
Rückgabewert	
String	Der Name der Eigenschaft in der Notation "Objekt.Eigenschaft"

isMultiValue()

Die Methode ermittelt, ob es sich bei der Eigenschaft um den Typ "Mehrfachwert" handelt

Signatur	
isMultiValue()	
Rückgabewert	
boolean	`true`: Mehrfachwert `false`: Einfachwert

1.2.2. Subobjekt params

params.getValue(...)

Über die Funktion `getValue(...)` wird der Wert eines direkten Skript-Parameters ausgelesen.

Signatur		
`Object getValue( name )` `Object getValue( name, index )`		
Importparameter		
`name`	String	Der Name des Parameters. Dieser wurde entweder innerhalb einer Datenquelle (**Daten-Objekt**) konfiguriert, oder im**Client-Skript**direkt übergeben. Bei Einsatz zusammen mit der Suchhilfe wird z.B. implizit der Parameter`ingesetzt`, der in dem Fall der aktuellen Eingabe im**Suchhilfe-Feld**entspricht.
`index`	Integer	Index des Wertes in der Liste.
Rückgabewert		
		Bei`getValue( name )`wird der entsprechende Wert des Parameters im entsprechenden Datentyp zurückgegeben (`String, Number, Date, ...`).

Signatur	
	Beigetvalue(name, index)wird der Wert an der jeweiligen Position innerhalb der Liste zurückgegeben.
Anwendungsbereich	
Die Funktion dient in den globalen Datenobjekten und den Server-Aktionen zur Parameter-Übergabe.	

Hinweis
Die Parameternamen `type` und `method` können aktuell nicht verwendet werden.

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

params.getValues(...)

Über die Funktion `getValues(...)` werden die Werte eines direkten Skript-Parameters ausgelesen.

Signatur		
`List<Object> getValues(name)`		
Importparameter		
`name`	String	Der Name des Parameters. Dieser wurde entweder innerhalb einer Datenquelle (**Daten-Objekt**) konfiguriert, oder im**Client-Skript**direkt übergeben.
Rückgabewert		
	`List<Object>`	Liste der Werte
Anwendungsbereich		
Die Funktion dient in den globalen Datenobjekten und in Server-Aktionen zur Parameter-Übergabe.		

Hinweis
Die Parameternamen `type` und `method` können aktuell nicht verwendet werden.

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

1.2.3. Subobjekt data

data.clear()

Die Funktion `clear()` löscht alle bisherigen Werte der Datenquelle. Diese sollte i.d.R immer anfangs einer loadData-Methode einer Datenquelle aufgerufen werden. Wenn man dies nicht aufrufen würde, gibt man letzendlich das Ergebnis des vorherigen Aufrufes zurück.

Signatur
`clear()`
Anwendungsbereich
Wenn `clear()` nicht aufgerufen wird, kann das stückweise Nachladen von Daten (z.B. Paging) serverseitig implementiert/beschleunigt werden.

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

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

data.addValue(...)

Über die Funktion `addValue(...)` wird einer Spalte der Ergebnistabelle ein Wert hinzugefügt. Bestehende Werte werden ab dem angegebenen Index um eine Stelle nach hinten verschoben.

Signatur		
`addValue( path, value )`		
`addValue( path, index, value )`		
Importparameter		
`path`	String	Der Name der Spalte
`index`	Integer	Index, an dem der Wert eingefügt werden soll. Wenn nicht angegeben, wird der Wert am Ende der Liste angehangen.
`value`	Object	Der neue Wert

```
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(...)

Die Funktion `setValue(...)` gibt einen Rückgabewert an d.ecs forms zurück.

Signatur		
`setValue( name, value )`		
`setValue( name, index, value )`		
Importparameter		
`name`	String	Der Name des Rückgabewertes. Der Name muss innerhalb einer Datenquelle zuvor samt Daten-Typ konfiguriert werden.
`index`	Integer	Die Position in der Liste, falls es sich um eine Liste handelt.
`value`	Object	Der Wert. Der Daten-Typ des Wertes (`String`, `Number`, `Date`, ...) muss dabei dem konfigurierten Typ im Datenmodell entsprechen.
Anwendungsbereich		
Wird hauptsächlich bei Datenquellen zum Übergeben der Werte an das Datenmodell verwendet. Wird aber auch bei der Suchhilfe verwendet. Dort werden die `id` und die `caption`(beides eine Liste) über`setValue`zurückgegeben.		

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

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

Achtung

Soll im Rahmen der Verarbeitung großer Datenmengen eine Vielzahl von Werten mit Index gesetzt werden, so kann das Verwenden dieser Methode zu Performance-Problemen führen. In diesem Fall wird empfohlen, die Werte zunächst in einer Collection zu sammeln und diese dann gebündelt per setValues(..) in das Datenmodell zu schreiben.

data.setValues(...)

Die Funktion `setValues(...)` gibt alle Werte eines Rückgabewertes an d.ecs forms zurück.

Signatur		
`setValues( name, value )`		
Importparameter		
`name`	String	Der Name des Rückgabewertes. Der Name muss innerhalb einer Datenquelle zuvor samt Daten-Typ konfiguriert werden.
`values`	List<Object>	Die Werte. Der Daten-Typ des Wertes (`String`, `Number`, `Date`, ...) muss dabei dem konfigurierten Typ im Datenmodell entsprechen.
Anwendungsbereich		
Wird hauptsächlich bei Datenquellen zum Übergeben der Werte an das **Datenmodell** verwendet. Wird aber auch bei der **Suchhilfe** verwendet. Dort werden die `id` und die `caption` (beides eine Liste) über `setValues` zurückgegeben.		

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

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

data.addValue(...)

Die Funktion `addValue(...)` fügt dem Rückgabewert, der an d.ecs forms zurückgegeben wird, einen Wert hinzu. Bestehende Werte werden ab dem angegebenen Index um eine Stelle nach hinten verschoben.

Signatur		

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

Importparameter		
`name`	String	Der Name des Rückgabewertes. Der Name muss innerhalb einer Datenquelle zuvor samt Daten-Typ konfiguriert werden.
`index`	Integer	Index, an dem der Wert eingefügt werden soll. Wenn nicht angegeben, wird der Wert am Ende der Liste angehangen.
`value`	Object	Der neue Wert. Der Daten-Typ des Wertes (`String`, `Number`, `Date`, ...) muss dabei dem konfigurierten Typ im Datenmodell entsprechen.
Anwendungsbereich		
Wird hauptsächlich bei Datenquellen zum Übergeben der Werte an das Datenmodell verwendet. Wird aber auch bei der Suchhilfe verwendet. Dort werden die `id` und die `caption` (beides eine Liste) über `addValue` zurückgegeben.		

```
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(...)

Die Funktion `getValue(...)` liest einen Wert aus dem Zwischenspeicher der Datenquellen aus, die zuvor beim Laden in diese Datenquelle geschrieben wurden.

Signatur		
`Object getValue( name )`		
`Object getValue( name, index )`		
Importparameter		
`name`	String	Der Name des Rückgabewertes. Der Name muss innerhalb einer Datenquelle zuvor samt Daten-Typ konfiguriert werden.
`index`	Integer	Die Position in der Liste, falls es sich um eine Liste handelt.
`value`	Object	Der Wert.
Rückgabewert		
		Bei `getValue( name )` wird der entsprechende Wert des Parameters im entsprechenden Datentyp zurückgegeben (`String, Number, Date,` ...).
		Bei `getValue( name, index )` wird der Wert an der jeweiligen Position innerhalb der Liste zurückgegeben.
		Der Daten-Typ des Wertes (`String`, `Number`, `Date`, ...) muss dabei dem konfigurierten Typ im Datenmodell entsprechen.
Anwendungsbereich		
Die Funktion kann aktuell nur innerhalb von Datenquellen genutzt werden.		

```
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(...)

Die Funktion `getValues(...)` liest alle Werte aus dem Zwischenspeicher der Datenquellen aus, die zuvor beim Laden in diese Datenquelle geschrieben wurden.

Signatur		
`Object getValues( name )`		
Importparameter		
`name`	String	Der Name des Rückgabewertes. Der Name muss innerhalb einer Datenquelle zuvor samt Daten-Typ konfiguriert werden.
Rückgabewert		
`List<Object>`		Liste der Werte.
Anwendungsbereich		
Die Funktion ist aktuell nur innerhalb von Datenquellen nutzbar.		

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

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

   return null;
}
```

data.deleteValue(...)

deleteValue(...)

Über die Funktion `deleteValue(...)` wird der Wert einer Spalte der Ergebnistabelle gelöscht.

Signatur		
`deleteValue( path )`		
`deleteValue( path, index )`		
Importparameter		
`path`	String	Der Name der Spalte
`index`	Integer	Index des Wertes in der Liste

```
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(...)

Über die Funktion `deleteValues(...)` werden alle Werte einer Spalte der Ergebnistabelle gelöscht.

Signatur		
`deleteValues( path )`		
Importparameter		
`path`	String	Der Name der Spalte

```
 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()

Über die Funktion `ds.data.getTable()` kann eine tabellarische Sicht auf alle vorhandenen Werte der Daten einer Datenquelle erreicht werden.

Diese Funktion sollte nur verwendet werden, wenn alle Eigenschaften, die sich in ds.data befinden, auch dieselbe Länge besitzen.

Signatur		
`getTable()`		
Rückgabewert		
`Tabelle`	Der Inhalt aller Eigenschaften in Tabellenform.	
Anwendungsbereich		
Falls der Wunsch oder die Notwendigkeit besteht, nicht Spalten-Orientiert sondern Zeilen-Orientiert auf die Daten zuzugreifen		

```
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. Subobjekt misc

Das Subobjekt **misc** enthält weitere Subobjekte:

- formParams

Subobjekt formParams

getInstanceId(...)

Die Funktion `getInstanceId()` erlaubt es, die aktuelle Formular-Instanz-ID auszulesen.

Signatur	
`String getInstanceId ( )`	
Rückgabewert	
`String`	Formular-Instanz-ID

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

request.getValue(...)

Mit `request.getValue(...)` kann ein Formular-Parameter ausgelesen werden. In der Regel ist dies ein Request-Parameter, mit dem das Formular aufgerufen wurde.

Signatur		
`String getValue( parameterName )`		
Importparameter		
`parameterName`	String	Der Name des Parameters.
Rückgabewert		
	String	Der Wert des Parameters als String.
Anwendungsbereich		
Auslesen eines beliebigen Formular-Parameters.		

```
/*
```

```
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" */
```

> Hinweis
>
> Diese Methode gibt immer Werte vom Typ `String` zurück. Um typisierte Werte zu erhalten, muss der Weg über eine typisierte Objekt-Eigenschaft aus dem Datenmodell gegangen werden, die diesen Request-Parameter ausliest. Schauen Sie hierzu in die d.ecs forms Dokumentation.

request.getValues(...)

Mit `request.getValues(...)` kann ein Formular-Parameter (mit mehreren Werten) ausgelesen werden. In der Regel ist dies ein HTTP-Request-Parameter, mit dem das Formular aufgerufen wurde.

Signatur		
`List<String> getValues( parameterName )`		
Importparameter		
`parameterName`	String	Der Name des Parameters.
Rückgabewert		
	List<String>	Die Werte des Parameters als String.
Anwendungsbereich		
Auslesen eines beliebigen Formular-Parameters.		

```
/*

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"] */
```

> Hinweis
>
> Diese Methode gibt immer Werte vom Typ `List<String>` zurück. Um typisierte Werte zu erhalten, muss der Weg über eine typisierte Objekt-Eigenschaft aus dem Datenmodell gegangen werden, die diesen Request-Parameter ausliest. Schauen Sie hierzu in die d.ecs forms Dokumentation.

i18n.getValue(...)

Mit `i18n.getValue(...)` kann eine Übersetzung aus dem System geladen werden.

Signatur		
`String getValue( key )`		
Importparameter		
`key`	String	Der Key der Übersetzung
Rückgabewert		
	String	Die Übersetzung
Anwendungsbereich		
Auslesen einer Übersetzung		

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

login.getUsername(...)

Die Funktion `login.getUsername()` erlaubt es, den Namen des angemeldeten Benutzers auszulesen.

> Hinweis
>
> Hierbei muss es sich nicht zwangsläufig um den d.3 Benutzernamen handeln. Bei einer Anmeldung in der Form "MyDomain\User" ist dieser Wert auch der Rückgabewert dieser Funktion.
>
> Bei der Verwendung von d.ecs identity provider kann dieses Verhalten je nach Konfiguration des LDAP Benutzer-Providers abweichen.

Signatur	
`String getUsername( )`	
Rückgabewert	
`String`	Name des angemeldeten Benutzers

```
/*

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. Subobjekt log

debug(...)

Die Funktion `debug(...)` schreibt einen Eintrag mit dem Loglevel "Debug" in das d.ecs forms server Log.

Debug wird im Standardfall nicht geloggt. Dies muss manuell aktiviert werden.

Signatur
`debug( msg )`
Importparameter

Signatur		
`msg`	String	Meldungstext, der in das Log geschrieben werden soll.
Anwendungsbereich		
Debug sollte nur zum Finden von Problemen benutzt werden.		

```
/*

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(...)

Die Funktion `info(...)` schreibt einen Eintrag mit dem Loglevel "`Info`" in das d.ecs forms server Log.

Signatur		
`info( msg )`		
Importparameter		
`msg`	String	Meldungstext, der in das Log geschrieben werden soll.
Anwendungsbereich		
Info sollte verwendet werden, um allgemeine Informationen zu loggen. Hier sollten z.B. keine Parameter zur eventuellen Fehlersuche geloggt werden.		

```
/*

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(...)

Die Funktion `warn(...)` schreibt einen Eintrag mit dem Loglevel "`Warn`" in das d.ecs forms server Log.

Signatur		
`warn( msg )`		
Importparameter		
`msg`	String	Meldungstext, der in das Log geschrieben werden soll.
Anwendungsbereich		
Warn sollte verwendet werden, wenn sich ein Problem ankündigt.		

```
/*
```

```
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(...)

Die Funktion `error(...)` schreibt einen Eintrag mit dem Loglevel "`Error`" in das d.ecs forms server Log.

Signatur		
`error( msg )`		
Importparameter		
`msg`	String	Meldungstext, der in das Log geschrieben werden soll.
Anwendungsbereich		
Error sollte verwendet werden, wenn ein Problem aufgetreten ist.		

```
/*

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. Subobjekt engine

Subobjekt sql

execute(...)

Führt ein beliebiges SQL-Statement aus.

Signatur		
`execute( alias, query, parameter )`		
Importparameter		
`alias`	String	Der Name der konfigurierten Datenbankverbindung.
`query`	String	Das SQL-Statement.
`parameter`	List<Object>	Eine Liste der Parameter des SQL-Statements.
	alternativ:	Eine Map der Parameter des SQL-Statements
	Map	
Anwendungsbereich		
Die Funktion führt beliebige SQL-Statements aus, bietet dabei aber keine Möglichkeit der Ergebnis-Auswertung. Sie ist damit nur für ausschließlich schreibende Operationen sinnvoll.		
Zur Datenauswertung sollte eine der anderen Methoden (z.B.executeAndStoreoderexecuteAndGet) verwendet werden.		

```
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(...)

Führt typischerweise ein SQL `Select`-Statement aus und übernimmt das Ergebnis des SQL-Statements direkt in den Zwischenspeicher der Datenquelle.

Das vorherige Ergebnis der Datenquelle wird dabei gelöscht. Es muss nicht mit `ds.data.clear()` geleert werden.

Die Spaltennamen der Datenbank-Tabelle werden dabei automatisch den Namen der Rückgabewerte der Datenquelle zugeordnet.

> Hinweis
>
> Bei einer Suchhilfe empfiehlt es sich mit dem SQL-Schlüsselwort `as` zu arbeiten um gegebenenfalls die Rückgabewerte für z.B. die Suchhilfe nach `id` und `caption` umzubenennen. Speziell bei einer Oracle-Datenbank muss der Name `id` und `caption` in Anführungszeichen geschrieben werden (siehe Beispiel unten), um die Kleinschreibung zu erzwingen.

Signatur		
`executeAndStore( alias, query, parameter )`		
Importparameter		
`alias`	String	Der Name der konfigurierten Datenbankverbindung.
`query`	String	Das SQL-Statement.
`parameter`	List<Object>	Eine Liste der Parameter des SQL-Statements.
	alternativ:	Eine Map der Parameter des SQL-Statements
	Map	
Anwendungsbereich		
Die Funktion führt beliebige SQL-Statements aus und übernimmt das Ergebnis des SQL-Statements direkt in den Zwischenspeicher der Datenquelle.		

```
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(...)

Die Metode `executeAndGet(...)` führt ein SQL-Statement aus und bietet die Möglichkeit dieses Ergebnis manuell auszuwerten.

Signatur		
`executeAndGet( alias, query, parameter )`		
Importparameter		
`alias`	String	Der Name der konfigurierten Datenbankverbindung.
`query`	String	Das SQL-Statement.
`parameter`	List<Object>	Eine Liste der Parameter des SQL-Statements.
	alternativ:	Eine Map der Parameter des SQL-Statements
	Map	
Rückgabewert		
	List<GroovyRowResult>	Eine Liste, wobei jeder Listeneintrag eine Zeile im Ergebnis entspricht. Auf die einzelnen Werte dieser Zeile muss, anders als sonst, direkt zugegriffen werden, also nicht über `row.getValue("name")` sondern `row.name` (siehe Beispiel).
Anwendungsbereich		
Verwenden Sie diese Methode für SELECT-Statements, wenn Sie deren Ergebnis nicht automatisch in die Datenquellen übernehmen möchten, sondern dies noch verändern möchten.		

```
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(...)

Die Methode `executeInsert(...)` führt typischerweise ein SQL `Insert`-Statement aus. Dabei werden eventuell durch das Statement erzeugte IDs zurückgegeben.

Signatur		
`executeInsert( alias, query, parameter )`		
Importparameter		
`alias`	String	Der Name der konfigurierten Datenbankverbindung.
`query`	String	Das SQL-Statement.
`parameter`	List<Object>	Eine Liste der Parameter des SQL-Statements.
	alternativ:	Eine Map der Parameter des SQL-Statements
	Map	
Rückgabewert		
	List<List<Object>>	Falls durch das Statement Werte erzeugt werden (`Auto-Increment`, etc.), so können diese hier ausgelesen werden.
Anwendungsbereich		
Führt ein SQL `Insert`-Statement aus. Dabei werden eventuell durch das Statement erzeugte IDs zurückgegeben.		

```
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(...)

Die Methode `executeUpdate(...)` führt typischerweise ein SQL `Update`-Statement aus und gibt dabei die Anzahl der durch das Update geänderten Zeilen zurück.

Signatur		
`executeUpdate( alias, query, parameter )`		
Importparameter		
`alias`	String	Der Name der konfigurierten Datenbankverbindung.
`query`	String	Das SQL-Statement.
`parameter`	List<Object>	Eine Liste der Parameter des SQL-Statements.
	alternativ:	Eine Map der Parameter des SQL-Statements
	Map	
Rückgabewert		
	Integer	Gibt die Anzahl der geänderten Zeilen zurück.
Anwendungsbereich		
Führt ein SQL `Update`-Statement aus und gibt dabei die Anzahl der durch das Update geänderten Zeilen zurück.		

```
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(...)

Die Methode `createSQLConnection(...)` stellt ein Groovy-SQL-Objekt auf Basis der übergebenen Datenbankverbindung bereit.

Signatur
`createSQLConnection( alias )`

Signatur		
Importparameter		
`alias`	String	Der Name der konfigurierten Datenbankverbindung.
Rückgabewert		
	groovy.sql.Sql	Groovy-SQL-Objekt
Anwendungsbereich		
Es wird empfohlen, nur dann direkt auf dem Groovy-SQL-Objekt zu arbeiten, wenn keine nativen d.ecs forms Funktionen im Bereich `form.engine.sql` für den gewünschten Anwendungsfall existieren.		

> Achtung
>
> Beachten Sie, dass erzeugte SQL-Objekte nach der Benutzung unbedingt mittels `close()`-Methode geschlossen werden müssen. Dies wird **NICHT** durch das d.ecs forms System gewährleistet.

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

Subobjekt d3

Innerhalb der **serverseitigen Skripte** kann auf die d.3 API zugegriffen werden.

Dabei gibt es zwei unterschiedliche Ansätze

1. den direkten Zugriff
2. den Zugriff über vereinfachende Methoden

Beim **direkten Zugriff** werden die d.3 API-Funktionen unterstützt, die im Anhang aufgeführt sind.

> Achtung
>
> Der **direkte Zugriff** ermöglicht Ihnen den Zugriff auf alle öffentlichen Funktionen der d.3 API, die im Anhang dieser Dokumentation aufgeführt sind. Die Dokumentation jener d.3 API Funktionen kann über das d.velop Serviceportal bezogen werden, wobei hierzu die Unterzeichnung einer Geheimhaltungsvereinbarung vorausgesetzt wird. Zusätzlich wird die Teilnahme an der Schulung d.3 Administration empfohlen. Bitte wenden Sie sich für weitere Fragen gerne an das Technologie Partner Management der d.velop AG (d.velop.competencenetwork@d-velop.de).

Für einige Standardaufgaben gibt es einige **vereinfachenden Methoden**. Diese kapseln den teilweise aufwändigen Umgang mit Parametern.

> Hinweis
>
> Wenn ein Formular aus dem d.3 Workflow heraus aufgerufen wird, werden die Login-Informationen (d.3 System und d.3 Benutzer) automatisch aus dem d.3 Client übernommen. Die Angabe eines d.3 Alias-Objektes ist in diesem Fall somit nur dann notwendig, wenn die d.3 API Aufrufe explizit von einem anderen als dem angemeldeten Benutzer durchgeführt werden sollen.

Direkter Zugriff

Innerhalb von serverseitigen Groovy-Skripten kann direkt auf die d.3 API zugegriffen werden. Hierzu muss ein zentrales Objekt erstellt werden, über das der Zugriff erfolgt.

Zugriff aus Formular-Kontext

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

Die Konfiguration des Objektes erfolgt analog zum Aufruf generischer d3fc-Funktionen.

Beispiel

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 );
}
```

Vereinfachter Zugriff

d.ecs forms bietet neben dem direkten Zugriff auf die d.3 API auch einen vereinfachten Zugriff. Dabei handelt es sich um Methoden, die komplexere d.3 API Aufrufe kapseln. Das ermöglicht es, auch ohne tiefere Kenntnisse über die Verwendung der d.3 API, Funktionen auf dem d.3 Server auszuführen.

Die Methoden führen die entsprechende d.3 Funktion direkt auf dem d.3 Server aus und geben je nach d.3 Return-Code einen Ausnahmefehler (Exception) mit der entsprechenden Meldung des d.3 Servers zurück. Folgende Funktionen werden unterstützt:

Methode	d.3 API Funktion
callGetDocumentTypeTitleAll	GetDocumentTypeTitleAll
callGetMultipleAttributes	GetMultipleAttributes
callGetValidValuesForAttribute	GetValidValuesForAttribute
callImportDocument_TextDocument	ImportDocument (mit einem Textdokument)
callReceiveNote	ReceiveNote
callSearchDocument	SearchDocument
callSendNote	SendNote
callUpdateAttributes	UpdateAttributes
callPutDocumentIntoWorkPath(...)	PutDocumentIntoWorkPath

Für detaillierte Informationen bzgl. der d.3 API Funktionen schauen Sie bitte in die Dokumentation der d.3 API.

callGetDocumentTypeTitleAll

Die Methode ermittelt alle Eigenschaften einer Dokumentart.

Signatur		
`callGetDocumentTypeTitleAll( docTypeShort )`		
Importparameter		
`docTypeShort`	String	Das Kürzel der Dokumentart, deren Attribute ermittelt werden sollen
Rückgabewert		
	Table	Die Export-Tabelle gemäß der d.3 API Funktion GetDocumentTypeTitleAll
Anwendungsbereich		
Die Funktion steht in serverseitigen Groovy-Skripten im Formular-Kontext oder dem d.ecs forms Objektmodell zur Verfügung		

```
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);
}
```

callGetMultipleAttributes

Die Methode ermittelt zu einem vorgegebenen Dokument die Attribute mit Mehrfachausprägung.

Signatur		
`callGetMultipleAttributes( docId)`		
Importparameter		
`docId`	String	Die Dokument-ID, zu deren Dokument die Mehrfacheigenschaften ermittelt werden sollen
Rückgabewert		
	Table	Eine Tabelle, die der 60er-Tabelle entspricht. Für jedes Mehrfacheigenschaften-Feld existiert also eine Spalte in der Tabelle.
	Exception	Wenn Returncode ungleich 0
Anwendungsbereich		
Die Funktion steht in serverseitigen Groovy-Skripten im Formular-Kontext oder dem d.ecs forms Objektmodell zur Verfügung		

```
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

Die Methode ermittelt zu einer vorgegebenen Dokumentart zu einem bzw. allen dokumentartabhängigen Attribut(en) den gültigen Wertevorrat, sofern dieser definiert ist.

Signatur		
`callGetValidValuesForAttribute( importParams)`		
Importparameter		
`importParams`	Map<String, Object>	Die Import-Parameter gemäß der d.3 API Funktion GetValidValuesForAttribute
Rückgabewert		
	Table	Eine Tabelle, in der jedes Doc-Field durch eine eigene Spalte repräsentiert wird.
	Exception	Wenn Returncode ungleich 0
Anwendungsbereich		
Die Funktion steht in serverseitigen Groovy-Skripten im Formular-Kontext oder dem d.ecs forms Objektmodell zur Verfügung		

```
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

Die Methode legt ein Textdokument im **d.3** Archiv ab.

Signatur		
`callImportDocument_TextDocument( importParams, textDocument)`		
Importparameter		
`importParams`	Map<String, Object>	Die Import-Parameter gemäß der d.3 API Funktion ImportDocument
`textDocument`	String	Der Inhalt des Dokumentes
Rückgabewert		
	String	Die Dokument-Id des archivierten Dokumentes
	Exception	Wenn Returncode ungleich 0
Anwendungsbereich		

Signatur
Die Funktion steht in serverseitigen Groovy-Skripten im Formular-Kontext oder dem d.ecs forms Objektmodell zur Verfügung

```
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(...)

Die Methode startet einen d.3 Workflow und ermöglicht es, dabei auch Workflow-Variablen anzugeben.

Signatur		
`callPutDocumentIntoWorkPath(docId, wflId, wflVars)`		
Importparameter		
`docId`	String	Die d.3 Dokument-ID
`wflId`	String	Die d.3 Workflow-ID (Process-ID)
`wflVars`	Map<String, Object>	Die d.3 Workflow-Variablen (oder <null>)
Rückgabewert		
	void	
	Exception	Wenn Returncode ungleich 0
Anwendungsbereich		
Die Funktion steht in serverseitigen Groovy-Skripten im Formular-Kontext zur Verfügung		

```
//########################################
// 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

Die Methode ermittelt die Inhalte einer Notiz zu einem Dokument.

Signatur		
`callReceiveNote( docId )`		
Importparameter		
`docId`	String	Die Dokument-Id des Dokumentes
Rückgabewert		
	String	Der Inhalt der Notizdatei
	Exception	Wenn Returncode ungleich 0 und ungleich 320
Anwendungsbereich		
Die Funktion steht in serverseitigen Groovy-Skripten im Formular-Kontext oder dem d.ecs forms Objektmodell zur Verfügung		

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

callSearchDocument

Die Methode ermittelt zu vorgegebenen Suchkriterien die Kenndaten der gewünschten Dokumente.

Signatur		
`callSearchDocument( searchParams )`		
Importparameter		
`searchParams`	Map<String, Object>	Die Import-Parameter gemäß der d.3 API Funktion SearchDocument
Rückgabewert		
	Table	Die Export-Tabelle gemäß der d.3 API Funktion SearchDocument
	Exception	Wenn Returncode ungleich 0 und ungleich 10
Anwendungsbereich		
Die Funktion steht in serverseitigen Groovy-Skripten im Formular-Kontext oder dem d.ecs forms Objektmodell zur Verfügung		

```
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

Ergänzt die Notiz eines Dokuments um einen neuen Eintrag, bzw. legt die Notiz mit diesem Eintrag an, falls die Notiz noch nicht vorhanden ist.

Signatur		
`callSendNote( docId, note )`		
Importparameter		
`docId`	String	Die Dokument-Id des Dokumentes
`note`	String	Inhalt der zu ergänzenden / anzulengenden Notizdatei
Anwendungsbereich		
Die Funktion steht in serverseitigen Groovy-Skripten im Formular-Kontext oder dem d.ecs forms Objektmodell zur Verfügung		

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

callUpdateAttributes

Aktualisiert die Kenndaten eines Dokumentes.

Signatur		
`callUpdateAttributes( docId, attributes)`		
Importparameter		
`docId`	String	Die Dokument-Id des zu ändernden Dokumentes
`attributes`	Map<String, Object>	Die Import-Parameter gemäß der d.3 API Funktion UpdateAttributes
Rückgabewert		
	void	
	Exception	Wenn Returncode ungleich 0
Anwendungsbereich		
Die Funktion steht in serverseitigen Groovy-Skripten im Formular-Kontext oder dem d.ecs forms Objektmodell zur Verfügung		

```
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(...)

Damit kann ein Dokument aus d.3 heruntergeladen werden. Diese Methode erwartet einen Output-Stream und schreibt das d.3 Dokument dort hinein.

Signatur		
`getDocumentAsStream( params , os )`		
Importparameter		
params	Map<String, String>	Die Parameter, mit denen das gewünschte Dokument ausgesucht wird.
os	OutputStream	Ein OutputStream wohin das Dokument geschrieben werden soll
Anwendungsbereich		
Die Funktion steht in serverseitigen Groovy-Skripten im Formular-Kontext oder dem d.ecs forms Objektmodell zur Verfügung		

Für die Parameter können die folgenden Optionen angegeben werden:

Name	Bedeutung	Pflichtparameter
docId	Gibt die Dokumenten-ID des zu öffnenden Dokuments an.	Ja
status	Gibt den gewünschten Status des Dokuments an (Be, Pr, Fr, Ar).	Nein
archiveIndex	Bei mehreren Archiv-Versionen kann hier die gewünschte angegeben werden. Wird keine Angabe gemacht, wird die neuste Archiv-Version gewählt.	Nein
dependentDocument	Wählt statt der Nutzdatei des Dokuments ein abhängiges Dokument (z.B. "t1") aus.	Nein

Hinweis

Das d.3 Dokument wird beim Aufruf dieser Methode sofort bereitgestellt und der OutputStream geschlossen. Das Aufrufen der d3api .execute() ist in diesem Fall nicht notwendig.

```
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()

Diese Methode ermittelt die d.3 Dokument-ID des Dokumentes, welches dem aktuellen Formular-Aufruf zu Grunde liegt.

Signatur	
`String getDocId ( )`	
Rückgabewert	
String	Aktuelle d.3 Dokument-ID.

setDocId()

Diese Methode setzt die d.3 Dokument-ID des Dokumentes, welches dem aktuellen Formular-Aufruf zu Grunde liegt.

> **Achtung**
>
> Die übergebene d.3 Dokument-ID darf nicht `null` oder leer sein. Ein nachträgliches Deselektieren eines d.3 Dokumentes ist mit Hilfe dieser Methode nicht möglich.

Signatur		
`setDocId ( docId )`		
Importparameter		
docId	String	d.3 Dokument-ID, die dem Formular-Kontext zu Grunde liegen soll

```
/*

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()

Diese Methode aktualisiert alle Eigenschaften, die Ihren Wert aus einem d.3 Dokument lesen. Zum Ausführen dieser Methode ist eine d.3 Dokument-ID erforderlich. Diese muss entweder initial durch die Formular-URL oder später per `form.engine.d3.setDocId()` übergeben worden sein.

> **Achtung**
>
> Ggf. lokal existierende Werte werden dabei überschrieben.

Signatur
`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( );
}
```

Subobjekt login

getRepositoryId()
Diese Methode ermittelt die ID des d.3 Repositories, welches der Anmeldung des aktuellen Formular-Aufrufs zu Grunde liegt.

Signatur	
`String getRepositoryId ( )`	
Rückgabewert	
String	Aktuelle d.3 Repository-ID.

getUserNameDms()
Diese Methode ermittelt den auf 10 Bytes gekürzten d.3 Benutzernamen, welcher der Anmeldung des aktuellen Formular-Aufrufs zu Grunde liegt.

Signatur	
`String getUserNameDms ()`	
Rückgabewert	
String	Aktueller Benutzername (10 Bytes)

getUserNameDmsLong()
Diese Methode ermittelt den maximal 30 Zeichen langen d.3 Benutzernamen, welcher der Anmeldung des aktuellen Formular-Aufrufs zu Grunde liegt.

Signatur	
`String getUserNameDmsLong ()`	
Rückgabewert	
String	Aktueller d.3 Benutzername (30 Zeichen)

d.3 Login über d.ecs identity provider
Ab **d.ecs forms** Version 3.1 besteht die Möglichkeit, Formularzugriffe über den **d.ecs identity provider** 2.0 (IDP2) zu authentifizieren und authorisieren.

Zur Verwendung der **d.3** API innerhalb eines über den IDP2 autentifizierten Formulars, ist beim Aufruf des Formulars zusätzlich die Repository-ID des zu verwenden **d.3** Systems erforderlich. Der **d.es forms designer** bietet eine dementsprechende Option. Lesen Sie dazu das Kapitel Formular im Browser öffnen in der Dokumentation zur Administration von d.ecs forms.

Bei der Verwendung einer d.3 API-Funktion wird eine spezielle Loginsitzung zwischen dem **d.ecs forms** server und dem **d.3 server** mithilfe des aktuellen IDP2-Tokens der aktuellen IDP2-Sitzung ausgehandelt. Dazu ist der d.3 server in der Version 8.1 oder höher erforderlich.

> Hinweis
>
> Empfehlung:
>
> Bei der Verwendung der d.3 API innerhalb eines Formulars, das über den d.ecs identity provider angemeldet wurde, sollte der erste Zugriff auf den d.3 server frühstmöglich stattfinden. (z.B. beim Laden des Formulars)
>
> So vermeiden Sie ein potentielles Problem mit einem abgelaufen IDP2-Token, da der IDP2-Token eine begrenzte Lebenszeit besitzt.
>
> Falls ansonsten ein Problem beim Erstellen einer d3-API-Sitzung entstehen sollte, muss das Formular danach neu geladen werden, um so eine gültige IDP2-Sitzung zu erhalten. Dabei gehen alle nicht gespeicherten Formularinhalte verloren.

Subobjekt d3fc
Innerhalb der **serverseitigen Skripte** können über das d3fc Protokoll beliebige d3fc Funktionen abgesetzt werden. Bei den d3fc-Funktionen ist es im Gegensatz zu den d.3 API-Funktionen möglich, beliebige Funktionen aufrufen zu können.

Bei dem Zielserver muss es sich nicht zwangsläufig um einen d.3 server handeln. Es können andere d3fc-basierende Server angesprochen werden. Es ist hier aber auch möglich, beliebige selbsterstellte API-Funktionen im d.3 server (ab Version 8) auf zu rufen. Lesen sie hierzu das Kapitel "Groovy API-Funktionen" in der Dokumentation "d.3 server scripting (groovy)" des d.3 servers.

Es wird gundsätzlich nur `String` als Datentyp unterstützt.

Innerhalb von serverseitigen Groovy-Skripten kann direkt auf die d3fc zugegriffen werden. Hierzu muss ein zentrales Objekt erstellt werden, über das der Zugriff erfolgt.

Zugriff aus Formular-Kontext

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

Dieses Objekt stellt die Basis für jede Kommunikation mit dem d3fc-Server dar. Mit den Methoden des d3fc-Objektes können Sie den Aufruf konfigurieren, ausführen und auswerten. Die Methoden der *Konfiguration* definieren die Art des Aufrufs sowie die erforderlichen Parameter. Sie müssen alle vor der *Ausführung* der *execute*-Methode gesetzt werden. Erst nach dem *execute* können die Methoden der *Auswertung* genutzt werden, um das Ergebnis des Aufrufes zu verarbeiten.

Anwendungsbeispiel

Ab d.3 Version 8.0 besteht im d.3 server die Möglichkeit, eigene d.3 Funktionsaufrufe hinterlegen zu können. Im Folgenden wird an einem einfachen Beispiel gezeigt, wie aus d.ecs forms heraus eine benutzerspezifische d.3 Funktion aufgerufen wird. Im Beispiel wird gezeigt, wie mit Import- und Export-Parametern und mit einer Import- und Export-Tabelle gearbeitet werden kann. Die benutzerspezifische d.3 Funktion ist ebenfalls in Groovy implementiert, aber nicht in d.ecs forms sondern im d.3 server zu hinterlegen.

Beispiel

```
/*
 * 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(...)

Diese Methode setzt den Connection-Alias (aus dem d.ecs forms Objektmodell), der für die Ausführung der Funktion genutzt werden soll.

Wenn kein Connection-Alias gesetzt wird, wird (sofern vorhanden) die d.3 Verbindung aus der d.3 Authentifizierung genutzt.

`setConnectionAlias( name )`		
Parameter		
`name`	`String`	Name des Connection-Alias

Beispiel

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

setFunctionName(...) (Pflicht)

Diese Methode setzt den Namen der zu verwendenten d.3 Funktion gemäß der d.3 API Dokumentation.

Signatur		
`setFunctionName( name )`		
Parameter		
`name`	`String`	Name der Funktion

Beispiel

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

setImportParams(...)

Diese Methode setzt die Import-Parameter der zu verwendenden Funktion.

Signatur		
`setImportParams ( importParams )`		
Parameter		
`importParams`	`Map<String, String>`	Eine Map mit allen erforderlichen Import-Parametern dieser Funktion.

Beispiel

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

setImportTable(...)

Diese Methode definiert eine ImportTabelle der zu verwendenden Funktion.

Signatur		
`setImportTable ( importTable )`		
Parameter		
`importTable`	`Map<String, List<String>>`	Eine Map mit allen Spalten als Liste.

Beispiel

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

setImportTableAsText(...)

Diese Methode setzt einen String als Nutzdaten, die im d3fc-Protokoll anstelle der Tabelle angehangen werden sollen

Signatur		
`setImporTableAsText( text, encoding)`		
Parameter		
`text`	`String`	Der Text
`encoding`	`String`	Das Encoding nach Java-Standard (z.B. UTF8)

Beispiel

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

setImportTableAsFile(...)

Diese Methode setzt die angegebene Datei als Nutzdaten, die im d3fc-Protokoll angehangen werden sollen

Signatur		
`setImportTableAsFile( file)`		
Parameter		
`file`	`File`	Eine Datei als java.io.File

Beispiel

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

setImportTableAsStream(...)

Diese Methode setzt den angegebenen Stream als Nutzdaten, die im d3fc-Protokoll angehangen werden sollen. Wichtig ist dabei die Angabe der länge des Stream in Bytes.

Signatur		
`setImportTableAsStream( stream, len)`		
Parameter		
`stream`	`InputStream`	Eine java.io.InputStream oder kompatibler InputStream
len	long	die Länge des Stream

Beispiel

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

getImportTable( )

Diese Methode stellt eine Tabelle bereit, welche zur Darstellung der Import-Tabelle gemäß der d.3 API Dokumentation genutzt werden kann.

Signatur	
`getImportTable()`	
Rückgabewert	
Tabelle	Objekt zum Aufbau einer Tabellenstruktur gemäß der d.3 API Dokumentation

Beispiel

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

```
d3fc.execute()
//...
```

setExportParams(...)

Einige d3fc-Server-Implementierungen, wie z.B. der d.3 server, erfordern das Vordefinieren von Export-Parametern. Die Methode setExportParams definiert die Export-Parameter, die nach der Ausführung zurückgegeben werden sollen. Die Referenz auf die resultierenden Export-Parameter müssen nach execute mit der Methode getExportParams() geholt werden.

Signatur		
`setExportParams ( exportParams )`		
Parameter		
`exportParams`	`Map<String, String>`	Eine Map mit allen erforderlichen Export-Parametern dieser Funktion.

Beispiel

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

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

setExportTableAsBytes(...)

Diese Methode muss verwendet werden, wenn dieser Funktionsaufruf einen Byte-Array erwarten soll. Die Referenz auf den resultierenden Byte-Array muss nach execute() mit der Methode getExportBytes() geholt werden.

Signatur		
`setExportTable AsBytes( asBytes )`		
Parameter		
`asBytes`	`boolean`	true oder false. Wenn true, dann kann nach Aufruf der Funktion die Methode getExportTableAsBytes() oder getExportTableAsText(encoding) verwendet werden, um auf die Daten zugreifen zu können

Beispiel

```
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( )

Diese Methode führt die zuvor parametrisierte Funktion aus.

Signatur
`execute ( )`

getReturnCode( )

Diese Methode stellt den Return-Code der ausgeführten Funktion bereit.

Signatur	
`getReturnCode()`	
Rückgabewert	
Integer	Return-Code der ausgeführten Funktion

getReturnMessage( )

Diese Methode stellt die Return-Message der ausgeführten Funktion bereit.

Signatur	
`getReturnMessage()`	
Rückgabewert	
String	Return-Message der ausgeführten Funktion

getExportParams( )

Diese Methode liefert die resultierenden Export-Parameter der ausgeführten Funktion.

Signatur	
`getExportParams()`	
Rückgabewert	
Map<String, String>	Map mit den Export-Parametern.

Beispiel

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

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

getExportTable( )

Diese Methode liefert die resultierende Export-Tabelle der ausgeführten Funktion.

Signatur	
`getExportTable()`	
Rückgabewert	
Tabelle	Tabellen-Struktur entsprechend der Export-Tabelle gemäß der d.3 API Dokumentation

Beispiel

```
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( )

Diese Methode liefert die Export-Daten als Byte-Array der ausgeführten Funktion. Diese kann nur verwendet werden, wenn vor der Aussführung `setExportTable AsBytes`(true) gesetzt wurde.

Signatur	
`getExportTableAsBytes( )`	
Rückgabewert	
bytes[]	Export-Daten als Byte-Array

getExportTableAsText(...)

Diese Methode liefert die Export-Daten der ausgeführten Funktion die als Byte-Array definiert wurde und konvertiert diese in einen String.

Signatur	
`getExportTableAsText(encoding )`	
Parameter	
String	Encoding (nach Java-Standard) für die Byte-nach-String Konvertierung (z.B. "UTF8" oder "ISO-8859-1")
Rückgabewert	
String	Export-Daten als String

Subobjekt io

getSecureTempFile()

Die Funktion `getSecureTempFile( )` erlaubt den Zugriff auf verschlüsselte Dateien im temporären Verzeichnis.

Unverschlüsselte Dateien lassen sich damit ebenfalls verwenden.

Signatur	
`getSecureTempFile(filename)`	
Parameter	
`filename`	Name der Datei innerhalb des temporären Ordners.
Rückgabewert	
`SecureTemp-File`	Ein Objekt für die weitere Verwendung mit dem Dateinamen aus filename innerhalb des temporären Ordners
Exception	
	IOException, wenn es Probleme mit dem Dateisystem geben sollte, oder auch wenn weniger als 500 Megabyte freier Festplattenplatz auf dem Laufwerk der Installation exisitiert

```
/*
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

Ein `SecureTempFile`-Objekt erlaubt den Zugriff auf verschlüsselte und unverschlüsselte Dateien im temporären Ordner. Mit der Funktion getSecureTempFile() kann eine Instanz eines solchen Objekts erhalten werden.

Das Objekt bietet Methoden an, mit denen auf die Datei zugegriffen werden kann.

getInputStream(...)

Mit dieser Methode kann ein InputStream auf die Datei geöffnet werden. Der InputStream liefert den entschlüsselten Inhalt der Datei.

`getInputStream( )`	
Rückgabewert	
java.io.InputStream	Ein Stream, der den Inhalt der Datei liefert

getLength(...)

Diese Methode liefert die Länge des unverschlüsselten Inhalts der Datei.

`getLength( )`	
Rückgabewert	
long	Die Anzahl an Bytes, die der unverschlüsselte Inhalt der Datei hat.

getTempFile()

Mittels Funktion `getTempFile(filename)` lässt sich mit einem lokal auf dem Server befindlichen temporären Ordner arbeiten, der innerhalb der aktuellen Formularsitzung verwendet werden kann. Eine Datei existiert zu diesem Zeitpunkt nicht und muss durch einen Funktionsaufruf erzeugt werden. Der Ordner wird also leer angelegt und zudem beim Logout einer Sitzung vollständig gelöscht.

> Hinweis
>
> Dateien, die per Upload-Element hochgeladen wurden, werden verschlüsselt im temporären Verzeichnis abgelegt.
>
> Um diese Dateien lesen zu können muss die Funktion getSecureTempFile() verwendet werden. Ein normaler FileInputStream würde nur die verschlüsselten Daten auslesen.

Signatur	
`getTempFile(filename)`	
Parameter	
`filename`	Name der Datei innerhalb des temporären Ordners.
Rückgabewert	
`File`	Eine Instanz von `java.io.File` mit dem Dateinamen aus filename innerhalb des temporären Ordners
Exception	
	IOException, wenn es Probleme mit dem Dateisystem geben sollte, oder auch wenn weniger als 500 Megabyte freier Festplattenplatz auf dem Laufwerk der Installation exisitiert

```

/*
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()

Die Funktion `getUrlForTempFile(filename)` erzeugt eine URL zu einer mit der Methode `getTempFile(filename)` erzeugten Datei. Diese URL ist nur innerhalb der aktuell laufenden Benutzersitzung gültig.

> Hinweis
>
> Um im späteren Verlauf dem aktuellen Formularbenutzer diesen Download anzubieten, kann die JavaScript Funktion `form.misc.http.download(url)` verwendet werden.

Signatur	
`getUrlForTempFile(filename)`	
Parameter	
`filename`	Name der Datei die zuvor mit der Methode `getTempFile(filename)` erzeugt worden sein muss
Rückgabewert	
`String`	Die URL als 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. Subobjekt instance

save()

Die Funktion `save()` speichert die konkreten Inhalte der Daten-Property in den Fremdsystemen UND, wenn eine Instanz-Id vorhanden ist, zusätzlich auch in der internen Persistenzschicht von d.ecs forms.

Signatur
`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()

Die Funktion `saveInstance()` speichert die Instanz des Formulars. Eine ggf. selektierte Subinstanz wird dabei mitgespeichert.

Signatur
`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(...)

Die Funktion `closeInstance()` schließt die Instanz des Formulars. Die Werte der Instanz werden dabei archiviert. Diese Instanz und auch dessen Instanz-ID können danach nicht mehr in Formularen verwendet werden.

Eine ggf. selektierte Subinstanz wird dabei mit geschlossen. Es dürfen dabei jedoch keine weiteren offene Subinstanzen existieren. Optional kann angegeben werden, nach wie vielen Tagen die Instanz komplett gelöscht werden soll.

Signatur		
`closeInstance() closeInstance( daysToLive )`		
Importparameter		
`daysToLive`	Integer	Der Parameter gibt an wie lange die Instanz noch archiviert bleiben soll. Nach daysToLive Tagen wird die Instanz komplett gelöscht. Bei Eingabe des Wertes '0' wird die Instanz zum nächst möglichen Zeitpunkt gelöscht. Dies ist nach maximal 6 Stunden oder nach einem Neustart des d.ecs forms servers.

```
/*

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(...)

Über die Funktion `setSubInstanceId(...)` kann die Id der Subinstanz gesetzt werden. Falls diese Subinstanz bereits existiert, also zuvor gespeichert wurde, wird deren Inhalt an dieser Stelle wieder geladen.

Signatur		
`setSubInstanceId( subInstanceId)`		
Importparameter		
`subInstanceId`	String	Die Id der Subinstanz. Subinstanzen sind Kinder einer Instanz und gelten nur innerhalb der aktuell verwendeten (Haupt-)Instanz. Die subInstanzId daf nicht null sein. (Siehe 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);
}
```

Achtung

Das Setzen einer Subinstanz gilt nur für die aktuelle Formularsitzung und wird nicht persistiert. Das Setzen muss also bei jedem Laden eines Formulars wiederholt werden.

Achtung

Innerhalb einer Transaktion ist das Setzen bzw. Ändern der Subinstanz Id nicht möglich.

removeSubInstanceId()

Über die Funktion `removeSubInstanceId()` lässt sich die mit `setSubInstanceId( subInstanceId)` gesetzte subinstanceId wieder entfernen. Anschließend wird wieder ausschließlich auf der Hauptinstanz gearbeitet.

Signatur
`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()

Die Funktion `saveSubInstance()` speichert nur die Subinstanz des Formulars.

Signatur
`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(...)

Die Funktion `closeInstance(...)` schließt die angegebene Subinstanz ab. Die Werte der Subinstanz werden dabei archiviert. Diese Subinstanz kann danach nicht mehr verwendet werden. Es kann auch keine weitere Subinstanz mit dieser ID innerhalb der aktuellen (Haupt-)Instanz mehr angelegt werden.

Signatur		
`closeSubInstance(subInstanceId)`		
Parameter		
`subInstanceId`	String	Subinstanz-ID, die geschlossen werden soll

```
/*

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()

Über die Funktion `getSubInstanceId()` lässt sich die mit `setSubInstanceId( subInstanceId)` gesetzte subinstanceId ermitteln.

Signatur		
`String getSubInstanceId()`		
Rückgabewert		
`subInstanceId`	String	Die Id der Subinstanz oder null wenn zuvor keine gesetzt wurde. Subinstanzen sind Kinder einer Instanz und gelten nur lokal innerhalb der aktuell verwendeten (Haupt-) Instanz.

```
/*

Use-case Description:

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

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

getSubContext(...)

Über die Funktion `getSubContext(subInstanceId)` lassen sich die Werte einer anderen Subinstanz auslesen. Das zurückgegebene Groovy-Objekt besitzt alle lesenden Methoden wie auch das bisher bekannte context-Objekt (form.context)

Signatur		
`SubContextInstance getSubContext( subInstanceId)`		
Importparameter		
`subInstanceId`	String	Die Id der Subinstanz. Die subInstanceId darf nicht null sein.

Signatur		
Rückgabewert		
	SubContextInstance	Lesender Zugriff auf alle Werte der Subinstanz.

```
/*
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

Ein SubContext wird bei der parallelen Verarbeitung von Instanzen verwendet. Er referenziert die Werte einer bereits in der Datenbank gespeicherten Subinstanz und bietet lesenden Zugriff darauf. Schreibender Zugriff ist nicht möglich. Als Speicherort muss zuvor im d.ecs forms designer die Option "Subinstanz" ausgewählt werden. Nur dann handelt es sich um separate Werte jeder Subinstanz.

getValue(...)

Die Methode ermittelt den ersten bzw. den indizierten Wert einer Eigenschaft (innerhalb der Subinstanz)

Signatur		
`getValue( propertyName )`		
`getValue(propertyName, index )`		
Importparameter		
`propertyName`	String	Name der Eigenschaft
index	Integer	Index
Rückgabewert		
	Object (entsprechend dem Datentyp)	Der Wert der Eigenschaft

getValues(...)

Die Methode ermittelt alle Werte einer Eigenschaft (innerhalb der Subinstanz)

Signatur		
`getValues( propertyName)`		
Importparameter		
`propertyName`	String	Name der Eigenschaft
Rückgabewert		
	List<Object> (entsprechend dem Datentyp)	Die Werte der Eigenschaft

getAllSubInstanceIds()

Über die Funktion `getAllSubInstanceIds()` lassen sich alle ID's der zu dieser Instanz gespeicherten Subinstanzen ermitteln.

Signatur		
`List<String> getAllSubInstanceIds()`		
Rückgabewert		
`subInstanceIds`	List<String>	Alle ID's der Subinstanzen als Liste von 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()

Über die Funktion `getRemoteContext()` können die aktuellen Instanzwerte aus der Datenbank ermittelt werden. Hierbei handelt es sich um einen rein lesenden Zugriff. Sollte die Instanz noch nie gespeichert worden und demnach nicht in der Datenbank vorhanden sein, liefert diese Methode `NULL`.

Signatur		
`RemoteContext getRemoteContext( )`		
Rückgabewert		
RemoteContext		Die aktuellen Instanzwerte der Datenbank, oder `NULL`, wenn die Instanz noch nicht in der Datenbank vorhanden ist.

```
/*

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

Über ein `RemoteContext` -Objekt können die Werte von Eigenschaften ermittelt werden, wie sie aktuell in der Datenbank vorhanden sind.

Zur Erzeugung eines `RemoteContext` Objektes kann die Methode `form.instance.getRemoteContext()` verwendet werden.

getValue(...)

Über die Funktion `getValue(...)` wird der Wert einer Eigenschaft im Remote-Kontext ausgelesen.

Signatur		
`Object getValue( path )`		
`Object getValue( path, index )`		
Importparameter		
`path`	String	Der Name der Eigenschaft im Remote-Kontext
`index`	Integer	Index des Wertes in der Liste

Signatur	
Rückgabewert	
	Bei `getValue( name )` wird der entsprechende Wert des Parameters im entsprechenden Datentyp zurückgegeben (`String, Number, Date, ...`).
	Bei `getValue( name, index )` wird der Wert an der jeweiligen Position innerhalb der Liste zurückgegeben.

getValues(...)

Über die Funktion `getValues(...)` wird der Wert einer Eigenschaft im Remote-Kontext als Liste ausgelesen.

Signatur		
`List<Object> getValues( path )`		
Importparameter		
`path`	String	Der Name der Eigenschaft im Remote-Kontext
Rückgabewert		
	List<Object>	Liste der Werte

getRemoteChanges(...)

Über die Funktion `getRemoteChanges(...)` können alle Wertänderungen, die durch einen anderen Benutzer in der Instanz-Datenbank parallel durchgeführt wurden, ermittelt werden. Dabei handelt es sich um einen Vergleich zwischen den aktuell in der Instanz-Datenbank vorhandenen Daten mit der Revision der Instanz-Datenbank zum Zeitpunkt des Ladens (bzw. des letzten Speicherns oder Mergens) in der aktuellen Formular-Sitzung.

Signatur	
`RemoteChanges getRemoteChanges()`	
Rückgabewert	
RemoteChanges	Objekt, welches sämtliche Wertänderungen der Datenbank gegenüber den lokalen Daten repräsentiert.

> Achtung
>
> Wertänderungen, die durch einen anderen Benutzer in der Instanz-Datenbank parallel durchgeführt wurden, beziehen sich ausschließlich auf die Hauptinstanz. Subinstanzen werden hier nicht berücksichtigt, da diese von ihrem Anwendungszweck nicht von mehreren Benutzern parallel genutzt werden sollen.

> Beispiel
>
> Für Beispiele zur Verwendung dieses APIs schauen Sie bitte in die Templates (Server-Aktionen) zur Übernahme von Remote-Daten im d.ecs forms designer

RemoteChanges

Ein Objekt vom Typ `RemoteChanges` repräsentiert die Änderung der Werte in der Instanz-Datenbank gegenüber dem Datenmodell der aktuellen Formular-Sitzung. Bei einer Änderung kann es sich um das Hinzufügen, das Aktualisieren oder das Entfernen eines Wertes handeln.

getChangesInProperties(...)

Diese Methode filtert die ermittelten Wertänderungen jeweils pro Eigenschaft. Es wird **kein** tabellarischer Zusammenhang zwischen den Eigenschaften eines Objektes hergestellt.

> Achtung
>
> Eigenschaften, die nicht in die Instanz gespeichert werden, werden hierbei nicht berücksichtigt.

Signatur		
`RemotePropertyChanges getChangesInProperties( )`		
`RemotePropertyChanges getChangesInProperties( propertiesOrObjects )`		
Importparameter		
propertiesOrObjects	String[]	Array von Eigenschaften oder Objekten, für die die Änderungen ermittelt werden sollen. Diese Methode wird aus Performance-Gründen empfohlen, wenn in einem großen Datenmodell die Änderungen von nur wenigen Eigenschaften ermittelt werden sollen.
Rückgabewert		
	RemotePropertyChanges	Objekt, welches sämtliche Wertänderungen der Datenbank gegenüber den lokalen Daten repräsentiert.

RemotePropertyChanges

Ein Objekt vom Typ `RemotePropertyChanges` repräsentiert eine Gruppe von Änderungen in der Instanz-Datenbank gegenüber dem Datenmodell der aktuellen Formular-Sitzung. Bei diesen Änderungen wird danach unterschieden, ob ein neuer Wert hinzugefügt, ein existierender Wert geändert oder aber gelöscht wurde.

getCreations()

Die Methode ermittelt alle Änderungen, bei denen ein neuer Wert hinzugefügt wurde.

Signatur	
`Collection<RemoteValueChange> getCreations()`	
Rückgabewert	
Collection<RemoteValueChange>	Collection aller entsprechenden Änderungen. Auf diese Collection können die Groovy-Standard-Methoden für Collections angewendet werden.

getUpdates()

Die Methode ermittelt alle Änderungen, bei denen ein existierender Wert geändert wurde.

Signatur	
`Collection<RemoteValueChange> getUpdates()`	
Rückgabewert	
Collection<RemoteValueChange>	Collection aller entsprechenden Änderungen. Auf diese Collection können die Groovy-Standard-Methoden für Collections angewendet werden.

getDeletions()

Die Methode ermittelt alle Änderungen, bei denen ein existierender Wert gelöscht wurde.

Signatur	
`Collection<RemoteValueChange> getDeletions()`	
Rückgabewert	
Collection<RemoteValueChange>	Collection aller entsprechenden Änderungen. Auf diese Collection können die Groovy-Standard-Methoden für Collections angewendet werden.

RemoteValueChange

Ein Objekt vom Typ `RemoteValueChange` repräsentiert die Änderung eines Wertes in der Instanz-Datenbank gegenüber dem Datenmodell der aktuellen Formular-Sitzung. Bei einer Änderung kann es sich um das Hinzufügen, das Aktualisieren oder das Entfernen eines Wertes handeln.

getObject()

Die Methode ermittelt den Namen des Objektes, zu dem die Eigenschaft aus `getProperty()` gehört.

Signatur	
`String getObject()`	
Rückgabewert	
String	Der Name des Objektes

getProperty()

Die Methode ermittelt den Namen der Eigenschaft, deren Wert durch das Objekt repräsentiert wird.

Signatur	
`String getProperty()`	
Rückgabewert	
String	Der Name der Eigenschaft

getRemoteIndex()

Die Methode ermittelt den Index des Wertes innerhalb der Werteliste in der Datenbank

Signatur	
`Integer getRemoteIndex()`	
Rückgabewert	
Integer	Der Index in der Datenbank

getRemoteValue()

Die Methode ermittelt den eigentlichen Wert in der Datenbank

Signatur	
`Object getRemoteValue()`	
Rückgabewert	
Object (entsprechend dem Datentyp)	Der Wert in der Datenbank

getLocalIndex()

Die Methode ermittelt den Index des Wertes innerhalb der aktuellen lokalen Werteliste

Signatur	
`Integer getLocalIndex()`	
Rückgabewert	
Integer	Der aktuelle lokale Index

getChangesInRows(...)

Diese Methode filtert die ermittelten Wertänderungen jeweils pro Zeile. Es wird ein tabellarischer Zusammenhang zwischen den Mehrfachwert-Eigenschaften eines Objektes hergestellt.

> Achtung
>
> Eigenschaften vom Typ 'Einfachwert' und Eigenschaften, die nicht in die Instanz gespeichert werden, werden hierbei nicht berücksichtigt.

Signatur
`RemoteRowChanges getChangesInRows( )`
`RemoteRowChanges getChangesInRows( objects )`
Importparameter

Signatur		
objects	String[]	Array von Objekten, für die die Änderungen ermittelt werden sollen. Diese Methode wird aus Performance-Gründen empfohlen, wenn in einem großen Datenmodell die Änderungen von nur wenigen Objekten ermittelt werden sollen.
Rückgabewert		
	RemoteRowChanges	Objekt, welches sämtliche Zeilenänderungen der Datenbank gegenüber den lokalen Daten repräsentiert.

Achtung

Bei `getChangesInRows(...)` werden nur Eigenschaften vom Typ `Mehrfachwert` berücksichtigt.

RemoteRowChanges

Ein Objekt vom Typ RemoteRowChanges repräsentiert eine Gruppe von Änderungen in der Instanz-Datenbank gegenüber dem Datenmodell der aktuellen Formular-Sitzung. Jedes Änderungsobjekt repräsentiert dabei eine Zeile innerhalb eines definierten Objektes. Bei diesen Änderungen wird danach unterschieden, ob ein neuer Wert hinzugefügt, ein existierender Wert geändert oder aber gelöscht wurde.

getRowCreations()

Die Methode ermittelt alle Änderungen, bei denen eine neue Zeile hinzugefügt wurde.

Signatur	
`Collection<RemoteRowValueChange> getRowCreations()`	
Rückgabewert	
Collection<RemoteRowValueChange>	Collection aller entsprechenden Änderungen. Auf diese Collection können die Groovy-Standard-Methoden für Collections angewendet werden.

getRowUpdates()

Die Methode ermittelt alle Änderungen, bei denen mindestens ein existierender Wert einer existierenden Zeile geändert wurde.

Signatur	
`Collection<RemoteRowValueChange> getRowUpdates()`	
Rückgabewert	
Collection<RemoteRowValueChange>	Collection aller entsprechenden Änderungen. Auf diese Collection können die Groovy-Standard-Methoden für Collections angewendet werden.

getRowDeletions()

Die Methode ermittelt alle Änderungen, bei denen eine existierende Zeile gelöscht wurde.

Signatur	
`Collection<RemoteRowValueChange> getRowDeletions()`	
Rückgabewert	
Collection<RemoteRowValueChange>	Collection aller entsprechenden Änderungen. Auf diese Collection können die Groovy-Standard-Methoden für Collections angewendet werden.

RemoteRowValueChange

Ein Objekt vom Typ `RemoteRowValueChange` repräsentiert die Änderung einer Zeile eines definierten Objektes in der Instanz-Datenbank gegenüber dem Datenmodell der aktuellen Formular-Sitzung. Bei

einer Änderung kann es sich um das Hinzufügen, das Aktualisieren oder das Entfernen einer Zeile handeln.

getObject()

Die Methode ermittelt den Namen des Objektes.

Signatur	
`String getObject()`	
Rückgabewert	
String	Der Name des Objektes

getProperties()

Die Methode ermittelt die Namen aller Eigenschaften vom Typ Mehrfachwert innerhalb des Objektes

Signatur	
`List<String> getProperties()`	
Rückgabewert	
List<String>	Liste der Namen

getChangedProperties()

Die Methode ermittelt die Namen aller Eigenschaften vom Typ Mehrfachwert innerhalb des Objektes, die von der Änderung betroffen sind

Signatur	
`List<String> getChangedProperties()`	
Rückgabewert	
List<String>	Liste der Namen

getRemoteIndex()

Die Methode ermittelt den Index der Zeile innerhalb des Objektes in der Datenbank

Signatur	
`Integer getRemoteIndex()`	
Rückgabewert	
Integer	Der Index in der Datenbank

getRemoteValue(String property)

Die Methode ermittelt den eigentlichen Wert einer bestimmten Eigenschaft (Spalte) innerhalb der Zeile in der Datenbank

Signatur		
`Object getRemoteValue( String property)`		
Importparameter		
`property`	String	Der Name der Eigenschaft (Spalte)
Rückgabewert		
	Object (entsprechend dem Datentyp)	Der Wert in der Datenbank

getChangeInProperty(String property)

Die Methode ermittelt die Wertänderung für eine bestimmte Eigenschaft (Zelle) innerhalb der Zeile in der Datenbank.

Signatur
`RemoteValueChange getChangeInProperty( String property)`

Signatur	
Rückgabewert	
RemoteValueChange	Die Wertänderung der Eigenschaft (Zelle) innerhalb der Zeile.

getLocalIndex()

Die Methode ermittelt den Index des Wertes innerhalb der aktuellen lokalen Werteliste

Signatur	
`Integer getLocalIndex()`	
Rückgabewert	
Integer	Der aktuelle lokale Index

markAsMerged(...)

Diese Methode markiert sämtliche in diesem Objekt ermittelten Wertänderungen als synchronisiert. Das hat folgende Auswirkungen

- Ein erneuter Aufruf von `getRemoteChanges(...)` gegen den Datenbank-Stand, der diesem Objekt zu Grunde liegt, führt zu keinem Ergebnis mehr
- Das Speichern der aktuellen Instanz führt zu keinem weiteren Konflikt, sofern der Stand in der Datenbank nach dem Aufruf von `getRemoteChanges(...)` nicht abermals durch einen anderen Benutzer aktualisiert wurde.

Signatur
markAsMerged()

beginRemoteTransaction()

Über die Funktion `beginRemoteTransaction()` wird eine datenbankgesteuerte Transaktion begonnen. Es ist sichergestellt, dass zu einem Zeitpunkt immer nur maximal eine Transaktion geöffnet sein kann. Andere Transaktionen auf der gleichen Formularinstanz werden während dessen blockiert.

> Achtung
>
> Um eine solche Transaktion öffnen zu können, muss eine Instanz-ID vorhanden sein.

Signatur		
`RemoteTransaction beginRemoteTransaction()` `RemoteTransaction beginRemoteTransaction( timeoutSeconds)`		
Importparameter		
`timeoutSeconds`	Integer	Zeit in Sekunden, nach der der Versuch eine Transaktion zu starten abgebrochen wird. Dies ist relevant, falls die Instanz durch eine andere Transaktion (zu lange) blockiert wird. Achtung Bitte beachten Sie, dass die Angabe eines Timeouts eine Implementierung des jeweiligen Datenbank-Managament-Systems (DBMS) ist. Aus diesem Grund kann es hierbei je nach verwendetem DBMS zu unterschiedlichem Verhalten kommen. Bitte prüfen Sie die für Ihren Anwendungsfall gewünschte Verhaltensweise daher in Verbindung mit Ihrem DBMS.
Rückgabewert		
	RemoteTransaction	Objekt mit einigen Methoden die innerhalb der aktiven Transaktion verwendet werden können:

```
/*
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

Transaktionen in **d.ecs forms** bieten die Möglichkeit, Änderungen am Datenmodell zunächst temporär durchzuführen, um diese dann abschließend entweder zu übernehmen oder ggf. zu verwerfen. Eine `RemoteTransaction` öffnet darüber hinaus eine sperrende Transaktion in der Instanz-Datenbank. Es ist dadurch sichergestellt, dass zu einem Zeitpunkt immer nur maximal eine Transaktion geöffnet sein kann. Andere Transaktionen auf der gleichen Formularinstanz werden während dessen blockiert. Das Blockieren gilt nur von der `beginRemoteTransaction()`- bis zur `commit()`-Zeile. Sollten sich zuvor während der laufenden Formularsitzung (z.B. durch einen anderen Benutzer) Änderungen an der Instanz ergeben haben, wird eine Exception geworfen. Wird diese nicht behandelt, führt dies im Clientskript zu dem Return-Code -3.

Zur Erzeugung eines `RemoteTransaction` Objektes kann die Methode form.instance.beginRemoteTransaction() verwendet werden.

> Achtung
>
> Solange nicht die `transaction.commit()` Methode auf dieser `RemoteTransaction` aufgerufen wird, werden keine finalen Änderungen an der Datenbank vorgenommen.
>
> **Beispiel**
>
> ```
> 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()

Muss zwingend aufgerufen werden, um die Transaktion abzuschließen und alle Änderungen tatsächlich zu übernehmen. Wird diese Methode innerhalb eines Skriptes nicht aufgerufen, wird ein Rollback durchgeführt.

Signatur
`commit()`

> Achtung
>
> Das Commit einer Transaktion beendet diese gleichzeitig. Soll anschließend weiterhin mit einer Transaktion gearbeitet werden, so muss eine solche erneut per `form.instance.beginRemoteTransaction()` geöffnet werden.
>
> Das Commit einer Transaktion hat ferner zur Folge, dass alle innerhalb der Transaktion an der Instanz durchgeführten Änderungen übernommen werden. Entsprechende Änderungen, die währenddessen **außerhalb** der Transaktion durchgeführt wurden, werden dadurch überschrieben. Aus diesem Grund wird dringend empfohlen, Änderungen an der Instanz bei aktiver Transaktion ausschließlich mit den Methoden `transaction.context.*` und `transaction.instance.*` durchzuführen.

Subobjekt context

Bietet eine zu `form.context` analoge Schnittstelle zum lokalen Datenmodell. Diese Schnittstelle verhält sich jedoch in der Form transaktional, dass die hier gemachten Änderungen erst dann wirksam werden, wenn die Transkation mit `commit()` beendet wird.

Subobjekt instance

Bietet eine zu `form.instance` analoge Schnittstelle. Diese beschränkt sich auf die folgenden Methoden:

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

> Hinweis
>
> Der Aufruf dieser Methoden innerhalb von `transaction.instance` bewirkt (analog zu `transaction.context`), dass die dort gemachten Änderungen erst beim Aufruf von `commit()` wirksam werden.

acceptCreation(...)

Über die Funktion `acceptCreation(...)` kann ein neuer Wert, der durch einen anderen Benutzer in der Instanz-Datenbank hinzugefügt wurde, in die lokalen Daten übernommen werden.

Signatur		
`acceptCreation( index, remoteChange)`		
`acceptCreation( remoteChange)`		
Importparameter		
`index`	Integer	Ziel-Index des Wertes im lokalen Datenmodell
remoteChange	RemoteValueChange	Änderungsobjekt, welches die Wertänderung der Datenbank gegenüber den lokalen Daten repräsentiert.

```
/*

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(...)

Über die Funktion `acceptUpdate(...)` kann eine Änderung an einem Wert, die durch einen anderen Benutzer in der Instanz-Datenbank gemacht wurde, in die lokalen Daten übernommen werden.

Signatur		
`acceptUpdate( remoteChange)`		
Importparameter		
remoteChange	RemoteValueChange	Änderungsobjekt, welches die Wertänderung der Datenbank gegenüber den lokalen Daten repräsentiert.

```
/*

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(...)

Über die Funktion `acceptDeletion(...)` kann das Löschen eines Wertes, der durch einen anderen Benutzer aus der Instanz-Datenbank entfernt wurde, in die lokalen Daten übernommen werden.

Signatur		
`acceptDeletion( remoteChange)`		
Importparameter		
remoteChange	RemoteValueChange	Änderungsobjekt, welches die Wertänderung der Datenbank gegenüber den lokalen Daten repräsentiert.

```
/*

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(...)

Über die Funktion `acceptRowCreation(...)` kann eine neue Zeile, die durch einen anderen Benutzer in der Instanz-Datenbank hinzugefügt wurde, in die lokalen Daten übernommen werden.

Signatur		
`acceptRowCreation( index, remoteRowValueChange)`		
`acceptRowCreation( remoteRowValueChange)`		
Importparameter		
`index`	Integer	Ziel-Index des Wertes im lokalen Datenmodell
remoteRowValueChange	R emoteRowValueChange	Änderungsobjekt, welches die Wertänderung der Datenbank gegenüber den lokalen Daten repräsentiert.

```
/*

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(...)

Über die Funktion `acceptRowUpdate(...)` können Änderungen an mindestens einem Wert in einer Zeile, die durch einen anderen Benutzer in der Instanz-Datenbank gemacht wurde, in die lokalen Daten übernommen werden.

Signatur		
`acceptRowUpdate( remoteRowValueChange)`		
`acceptRowUpdate( remoteRowValueChange, includeUnchangedRemoteColumns)`		
Importparameter		
remoteRowValueChange	RemoteRowValueChange	Änderungsobjekt, welches die Wertänderung der Datenbank gegenüber den lokalen Daten repräsentiert.
includeUnchangedRemoteColumns	Boolean	Wenn true, dann werden auch die Werte der Spalten übernommen, die NICHT parallel geändert wurden. (Default: false)

> Achtung
>
> Wenn `includeUnchangedRemoteColumns` mit '`false`' (Default) übergeben wird, dann werden ausschließlich die Werte in das lokale Datenmodell überführt, die in der Datenbank tatsächlich geändert wurden. Lokale Änderungen in den übrigen Spalten werden nicht überschrieben.

```
/*

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(...)

Über die Funktion `acceptRowDeletion(...)` kann das Löschen einer Zeile, die durch einen anderen Benutzer aus der Instanz-Datenbank entfernt wurde, in die lokalen Daten übernommen werden.

Signatur		
`acceptRowDeletion( remoteRowValueChange)`		
Importparameter		
remoteRowValueChange	R emoteRowValueChange	Änderungsobjekt, welches die Wertänderung der Datenbank gegenüber den lokalen Daten repräsentiert.

```
/*

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(...)

Über die Funktion `hasLocalChanges(...)` kann geprüft werden, ob eine Eigenschaft oder ein Wert seit dem letzten Speicherzeitpunkt (bzw. ggf. dem letzten Laden) lokal geändert wurde. Die Funktion kann folgendermaßen genutzt werden:

- Ermitteln von lokalen Änderungen, die mit Wertänderungen in der Datenbank in Konflikt stehen
- Ermitteln von lokalen Änderungen innerhalb eines Objektes
- Ermitteln von lokalen Änderungen innerhalb einer Eigenschaft

Signatur		
`boolean hasLocalChanges( remoteChange)`		
`boolean hasLocalChanges( objectOrPropertyName)`		
Importparameter		
remoteChange	RemoteValueChange	Änderungsobjekt, welches die Wertänderung der Datenbank gegenüber den lokalen Daten repräsentiert.
objectOrPropertyName	String	Name eines Objekts oder einer Eigenschaft
Rückgabewert		
	boolean	`true`, wenn der Wert oder die Eigenschaft lokale Änderungen aufweist.

```
/*

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(...)

Über die Funktion `hasLocalRowChanges(...)` kann geprüft werden, ob eine Zeile seit dem letzten Speicherzeitpunkt (bzw. ggf. dem letzten Laden) lokal geändert wurde. Dabei ist es unerheblich, in welcher Spalte der Zeile die lokale Änderung stattgefunden hat. Die Funktion kann folgendermaßen genutzt werden:

- Ermitteln von lokalen Änderungen, die mit Zeilenänderungen in der Datenbank in Konflikt stehen

Signatur		
`boolean hasLocalRowChanges( remoteChange)`		
Importparameter		
remoteChange	RemoteRowValueChange	Änderungsobjekt, welches die Zeilenänderung der Datenbank gegenüber den lokalen Daten repräsentiert.
Rückgabewert		
	boolean	`true`, wenn der Wert oder die Eigenschaft lokale Änderungen aufweist.

> Achtung
>
> Wenn die Zeile nicht existiert, wird `false` zurückgegeben

```
/*

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. Subobjekt convert

toPDF(...)

Über die Funktion `toPDF()` kann ein Formular in ein PDF-Dokument konvertiert werden.

Signatur		
ConverterInstance toPDF()		
Rückgabewert		
`ConverterInstance`		Mit diesem Objekt kann der Konvertierungsprozess weiter konfiguriert werden.

```
/*

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(...)

Über die Funktion `doNotPrint()` können Elemente selektiv aus dem erzeugten PDF ausgeblendet werden.

Signatur		
`void doNotPrint( elementId ) void doNotPrint( elementsArray )`		
`void doNotPrint( elementsList )`		
Importparameter		
elementId	String	Der Name eines Formular-Elements, das ausgeblendet werden soll.
`elementsArray`	String[]	Ein Array mit den Namen mehrerer Formular-Elemente, die ausgeblendet werden sollen.
`elementsList`	List<String>	Eine Liste mit den Namen mehrerer Formular-Elemente, die ausgeblendet werden sollen.

> Achtung
>
> Die Funktion unterstützt keine Tabellenspalten. Wird der Name einer Tabellenspalte übergeben, wird eine Exception ausgelöst.

```
/*

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(...)

Mit der Funktion `execute()` wird die PDF-Konvertierung mit den gewählten Parametern durchgeführt.

Signatur
`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(...)

Über die Funktion `setFormId()` kann ein anderes Formular für die PDF-Konvertierung ausgewählt werden.

Signatur		
void setFormId(formId)		
Importparameter		
`formId`	String	Die eindeutige ID des Formulars, das für die Konvertierung verwendet werden soll.

```
/*

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(...)

Über die Funktion `setFormVersion()` kann ein anderer Formular-Build für die PDF-Konvertierung ausgewählt werden.

Signatur		
`void setFormVersion( formVersion )`		
Importparameter		
`formVersion`	Integer	Die Build-Nummer des gewünschten Formulars.

```
/*
```

```
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(...)

Über die Funktion `setLocale()` kann die Sprache für die PDF-Konvertierung gewählt werden.

Signatur		
`void setLocale( locale )`		
Importparameter		
locale	String	Ein String mit dem Sprachcode der gewünschten Spracheinstellung.

> Hinweis
>
> Verwenden Sie den Parameter `locale` als String beachten Sie das IETF BCP 47 Format. Also zum Beispiel:
>
> "de" für Deutsch.
>
> "de-AT" für Deutsch (Österreich)

```
/*

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(...)

Über die Funktion `setOutputStream()` muss ein OutputStream angegeben werden. Das generierte PDF-Dokument wird beim Aufruf von execute() dort hin geschrieben werden.

Signatur		
`void setOutputStream(os)`		
Importparameter		
os	OutputStream	Ein 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(...)

Über die Funktion `setSubInstanceId()` wird eine andere Subinstanz als Basis für die PDF-Konvertierung ausgewählt.

Signatur		
`void setSubInstanceId( subInstanceId )`		
Importparameter		
`subInstanceId`	String	Der Name der gewünschten Subinstanz.

```
/*

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. Datenstrukturen

Tabelle

Diese Datenstruktur repräsentiert Tabellen mit Zeilen und Spalten. Eine Tabelle ist iterierbar, was bedeutet, dass sie in den entsprechenden Groovy-Loop-Konstrukten (for-each, each, find etc.) genutzt werden kann. Dabei wird über eine Tabellenzeile iteriert.

Eine Tabelle wird z.B. zum Abbilden der Import- und Export-Tabellen bei d.3 API-Funktionen oder auch in Datenquellen (ds.data.getTable()) verwendet.

getValue(...)

Die Methode ermittelt den Wert einer Tabellenzelle

Signatur		
`getValue( colName, index )`		
Importparameter		
`colName`	String	Der Name der Spalte
`index`	Integer	Zeilenindex
Rückgabewert		
	Object (entsprechend dem Datentyp)	Der Wert der entsprechenden Zelle

getValues(...)

Die Methode ermittelt alle Werte einer Spalte

Signatur		
`getValues( colName )`		
Importparameter		
`colName`	String	Der Name der Spalte
Rückgabewert		
	List<Object> (entsprechend dem Datentyp)	Die Werte der entsprechenden Spalte

size( )

Die Methode ermittelt die maximale Anzahl der Zeilen

Signatur	
`size( )`	
Rückgabewert	
Integer	Die maximale Anzahl der Zeilen

iterator( )

Die Methode erstellt einen Iterator, mit dem über die Tabellenzeilen iteriert werden kann.

Signatur	
`iterator( )`	
Rückgabewert	
Iterator<Tabellenzeile>	Ein Iterator über alle Zeilen

setValue(...)

Die Methode setzt den Wert einer Zelle

Signatur		
`setValue( colName, index, value )`		
Importparameter		
`colName`	String	Der Name der Spalte
`index`	Integer	Zeilenindex
`value`	Object (entsprechend dem Datentyp)	Der neue Wert

addValue(...)

Die Methode fügt einer Spalte in der Tabelle einen Wert hinzu. Bestehende Werte werden ab dem angegebenen Index um eine Stelle nach hinten verschoben.

Signatur		
`addValue( colName, value )` `addValue( colName, index, value )`		
Importparameter		
`colName`	String	Der Name der Spalte in der Tabelle
`index`	Integer	Index, an dem der Wert eingefügt werden soll. Wenn nicht angegeben, wird der Wert am Ende der Liste angehangen.
`value`	Object (entsprechend dem Datentyp)	Der neue Wert

setValues(...)

Die Methode setzt alle Werte einer Spalte

Signatur		
`setValues( colName, values )`		
Importparameter		
`colName`	String	Der Name der Spalte
`values`	List<Object> (entsprechend dem Datentyp)	Die neuen Werte

deleteValue(...)

Die Methode leert den Wert einer Zelle

Signatur		
`deleteValue( colName, index )`		
Importparameter		
`colName`	String	Der Name der Spalte
`index`	Integer	Zeilenindex

deleteValues(...)

Die Methode leert alle Werte einer Spalte

Signatur		
`deleteValues( colName )`		
Importparameter		
`colName`	String	Der Name der Spalte

Tabellenzeile

Diese Datenstruktur repräsentiert eine Zeile einer Tabelle. Diese kann explizit über die iterator()-Funktion, oder implizit durch die Nutzung von Loop-Konstrukten (for, each, find etc.) ermittelt werden.

Ein Objekt vom Typ Tabellenzeile ist kompatible zu einer `Map<String,Object>` wobei dessen Methoden `containsValue()`, `remove()` und `clear()` nicht implemetiert sind. Ansonsten lässt sich ein solches Objekt wie eine Map mit dessen Methoden `get()` oder `put()` verwenden.

getValue(...)

Die Methode ermittelt den Wert einer Tabellenzelle

Signatur		
`getValue( colName )`		
`get( colName )`		
Importparameter		
`colName`	String	Der Name der Spalte
Rückgabewert		
	Object (entsprechend dem Datentyp)	Der Wert der entsprechenden Zelle

setValue(...)

Die Methode setzt den Wert einer Zelle

Signatur		
`setValue( colName, value )`		
`put( colName, value )`		
Importparameter		
`colName`	String	Der Name der Spalte
value	Object (entsprechend dem Datentyp)	Der neue Wert

getRowNumber( )

Die Methode gibt die Zeilennummer der aktuellen TableRow an

Signatur	
`getRowNumber( )`	
Rückgabewert	
Integer	Die Zeilennummer

1.2.10. Allgemeine Methoden

fail(...)

Mit `fail(...)` kann im Fehlerfall die Ausführung eines Skriptes beendet und eine Nachricht an den Client weitergeleitet werden.

Signatur		
`void fail( errorCode, errorMessage )`		
Importparameter		
`errorCode`	int	Ein positiver Fehlercode.
`errorMessage`	String	Ein beschreibender Fehlertext.
Anwendungsbereich		
Werfen einer selbst definierten Ausnahme im Fehlerfall. Der Client kann in der onFailure-Funktion den Fehlercode und den Fehlertext auswerten. Der Aufruf dieser Funktion beendet die Ausführung des Skriptes.		

> Achtung
>
> Alle Änderungen, die vor dem Aufruf von `fail(...)` mit `context.setValue(...)` bzw. `context.setValues(...)` am Formular-Kontext vorgenommen wurden, bleiben erhalten.

```
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-Module

modules(...)

Die Funktion `modules(...)` ermöglicht den Zugriff auf ein Groovy-Modul, das im Formular-Kontext oder Objektmodell referenziert wird.

Signatur		
`Module modules( moduleName )`		
Importparameter		
`moduleName`	String	Der Name des Moduls, welches geladen werden soll.
Rückgabewert		
	Module	Das angeforderte Modul.

Mit dieser Funktion ist es möglich, auf globale Module zuzugreifen.

> Achtung
>
> Stellen Sie sicher, dass das gewünschte Modul im Formular-Kontext oder dem Objektmodell inkludiert wird.
>
> Wird ein Modulname übergeben, der im Formular-Kontext nicht existiert, kommt es zu einem Fehler und das Formular wird beendet.

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

1.3. Abgekündigte Methoden

Subobjekt	Methode	Abgekündigt mit Version	Hinweise für möglichen Ersatz

Innerhalb des d.3 API Objektes (`form.engine.d3.createD3Api()`)	`d3api.setImportBytes()`	3.0.0	`d3api.setImportTableAsStream()`
Innerhalb des d.3 API Objektes (`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

Die d.ecs forms engine kann beim Speichern die Werte einer Datenquelle analysieren und in die drei Bereiche aufteilen:

- **Create**: Werte, die in d.ecs forms hinzugefügt wurden
- **Update:** Werte, die in d.ecs forms verändert wurden
- **Delete**: Werte, die in d.ecs forms gelöscht wurden

Dies geschieht aufgrund von Vergleichsoperationen zum letzten Lade- oder Speichervorgang. Bei Create wird alles zusammengefasst, was im Formular seit dem letzten Speichern hinzugefügt wurde. Bei Update sind nur die Änderungen vorhanden, usw.

Um diesen Mechanismus zu nutzen, müssen anstelle der saveData-Methode die folgenden drei Methoden im serverseitigen (Groovy) Skript implementiert werden:

- processCreate
- processUpdate
- processDelete

Diese Methoden verhalten sich ähnlich wie die saveData-Methode. Der Unterschied ist nur die Vorfilterung in die drei Bereiche create, update und delete. Diese Methoden werden nur aufgerufen, wenn jeweils ein Create, Update oder Delete seit dem Erzeugen der Formular-Instanz oder dem letzten Speichern des Formulars (`form.instance.save()`) stattgefunden hatte.

Voraussetzung für folgendes Beispiel-Skript ist eine Datenquelle (Objektmodell) mit den Eigenschaften name, vorname, adresse, plz, ort und nr (jeweils vom Typ Mehrfachwert), eine dazu passende Datenbanktabelle und einen dazu passenden DB-Alias (myDB).

CRUD-Datenquelle

```
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)

Serverseitig können Webservices angesprochen werden.

> Achtung
>
> Unter Umständen wird dabei eine Java-Library (JAR-Datei) benötigt. Diese muss dem System (ab d.ecs forms 3.0) als Bibliothek hinzugefügt und in dem entsprechenden Formular-Kontext oder Objektmodell referenziert werden. Mehr Informationen zum Anlegen und Nutzen von JAR-Dateien als Bibliotheken finden Sie im d.ecs forms Administrationshandbuch.

JAX-WS

Empfohlen ist die Verwendung von JAX-WS, da diese in Java standardmäßig integriert ist. Dabei muss außerhalb d.ecs forms eine Java-Library erstellt werden (siehe Dokumentation JAX-WS), die den Webservice-Aufruf implementiert und in einer einfachen Schnittstelle kapselt. Die Kompexität des Webservice ist somit verborgen. Ein weiterer Vorteil ist, dass diese Library einzeln verwendbar/testbar ist.

1.6. Anhang

1.6.1. Liste aller unterstützten d.3 API Funktionen für direkten Zugriff

Hier eine Auflistung der d.3-Funktionen, die im Kapitel Direkter Zugriff verwendet werden können

> Hinweis
>
> Hinweise zum Bezug der Dokumentation der API-Funktionen finden Sie im Kapitel Subobjekt d3.

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

1.7. Weitere Informationsquellen und Impressum

Wenn Sie Ihre Kenntnisse rund um die d.velop-Software vertiefen möchten, besuchen Sie die digitale Lernplattform der d.velop academy unter https://dvelopacademy.keelearning.de/.

Mithilfe der E-Learning-Module können Sie sich in Ihrem eigenen Tempo weiterführende Kenntnisse und Fachkompetenz aneignen. Zahlreiche E-Learning-Module stehen Ihnen ohne vorherige Anmeldung frei zugänglich zur Verfügung.

Besuchen Sie unsere Knowledge Base im d.velop service portal. In der Knowledge Base finden Sie die neusten Lösungen, Antworten auf häufig gestellte Fragen und How To-Themen für spezielle Aufgaben. Sie finden die Knowledge Base unter folgender Adresse: https://kb.d-velop.de/

Das zentrale Impressum finden Sie unter https://www.d-velop.de/impressum.