d.velop

DSSAGCurrent2023Q4 (d.3 hook & server scripting api (groovy))

Table of Contents

1. d.3 hook & server scripting api (groovy)

1.1. Introduction

1.1.1. About this documentation

In this documentation, you will learn how to develop d.3 hook functions and scripts with Groovy and which interfaces and functions d.3 server (version 8.1.0 and higher) provides for this purpose.

Passing this documentation or parts of it on to someone else is not permitted. For requests regarding the development partnership only the online documentation applies.

Please note that this interface also enables your software to access data stored and configured in d.velop documents belonging to your customers, and to influence internal system processes. Please proceed carefully and note that your application exists in interaction with other applications. The improper use of this interface can lead to changed application procedures and the loss of data.

Software development with this programming interface is considered custom programming. The program code created by you is not subject to the maintenance and support conditions of the products of d.velop AG. d.velop Support will be happy to help you with any questions. If your request cannot be traced back to an error in our products, this support will be subject to additional charges.

For any questions about the prerequisites and about software development, please contact the Technology Partner Management of d.velop AG.

1.1.2. Prerequisites

To be able to use all Java or Groovy functions, you have to enable Java or Groovy support in d.3 config.

> **Note**
> Java or Groovy support is enabled by default in d.3 server from version 8.0.0. d.3 server already has the Java Runtime Environment from Sun/Oracle enabled.
>
> For additional information, see the d.3 admin documentation.

Versions 8.0.0 and later of d.3 server already have the Java Runtime Environment from Sun/Oracle enabled.

1.1.3. Groovy

Groovy is a dynamic scripting language for Java Virtual Machine. Thanks to its close integration with Java, it enables you to access all the content of the Java environment, including many extensive libraries.

Groovy includes functions that are not available in Java. These include native syntax for maps, lists and regular expressions as well as a simple template system with which you can create HTML and SQL code. Furthermore, Groovy offers an XQuery-like syntax for running object trees, operator overloading and a native representation for BigDecimal and BigInteger.

Unlike other scripting languages, Groovy is not executed via an interpreted abstract syntax tree, but is translated directly into Java bytecode before a script is executed. Groovy is more similar to the syntax of Ruby and Python than the syntax of Java or BeanShell.

Further information on Groovy can be found on the official Groovy website at http://www.groovy-lang.org/.

1.2. Development environment

Groovy code is contained in text files that you can edit with a simple text editor.

For effective development of hooks, however, we recommend using a development environment such as Eclipse, which assists users with functions such as autocompletion of code.

The Java classes of the d.3 server interface and Groovy support as well as the Groovy interpreter are located in the Java archive **groovyhook.jar** in the d.3 server program directory (by default at C:\d3\d3server.prg). In Eclipse, you can integrate the Java archive at **Project Properties > Java Build Path as External JAR**.

In this case, install a Groovy plug-in for Ecplise to get the best possible support for syntax highlighting and command autocompletion.

> **Note**
> When developing and testing Groovy program code, enable the d.3 config parameter **Enable reloading of Groovy hook files on change** (**RELOAD_ON_CHANGE**).
>
> This means that when Groovy hook files are saved, the Groovy hook files are automatically reloaded by the server processes and the code changes are immediately active.

> **Warning**
> For security reasons, do not activate the reload option in the live system. Otherwise, changes to the Groovy scripts would become active immediately in your live system.

1.2.1. Using Eclipse as a development environment

If you want to use Eclipse as the environment for hook development, you must first configure Eclipse.

This is how it works

1. Download the Java Development Kit (JDK) and perform the installation.

   > **Warning**
   > Due to the change in the license conditions by Oracle, the Java distribution is no longer free of charge. This applies to JDK/JRE updates obtained from Oracle after January 2019.
   >
   > In the future, all affected products will be delivered with OpenJDK (https://openjdk.java.net/), while hotfixes use the Oracle Java 8 update 201.

2. Download Eclipse IDE for Java Developers at http://www.eclipse.org/ and unzip the file.
3. Open **eclipse.exe**.
4. Select a suitable directory for your workspace. The workspace is a directory in which Eclipse manages your projects. Do not specify a directory of the d.3 server, but rather a directory in your own files.

Once you have started Eclipse, install the Groovy plug-in for Eclipse.

This is how it works

1. Under **Help**, select the option **Install New Software...**.
2. Under https://github.com/groovy/groovy-eclipse/wiki , select the update site corresponding to your Eclipse version. An update site is a web address which Eclipse can use to automatically install software.

3. Enter your previously selected update site under **Work with** and confirm your entry.
4. Enable the feature **Groovy-Eclipse (Required)**.
5. Carry out the installation.

Your Eclipse installation is now ready for programming d.3 hooks with Groovy.

> **Note**
> The first time you start Eclipse in a newly created workspace, you may have to switch from the welcome page to the default view using **Workbench**.
>
> **Target systems without internet access**
>
> You can also configure the Eclipse development environment on a separate computer. Once you have successfully completed the configuration, you can copy the Eclipse folder to your target system without additional installation.

Creating hook projects

To program hooks, you must create a hook project.

This is how it works

1. Navigate to **File > New > Project**.
2. Select **Groovy Project** and assign a meaningful name on the next page.
3. Add a new source code directory by selecting the option **Link Source** under **Build Settings** after right-clicking on the project.
4. Under **Linked folder location**, enter the directory defined in d.3 admin for your Groovy hooks (see Groovy hook directories for customer-specific program adaptations).
5. Assign a descriptive name under **Folder name** (e.g. **Hooks**) and confirm the name with **Finish**.
6. Create the project with **Finish**.
7. Delete the folder **src** in the project you have just created.
8. Search for the file **groovyhook.jar** in the installation directory of your d.3 server (by default C:\d3\d3server.prg) and copy the file into your Eclipse project.
9. Add **groovyhook.jar** by right-clicking on **Build Path** and selecting **Add to Build Path**. If both the Eclipse development environment and the d.3 server installation are installed on the same computer, you can also integrate **groovyhook.jar** directly as an external JAR without copying.
10. Add **groovyhook-javadoc.jar** to the build path by selecting **groovyhook.jar** under **Libraries**, edit the option **Javadoc location** and enter the file under **Javadoc in archive** as **External file**.

> **Warning**
> If the version of the Groovy libraries automatically included in the Java build path for the Groovy project is newer than the library contained in the **groovyhook.jar**, Eclipse reports a version conflict. In this case, you must remove the entries **Groovy DSL Support** and **Groovy Libraries** from the Java build path in the **Libraries** tab.

11. Create new hook classes by selecting the type **Groovy Class** in the context menu of the **Hooks** folder under **New | Other**.
12. Give your hook a descriptive name and confirm the name with **Finish**.

You have successfully created a hook project for which you can now implement your hook functions.

> **Note**
> If you have configured in d.3 admin that hooks are automatically reloaded when changes are made, you only need to save in Eclipse to test the changes. If you have deactivated automatic reloading, you must restart your repository processes in order to apply the changes.

1.2.2. Using IntelliJ IDEA as a development environment

If you want to use IntelliJ IDEA for hook development as a development environment, you must first configure IntelliJ IDEA.

This is how it works

1. Download the Java Development Kit (JDK) and perform the installation.

> **Warning**
> Due to the change in the license conditions by Oracle, the Java distribution is no longer free of charge. This applies to JDK/JRE updates obtained from Oracle after January 2019.
>
> In the future, all affected products will be delivered with OpenJDK (https://openjdk.java.net/), while hotfixes use the Oracle Java 8 update 201.

2. Download the free community edition of IntelliJ IDEA (https://www.jetbrains.com/idea/download/#section=windows) and carry out the installation.
3. Download the Groovy library (http://groovy-lang.org/download.html) and carry out the installation.
4. Start IntelliJ IDEA.
5. Select a suitable directory for your project.
6. Click on the project folder and select the option **Sources Root** under **Mark Directory as**.

Making known the Groovy API of d.3

Next, make the Groovy API known in your project.

This is how it works

1. Navigate to **File > Project Structure**.
2. Under **Libraries**, add the file **groovyhook.jar** from the program directory of d.3 server.
3. Click on the plus icon and select **Java**.
4. Select **groovyhook.jar** from the program directory of d.3 server. You will receive a message that the library has been added to your project.
5. Confirm the message with **OK**.
6. Save your settings.

You have successfully completed the configuration of IntelliJ IDEA and made known the Groovy API of d.3. You can now develop Groovy hooks using the IntelliJ development environment.

More information about IntelliJ IDEA

The documentation and tutorials for IntelliJ IDEA can be found at https://www.jetbrains.com/idea/documentation/.

1.3. Groovy basics

This chapter provides you with an overview of basic information on using Groovy to help you get started.

Groovy documentation

http://www.groovy-lang.org/

http://www.groovy-lang.org/style-guide.html

http://grails.asia/groovy-list-tutorial-and-examples

Introduction to Groovy

http://www.oio.de/public/java/groovy/groovy-einfuehrung.htm

http://www.oio.de/public/java/groovy-closures-artikel.htm

http://www.javabeat.net/introduction-to-groovy-scripting-language/

Tutorials

https://www.timroes.de/2015/06/27/groovy-tutorial-for-java-developers/

http://mrhaki.blogspot.de/

http://mrhaki.blogspot.de/2009/10/groovy-goodness-finding-data-in.html

Groovy vs. Java

http://www.groovy-lang.org/differences.html

1.3.1. Variables and strings

In this chapter you will find information on how to use variables and strings.

> **Note**
> You can output content with the command **println**. In d.3, this command is output as an info message in the log file. Alternatively, you can also use the log function **d.log.info (..)** directly.

Defining variables

You can define variables with Groovy using dynamic typing by specifying the keyword **def**. In Groovy, you can also work with the common Java types. Make sure you only use the same variables for one type.

```
package com.dvelop.scripts;
import com.dvelop.d3.server.core.D3Interface;


D3Interface d3 = getProperty("d3");

def x = 42;
d3.log.info( "$x --> " + x.getClass() );

x = "Hello World";
d3.log.info( "$x --> " + x.getClass() );
// Output:
// 30.11 09:44:48,258 Master 10080CD8 D3B: 42 --> class java.lang.Integer
// 30.11 09:44:48,258 Master 10080CD8 D3B: Hello World --> class
java.lang.String
```

What are GStrings?

With Groovy, you can resolve variables within a string by integrating variables with a preceding $ character into the string. This concept is referred to as a GString.

```
package com.dvelop.scripts;
import com.dvelop.d3.server.core.D3Interface;

D3Interface d3 = getProperty("d3");

def x = "World";
d3.log.info( "Hello, $x" );

// Output:
// 30.11 09:44:48,258 Master   10080CD8 D3B: Hello, World
```

You can also access individual substrings within this notation by using curly brackets. You can also access individual letters within a string in Groovy. In this case, the notation is similar to that for accessing array elements.

```
package com.dvelop.scripts;
import com.dvelop.d3.server.core.D3Interface;

D3Interface d3 = getProperty("d3");

def firstName = "Douglas";
def name      = "Adams";
d3.log.info( "Hello, ${firstName[0]}. $name" );

// Output:
// 30.11 09:44:48,285 Master   10080CD8 D3B: Hello, D. Adams
```

Multi-line strings

You can define strings in Groovy over several lines. To do this, you need three quotation marks at the beginning and end of the corresponding string.

```
package com.dvelop.scripts;
import com.dvelop.d3.server.core.D3Interface;

D3Interface d3 = getProperty("d3");

def s = """This is
a multiline
string""";

d3.log.info( s );
// Output:
// 30.11 09:50:34,925 Master   10080CD8 D3B: This is
// 30.11 09:50:34,925 Master   10080CD8 D3B: a multiline
// 30.11 09:50:34,925 Master   10080CD8 D3B: string
```

1.3.2. Conditions

The conditions in Groovy are largely identical to the conditions in Java. Below you will find information on the special features of the conditions in Groovy.

Using an operator for safe navigation

If you want to check a value within an object structure, you must ensure that the individual elements of the structure are not equal to the value **null**. You have the following options:

Option 1

```
if( company.getContact() != null && company.getContact().getAddress() !=
null && company.getContact.getAddress().getCountry() == Country.NEW_ZEALAND
) { ... }
```

Option 2

```
if( company.getContact()?.getAddress()?.getCountry() == Country.NEW_ZEALAND
) { ... }
```

If the object or one of the object components does not exist, the value **null** and no error message is returned.

Elvis operator

Groovy has a compressed notation for "if" statements. A question mark is always used for true branches and a colon for false branches.

```
package com.dvelop.scripts;
import com.dvelop.d3.server.core.D3Interface;

D3Interface d3 = getProperty("d3");

def testValue = null;
def name = testValue != null  ? testValue : "default";
d3.log.info( "Normal -->" + Name );
// Output
// 10.12 10:58:23,238 Master   15041A9C D3B: Normal -->default
```

If you only want to assign a different value if the checked variable is not contained, you can also work with an abbreviated notation.

```
package com.dvelop.scripts;
import com.dvelop.d3.server.core.D3Interface;

D3Interface d3 = getProperty("d3");

def testValue = null;
def name = testValue ?: "default";
d3.log.info( "Elvis -->" + Name );
// Output
// 10.12 10:58:23,239 Master   15041A9C D3B: Elvis -->default
```

Switch statement

Unlike Java, Groovy is not limited to numerical values for switch statements.

```
package com.dvelop.scripts;
import com.dvelop.d3.server.core.D3Interface;

D3Interface d3 = getProperty("d3");

def testValue = "ABC";
switch( testValue ){
   case 100: // Integer
      d3.log.info( "The number 100" );
      break;
   case "ABC": // String
      d3.log.info( "The string ABC" );
```

```
      break;
   case Long: // Class
      d3.log.info( "A Long value" );
      break ;
   case ['alpha','beta','gamma']: // List
      d3.log .info( "alpha, beta or gamma" );
      break;
   case {it > -0.1 && it < 0.1}: // Closure
      d3.log.info( "A number near zero" );
      break;
   case null: // null
      d3.log.info( "An empty value " );
      break;
   case ~/Groov.*/ : // Regulare Expression
      d3.log.info( "Begins with Groov" );
      break;
   default:
      d3.log.info( "Something completely different" );
}
```

1.3.3. Loops

The handling of loops in Groovy is identical to Java. Unlike Java, however, Groovy has what are known as "closures," which are described in the following chapter. As curly brackets are intended for closures, the initial filling of lists and arrays is done with square brackets.

For-loop

The following example shows you how "for" loops are structured:

```
package com.dvelop.scripts;
import com.dvelop.d3.server.core.D3Interface;

D3Interface d3 = getProperty("d3");

def testList = [ "This", "is", "example", "content" ];
int n        = 0;
// A classic for-statement
for( n = 0; n < testList.size(); n++ ){
   d3.log.info( "$n. For --> " + testList[n] );
}
// Output
// 10.12 12:33:11,996 Master   15041A9C D3B: 0. For --> This
// 10.12 12:33:11,997 Master   15041A9C D3B: 1. For --> is
// 10.12 12:33:11,997 Master   15041A9C D3B: 2. For --> example
// 10.12 12:33:11,997 Master   15041A9C D3B: 3. For --> content


// A for-statement with collection
n = 0;
for( def item in testList ){
   d3.log.info( (n++) +  ". For-(Collection) --> " + item );
}

// Output
// 10.12 12:33:11,998 Master   15041A9C D3B: 0. For-(Collection) --> This
// 10.12 12:33:11,999 Master   15041A9C D3B: 1. For-(Collection) --> is
```

```
// 10.12 12:33:11,999 Master   15041A9C D3B: 2. For-(Collection) -->
example
// 10.12 12:33:11,999 Master   15041A9C D3B: 3. For-(Collection) -->
content
```

Each statements

The following example shows you how "each" statements are structured:

```
package com.dvelop.scripts;
import com.dvelop.d3.server.core.D3Interface;

D3Interface d3 = getProperty("d3");

// Each-Statement
n = 0;
testList.each{
   d3.log.info( (n++) +  ". Each --> " + it );
}
// Output
// 10.12 12:33:12,008 Master   15041A9C D3B: 0. Each --> This
// 10.12 12:33:12,008 Master   15041A9C D3B: 1. Each --> is
// 10.12 12:33:12,008 Master   15041A9C D3B: 2. Each --> example
// 10.12 12:33:12,009 Master   15041A9C D3B: 3. Each --> content
// Each-Statement with index
testList.eachWithIndex { val, idx ->
   d3.log.info( idx +  ". Each with index --> " + val );
}
// Output
// 10.12 12:33:12,010 Master   15041A9C D3B: 0. Each with index --> This
// 10.12 12:33:12,010 Master   15041A9C D3B: 1. Each with index --> is
// 10.12 12:33:12,010 Master   15041A9C D3B: 2. Each with index --> example
// 10.12 12:33:12,010 Master   15041A9C D3B: 3. Each with index -->
content
```

Collection statements

The following example shows you how "collection" statements are structured:

```
package com.dvelop.scripts;
import com.dvelop.d3.server.core.D3Interface;

D3Interface d3 = getProperty("d3");

n = 0;
def newList = testList.collect { it; }
newList.each{
   d3.log.info( (n++) + ". Collect --> " + it );
}
// Output
// 10.12 12:33:12,012 Master   15041A9C D3B: 0. Collect --> This
// 10.12 12:33:12,012 Master   15041A9C D3B: 1. Collect --> is
// 10.12 12:33:12,012 Master   15041A9C D3B: 2. Collect --> example
// 10.12 12:33:12,012 Master   15041A9C D3B: 3. Collect --> Content
```

1.3.4. Closures

Closures are smaller and unnamed functions that are directly linked to a variable.

Variables with a function

The following example shows you how variables with a function are structured:

```
package com.dvelop.scripts;
import com.dvelop.d3.server.core.D3Interface;

D3Interface d3 = getProperty("d3");

// Print Hello world ------------------------------------------
def optionOne = {
   d3.log.info( "Option 1: Hello World" );
}
optionOne();

// Output
// 09.12 13:30:35,082 Master   12041F98 D3B: Option 1: Hello World
```

Closures with variables with fixed types

The following example shows you how closures are structured that contain variables with a fixed type:

```
// Closures with parameters -------------------------------------
def power = { int x, int y ->
   return Math.pow( x, y ); }
d3.log.info( "Option 2: " + power( 2, 3 ));

// Output
// 09.12 13:30:35,093 Master   12041F98 D3B: Option 2: 8.0
```

Closures with untyped variables

The following example shows you how closures with an untyped variable are structured:

```
package com.dvelop.scripts;
import com.dvelop.d3.server.core.D3Interface;

D3Interface d3 = getProperty("d3");

// Closure with one dynamic typing variable --------------------
def optionThree = { what ->
   d3.log.info( what ); }
optionThree "Option 3: Hello World"; // same as optionThree( "Option 3:
Hello World" );

// Output
// 09.12 13:30:35,094 Master   12041F98 D3B: Option 3: Hello World
```

Closures with implicit "it" variables

The following example shows you how closures with implicit "it" variables are structured:

```
// Closure with an implizit argument --------------------------
def optionFour = { d3.log.info( it ); }
optionFour "Option 4: Hello World"; // same as optionFour("Option 4: Hello
World");

// Output
// 09.12 13:30:35,095 Master   12041F98 D3B: Option 4: Hello World
```

Closures without variables

The following example shows you how closures that explicitly do not contain any variables are structured:

```
package com.dvelop.scripts;
import com.dvelop.d3.server.core.D3Interface;

D3Interface d3 = getProperty("d3");

// Closure without any argument -------------------------------
def optionFive = { ->
   d3.log.info( "Option 5: This closure does not take any arguments." ); }
optionFive();
// Output
// 09.12 13:30:35,095 Master   12041F98 D3B: Option 5: This closure does
not take any arguments.
```

If you want a value to be returned, you can also do this without a return. This causes the last value to be returned:

```
package com.dvelop.scripts;
import com.dvelop.d3.server.core.D3Interface;

D3Interface d3 = getProperty("d3");

// Optional return value --------------------------------------
def square = { it * it };
d3.log.info( "Option 6: " + square(4));

// Output
// 09.12 13:30:35,127 Master   12041F98 D3B: Option 6: 16
```

1.3.5. Database connection

Accessing a d.3-internal database (d.3 SQL database)

You can use the d.3 SQL interface to access database tables within the d.3 database.

```
//(1)
package com.dvelop.scripts;
import com.dvelop.d3.server.core.D3Interface;

D3Interface d3 = getProperty("d3");

//(2)
def resultRows = d3.sql.executeAndGet( "SELECT name FROM CustoemrData" );
//(3)
resultRows.each{ println it.name; }
```

Notes on the individual sections

- To use the d.3 SQL interface, you need the library **D3**, which you can import if required. As the variable **d3** is implicitly available via the d3-server-interface call, you can make the variable available in the script using the function **getProperty("d3")**. This means you can also access the variable during programming for autocompletion of the code.
- You can send an SQL statement to the database table using various implementations or functions. The results are transferred to a map structure.

- You can further process the data from the results using an "each" statement.

Further information on the database connection

For queries where you often only receive one hit or explicitly only want to receive one hit, you can use the **firstRow()** method.

```
def sqlQuery  = "SELECT max(product_count) as value FROM productDB WHERE
product_id = ? ";
def sqlParams = [ 4711 ];
def firstRow  = d3.sql.firstRow (sqlQuery, sqlParams);
int max_no    = firstRow[0] + 1;
```

Alternatively, you can use **AS highestNo** in the SQL command to achieve improved readability and therefore better code.

```
def sqlQuery  = "SELECT max(product_count) as highestNo FROM productDB
WHERE product_id = ? ";
def sqlParams = [ 4711 ];
def firstRow  = d3.sql.firstRow (sqlQuery, sqlParams)
int max_no    = firstRow.highestNo + 1;
```

Accessing an external database

Below you will find more detailed information on how to access an external database.

> **Note**
> To access external databases, you must download and deploy the necessary JDBC drivers from the respective database manufacturers. You can then use the standard SQL interface to establish a connection to the database and issue an SQL statement.

```
//(1)
import groovy.sql.Sql;
//(2)
def dbConnection = Sql.newInstance( "jdbc:sqlserver://
localhost:1433;databaseName=Name", "User", "Password" );
//(3)
def resultRows = dbConnection.rows( "SELECT name FROM CustomerData " );
//(4)
resultRows.each{ println it.name; }
```

Notes on the individual sections

- To be able to use the d.3 SQL interface for external database access, you need the standard Groovy library for SQL, which you must import if required.
- To access an external database, you must establish a connection to the database. You can also use a standard Groovy function to set up a **newInstance** connection. Further information can be found in the documentation of the respective database manufacturer and the Groovy documentation.
- You can send an SQL statement to the database table using various implementations or functions. The results are transferred to a map structure.
- You can further process the data from the results using an "each" statement.

1.3.6. d.3 special cases

Reading out d.3 configuration parameters

If you use a Groovy script in both the test and live environments, you can configure the script by querying configuration and server parameters so that the script works in both environments without

adjustments. Porting or adaptation is therefore no longer necessary. The possible parameters can be found in the relevant d.3 documentation.

The following example shows you how to query the d.3 configuration parameters:

```
d3.log.error( d3.config.value( "d3fc_server_id" ) );   // Server id
d3.log.error( d3.config.value( "d3fc_server_name" ) ); // Server name
d3.log.error( d3.config.value( "db_server" ) );        // Database type
d3.log.error( d3.config.value( "CUR_60ER_FIELD_NR" ) );// Current max .
sizce of 60-ies feild

d3.log.error( d3.natives.getd3fcLanguage() );        // Get the current
language

// Output
// 08.12 15:25:39,090 Master   130C1308 D3B: B
// 08.12 15:25:39,090 Master   130C1308 D3B: localhost
// 08.12 15:25:39,090 Master   130C1308 D3B: MSQL
// 08.12 15:25:39,091 Master   130C1308 D3B: 100
```

As of version 8.1, you can also determine parameters such as the current API language or the app version in Groovy.

```
import com.dvelop.d3.server.Document
import com.dvelop.d3.server.DocumentType
import com.dvelop.d3.server.Entrypoint
import com.dvelop.d3.server.User
import com.dvelop.d3.server.core.D3Interface
public class d3RemoteInterfaceExample{

  @Entrypoint( entrypoint = "hook_insert_entry_10" )
//-----------------------
  public int getCustomerDataForInvoice( D3Interface d3, User user,
DocumentType docTypeShort, Document doc ){
    d3.log.error( "App-Version: "  + d3.remote.getVersion());
    d3.log.error( "APP-ID: "       + d3.remote.getVersion()[0..2]);
    d3.log.error( "App-Language: " + d3.remote.getLanguage());
    return 0;
  } // end of getCustomerDataForInvoice

}// end of d3RemoteInterfaceExample
```

d.3 system properties

The d.3 interface defines the following system properties as Java system properties:

Property name	Description	Example values
d3.server.home	The current d.3 server program directory	"D:\d3\d3server.prg"
d3.server.version	The d.3 server version number	"08.01.00.05"
d3.server.systemUser	The system user name of the current d.3 server process	"D3Server", "D3Async", "hostimp", "Master"
d3.repository.uuid	The archive UUID of the d.3 repository	"8be6de08-d009-4c2b-a33d-38a3a6458617"

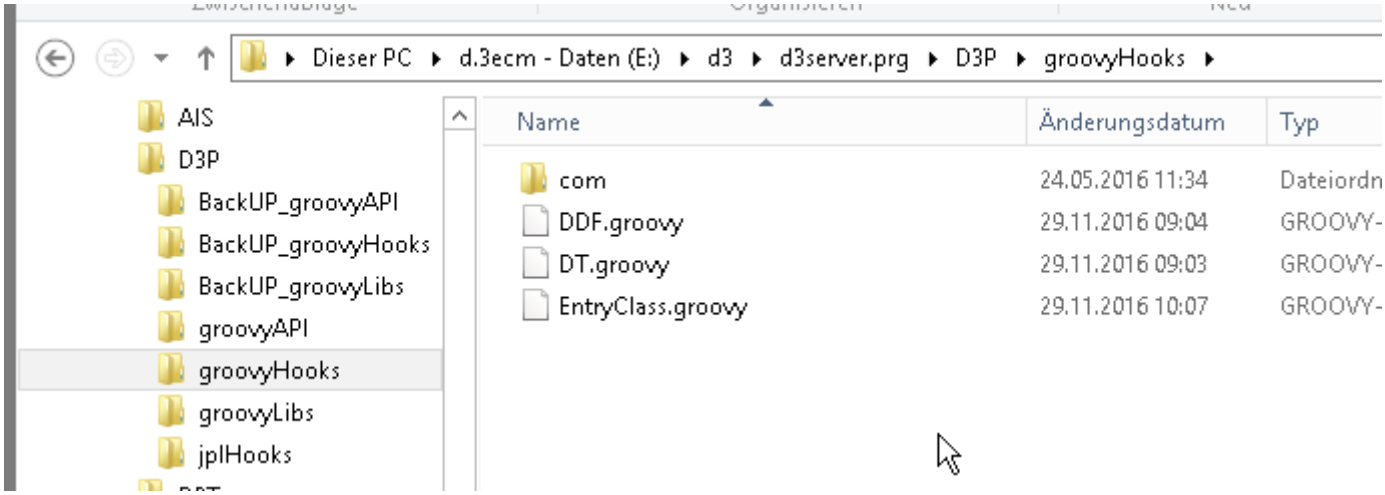
You can query the system properties with the following call: **System.getProperty("property name")**

```
import javax.swing.JOptionPane
import javax.swing.UIManager
```

```
def scriptRequireD3Version = "08.01.00.19"

UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());

if (System.getProperty("d3.server.version") < scriptRequireD3Version)
{
    showMessageDialog("This script requires d.3 server version <i>$
{scriptRequireD3Version}</i> or later!", "Check d.3 server version")
    return
}

def showMessageDialog(String messageHtml, String title)
{
    JOptionPane.showMessageDialog(null, "<html>" + messageHtml + "</html>",
title, JOptionPane.ERROR_MESSAGE);
}
```

Class for global constants

In JPL, global constants are used to reference the database positions of the advanced properties and are controlled centrally. With Groovy, this is not realized via a separate package structure, but only in the root directory with all the necessary files.



As an example, create two classes that are referenced within the hook functions.

Class for doc-dat field positions

```
class DDF {
   static final int FIRSTNAME = 4; // DB positions
   static final int LASTNAME  = 5;
   static final int STATE_ID  = 6;
}// end of DDF
```

Class for document and dossier types

```
class DT {
   static final String INVOCIE      = "DRECH"; // Doc types
   static final String ORDER        = "DBEST";
   static final String EMPLOYEE_FOLDER = "APERS"; // Folder types
}// end of DT
```

Using the global constants within a hook function

As the classes are located in the same directory, you can now use the classes in the hook files or hook functions.

```
import com.dvelop.d3.server.Document
import com.dvelop.d3.server.DocumentType
import com.dvelop.d3.server.Entrypoint
import com.dvelop.d3.server.User
import com.dvelop.d3.server.core.D3Interface

class D3Hooks {

   @Entrypoint( entrypoint = "hook_insert_entry_10" )
   public int myEntryPoint( D3Interface d3, User d3User, DocumentType
docTypeShort, Document doc ){
      d3.log.error( "+++ MyGlobal-Test+++++ " + DDF.FIRSTNAME);
      d3.log.error( "+++ MyGlobal-Test+++++ ${DDF.LASTNAME}" );
      d3.log.error( "+++ MyGlobal-Test+++++ $DDF.STATE_ID" );

      d3.log.error( "+++ MyGlobal-Test+++++ " + DT.INVOICE);
      d3.log.error( "+++ MyGlobal-Test+++++ " + DT.ORDER );
      d3.log.error( "+++ MyGlobal-Test+++++ " + DT.EMPLOYEE_FOLDER);
      return 0;
   }// end of myEntryPoint
}// end of D3Hooks
```

1.4. Debugging

In d.3 config under **Java/Groovy**, activate the setting **Java Remote Debugging** (**JAVA_REMOTE_DEBUGGING**) to fix errors in Groovy code. If you activate the setting, the d.3 server processes start the Java Virtual Machine in debug mode. You can then connect to a d.3 server process via Remote Java Debugger to fix the errors in the Groovy hooks.

The port **43400** is used for communication. As each d.3 process starts its own Java Virtual Machine (JVM), the ports used are incremented, i.e. the first process started using **JAVA_REMOTE_DEBUGGING** opens port **43400**, the second process opens port **43401** and so on. The ports appear when the JVM is started with the message **Java Remote Debugging Port** in the d.3 log.

> **Note**
> The JVM is started by d.3 the first time Groovy code is accessed. The JVM is therefore not usually available immediately after the process is started.

Further information on the subject of debugging can be found in the usual technical literature and on the Internet.

1.4.1. Remote debugging with Eclipse

Remote debugging is a method for finding and correcting errors in hook functions. With remote debugging, each d.3 server process opens a port to which you connect via Eclipse. You can then check the function sequence.

The d.3 server processes start the Java Virtual Machine (JVM) in debug mode.

> **Note**
> The JVM is started by d.3 the first time Groovy code is accessed. The JVM is therefore not usually available immediately after the process is started.

The first d.3 server process started uses the port **43400**, the second process uses the port **43401** and so on.

Please note that the d.3 processes d3_server, d3_async and hostimport also support remote debugging. An archive is usually supported by multiple server processes. Therefore, always ensure that you are connected to the correct server process, e.g. by only running one d.3 server process at a time.

> **Warning**
> While you are performing remote debugging, the d.3 server process cannot process any other tasks. Accordingly, do not perform remote debugging in live operation.

Performing remote debugging – This is how it works

1. Navigate to **d.3 admin > d.3 config > Java/Groovy** and activate **Java Remote Debugging**.
2. In your Eclipse project, navigate to **Run > Debug Configurations**.
3. Create a new entry under **Remote Java Application**.
4. Enter the connection data of the system under **Connection Properties**.
5. Click **Debug**.
6. Set breakpoints in the source code at which to pause the execution. This allows you to check and continue the process manually.

1.4.2. Remote debugging with IntelliJ IDEA

You can troubleshoot Groovy hook functions in a d.3 server process using IntelliJ IDEA for remote debugging. You must first add the debug configuration. You can then perform remote debugging.

> **Warning**
> While you are performing remote debugging, the d.3 server process cannot process any other tasks. Accordingly, do not perform remote debugging in live operation.

This is how it works

1. In IntelliJ IDEA, select **Edit Configurations** from the selection menu for run configurations.
2. Click on the plus icon in the new window and select **Remote**.
3. Enter the following connection data to the d.3 application server on which the hooks and the d.3 server processes are executed:
   - **Host**: Host address or FQDN
   - **Port**: Debug port of the d.3 server process
4. Select the project module from IntelliJ that contains your hook project.
5. Add breakpoints to all desired lines of code by clicking next to the line numbers of the lines of code. Execution of the code is interrupted at the breakpoints.
6. Stop all d.3 server processes and restart a d.3 server process.
7. Start the server interface.
8. Check the log to see which port the server process is using. Proceed as follows, for example:
   - Search for "Remote debugging" in the log. A log line may look like this, for example: **04.04 14:46:47,695 D3SRV_P 68B85950 : Java remote debugging port: 43403**. In this example, the determined port is therefore **43403**.
9. Enter the determined port in the configuration.
10. Click the bug icon or press **Shift + F9** to start remote debugging. When the hooks or scripts are called, the breakpoints pause the process. This allows you to check and continue the process manually.

Further information on debugging with IntelliJ IDEA can be obtained from the manufacturer JetBrains: https://www.jetbrains.com/idea/documentation/.

1.5. Groovy hook types

Groovy hooks are defined interfaces (entry points) in our systems where you can create individual functions. Use a Groovy script to create the functions.

There are different types of Groovy hooks:

- Entry points are event-driven script interfaces
- Validation hooks for extended validation of values
- Provision of dynamic datasets
- Extended document classes

1.5.1. Groovy interface in d.3 admin

In d.3 admin under **System settings > d.3 config**, make configurations in the areas **Hook functions** and **Java/Groovy** in order to be able to use the Groovy interface. Please note the following information on the configurations.

Hook functions

- **Activating Groovy hooks**
  Enter the directories containing the Groovy program adaptations.
- **Reloading Groovy hooks**
  If you activate the switch, Groovy hook files are reloaded as soon as you save changes to the Groovy hook files. This means you do not have to restart a d.3 process, and the changes to the code are active immediately. Only use this function during development and not in a live system.
- **Activating JPL and Groovy hooks for the same entry point**
  If you activate this switch, JPL hook functions and Groovy hook methods can be registered for the same hook entry point. This option is intended to make it easier to replace existing JPL code with modern Groovy hooks, as it allows the logic to be updated successively and as needed.
  If the **Hook function** column under **Execute hook functions** contains the name of a JPL hook function and there is also an annotation for the entry point in a Groovy hook module, the function and the method for this entry point are registered. During execution, the JPL hook function is called first and the annotated Groovy hook method(s) immediately afterwards.

Java/Groovy

- **Java/Groovy Support**
  Activates support for the execution of Java and Groovy code in d.3.
- **Java CLASSPATH**
  File path(s) to your Java classes, separated with semicolons. Specify one or more directories in which the CLASS files are stored. Alternatively, enter the file name of a JAR file or use a CLASSPATH file.
- **Java/Groovy API functions**
  Activates the plugin interface for API functions that were developed in Java or Groovy. Groovy scripts or JAR files that implement d.3 API functions are loaded from the directory. We recommend that you do not activate this setting.
- **Java Remote Debugging**
  Starts the Java Virtual Machine (JVM) in debug mode. You can connect to a d.3 server process via Remote Java Debugger to fix errors in the Groovy hooks executed in it. Port **43400** is used for communication. As each d.3 process starts its own Java Virtual Machine (JVM), the ports used are incremented. The first process started with **JAVA_REMOTE_DEBUGGING** enabled opens port **43400**, the second opens port **43401** and so on. The determined port appears in the message **Java Remote Debugging Port** in the d.3 log when the JVM is started.

> **Note**
> The JVM is started by d.3 the first time Groovy code is accessed. The JVM is therefore not usually available immediately after the process is started.

1.5.2. Programming hook functions

Required Java libraries

The Groovy context requires libraries that are available by integrating the file **groovyhook.jar**. You can reference the libraries in the Groovy files.

Global libraries

To use the **D3Interface** object, you need the **com.dvelop.d3.server.core.D3Interface** library.

> **Note**
> If you have problems with the JavaDoc documentation and Groovy templates when implementing the **D3Interface** object, you can also use the **import com.dvelop.d3.server.core.D3** base library for programming. However, switch back to the **com.dvelop.d3.server.core.D3Interface** library for live use.

Special libraries for the individual hook types

Annotation	Use	Syntax	Required Java library
@Entrypoint	d.3 entry points	**@Entrypoint(entrypoint="name_in_admin", order* = n)**	**import com.dvelop.d3.server.Entrypoint;**
@ValueSet	Dataset hooks	**@ValueSet(entrypoint="name_in_admin", order* = n)**	**import com.dvelop.d3.server.ValueSet;** **import com.dvelop.d3.server.RepositoryField;**
@Validation	Validation hooks	**@Validation(entrypoint="name_in_admin", order* = n)**	**import com.dvelop.d3.server.Validation;**
@DocumentClass	Document class hooks	**@DocumentClass(entrypoint="name_in_admin", order* = n)**	**import com.dvelop.d3.server.DocumentClass;**
@Condition	Filter for specific document classes	**@Condition(doctype = ["DRECH", "DBEST", "DLIEF"])**	**import com.dvelop.d3.server.Condition;**

> **Note**
> *You can define the optional parameter **order** to define an order of processing if several Groovy functions are defined on a single entry point.

Style guide recommendation

To deploy the function, you must create at least one class of type **public**; **public** can also be omitted in Groovy contexts. You can deploy a separate class or a separate file for each type of hook function. Below are some example class names:

Class name	Description
D3Hooks	For handling entry points
D3DataSets	For implementing datasets
D3Validate	For deploying validation functions at property level
D3DocClasses	For realizing specific document classes
D3FolderScheme	You can deploy this class if you need extended dossier schemes.

Registering a Groovy method as a hook function

The following annotations are available for registering the various hook types:

```
import com.dvelop.d3.server.Entrypoint;

public class MyHooks{
```

```
   @Entrypoint(entrypoint="hook_insert_entry_10", order = 1 )
   @Condition( doctype= ["DRECH", "DBEST", "DLIEF" ] )
   int checkIncommingDocs(D3Interface d3, User user, DocumentType docType,
Document doc){
      println "The function checkIncommingDocs was called inside the hook
function entry Point hook_insert_entry_10";
      return 0;
   } // end of checkIncommingDocs
}// end of MyHooks
```

Registration consists of entering the annotation in the source code directly in front of the Java/Groovy method that is to be called for the hook entry point specified by **entrypoint**.

> **Note**
> A Groovy class that is to be loaded and registered by d.3 must provide a public constructor without parameters (**public no-argument constructor**). The requirement is also fulfilled if the constructor is missing because in this case, the Java default constructor exists implicitly.

Return value of hook methods

A number (integer) is expected as the return value. The server evaluates the returned value as an error code as follows:

- Value = 0: Success
- Value <> 0: Error in the hook function. Depending on the hook function, this may result in the action during which the hook function was executed being aborted.

> **Note**
> In Groovy, the keyword **return** for exiting a method with a return value is optional. You can therefore omit the keyword.
>
> If you do not specify the keyword, Groovy uses the last variable value before the closing bracket and implicitly returns this value as the return value. In this case, errors or unwanted return values may occur. We therefore recommend explicitly ending a hook method with **return**. If the return value is not relevant, you can use **return 0**.
>
> Groovy hook functions for the d.3 entry points are configured automatically. Please note the following information:
>
> - You do not have to enter the identifiers for d.3 entry points individually under **d.3 admin > d.3 config >Hook functions > Execute hook functions**.
> - If an annotation for an entry point is found when loading a Groovy class, the annotated method is registered.
> - In the Config module, the registration label **<groovy hook>** is entered after the entry point.
> - It is possible to register several methods per entry point.

> **Warning**
> Since updating d.3 admin can take some time, you can also use the output in the log file. The successful loading of the Groovy functions for the entry points is also documented in the log file.

Using Java libraries

If you need Java libraries from third-party providers or Java libraries you have created yourself when implementing a hook function (e.g. to address your CRM system), you can specify these libraries specifically for each hook.

In addition to your existing file **<hook name>.groovy**, create another file **<hook name>.classpath**. In the CLASSPATH file, you define entries for the Java classpath line by line. You can use absolute paths and paths relative to the current directory that lead to JAR files. In this way, the Java libraries are isolated from each other so that you can use different versions of libraries in several hooks.

> **Note**
> You cannot integrate JDBC drivers on a hook-specific basis. The drivers must be loaded globally.

Isolation of hook classes

Each hook class is loaded by its own Groovy classloader instance. This means that a hook object cannot access the properties of other hook objects. However, you can make any Groovy class visible in another Groovy class using the **import** command. You can instantiate the Groovy class or, in the case of static elements, call it directly.

If there are several thematically related hook classes, you can outsource shared code to a separate class and therefore a separate module. Example: If a hook is used during import to validate whether entered customer data has been saved in the CRM system and a dataset of possible customers is to be offered at the same time, there is typically shared code. Implement the code in a separate Groovy class. The Groovy class can in turn be used by the other Groovy hook classes.

Package structure

Just like Java classes, Groovy classes are also organized into packages. State the name of the package at the beginning of the source file before the import instructions and the first class definition. You can use the standard form of structuring based on domain names in reverse order. If you do not specify a package, the default package is used.

> **Warning**
> In some versions you cannot use packages.

Recommendations for packages

- **com.dvelop.scripts**
  Groovy scripts that are called in the context of a server interface or by d.3 process manager and are therefore saved in the **ext_groovy** directory. The resulting directory is e.g. **ext_groovy\com\dvelop\scripts**.
- **com.dvelop.hooks**
  You should define the Groovy hook functions that serve the individual hook entry points in a separate package. The resulting directory is e.g. **d3server.prg\D3T\groovyHooks\com\dvelop\hooks**.
- **com.dvelop.api**
  If you provide your own API functions using Groovy functions, you can use this package. The resulting directory is e.g. **d3server.prg\D3T\groovyAPI\com\dvelop\api**.

1.5.3. d.3 dynamic feedback from the hook functions

If you also want dynamic texts from a server action to be transferred to the client side (e.g. for dynamic error messages), you can use the hook property **additional_info_text** from every hook.

The getProperty and setProperty functions are not available for the following hook integration types:

- validation hook
- document class hook
- value set hook

The functions are only available for the “classic” d.3 entry points.

Call

```
d3.hook.setProperty("additional_info_text", "My message for the Client
side!")

//Message depending on the language of the client request
if(d3.remote.getLanguage() == "049"){
  d3.hook.setProperty("additional_info_text", "Meine Nachricht für die
Client-Seite.")
} else {
  d3.hook.setProperty("additional_info_text", "My message for the Client
side!")
}
```

If the hook is called by a d.3 server process (not **hostimp** or **async**), the text is transferred to the client as an additional export parameter in the current API call.

> **Note**
> The text is not displayed across all clients.

1.5.4. Number range for return values

The return values from the hook functions are calculated with an offset value. The value that can be evaluated on the client side is the result of the calculation "offset value minus range value".

Entry point	Range	Offset	Result (offset value – range value)	Example (offset – custom value)
ImportDocument ValidateAttributes	-8000 -> -9999	10000	18000 ->19999	10000 - (-8000) = 18000
ImportNewVersionDocument	-8000 -> -9999	20000	28000 ->29999	20000 - (-8500) = 28500
DeleteDocument	-1900 -> -1999	4000	5900 -> 5999	4000 - (-1925) = 5925
[All others]	-1 -> -499	9500	9501 -> 9999	9500 - (-250) = 9750

Example

```
@Entrypoint(entrypoint = "hook_insert_entry_10")
int myCustomReturnValueImport(D3Interface d3, User user, DocumentType
docType, Document doc) {
        if(docType.id == "DDOKU" && doc.field[10] == null){
        //Client returns error message with id "18000" in msglib.usr
                return -8000
        }
}

@Entrypoint( entrypoint = "hook_new_version_entry_10" )
int myCustomReturnValueNewVersion( D3Interface d3, Document doc, String
fileSource, String fileDestination, User user, DocumentType docType ){
        if(docType.id == "DDOKU" && doc.field[10] == null){
        //Client returns error message with id "28000" in msglib.usr
                return -8000
        }
}

@Entrypoint( entrypoint = "hook_delete_entry_10" )
int myCustomReturnValueDelete( D3Interface d3, Document doc, User user,
DocumentType docType ){
        if(docType.id == "DDOKU" && doc.field[10] == null){
        //Client returns error message with id "5925" in msglib.usr
                return -1925
        }
}

@Entrypoint(entrypoint = "hook_upd_attrib_entry_20")
int myCustomReturnValueUpdAtt(D3Interface d3, Document doc, User user,
DocumentType docType, DocumentType docTypeNew) {
        if(docType.id == "DDOKU" && doc.field[10] == null){
        //Client returns error message with id "9750" in msglib.usr
                return -250
        }
}
```

msglib.usr

```
'Erste Spalte  = Fehlernummer
'Zweite Spalte = Sprache (049=Deutsch, 001=Englisch)
'Dritte Spalte = Typ
'   d.3-Server Fehlermeldungen
'   0 = Warnung
'   1 = Fehler
'   2 = Information
'   3 = Bestätigung
'Vierte Spalte = Nachricht


18000,049,2,"Bitte Angabe für Feld 10 machen. (Import, RetCode -8000)"
18000,001,2,"Please enter information for field 10. (Import, RetCode
-8000))"

28000,049,2,"Bitte Angabe für Feld 10 machen. (NewVersion, RetCode -8000))"
```

```
28000,001,2,"Please enter information for field 10. (NewVersion, RetCode
-8000))"

5925,049,2,"Bitte Angabe für Feld 10 machen. (Delete, RetCode -1925))"
5925,001,2,"Please enter information for field 10. (Delete, RetCode -1925))"

9750,049,2,"Bitte Angabe für Feld 10 machen. (UpdAtt, RetCode -250)"
9750,001,2,"Please enter information for field 10. (UpdAtt, RetCode -250)"
```

The file **msglib.usr** is created or edited in the client directory in which the file **msglib.dll** is located. Example for d.velop documents: **D:\d3\d.3one\dms\bin\msglib.usr**

1.5.5. Using the transport system for Groovy functions

You can use the transport system to transfer settings between d.3 repositories. You can also transfer Groovy hook modules.

In the edit mode of d.3 admin, you can define projects that contain the settings to be transported. To assign a Groovy hook module to a transport project, enter the project name in the Groovy hook module using the annotation **@TransportProject**.

The annotation **@TransportProject** is defined at class level, which is why you need to introduce the annotation to a Java/Groovy class definition. You can enter one or more project names or the GUIDs of the projects as annotation parameters.

```
import com.dvelop.d3.server.TransportProject;

// OPTION 1: Projectname
@TransportProject("myProject")
public class MyTestHooks {

} // end of MyTestHooks

// OPTION 2: Project-GUID
@TransportProject("6ACDA408-3638-4B70-8E0D-036CC9559F7E")
public class MyTestHooks {

} // end of MyTestHooks

// OPTION 3: Or Project names or numbers
@TransportProject(["ProjectNewInstallation", "931608C4-E075-4E89-
AA35-66E6FD74770B"])
public class MyTestHooks {
   // …
} // end of MyTestHooks
```

Please note the following information on transporting Groovy hook modules:

- Do not change the file name of the modules once a milestone that belongs to one of the annotated projects has been closed for the first time.
- Only use one class per module.
- The modules are included in each milestone of the annotated projects.
- When importing a milestone, each module is replaced, i.e. overwritten.
- During import, the system searches for the file name of the modules in the directories set by the parameter **HOOK_GROOVY_DIRS_CUSTOMER**. If the file is found, the file is overwritten.
- If no file with the name was found, the file is copied to the first directory from **HOOK_GROOVY_DIRS_CUSTOMER**.

- If no directory is configured via **HOOK_GROOVY_DIRS_CUSTOMER**, a subfolder **groovy_hooks** is created in the d.3 configuration directory (location of **d3config.ini**) and the file is saved in the subfolder.
- When the module file is saved in the target repository for the first time, you must restart the d.3 processes.
- If a module file already exists and the module file is replaced, the parameter **HOOK_GROOVY_RELOAD_ON_CHANGE** determines whether the module is activated directly.
- If you want to delete a Groovy hook module in the target system, you must remove the annotation of the module in the source system and delete the file in the target system.

1.5.6. d.3 entry points

General

The entry points are triggered by user actions and then processed one after the other. If a hook function within the processing chain is exited with a value not equal to 0, the chain is interrupted. Example: During the manual import of an invoice, it is determined that the customer number contains an incorrect value. In this case, the invoice is prevented from being saved in the system.

The following sub-chapters describe the available entry points for hook functions in d.3. For the parameters of a hook function described, the d.3 archive objects available in the context of the call are transferred to the Groovy hook functions. The d.3 interface is always transferred first when a Groovy hook function is called.

Example

```
import com.dvelop.d3.server.Entrypoint
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;

class MyHooks{
    @Entrypoint(entrypoint="hook_insert_entry_10")
    public int checkIncommingDocuments(D3Interface d3, User user,
DocumentType docType, Document doc){
          println "groovy hook insert_entry_10";
          doc.field[1] = "<New Value>";
          return 0;
    }// end of checkIncommingDocuments
} // end of MyHooks
```

> **Note**
>
> You can also register multiple methods per entry point. If the order of the call is relevant, you can specify the order with the numerical attribute **order** in the annotation. The default value for the **order** attribute is "1".
>
> Before all methods of a class are called for the same entry point, the methods are sorted in ascending order according to the **order** attribute. If a method is to be called after another method for the same entry point, the **order** value of the method called later must be greater.

Example

```
// ...
    @Entrypoint(entrypoint="hook_insert_entry_10", order = 2)
    def checkIncommingDocuments_2(D3Interface d3, User user, DocumentType
docType, Document doc){
          println "Second method to handle this entrypoint
hook_insert_entry_10";
          doc.field[2] = "<New Value>";
          return 0;
    }// end of checkIncommingDocuments
}/ end of MyHooks
```

> **Note**
> A second annotation type is **@Condition**. You can use this annotation type to define conditions for calling an annotated method.
>
> You can use the **doctype** property to specify IDs of d.3 document types. If the entry point has a parameter of type **Document** or a **DocumentType**, the contained document type IDs are compared with the condition. The method is called if there is at least one match.

```
// ...
    @Entrypoint(entrypoint="hook_insert_entry_10", order=2)
    @Condition(doctype= [ "DA1" ] )
    def insertEntry10_2(D3Interface d3, User user, DocumentType docType,
Document doc)
    {
// ...
```

Example – Specifying multiple document type IDs

```
// ...
    @Condition(doctype=["DA1", "DA2", "DA3"])
    def insertEntry10_2(D3Interface d3, User user, DocumentType docType,
Document doc)
    {
// ...
```

Example – Multiple IDs using constant variables

```
// ...
 static final String Photo                                = "DFOTO";  //
Document type Photo
 static final String curriculumVitae                      = "DLELA";  //
Document type Curriculum vitae
 static final String personnelMasterData                  = "DPSB";   //
Document type personal master data
 static final String certificates                         = "DZEUG";  //
Document type certificates
 @Condition(doctype=[Photo, curriculumVitae, personnelMasterData,
certificates])
// ...
```

Dependent files

hook_dep_doc_entry_10

```
int hook_dep_doc_entry_10(D3Interface d3, Document doc, String status,
Integer fileId, UserOrUserGroup editor, String depExt, Integer transfer)
```

Call time: Before entering a dependent file in the database

Parameters	Description
d3	the d.3 interface
doc	The document to which the dependent file belongs
status	Current status of the document (**Be**, **Pr**, **Fr**, **Ar**)
fileId	Version of the document
editor	Editor or verification group of the document if the status is Processing or Verification
depExt	Data extension of the dependent file
transfer	1: Call during status transfer

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.UserOrUserGroup;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_dep_doc_entry_10" )
   // (4)
   public int doSomething (D3Interface d3, Document doc, String status,
Integer fileId, UserOrUserGroup editor, String depExt, Integer Transfer ){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
5. An output is created in the log file using the log function.
6. The function is ended with the return value **0** .

hook_dep_doc_exit_10

```
int hook_dep_doc_exit_10(D3Interface d3, Document doc, String status,
Integer fileId, UserOrUserGroup editor, String depExt, Integer transfer)
```

Call time: After entering the dependent file in the database.

Parameters	Description
d3	the d.3 interface
doc	The document for which the dependent file was entered
status	Current status of the document
fileID	Version of the document
editor	Editor or verification group of the document if the status is Processing or Verification
depExt	Data extension of the dependent file
transfer	1: Call during status transfer

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.UserOrUserGroup;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_dep_doc_exit_10" )
   // (4)
   public int doSomething(D3Interface d3, Document doc, String status,
Integer fileId, UserOrUserGroup editor, String depExt, Integer Transfer){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

1. Import of the required libraries from the *groovyhook.jar* file.
2. Deploy a custom class of type *public*; *public* can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name *doSomething*.
5. An output is created in the log file using the log function.
6. The function is ended with the return value *0* .

Updating the property values (UpdateAttributes)

hook_upd_attrib_entry_20

```
int hook_upd_attrib_entry_20(D3Interface d3, Document doc, User user,
DocumentType docType, DocumentType docTypeNew)
```

Available fields: All fields transferred during the API call. However, only the **dok_dat_** fields and the **text** field can be changed.

Call time: The new attributes have been received but not yet checked for plausibility.

Parameters	Description
d3	the d.3 interface
doc	Document object with the property values to be updated. The values can be changed in this hook function.
user	the executing user
docType	Document type before the property update
docTypeNew	Document type after property update

> **Note**
> The values **docType** and **docTypeNew** only differ if there was actually a change of document type.

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.Condition;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_upd_attrib_entry_20" )
   // (4)
   @Condition( doctype = [ "XXXX" ] )
   // (5)
   public int doSomething( D3Interface d3, Document doc, User user,
DocumentType docType, DocumentType docTypeNew ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

[de]

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.

6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_upd_attrib_exit_10

> **Note**
> This hook function is only executed if no errors occurred before. If the function returns a value other than zero, the update is cancelled and the changes are undone.

```
int hook_upd_attrib_exit_10(D3Interface d3, Document doc, Integer
errorCode, User user, DocumentType docType, DocumentType docTypeOld,
Document docOld)
```

Call time: Immediately before the DB transaction is terminated.

Parameters	Description
d3	the d.3 interface
doc	Document object with the updated property values. The values can be changed in this hook function.
errorCode	0: Update successful <> 0: Error number, supplied by the database server
user	the executing user
docType	Document type after property update
docTypeOld	Document type before the property update
docOld	Document object before the property update

> **Note**
> The values **docType** and **docTypeOld** only differ if there was actually a change of document type.

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.Condition;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_upd_attrib_exit_10" )
   // (4)
   @Condition( doctype = [ "XXXX" ] )
   // (5)
   public int doSomething( D3Interface d3, Document doc, Integer errorCode,
User user, DocumentType docType, DocumentType docTypeOld ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
```

```
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_upd_attrib_exit_20

> **Note**
> This hook function is always executed, i.e. the function is executed even if an error occurred beforehand. We therefore recommend evaluating the parameter **errorCode**.

Parameters	Description
d3	the d.3 interface
doc	Document object with the updated property values
errorCode	0: Update was successful <> 0: Error number, supplied by the database server
user	the executing user
docType	Document type after property update
docTypeOld	Document type before the property update
docOld	Document object before the property update

```
int hook_upd_attrib_exit_20(D3Interface d3, Document doc, Integer
errorCode, User user, DocumentType docType, DocumentType docTypeOld,
Document docOld)
```

Call time: After ending the database transaction.

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.Condition;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_upd_attrib_exit_20" )
```

```
   // (4)
   @Condition( doctype = [ "XXXX" ] )
   // (5)
   public int doSomething( D3Interface d3, Document doc, Integer errorCode,
User user, DocumentType docType, DocumentType docTypeOld ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

Importing documents (Import Document)

hook_hostimp_entry_10

> **Warning**
> This entry point is currently not compatible with the Groovy hook interface and cannot yet be used. Use the JPL variant of the entry point.

```
int hook_hostimp_entry_10(D3Interface d3,  String importDir, String
fileName, Document doc, DocumentType docType, String newImport)
```

Call time: Only called during host import, directly after reading in the **default.ini** and the JPL attribute file.

The Unicode conversion has not yet been carried out at this point. The values of the translatable datasets have also not yet been converted. It is possible to change the transferred property values.

Parameters	Description
d3	the d.3 interface
ImportDir	Directory from which the file is imported
fileName	File name of the file to be imported
doc	The document object to be imported
docType	Document type of the document to be imported
newImport	1: import of a new document 0: Import a new file version for an existing document

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
```

```
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.Condition;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_hostimp_entry_10" )
   // (4)
   @Condition( doctype = [ "XXXX" ] )
   // (5)
   public int doSomething(D3Interface d3,  String importDir, String
fileName, Document doc, DocumentType docType, String newImport){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_insert_entry_10

```
hook_insert_entry_10 (D3Interface d3, User user, DocumentType docType,
Document doc)
```

Call time:

- Before the import. The system only checked whether the connection to the database is functioning properly. It is possible to change the transferred property values.
- During data validation for a subsequent document import (**API ValidateAttributes** with parameter **"function" = "Insert"**).

The document properties can be accessed via the document object. You can change the values of the advanced properties in the hook.

Parameters	Description
d3	the d.3 interface

Parameters	Description
user	the executing user
docType	Document type of the new document to be imported
doc	The new document object

> **Note**
> Scenario: When a document is saved in the d.3 system, the error message "Hello World" should be displayed in the d.3 log file.

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.Condition;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_insert_entry_10" )
   // (4)
   @Condition( doctype = [ "XXXX" ] )
   // (5)
   public int doSomething( D3Interface d3, User user, DocumentType docType,
Document doc ) {
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_insert_entry_20

```
hook_insert_entry_20 (D3Interface d3, Document doc, DocumentType docType,
User user)
```

Call time:

- Before the import. The file has already been transferred to the target directory.
- The SQL commands for saving the document metadata have been created.
- The transferred document properties can no longer be changed.
- The property values have not yet been checked for validity (value range, reg. expression, min/max range, etc.).

Parameters	Description
d3	the d.3 interface
doc	The document object to be imported
docType	Document type of the document to be imported
user	the executing user

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.Condition;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_insert_entry_20" )
   // (4)
   @Condition( doctype = [ "XXXX" ] )
   // (5)
   public int doSomething( D3Interface d3, Document doc, DocumentType
docType, User user ) {
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_insert_exit_10

```
int hook_insert_exit_10 (D3Interface d3, Document doc, String
fileDestination, Integer importOk, User user, DocumentType docType)
```

Call time: After the import. The database transaction has not yet been closed. You can still force an undo to undo the import.

Parameters	Description
d3	the d.3 interface
doc	The new document
fileDestination	Path and name of the target file (information on where the target file was saved)
importOk	1: no error has occurred so far 0: an error occurred when importing the document
user	the executing user
docType	Document type of the new document

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.Condition;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_insert_exit_10" )
   // (4)
   @Condition( doctype = [ "XXXX" ] )
   // (5)
   public int doSomething(D3Interface d3, Document doc, String
fileDestination, Integer importOk, User user, DocumentType docType ) {
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.

6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_insert_exit_20

```
int hook_insert_exit_20(D3Interface d3, Document doc, String
fileDestination, Integer importOk, User user, DocumentType docType)
```

Call time: After the import. The database transaction was closed (**COMMIT** or **ROLLBACK**). A successful import can no longer be undone.

Parameters	Description
d3	the d.3 interface
doc	The new document
fileDestination	Path and name of the target file (information on where the target file was saved)
importOk	1: no error has occurred so far 0: an error occurred when importing the document
user	the executing user
docType	Document type of the new document

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.Condition;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_insert_exit_20" )
   // (4)
   @Condition( doctype = [ "XXXX" ] )
   // (5)
   public int doSomething( D3Interface d3, Document doc, String
fileDestination, Integer importOk, User user, DocumentType docType ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called.

The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.

6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_insert_exit_30

```
int hook_insert_exit_30 (D3Interface d3, Document doc, String
fileDestination, Integer importOk, User user, DocumentType docType)
```

Call time: After the import. The database transaction has already been closed.

> **Note**
> You can use the function in the same way as the hook_insert_exit_20 function.

Releasing documents (Release Document)

hook_release_entry_10

> **Note**
> If this hook function returns a value not equal to 0, the release is cancelled.

```
int hook_release_entry_10(D3Interface d3, Document doc, User user,
DocumentType docType, String unblock)
```

Call time: Directly before starting the document release. The system has determined that the user has authorization to release the document.

Parameters	Description
d3	the d.3 interface
doc	The document to be released
user	the executing user
docType	Document type of the document to be released
unblock	1: When the document is unblocked If the document is not unblocked, no value appears.

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.Condition;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_release_entry_10" )
   // (4)
   @Condition( doctype = [ "XXXX" ] )
   // (5)
```

```
   public int doSomething( D3Interface d3, Document doc, User user,
DocumentType docType, String unblock ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_release_exit_10

```
int hook_release_exit_10(D3Interface d3, Document doc, User user, Integer
errorCode, DocumentType docType, String unblock)
```

Call time: After the release has been carried out and the database transaction has been completed.

Parameters	Description
d3	the d.3 interface
doc	The released document
user	the executing user
errorCode	0: Release was successful If the release was not successful, an error code appears.
docType	Document type of the released document
unblock	1: When the document is unblocked Not equal to 1: normal release

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.Condition;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_release_exit_10" )
```

```
   // (4)
   @Condition( doctype = [ "XXXX" ] )
   // (5)
   public int doSomething( D3Interface d3, Document doc, User user, Integer
errorCode, DocumentType docType, String unblock ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

Verifying a document (Verify Document)

hook_verify_entry_10

> **Note**
> If this hook function returns a value not equal to 0, the verification is cancelled.

```
int hook_verify_entry_10(D3Interface d3, Document doc, Integer versionId,
User user)
```

Call time: Directly before starting the database transaction. The system has determined that the user has permission to verify the document.

Parameters	Description
d3	the d.3 interface
doc	The document to be checked
versionId	Number of the document version to be checked
user	the executing user

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;
```

```
// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_verify_entry_10" )
   // (4)
   public int doSomething( D3Interface d3, Document doc, Integer versionId,
User user ){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_verify_exit_10

```
int hook_verify_exit_10(D3Interface d3, Document doc, Integer versionId,
User user, Integer errorCode)
```

Call time: After performing the verification. After completion of the database transaction.

Parameters	Description
d3	the d.3 interface
doc	The verified document
versionId	Number of the verified document version
user	the executing user
errorCode	0: Verification successful If verification was not successful, the database error number appears.

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
```

```
   @Entrypoint( entrypoint = "hook_verify_exit_10" )
   // (4)
   public int doSomething( D3Interface d3, Document doc, Integer versionId,
User user, Integer errorCode ){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

Document search (GetDocumentList / SearchDocument)

hook_search_entry_05

```
int hook_search_entry_05(D3Interface d3, User user, DocumentType docType,
Document searchContext)
```

Call time: Before searching for documents. Before accessing the full-text engine.

In this hook function, you can use the hook property field **searchtext_expression** to customize the full-text search term. Example:

```
// den aktuellen Volltext-Suchbegriff ausgeben
println d3.hook.getProperty("searchtext_expression")
// den Volltext-Suchbegriff verändern
d3.hook.setProperty("searchtext_expression", "mein Hook Volltext-
Suchbegriff")
```

You can also adjust the full-text search term using **searchContext**.

Parameters	Description
d3	the d.3 interface
user	the executing user
docType	The document type of the documents searched for (if specified)
searchContext	The search terms can be accessed via a document object

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
```

```
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_search_entry_05" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( D3Interface d3, User user, DocumentType docType,
Document searchContext ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

If you want to work with a search term or output a search term, you can use the **getImportParams()** method of **D3RemoteInterface**. Example:

```
@Entrypoint(entrypoint = "hook_search_entry_05")
 int hook_search_entry_05(D3Interface d3, User user, DocumentType docType,
Document searchContext)
 {
     d3.log.info("### hook_search_entry_05 ---- START")

     //(1)
     String searchText

     Map<String, Object> dict;

     dict = d3.remote.getImportParams();

     //(2)
     searchText = dict.get("searchtext_expression").toString()
```

```
    d3.log.info("The searchtext is $searchText");
    d3.log.info("### hook_search_entry_05 ---- END")

    return 0;
}
```

Description of the blocks

1. The import parameters of the search (**GetDocumentList**) are queried and saved in a map using **getImportParams()**. The import parameter for full text is called **searchtext_expression** and represents the key.
2. The value is queried via the key and saved in a string. The string is output in the log.

You can also output all keys and values of the map and thus determine all import parameters with their values. Example:

```
   @Entrypoint(entrypoint = "hook_search_entry_05")
    int hook_search_entry_05(D3Interface d3, User user, DocumentType
docType, Document searchContext)
    {
        d3.log.info("### hook_search_entry_05 ---- START")

        Map<String, Object> dict;

        dict = d3.remote.getImportParams();

                //(1)
        for ( String key : dict.keySet() ) {
            d3.log.info("### hook_search_entry_05 Searchitem: Key: " + key);
        }

                //(2)
        for ( Object value : dict.values() ) {
            d3.log.info("### hook_search_entry_05 Searchitem: Value: " +
value);
        }

        d3.log.info("### hook_search_entry_05 ---- END")
        return 0;
    }
```

Description of the blocks

1. **KeySet** is determined using a **for** loop.
2. All values are determined using a **for** loop.

hook_search_entry_10

```
int hook_search_entry_10(D3Interface d3, User user, DocumentType docType,
Document searchContext)
```

Call time:

- Before searching for documents. The transferred search criteria have not yet been checked for plausibility. The conversion of the search criteria to lower case or upper case (if enabled) has not yet been carried out.
- During data validation for a subsequent search (API **ValidateAttributes** with parameter **"function" = "Search"**).

The search terms have been adopted. You can change the search terms in the hook using **searchContext**.

Parameters	Description
d3	the d.3 interface
user	the executing user
docType	The document type of the documents searched for (if specified)
searchContext	The search terms can be accessed via a document object

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_search_entry_10" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( D3Interface d3, User user, DocumentType docType,
Document searchContext ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_search_entry_20

```
int hook_search_entry_20(D3Interface d3, User user, DocumentType docType)
```

Call time: Before searching for documents. The **SELECT** command for searching for documents has already been compiled according to the search criteria.

Parameters	Description
d3	the d.3 interface
user	the executing user
docType	The document type of the documents searched for (if specified)

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_search_entry_20" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( D3Interface d3, User user, DocumentType docType
){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_search_exit_30

```
hook_search_exit_30(D3Interface d3, User user, Integer errorCode, Integer
noResults, Integer noRefused, DocumentType docType)
```

Call time: At the end of the search, before the results are delivered to the client.

Parameters	Description
d3	the d.3 interface
user	User who performs the search

Parameters	Description
errorCode	0: Success If an error occurs, an error code appears.
noResults	Number of hits
noRefused	Number of refused hits
docType	The document type of the documents searched for (if specified)

Return value: is ignored

> **Note**
> You can use the hook property **no_results_refused** to disable display of the number of refused hits to the caller. Use this property, for example, if you do not want users to see that there are search results.
>
> Call: **d3.hook.setProperty("no_results_refused", "0")**

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_search_exit_30" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( D3Interface d3, User user, Integer errorCode,
Integer noResults, Integer noRefused, DocumentType docType ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.

6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

Importing a new version (ImportNewVersionDocument)

hook_new_version_entry_10

> **Note**
> The function is called at **ValidateAttributes nextcall=ImportNewVersionDocument**.
>
> As of d.3 version 8, the document storage has been restructured, which is why the parameters **fileSource** and **fileDestination** no longer have any contents. However, the parameters still exist, meaning the hook interface does not change. The parameters must therefore still be accepted. An empty string is transferred as the value.

```
int hook_new_version_entry_10(D3Interface d3, Document doc, String
fileSource, String fileDestination, User user, DocumentType docType)
```

Call time: The system checked whether the document already exists in d.3. The document properties can still be changed.

Parameters	Description
d3	the d.3 interface
doc	Document for which a new file version is to be imported
fileSource	deprecated with version 8.0
fileDestination	deprecated with version 8.0
user	the executing user
docType	Document type of the file version to be imported

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_new_version_entry_10" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( D3Interface d3, Document doc, String fileSource,
String fileDestination, User user, DocumentType docType ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_new_version_entry_20

> **Note**
> As of d.3 version 8, the document storage has been restructured, which is why the parameters **fileSource** and **fileDestination** no longer have any contents. However, the parameters still exist, meaning the hook interface does not change. The parameters must therefore still be accepted. An empty string is transferred as the value.

```
int hook_new_version_entry_20(D3Interface d3, Document doc, String
fileSource, String fileDestination, User user, DocumentType docType)
```

Call time: The system successfully checked whether the new file version exists. The version has not yet been imported. The database transaction has not yet been started. The document properties have been validated and can no longer be changed.

Parameters	Description
d3	the d.3 interface
doc	Document for which a new file version is to be imported
fileSource	deprecated with version 8.0
fileDestination	deprecated with version 8.0
user	the executing user
docType	Document type of the file version to be imported

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_new_version_entry_20" )
   // (4)
   @Condition( doctype = ["XXXX"] )
```

```
   // (5)
   public int doSomething( D3Interface d3, Document doc, String fileSource,
String fileDestination, User user, DocumentType docType ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_new_version_entry_30

> **Note**
> As of d.3 version 8, the document storage has been restructured, which is why the parameters **fileSource** and **fileDestination** no longer have any contents. However, the parameters still exist, meaning the hook interface does not change. The parameters must therefore still be accepted. An empty string is transferred as the value.

```
int hook_new_version_entry_30(D3Interface d3, Document doc, String
fileSource, String fileDestination, User user, DocumentType docType)
```

Call time: Executed after hook_new_version_entry_20. You can use the information in the same way as hook_new_version_entry_20.

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_new_version_entry_30" )
   // (4)
   @Condition( doctype = ["XXXX"] )
```

```
   // (5)
   public int doSomething( D3Interface d3, Document doc, String fileSource,
String fileDestination, User user, DocumentType docType ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_new_version_exit_10

```
int hook_new_version_exit_10(D3Interface d3, Document doc, Integer
errorCode, User user, DocumentType docType, Document docOld)
```

Call time: The database transaction was started. All properties including the multi-value properties (60 fields) have been updated.

Parameters	Description
d3	the d.3 interface
doc	document for which a new file version was imported
errorCode	1: Error while updating the properties 0: The update of the properties was successful
user	the executing user
docType	document type of the imported file version
docOld	Document object before the property update

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
```

```
   @Entrypoint( entrypoint = "hook_new_version_exit_10" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething(D3Interface d3, Document doc, Integer errorCode,
User user, DocumentType docType, Document docOld){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_new_version_exit_20

> **Note**
> As of d.3 version 8, document storage has been restructured, which is why the parameter **fileDestination** no longer has any contents and has been discontinued.

```
int hook_new_version_exit_20(D3Interface d3, Document doc, String
fileDestination, Integer importOk, User user, DocumentType docType,
Document docOld)
```

Call time: Also the multi-value property fields (60-fields) were updated. The new file version has been imported, together with any associated dependent files. The database transaction is ended (**COMMIT** or **ROLLBACK**).

Parameters	Description
d3	the d.3 interface
doc	document for which a new file version was imported
fileDestination	deprecated with version 8.0
ImportOk	1: Import of the new version was successful 0: Import of the new version was cancelled with an error
user	the executing user
docType	document type of the imported file version
docOld	Document object before the property update

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
```

```
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_new_version_exit_20" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( D3Interface d3, Document doc, String
fileDestination, Integer importOk, User user, DocumentType docType,
Document docOld){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_new_version_exit_30

```
int hook_new_version_exit_30(D3Interface d3, Document doc, Integer
importOk, Integer errorCode, User user, DocumentType docType, Document
docOld)
```

Call time: Equivalent to **hook_new_version_exit_20**.

Parameters	Description
d3	the d.3 interface
doc	document for which a new file version was imported
ImportOk	1: Import of the new version was successful 0: Import of the new version was cancelled with an error
errorCode	In case of error (**importOk=0**): Error code of the encountered error
user	the executing user

docType	document type of the imported file version
docOld	Document object before the property update

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_new_version_exit_30" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( D3Interface d3, Document doc, Integer importOk,
Integer errorCode, User user, DocumentType docType, Document docOld){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

Creating/editing TIFF or PDF documents

hook_rendition_entry_10

```
int hook_rendition_entry_10(D3Interface d3, Document doc, User user)
```

Call time: Before the start of the rendition, if the rendition was called by a d.3 user (via d.3 API or Server API). The function is not called if an automatic call is made via saved rules.

> **Note**
> You can cancel the call using a return value not equal to 0.

Parameters	Description
d3	the d.3 interface
doc	The document from which a TIFF or PDF rendition is to be created
user	the d.3 user who requested the rendition

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_rendition_entry_10" )
   // (4)
   public int doSomething( D3Interface d3, Document doc, User user ){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_rendition_entry_20

```
int hook_rendition_entry_20(D3Interface d3, Document doc, DocumentType
docType, String sourcePath, String sourceFilename, String destFilename)
```

Call time: Directly before the rendition job is sent to **d.ecs rendition service**.

Note

In this hook function, render options are transferred to d.ecs rendition service via the hook property fields **rendition_parameter_name** and **rendition_parameter_value**.

Example:

```
d3.hook.setProperty("rendition_parameter_name", 1,
"PRINT_FORMAT")
d3.hook.setProperty("rendition_parameter_value", 1, "A2")
```

Parameters	Description
d3	the d.3 interface
doc	The document from which a TIFF or PDF rendition is to be created
docType	Document type of this document
sourcePath	Source path of the master file
sourceFilename	File name of the master file
destFilename	File name of the finished rendition file

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_rendition_entry_20" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( D3Interface d3, Document doc, DocumentType
docType, String sourcePath, String sourceFilename, String destFilename ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called.

The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.

6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_rendition_exit_30

```
int hook_rendition_exit_30(D3Interface d3, Document doc, String destStatus,
String tiffFilename, Integer errorCode, String fileType)
```

Call time: After retrieval of the finished TIFF/PDF file from d.ecs rendition server.

Parameters	Description
d3	the d.3 interface
doc	The document from which a TIFF or PDF rendition was created
destStatus	Target status of the document (**B**, **P**, **F**, **A**)
tiffFilename	Target path + file name of the rendition file
errorCode	0: Success -1: Error when retrieving the file from d.ecs rendition service (see d.3 log file)
fileType	File type that was rendered (**P1**, **T1**, **TXT**)

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_rendition_exit_30" )
   // (4)
   public int doSomething( D3Interface d3, Document doc, String destStatus,
String tiffFilename, Integer errorCode, String fileType ){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.

6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

Login

hook_val_passwd_entry_10

```
int hook_val_passwd_entry_10(D3Interface d3, User user, String appLanguage,
String appVersion)
```

Call time: Before the user name and password are validated by the API function **ValidatePasswordForUser**. A long user name has already been swapped for the internal name. Sign-in data cannot be changed.

Parameters	Description
d3	the d.3 interface
user	User to be logged on (d.3 short or long name, LDAP user name)
appLanguage	Language ID transferred by the application, e.g. **049** for German or **001** for English
appVersion	Version string transferred by the application: • Characters 1 to 3: Module identifier, e.g. 200 for d.xplorer • Characters 4 to 6: Version of the module, e.g. 800 for version 8.0.0 • Characters 7 to 8: Log level, e.g. 9 for DEBUG

Return:

A value **!= 0** leads to a change in the return value of the API function **ValidatePasswordForUser** and thus to the cancelling of the sign-in process. The return value of the hook is subtracted from 9500 (example for the value -1: 9500-(-1) = 9501). The number is returned to the client and must therefore be saved in **msglib.usr**.

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_val_passwd_entry_10" )
   // (4)
   public int doSomething( D3Interface d3, User user, String appLanguage,
String appVersion ){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.

4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_val_passwd_exit_10

```
int hook_val_passwd_exit_10(D3Interface d3, int errorCode, User user,
String appLanguage, String appVersion)
```

Call time: The test of user name and password in relation to the d.3 user master or possibly a directory server (via LDAP/Kerberos) has been performed. The result is fixed and is transferred as the parameter **error**.

Parameters	Description
d3	the d.3 interface
errorCode	Error code of the user name/password check: • 0: Success • 0002: User name/password invalid
user	d.3 user to be signed in
appLanguage	Language ID transferred by the application, e.g. **049** for German or **001** for English
appVersion	Version string transferred by the application:

Return: A return value **!= 0** leads to cancellation (see hook_val_passwrd_entry_10).

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_val_passwd_exit_10" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( D3Interface d3, int errorCode, User user, String
appLanguage, String appVersion ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

Deleting a document (DeleteDocument)

hook_delete_entry_10

```
int hook_delete_entry_10(D3Interface d3, Document doc, User user,
DocumentType docType)
```

Call time: Before deletion of the document. The system successfully checked whether the user is allowed to delete the document. The deletion process can still be cancelled by a return value not equal to 0.

Parameters	Description
d3	the d.3 interface
doc	The document to be deleted
user	The user who wants to delete the document
docType	Document type of the document to be deleted

Privileged deletion:

By default, documents are deleted by moving them to the internal recycle bin. However, you can use privileged deletion to immediately delete a document version completely from the system (database, document tree and, if necessary, from the secondary storage system). You need a separate license from d.velop AG and may require an additional consultation. In this entry point, you can activate privileged deletion by using the hook property **DELETE_PRIVILEGED**.

Property	Description
DELETE_PRIVILEGED	0: Delete by moving to the recycle bin (default value) 1: Privileged deletion Removes documents irretrievably from the system.

Example:

```
d3.hook.setProperty("DELETE_PRIVILEGED", "1")
```

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;
```

```
// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_delete_entry_10" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( D3Interface d3, Document doc, User user,
DocumentType docType ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_delete_exit_10

```
int hook_delete_exit_10(D3Interface d3, Document doc, User user, Integer
errorCode, DocumentType docType)
```

Call time: After the deletion of a document. The deletion process can no longer be cancelled at this entry point. You can use the parameter **errorCode** to check whether the deletion was successful.

Parameters	Description
d3	the d.3 interface
doc	The document to be deleted
user	User who is deleting the document
errorCode	0: Document was successfully deleted If the deletion fails, an error code appears.
docType	Document type of the document to be deleted

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
```

```
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_delete_exit_10" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( D3Interface d3, Document doc, User user, Integer
errorCode, DocumentType docType ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Description of the blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name *doSomething*.
6. An output is created in the log file using the log function.
7. The function is ended with the return value *0* .

Deleting links (unlink)

hook_unlink_entry_30

```
int hook_unlink_entry_30(D3Interface d3, Document docFather, Document
docChild)
```

Call: Directly before execution of the database command to remove the link.

Parameters	Description
d3	the d.3 interface
docFather	The parent document
docChild	The child document

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
```

```
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_unlink_entry_30" )
   // (4)
   public int doSomething( D3Interface d3, Document docFather, Document
docChild ){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; public can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the order specified there. The d.3 object is always the first parameter passed when a Groovy hook function is called. The function name does not have to match the entry point like it does in JPL. The function name **doSomething** must be assigned a meaningful name.
5. An output is generated in the log file using a simple log function.
6. The function is finished with a return value **0**.

hook_unlink_exit_10

```
int hook_unlink_exit_10(D3Interface d3, Document docFather, Document
docChild, Integer unlinkErrorCode, Integer errorCode)
```

Call: After the unlinking of two documents.

Parameters	Description
d3	the d.3 interface
docFather	The parent document
docChild	The child document
unlinkErrorCode	0: Unlink was successful -1: Parent and child element are identical or an element does not exist -2: The two documents are not linked -4: A database error occurred when removing the link (see **errorCode**)
errorCode	0 = OK Else: database or hook error

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
```

```
   // (3)
   @Entrypoint( entrypoint = "hook_unlink_exit_10" )
   // (4)
   public int doSomething( D3Interface d3, Document docFather, Document
docChild, Integer unlinkErrorCode, Integer errorCode ){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; public can also be omitted in Groovy contexts.
3. To assign a Groovy-function to an entry point a registration of the function for an entry point applies by a Groovy-annotation with the predefined values.
4. The different entry types require the parameters described in the Groovy hook documentation in the order specified there. The d.3 object is always the first parameter passed when a Groovy hook function is called. The function name does not have to match the entry point like it does in JPL. The function name **doSomething** must be assigned a meaningful name.
5. An output is generated in the log file using a simple log function.
6. The function is finished with a return value **0**.

Mailbox (SendHoldFile)

hook_ack_holdfile_exit_10

```
int hook_ack_holdfile_exit_10(D3Interface d3, User user, Document doc,
Integer holdfileId)
```

Acknowledgement of a mailbox entry.

Call time:

After the acknowledgement of a mailbox entry by calling the API function **AcknowledgeReceivedHoldFile**.

Acknowledgment can no longer be prevented, as the call takes place after acknowledgment.

Parameters	Description
d3	the d.3 interface
user	User who triggered the acknowledgement
doc	The acknowledged document
holdfileId	ID of the mailbox entry

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
```

```
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_ack_holdfile_exit_10" )
   // (4)
   public int doSomething( D3Interface d3, User user, Document doc, Integer
holdfileId ){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

<listitem>
Import of the required libraries from the **groovyhook.jar** file.
</listitem>
<listitem>
Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
</listitem>
<listitem>
To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
</listitem>

1. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name doSomething.
2. An output is created in the log file using the log function.
3. The function is ended with the return value **0** .

Redlining (WriteRedline)

hook_write_redline_entry_10e

```
int hook_write_redline_entry_10(D3Interface d3, Document doc, User user,
DocumentType docType)
```

Call time: The function is executed before a redlining file is written.

Parameters	Description
d3	the d.3 interface
doc	The document for which the redlining file is stored
user	the executing user
docType	Document type of the document

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
```

```
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_write_redline_entry_10" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( D3Interface d3, Document doc, User user,
DocumentType docType ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

<listitem>
Import of the required libraries from the **groovyhook.jar** file.
</listitem>
<listitem>
Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
</listitem>
<listitem>
To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
</listitem>
<listitem>
The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
</listitem>
<listitem>
An output is created in the log file using the log function.
</listitem>
<listitem>
The function is ended with the return value **0** .
</listitem>

- Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.

hook_write_redline_exit_30

```
int hook_write_redline_exit_30(D3Interface d3, Document doc, User user,
DocumentType docType)
```

Call: After writing the redlining file (via d.3-API-call **WriteRedline**).

Parameters	Description
d3	the d.3 interface
doc	The document for which a redlining file was stored

Parameters	Description
user	the executing user
docType	Document type of the document

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_write_redline_exit_30" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( D3Interface d3, Document doc, User user,
DocumentType docType ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the *groovyhook.jar* file.
2. Deploy a custom class of type *public*; *public* can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional *condition* annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

Sending a resubmission (Send Holdfile)

hook_holdfile_entry_10

```
int hook_holdfile_entry_10(D3Interface d3, Document doc, UserOrUserGroup
recipient, UserOrUserGroup sender, Integer chainId, String notice, String
wvType)
```

Call time: Called before the transfer parameters are verified. The values of the transfer parameters are also available in the following hook property fields:

- **d3server_empfaenger_wv[1]**

- **d3server_sender_wv[1]**
- **d3server_kette_id**

These values can be changed using the call **d3.hook.property()**.

Parameters	Description
doc	The document to be placed in the resubmission
recipient	User or group object of the recipient
sender	User or group object of the sender
chainId	Chain ID to be used for this mailbox entry
notice	Subject text of the mailbox notification
wvTyp	Type ID of the mailbox notification Possible values: "" = normal mailbox notification "W" = workflow notification ... = other (possibly self-defined values)

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_holdfile_entry_10" )
   // (4)
   public int doSomething( D3Interface d3, Document doc, UserOrUserGroup
recipient, UserOrUserGroup sender, Integer chainId, String notice, String
wvType ){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
5. An output is created in the log file using the log function.
6. The function is ended with the return value **0** .

hook_holdfile_entry_20

```
int hook_holdfile_entry_20(D3Interface d3, Document doc, UserOrUserGroup
recipient, UserOrUserGroup sender, Integer chainId, String notice, String
wvType)
```

Call time: Called if the date etc. has already been checked for plausibility. However, the recipient's rights to the document have not yet been checked. The values can no longer be changed here.

For import parameters, see hook_holdfile_entry_10 (doc_id, recipient, sender, chain_id).

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_holdfile_entry_20" )
   // (4)
   public int doSomething( D3Interface d3, Document doc, UserOrUserGroup
recipient, UserOrUserGroup sender, Integer chainId, String notice, String
wvType ){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
5. An output is created in the log file using the log function.
6. The function is ended with the return value **0** .

hook_holdfile_entry_30

```
int hook_holdfile_entry_30(D3Interface d3, Document doc, UserOrUserGroup
recipient, UserOrUserGroup sender, Integer chainId, String notice, String
wvType)
```

Call time: Called up directly before the entry in the database, provided the recipient's rights to the document have already been checked. The values can no longer be changed here.

For import parameters: see hook_holdfile_entry_10 (doc_id, recipient, sender, chain_id).

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_holdfile_entry_30" )
   // (4)
   public int doSomething( D3Interface d3, Document doc, UserOrUserGroup
recipient, UserOrUserGroup sender, Integer chainId, String notice, String
wvType ){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
5. An output is created in the log file using the log function.
6. The function is ended with the return value **0** .

hook_holdfile_exit_10

```
int hook_holdfile_exit_10(D3Interface d3, Document doc, UserOrUserGroup
recipient, UserOrUserGroup sender, Integer chainId, Integer errorCode)
```

Call time: Directly after the database command that activates the resubmission.

Parameters	Description
d3	the d.3 interface
doc	The document for which a mailbox notification has been set
recipient	The recipient of the notification
sender	The sender of the notification
chainId	Chain ID to be used for this mailbox entry
errorCode	0: everything OK Otherwise: Database error code on entry of the resubmission into the database

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
```

```
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_holdfile_exit_10" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( D3Interface d3, Document doc, UserOrUserGroup
recipient, UserOrUserGroup sender, Integer chainId, Integer errorCode ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
5. An output is created in the log file using the log function.
6. The function is ended with the return value **0** .

Sending e-mails on resubmission

hook_send_email_entry_10

```
int hook_send_email_entry_10(D3Interface d3, Document doc, String
recipient, String sender, String subject, Integer trigger, String url_link)
```

Call: Before sending an e-mail.

Parameters	Description
d3	the d.3 interface
doc	The document for which the mailbox notification e-mail is sent
recipient	Recipient of the e-mail (d.3 user name or e-mail address)
sender	Sender of the e-mail (d.3 user name)
subject	Subject text
trigger	0 = no resubmission e-mail 1 = e-mail for resubmission 2 = e-mail for workflow resubmission

Parameters	Description
url_link	HTTP link for the view in d.3one

> **Note**
> The parameter is only filled if d.3one is installed.

In this hook function, e-mail properties can be set using the following hook property values:

Property	Description
api_email_body_file	Load e-mail body from a file. Name and path of a file containing the body text
api_email_mail_format	"html": HTML format Otherwise: Text-format (Standard)
api_email_attach	1 = attach the document to the e-mail 0 = do not attach document (default)

```
d3.hook.setProperty("api_email_body_file", "D:/hooks/data/myBody.html")
d3.hook.setProperty("api_email_mail_format", "html")
d3.hook.setProperty("api_email_attach", "1")
```

The properties will be reset after sending an e-mail successfully. The e-mail function can be cancelled with a return value not equal to 0.

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_send_email_entry_10" )
   // (4)
   public int doSomething( D3Interface d3, Document doc, String recipient,
String sender, String subject, Integer trigger, String url_link ){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.

5. An output is created in the log file using the log function.
6. The function is ended with the return value **0** .

hook_send_email_entry_20

```
int hook_send_email_entry_20(D3Interface d3, Document doc, String
recipient, String sender, String subject, Integer trigger, String url_link)
```

Call: Before sending an e-mail. E-mail address has been determined, group resolution has been performed.

> **Note**
> The hook function is only called once, i.e. not separately for each e-mail sent.

Parameters	Description
d3	the d.3 interface
doc	The document for which the mailbox notification e-mail is sent
recipient	Recipient of the e-mail (d.3 user name or e-mail address)
sender	Sender of the e-mail (d.3 user name)
subject	Subject text
trigger	0 = no resubmission e-mail 1 = e-mail for resubmission 2 = e-mail for workflow resubmission
url_link	HTTP link for the view in d.3one Note: The parameter is only filled if d.3one is installed.

In this hook function, e-mail properties can be set using the following hook property values:

Property	Description
api_email_body_file	Load e-mail body from a file. Name and path of a file containing the body text
api_email_mail_format	"html": HTML format Otherwise: Text-format (Standard)
api_email_attach	1 = attach the document to the e-mail 0 = do not attach document (default)

```
d3.hook.setProperty("api_email_body_file", "D:/hooks/data/myBody.html")
d3.hook.setProperty("api_email_mail_format", "html")
d3.hook.setProperty("api_email_attach", "1")
```

The properties will be reset after sending an e-mail successfully. The e-mail function can be cancelled with a return value not equal to 0.

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
```

```
   @Entrypoint( entrypoint = "hook_send_email_entry_20" )
   // (4)
   public int doSomething( D3Interface d3, Document doc, String recipient,
String sender, String subject, Integer trigger ){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
5. An output is created in the log file using the log function.
6. The function is ended with the return value **0** .

hook_send_email_exit_10

```
int hook_send_email_exit_10(D3Interface d3, Document doc, String recipient,
String sender, String subject, Integer retCode)
```

Call: After sending an e-mail.

Parameters	Description
d3	the d.3 interface
doc	The document for which the email was sent
recipient	Recipient of the e-mail (d.3 user name or e-mail address)
sender	Sender of the e-mail (d.3 user name)
subject	Subject text
retCode	1 = Message has been sent 0 = Message could not be sent

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_send_email_exit_10" )
   // (4)
   public int doSomething( D3Interface d3, Document doc, String recipient,
String sender, String subject, Integer retCode ){
```

```
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
5. An output is created in the log file using the log function.
6. The function is ended with the return value **0** .

Blocking a document

hook_block_entry_10

```
int hook_block_entry_10(D3Interface d3, Document doc, User user)
```

Call time: Before blocking a document with the status **Release**.

Parameters	Description
d3	the d.3 interface
doc	The document to be blocked
user	the executing user

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_block_entry_10" )
   // (4)
   public int doSomething( D3Interface d3, Document doc, User user ){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
5. An output is created in the log file using the log function.
6. The function is ended with the return value **0** .

hook_block_exit_10

```
int hook_block_exit_10(D3Interface d3, Document doc, User user)
```

Call time: After blocking a document with the status **Release**.

Parameters	Description
d3	the d.3 interface
doc	The currently blocked document
user	the executing user

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_block_exit_10" )
   // (4)
   public int doSomething( D3Interface d3, Document doc, User user ){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
5. An output is created in the log file using the log function.
6. The function is ended with the return value **0** .

Master data

hook_on_user_change_exit_10

```
int hook_on_user_change_exit_10(D3Interface d3, User actionUser, User
newUser, User oldUser)
```

In this entry point, adjustments can be made to a user object that has just been changed or is to be created.

Neither the creation of the user nor a change of the user can be prevented in this hook. This means that if this hook makes changes to the user object that cannot be written to the database (for example, because the maximum column lengths have been exceeded), the object is created as if the hook had not been executed.

Parameters	Description
d3	the d.3 interface
actionUser	the executing user
newUser	The new or changed user
oldUser	The unmodified user object

The following user properties can be changed via the **newUser** object using the corresponding setters:

Name of the property	Description
email	E-mail address of the user
phone	Phone number of the user
plant	Plant of the user
department	Department of the user
optField(int idx)	Optional fields for the user (array indices 1-10)

If the hook is ended with a return code not equal to "0", the changes made to the user object by the hook are ignored.

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_on_user_change_exit_10" )
   // (4)
   public int doSomething( D3Interface d3, User actionUser, User newUser,
User oldUser ){
      // (5)
      d3.log.error("Hello world!");
          // (6)
      def retval
      retval = newUser.setOptField(1, "myValue");
      retval = newUser.setPhone("012345-456789");
      retval = newUser.setPlant("myPlant");
      retval = newUser.setDepartment("myDepartment");
```

```
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
5. An output is created in the log file using the log function.
6. The function is ended with the return value **0** .

Status transfer

hook_transfer_entry_30

```
int hook_transfer_entry_30(D3Interface d3, User user, Document doc, Integer
fileId, String sourceStatus, String destStatus, UserOrUserGroup destEditor)
```

Call time: Before a document is transferred to another status.

Parameters	Description
d3	the d.3 interface
user	the executing user
doc	The document to be transferred
fileId	File ID of the document version
sourceStatus	Source status of the document (B for Processing, P for Verification, F for Release, A for Archive)
destStatus	Target status of the document (B for Processing, P for Verification, A for Archive)
destEditor	Target status **Processing**: User name or group name Target status **Verification**: Group name Otherwise empty

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.UserOrUserGroup;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_transfer_entry_30" )
   // (4)
```

```
   public int doSomething( D3Interface d3, User user, Document doc, Integer
fileId, String sourceStatus, String destStatus, UserOrUserGroup destEditor
){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
5. An output is created in the log file using the log function.
6. The function is ended with the return value **0** .

hook_transfer_exit_30

```
int hook_transfer_exit_30(D3Interface d3, User user, Document doc, Integer
fileId, String sourceStatus, String destStatus, UserOrUserGroup destEditor,
Integer errorCode)
```

Call time: After a document is transferred to another status.

Parameters	Description
d3	the d.3 interface
user	the executing user
doc	The document that was transferred
fileId	File ID of the document version
sourceStatus	Source status of the document (B, P, F, A)
destStatus	Target status of the document (B for Processing, P for Verification, A for Archive)
destEditor	Target status **Processing**: User object or group object Target status **Verification**: User or group object; null if no reviewer group specified
errorCode	0 = status transfer successful Error code otherwise

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.UserOrUserGroup;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
```

```
   @Entrypoint( entrypoint = "hook_transfer_exit_30" )
   // (4)
   public int doSomething( D3Interface d3, User user, Document doc, Integer
fileId, String sourceStatus, String destStatus, UserOrUserGroup destEditor,
Integer errorCode ){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
5. An output is created in the log file using the log function.
6. The function is ended with the return value **0** .

Validating property values (ValidateAttributes)

hook_validate_import_entry_10

```
int hook_validate_import_entry_10(D3Interface d3, User user, DocumentType
docType, Document doc, String nextcall)
```

> **Note**
> If this hook function returns a value other than 0, the validation of the search terms is cancelled.
>
> The API function **ValidateAttributes** returns the value **9500-(X)** (general error code in custom-specific hook-function). **X** is the return value of the hook.
>
> It is recommended to use a negative return value in the hook so that the output of the API call is >9500, as this range has been kept free.
>
> It is then possible to use a **msglib.usr** file with any text for the return code (e.g. 9542) on the client computers via the client distribution.
>
> You can find further information here: Number range for return values.

Call time: Only the properties of the new document to be imported were assigned.

Parameters	Description
d3	the d.3 interface
user	the executing user
docType	Document type of the document to be validated
doc	The document to be validated before importing
nextcall	The value of the parameter "nextcall" of the API function ValidateAttributes

> **Note**
> This function is executed in the context of the API function **ValidateAttributes**. This means that the function is not executed when a document is imported via the host import. The function is executed when an import is carried out via d.3 import, as this API function is called by this program before a document is imported.

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_validate_import_entry_10" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( D3Interface d3, User user, DocumentType docType,
Document doc, String nextcall ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
5. An output is created in the log file using the log function.
6. The function is ended with the return value **0** .

hook_validate_search_entry_10

```
int hook_validate_search_entry_10(D3Interface d3, User user, DocumentType
docType, Document searchContext, String nextcall)
```

> **Note**
> If this hook function returns a value other than 0, the validation of the search terms is cancelled. The API function **ValidateAttributes** returns the value **9500-(X)** (general error code in custom-specific hook-function). **X** is the return value of the hook. It is recommended to use a negative return value in the hook so that the output of the API call is >9500, as this range has been kept free. It is then possible to use a **msglib.usr** file with any text for the return code (e.g. 9542) on the client computers via the client distribution.

Call time: Only the search terms were transported to the corresponding context fields.

Parameters	Description
d3	the d.3 interface
user	the executing user
docType	Document type of the documents searched for if the search is document type-specific; otherwise empty document type object
searchContext	Document object via which the search terms can be read and changed
nextcall	The value of the parameter **nextcall** of the API function **ValidateAttributes**

This function is executed in the context of the API function **ValidateAttributes**.

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_validate_search_entry_10" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( D3Interface d3, User user, DocumentType docType,
Document searchContext, String nextcall ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the *groovyhook.jar* file.
2. Deploy a custom class of type *public*; *public* can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional *condition* annotation.

5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_validate_update_entry_10

```
int hook_validate_update_entry_10(D3Interface d3, User user, DocumentType
docType, Document doc, String nextcall)
```

Parameters	Description
d3	the d.3 interface
user	the executing user
docType	Document type of the document to be updated
doc	The d.3 document whose properties are to be updated. These properties can still be changed here. Otherwise, if no document ID is specified, the empty string is passed
nextcall	The value of the parameter **nextcall** of the API function **ValidateAttributes**

This hook function is called in the context of the API function **ValidateAttributes** .

If a property is changed for several documents at the same time using **changeatt.dxp**, the entry point **hook_validate_update_entry_10** does not take effect, as no validation (**ValidateAttributes**) takes place.

This validation takes place when an attribute is adjusted for a document via the properties.

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_validate_update_entry_10" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( D3Interface d3, User user, DocumentType docType,
Document doc, String nextcall ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the *groovyhook.jar* file.

2. Deploy a custom class of type *public*; *public* can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional *condition* annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

Linking documents or dossiers (LinkDocuments)

hook_link_entry_30

```
int hook_link_entry_30(D3Interface d3, Document docFather, Document
docChild)
```

> **Note**
> This hook function is activated only if no errors occurred previously. If this function returns a value other than 0, the link action is aborted with an error.

Call time: All link data is correct. Directly before the database command that registers the link.

Parameters	Description
docFather	The parent document or dossier
docChild	The child document

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_link_entry_30" )
   // (4)
   public int doSomething( D3Interface d3, Document docFather, Document
docChild ){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.

4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
5. An output is created in the log file using the log function.
6. The function is ended with the return value **0** .

hook_link_exit_10

```
int hook_link_exit_10(D3Interface d3, Document docFather, Document
docChild, Integer linkError, Integer errorCode)
```

Call time: Directly after executing the database command that enters the link.

Parameters	Description
docFather	The parent document or dossier
docChild	The child document
linkCode	0: Linking was successful -1: Parent and child element are identical or one of the two does not exist -2: The two documents are already linked -3: The two documents are already linked in reverse hierarchy -4: A database error occurred when entering the link in the database (see **errorCode**) Otherwise: Database error number when entering the link in the database
errorCode	0: OK Otherwise: Database or hook error

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_link_exit_10" )
   // (4)
   public int doSomething( D3Interface d3, Document docFather, Document
docChild, Integer errorCode ){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.

4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
5. An output is created in the log file using the log function.
6. The function is ended with the return value **0** .

Web-Publication

hook_webpublish_entry_10

```
int hook_webpublish_entry_10(D3Interface d3, Document doc, User user,
Integer publish)
```

Call time: Before or after a document is published online.

Parameters	Description
d3	the d.3 interface
doc	the document to be published or withdrawn
user	the executing user
publish	1: Document is published 0: Publication is withdrawn

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_webpublish_entry_10" )
   // (4)
   public int doSomething( D3Interface d3, Document doc, User user, Integer
publish ){
      // (5)
      d3.log.error("Hello world!");
      // (6)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.

5. An output is created in the log file using the log function.
6. The function is ended with the return value **0** .

hook_webpublish_entry_20

```
int hook_webpublish_entry_20(D3Interface d3, Document doc, User user,
DocumentType docType, Integer publish)
```

Call time: Before or after a document is published online.

Parameters	Description
d3	the d.3 interface
doc	the document to be published or withdrawn
user	the executing user
docType	The associated document type
publish	1: Document is published 0: Publication is withdrawn

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_webpublish_entry_20" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( D3Interface d3, Document doc, User user,
DocumentType docType, Integer publish ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the *groovyhook.jar* file.
2. Deploy a custom class of type *public*; *public* can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional *condition* annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called.
6. An output is created in the log file using the log function.

7. The function is ended with the return value **0** .

hook_webpublish_entry_30

```
int hook_webpublish_entry_30(D3Interface d3, Document doc, User user,
DocumentType docType, Integer publish)
```

Call time: Before or after a document is published online.

d3	the d.3 interface
doc	the document to be published or withdrawn
user	the executing user
docType	The associated document type
publish	1: Document is published 0: Publication is withdrawn

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_webpublish_entry_30" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( D3Interface d3, Document doc, User user,
DocumentType docType, Integer publish ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the *groovyhook.jar* file.
2. Deploy a custom class of type *public*; *public* can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional *condition* annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called.
6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_webpublish_exit_10

```
int hook_webpublish_exit_10(D3Interface d3, Document doc, User user,
Integer errorCode, DocumentType docType, Integer publish)
```

Call time: Before or after a document is published online.

Parameters	Description
d3	the d.3 interface
doc_id	The document that was published or withdrawn
user	the executing user
errorCode	0= Action successful Error code otherwise
docType	Document type short name
publish	1: Document has been published 0: Publication has been withdrawn

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_webpublish_exit_10" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( D3Interface d3, Document doc, User user, Integer
errorCode, DocumentType docType, Integer publish ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the *groovyhook.jar* file.
2. Deploy a custom class of type *public*; *public* can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional *condition* annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called.
6. An output is created in the log file using the log function.

7. The function is ended with the return value **0** .

hook_webpublish_exit_20

```
int hook_webpublish_exit_20(D3Interface d3, Document doc, User user,
Integer errorCode, DocumentType docType, Integer publish)
```

Call time: Before or after a document is published online.

Parameters	Description
d3	the d.3 interface
doc_id	The document that was published or withdrawn
user	the executing user
errorCode	0= Action successful Otherwise: Error code
docType	Document type short name
publish	1: Document has been published 0: Publication has been withdrawn

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_webpublish_exit_20" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( D3Interface d3, Document doc, User user, Integer
errorCode, DocumentType docType, Integer publish ){
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the *groovyhook.jar* file.
2. Deploy a custom class of type *public*; *public* can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional *condition* annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called.

6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

hook_webpublish_exit_30

```
int hook_webpublish_exit_30(D3Interface d3, Document doc, User user,
Integer errorCode, DocumentType docType, Integer publish)
```

Call time: Before or after a document is published online.

Parameters	Description
d3	the d.3 interface
doc_id	The document that was published or withdrawn
user	the executing user
errorCode	0= Action successful Otherwise: Error code
docType	Document type short name
publish	1: Document has been published 0: Publication has been withdrawn

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_webpublish_exit_30" )
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   public int doSomething( 3Interface d3, Document doc, User user, Integer
errorCode, DocumentType docType, Integer publish {
      // (6)
      d3.log.error("Hello world!");
      // (7)
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the *groovyhook.jar* file.
2. Deploy a custom class of type *public*; *public* can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional *condition* annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called.

6. An output is created in the log file using the log function.
7. The function is ended with the return value **0** .

Workflow

hook_workflow_cancel_exit_20

```
int hook_workflow_cancel_exit_20(D3Interface d3, Document doc, String
wflId, String stepId, User user)
```

Call time: After the workflow for a document has been cancelled. The cancellation of the workflow cannot be stopped here. This hook function is only activated if no error has previously occurred.

Parameters	Description
doc	The document whose workflow run was cancelled
wflId	ID of the workflow
stepId	ID of the workflow step
user	The executing d.3 user

```
// (1) Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

// (2)
public class D3Hooks{
   // (3)
   @Entrypoint( entrypoint = "hook_workflow_cancel_exit_20" )
   // (4
   public int doSomething( 3Interface d3, Document doc, String wflId,
String stepId, User user {
      // (5
      d3.log.error("Hello world!");
      // (6
      return 0;
   } // end of doSomething
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
5. An output is created in the log file using the log function.
6. The function is ended with the return value **0** .

1.5.7. Validation hooks

You can use validation hook functions for individual input validation (plausibility hooks).

- The function name (e.g. **ITValueValidation)** is entered for the property under **Plausibility check/Hook function**.
- The annotation is called: **@Validation(entry point="<identifier in d.3 administration>")**
- This function is only executed if a value was specified for the corresponding property in the client.
- This function is only executed after **hook_insert_entry_10**.
- Return value:
  - **0** on success
  - <> **0** in the event of an error

```
int myValidationHook(D3Interface d3, String value, Document doc)
```

Parameters	Description
d3	the d.3 interface
value	the property value to be checked
doc	The document to which the property belongs

```
@Validation(entrypoint="ITValueValidation")
int validateValue(D3Interface d3, String value, Document doc)
{
   if( value == "Peter" && doc.field["name"] == "Smith"){
      return 0;
   }
   else{
      return -8000;
   }
}// end of validateValue
```

Validation of an order number to a valid format

> **Note**
> Scenario:
>
> The order number should always match the format "two numbers-two letters-five numbers" (/[0-9]{2}-[a-zA-Z]{2}-[0-9]{5}/). This operation is suitable as an example to demonstrate the validation function.

Define a validation function for the **order number** document property.

```
//(1)
// Global d.3 libraries
import com.dvelop.d3.server.core.D3;

// Libraries to handle the diferent hook types
import com.dvelop.d3.server.Validation;
//(2)
public class D3Validate{
//(3)
   @Validation( entrypoint = "checkOrderNumber" )
//(4)
   public int checkOrderNumber( D3 d3, def currentValue, Document doc ){
//(5)
      def tmpValue  = currentValue;
      def matchFlag = ( tmpValue ==~ /[0-9]{2}-[a-zA-Z]{2}-[0-9]{5}/ );
//(6)
      return( matchFlag ? 0 : -1 );
```

```
   } // end of checkOrderNumber
}// end of D3Validate
```

The operation can be implemented in less time with Groovy.

```
//(1)
// Import the required d.3 classes
import com.dvelop.d3.server.core.D3;
import com.dvelop.d3.server.Validation;

 //(2)
 public class D3Validate {
   //(3)
   @Validation( entrypoint = "checkOrderNumber" )
   //(4)
   public int checkOrderNumber( D3 d3, def currentValue ) {
      //(6)
      return( ( currentValue ==~ /[0-9]{2}-[a-zA-Z]{2}-[0-9]{5}/ ) ? 0 : -1
);
   }// end of checkOrderNumber
}// end of D3Validate
```

Comments on the individual blocks

1. Import of the required classes
2. Creating an own class.
3. To use a Groovy method to validate a document property, the method is registered via the **@Validation** annotation for a validation function configured in d.3 admin.
4. The method then takes the parameters described above in the specified order.
5. You can now validate the transferred value within the method. In the example, it is verified using a regular expression.
6. If the value is valid, a **0** is returned. If the value is invalid, a **1** is returned.

1.5.8. Dataset hooks

You can use dataset hooks to generate dynamic datasets.

- The function name (e.g. **customerNumbers)** is entered for the property under **Value set/Hook function**.
- The annotation is called: **@ValueSet(entrypoint="<identifier in d.3 administration>")**
- A maximum of 10,000 values can be returned with a dataset hook.
- Sort order: With this, the preset order of values by the Groovy hook-function is maintained.

```
def myValueSetHook(D3Interface d3, RepositoryField repoField, User
user, DocumentType docType, Integer rowNo, Integer validate, Document
attribContext)
```

Parameters	Description
d3	the d.3 interface
repoField	the property field for which the dataset is defined the desired values can be passed via the **provideValuesForValueSet()** method
user	the executing user
docType	Document type containing the dataset
rowNo	row number for the multi-value properties
validate	Call for value validation (0/1)

Parameters	Description
attribContext	Document object with the attribute context. This object contains the other search criteria during the search. This object contains the other attributes already filled during import and update.

Simple datasets

Start with a static, simple dataset that is made available via the script.

Simple static dataset

```
// (1) Global d.3 libraries
-----------------------------------------------------
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the different hook types
-------------------------------
import com.dvelop.d3.server.ValueSet;

// Special libraries
-------------------------------------------------------------
import com.dvelop.d3.server.RepositoryField;

// (2)
class SimpleValueSet{
   // (3)
   @ValueSet( entrypoint = "customerNumbers" )
   // (4)
   def getCustomerNumber( D3Interface d3, RepositoryField reposField, User
user, DocumentType docType, int rowNo, int validate, Document doc ){

      // (5) Define static list of customer numbers
----------------------------
      def customerList = ["4711", "4712", "4713", "4714" ];

      // (6) Prepare List for interaction
-----------------------------------
      if( customerList.size() > 0 ){
         reposField.provideValuesForValueSet( customerList );
      }
   } // end of getCustomerNumber
}// end of SimpleValueSet
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.

5. An output is created in the log file using the log function.
6. The function is ended with the return value **0** .

Static datasets internal database

So that a dataset does not have to be maintained in two places, the data can be determined from the controlling system and made available as a selection list.

```
// (1) Global d.3 libraries
-----------------------------------------------------
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the different hook types
------------------------------
import com.dvelop.d3.server.ValueSet;

// Special libraries
------------------------------------------------------------
import com.dvelop.d3.server.RepositoryField;

// (2)
class StaticValueSet{
   // (3)
   @ValueSet( entrypoint = "customerNumbers" )
   // (4)
   def getCustomerNumber( D3Interface d3, RepositoryField reposField, User
user, DocumentType docType, int rowNo, int validate, Document doc ){

      // (5) Prepare sql statmenet
------------------------------------------------------------
      def sqlQuery = "SELECT customerNo FROM CustomerData ORDER BY
customerNo DESC"; // !! ATTENTION

      // (6) Execute sql statmenet
------------------------------------------------------------
      def resultRows = d3.sql.executeAndGet( (String) sqlQuery );

      // (7) Prepare list for user interface
-------------------------------------------------
      if( resultRows.size() > 0 ){
         reposField.provideValuesForValueSet( resultRows.collect{
it.customerNo } );
      }
   } // end of getCustomerNumber
}//end of StaticValueSet
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.

4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point.
5. Deployment of an SQL statement for determining the required values from the database.
6. Executing the SQL statement against the d.3 database table.
7. Deployment of values as a selection list for the user.

Static datasets external database

So that a dataset does not have to be maintained in two places, the data can be determined from the controlling system and made available as a selection list.

```
// (1) Global d.3 libraries
-----------------------------------------------------
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the different hook types
-------------------------------
import com.dvelop.d3.server.ValueSet;

// Special libraries
------------------------------------------------------------
import com.dvelop.d3.server.RepositoryField;

// (2)
class StaticValueSet{
   // (3)
   @ValueSet( entrypoint = "customerNumbers" )
   // (4)
   def getCustomerNumber( D3Interface d3, RepositoryField reposField, User
user, DocumentType docType, int rowNo, int validate, Document doc ){

      // (5) Prepare database Connection -----------------------------------
      def dbConnection = Sql.newInstance( "jdbc:sqlserver:<ServeRName>\
\<InstanceName>:0;databaseName=<DateBaseName>", "<DatabaseUser>",
„<Password>" );

      // (6) Prepare sql statmenet -----------------------------------------
      def sqlQuery  = "SELECT customerNo FROM CsutomerData ORDER BY
customerNo DESC"; // !! ATTENTION

      // (7) Execute sql statmenet -----------------------------------------
      def resultRows = dbConnection.rows( (String) sqlQuery );

      // (8) Prepare list for user interface -------------------------------
      if( resultRows.size() > 0 ){
         reposField.provideValuesForValueSet( resultRows.collect{
it.customerNo } );
      }
   } // end of getCustomerNumber
}// end of StaticValueSet
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point.
5. Structure of the JDBC connection to the external database. You must provide the applicable JDBC driver.
6. Deployment of an SQL statement for determining the required values from the database.
7. Executing the SQL statement against the d.3 database table.
8. Deployment of values as a selection list for the user.

Dynamic datasets

So that a dataset does not have to be maintained in two places, the data can be determined from the controlling system and made available as a selection list. It is then possible to consider search criteria dynamically.

We recommend creating a view on the controlling system's table so no user data has to be entered into scripts.

```
// (1) Global d.3 libraries
-------------------------------------------------
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the different hook types
-------------------------------
import com.dvelop.d3.server.ValueSet;

// Special libraries
-------------------------------------------------------------
import com.dvelop.d3.server.RepositoryField;

//-------------------------------------------------------------------------
---
// (2)
public class DynamicValueSet{
   // (3)
   @ValueSet(entrypoint = "customerNumbers" )
   // (4)
   def getCustomerNumber( D3Interface d3, RepositoryField reposField, User
user, DocumentType docType, int rowNo, int validate, Document doc ){

      // (5) Get needed values from the document properties
------------------
      def customerNo = doc.field[1];
      def zipCode    = doc.field[6];

      //(6) If needed filter on the customer no
------------------------------
      if( customerNo == null || ( customerNo != null && customerNo.size() <
3  )) {
```

```
         reposField.provideValuesForValueSet( "Please enter at least 3
characters!" );
      }

      // (7) Prepare the sql-statement with the needed params
-----------------
      def sqlQuery = """SELECT customerNo  + ' ' + name AS 'completeName'
                          FROM CustomerData WHERE 1 = 1 """;

      def sqlParams = [];

      sqlQuery += " AND customerNo LIKE ? OR name LIKE ?";

      sqlParams.add( customerNo + "%" ); // for the first questionmark
after customerNo
      sqlParams.add( customerNo + "%" ); // for the second questionmark
after name

      if( zipCode != null && zipCode != "" ){
         sqlQuery += " AND zipCode LIKE ?";
         sqlParams.add( zipCode + "%" );
      }

      // (8) Using external database
-----------------------------------------
      def dbConnection = Sql.newInstance( "jdbc:sqlserver:<ServerName>\
\<InstacneName>:0;databaseName=<DatabaseName>", "<DatabaseUser>",
„<Password>" );

      // (9) Using external database
-----------------------------------------
      def resultRows = dbConnection.rows( sqlQuery, sqlParams );

      // (10)
-----------------------------------------------------------------------
      if( resultRows.size() > 0 ){

reposField.provideValuesForValueSet( resultRows.collect{ it.completeName } )
;
      }
   } // end of getCustomerNumber
} // end of DynamicValueSet
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point.
5. Definition of the filter variables and transfer of the advanced properties.
6. The contents of the advanced properties are checked. Generation is canceled if content is missing or incomplete.

7. Composition of the SQL statement including dynamic assignment of the deployed filter criteria.
8. Structure of the JDBC connection to the external database. You must provide the applicable JDBC driver.
9. Executing the SQL statement against the d.3 database table.
10. Deployment of values as a selection list for the user.

Translating dynamic hook datasets

You can translate dynamic hook datasets.

- The annotation is: **@ValueSetTranslation(entrypoint="<identifier in d.3 administration>")**
- The entry point name matches the name of the corresponding @ValueSet hook, e.g. **MyMonths.**
- The translation hook must always return all values for the corresponding property field in the required language, not just the values that are relevant for the currently active user or context.
- The translations are cached by the d.3 server. The retention period of the values in the cache depends on the implementation and may change with future versions.
  You can enforce reloading of the translations at any time using the **d3.getArchive().removeTranslationFromCache()** function. You can call this function from any position not just from a dataset hook.
- Note: This interface is already prepared to support regional dialects. However, please note that d.3 currently does not yet fully support this feature.
- Only the memory value is transferred to the full text engine (d.search), not the translations. A full-text search by translations is therefore not supported.
- The combination of language and displayed value may only be used once per dataset. If this cannot be guaranteed, we recommend using the memory value as a prefix or suffix to the translation in order to ensure uniqueness.
- If there is not a translation for each value in the dataset, you must make sure that these untranslated values do not match the translations of other values in this dataset.

def myValueSetTranslation(D3Interface d3, Translation transl)

Parameters	Description
d3	The d.3 interface
transl	The translation object. The requested target language and the respective entry point are predefined in this object and must not be changed. Values can be entered using the **set** method.

Example hook

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Translation
import com.dvelop.d3.server.ValueSetTranslation


@ValueSetTranslation(entrypoint="MyMonths")
def myMonthsTranslation(D3Interface d3, Translation transl) {
    def lang = transl.locale.language

    if (lang == "de") {
        if (transl.locale.country == "AT")
            transl.set("01", "Jänner");
        else
            transl.set("01", "Januar");
        transl.set("02", "Februar");
        transl.set("03", "März");
```

```
            // ...
        } else if (lang == "th") {
            transl.set("01", "");
            transl.set("02", "");
            transl.set("03", "");
            // ...
        } else {
            transl.set("01", "January");
            transl.set("02", "February");
            transl.set("03", "March");
            // ...
        }
    }
```

Respective dynamic dataset function

Example hook

```
@ValueSet(entrypoint="MyMonths")
def myMonthsList(D3Interface d3, RepositoryField repoField, User user,
DocumentType docType, Integer row_no, Integer validate, Document
attribContext) {
   List<String> names = ["01", "02", "03" /*, ...*/];
   repoField.provideValuesForValueSet(names);

   boolean translationGotOutdated = false;
   if(translationGotOutdated){
      d3.getArchive().removeTranslationFromCache("MyMonths", Locale.GERMAN);
      d3.getArchive().removeTranslationFromCache("MyMonths", new
Locale("de", "AT"));
   }
}// end of myMonthsList
```

Further examples of dataset hooks can be found here.

1.5.9. Document class hooks

Document class hooks can be used to determine dynamic authorizations that cannot be mapped with d.3 document classes and restriction sets.

- To be specified in the administration area using: **@D3HOOK** ("Identifier for the hook")
- The annotation is: **@DocumentClass(entrypoint="<the identifier specified by @D3HOOK>")**

```
int myDocumentClassHook(D3Interface d3, String value, DocumentType docType,
String userId, Document doc)
```

Parameters	Description
d3	the d.3 interface
value	Value of the document property for which the hook function was called
docType	The document type of the document to be checked
userId	d.3 user ID of the executing user
doc	The document to be checked

Return value:

1: eligible

0: no access

Example hook

```
// (1) Global d.3 libraries
-------------------------------------------------------
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Condition;

// Libraries to handle the different hook types
------------------------------
import com.dvelop.d3.server.DocumentClass;
// (2)
public class D3DocumentClassHook{
   // (3)
   @DocumentClass(entrypoint="myDocumentClassHook")
   // (4)
   @Condition( doctype = ["XXXX"] )
   // (5)
   def myDocumentClassHook(D3Interface d3, String value, DocumentType
docType, String userId, Document doc){
      if (value > 10000 && docType.id == "DINV"){
         if (value <= 20000 && doc.owner == "Meyer"){
            return 1;
         }
         else if (value > 20000 && doc.owner == "Smith"){
            return 1;
         }
         else {
            return 0;
         }
      }
      else
         return 1;
      }
   }// end of myDocumentClassHook
}// end of D3DocumentClassHook
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point.

> **Note**
> Document class hooks are called for each individual document in the result set during a search.
>
> This can lead to performance problems when validating rights and therefore slow down the document search, especially if SQL commands are issued in the document class hook.
>
> There is an even greater impact on performance if document class hooks are used for multi-value property fields (60 fields), as these are then also called for each line of each multi-value property that has a value.

> **Warning**
> Document class hooks are executed in parallel in several threads during the document search. These hook functions may therefore only contain and call thread-safe code.

1.6. Groovy hook examples

1.6.1. Entry points

> **Note**
> These libraries must be imported in order to use entry points:
>
> **Global d.3 libraries**
>
> - import com.dvelop.d3.server.core.D3Interface
> - import com.dvelop.d3.server.Document
> - importcom.dvelop.d3.server.User
> - import com.dvelop.d3.server.DocumentType
>
> **Specific libraries for the entry points**
>
> - import com.dvelop.d3.server.Entrypoint
> - import com.dvelop.d3.server.Condition

> **Note**
> You can define any number of functions for any hook entry point. You can therefore use the entry point **hook_insert_entry_10** to check the order number of an invoice, check the personnel number of a leave request or complete the data of an order.

The different entry types require a defined number of parameters in a predefined sequence. The d.3 object is always the first parameter transferred when a Groovy hook function is called.

> **Note**
> For the hook entry points **datasets**, **validation** and **search**, an additional parameter of type **document** must always be appended at the end.

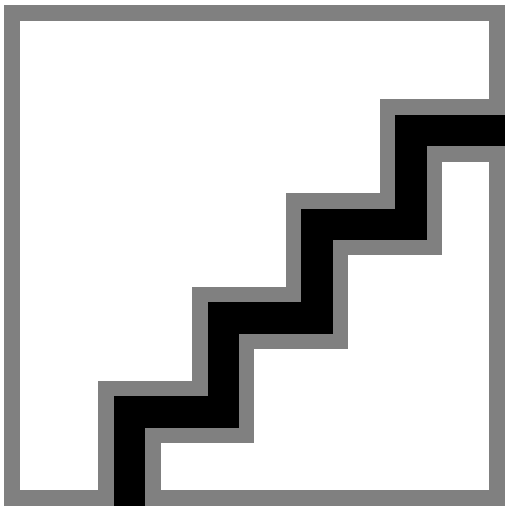
A chain of entry points

If, for example, a document is stored via directory monitoring (host import), the following chain of entry points is processed on the server side:

- hook_hostimp_entry_10 →hook_insert_entry_10 → hook_insert_entry_20 → hook_insert_exit_10 → hook_insert_exit_20 → hook_insert_exit_30

If, for example, a document is stored via manual import, the following chain of entry points is processed on the server side:

- hook_validate_import_entry_10 → hook_insert_entry_10 → hook_insert_entry_20 → hook_insert_exit_10 → hook_insert_exit_20 → hook_insert_exit_30



If a function is terminated at an entry point with an error or a value not equal to **0** or returns a value not equal to **0**, the entire chain is interrupted and the document is not archived, for example.

InsertEntry_10

Hello World!

> **Note**
> Scenario:
>
> When a document is stored in the d.3 system, only the error message "Hello World" should be displayed in the d.3 log file.

A hook function is saved for the entry point " **hook_insert_entry_10**", which checks the customer number.

Example

```
package com.dvelop.hooks;

//(1)
// Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;

//(2)
public class D3Hooks{
```

```
//(3)
   @Entrypoint( entrypoint = "hook_insert_entry_10" )
//-----------------------
//(4)
   public int insertEntry_10( D3Interface d3, User user, DocumentType
docTypeShort, Document doc ){
//(5)
      d3.log.error("Hello world!");
//(6)
      return 0;
   } // end of insertEntry_10
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. In addition, a parameter of type **Document** is required as the last parameter.
5. An output is now generated in the log file using a simple log function.
6. The function is finished with a return value **0**.

Checking and adding document properties during import

> **Note**
> Scenario:
>
> When a document is stored in the d.3 system, the **customer number** property is to be checked against a database and the remaining customer data automatically added if necessary.
>
> The required customer data in this example is stored in a database table within the d.3 database, so the data can be retrieved via a simple SQL implementation.

A hook function is saved for the entry point " **hook_insert_entry_10**", which checks the customer number.

Example

```
package com.dvelop.hooks;

//(1)
// Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the diferent hook types
import com.dvelop.d3.server.Entrypoint;

//(2)
public class D3Hooks{
```

```
//(3)
@Entrypoint( entrypoint = "hook_insert_entry_10" )
//(4)
public int insertEntry_10( D3Interface d3, User user, DocumentType
docTypeShort, Document doc ){
//(5)
        if( docTypeShort.id == "DRECH" ){
          def customerID = doc.field[14];
//(6)
         if( customerID != "" ){
             def sqlQuery   = "SELECT name, street, zipCode, City FROM
CustomerData WHERE customerNo = ? ";
             def sqlParams  = [ customerID ];
             def resultRows = d3.sql.executeAndGet( sqlQuery, sqlParams );
//(7)
            if( resultRows.size() == 1 ){
               doc.field["Strasse"] = resultRows[0].street;
               doc.field["PLZ"]     = resultRows[0].zipCode.toString(); //
ATTENTION!
               doc.field[10]        = resultRows[0].city;
               return  0;
             }
(8)
            else {
               return  -1;
             }
          }
      }
       return  0;
    } // end of insertEntry_10
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. In addition, a parameter of type **Document** is required as the last parameter.
5. Validation can now be carried out within the function for a specific document type; the document type is restricted via the **pDocTypeShort.id** property. In this example, the restriction is made using an IF statement, but you can also restrict the validation to one or more document types using **Condition**.
6. Once the customer number has been determined from the document properties, the customer number can be checked against the database. If the database table is in the d.3 DB address space, the native database interface of the d.3 server can be used by means of the d.3 SQL implementation.
7. As all document properties are available for both reading and writing in the context of the entry point **hook_insert_entry_10**, the additional customer data can now be added. The properties can be referenced via the database position or directly with a descriptive property name. Decide on one type of referencing.
8. If no valid customer number could be determined, a return value not equal to **0** is returned and the processing chain is interrupted at this point.

InsertExit_20

Entry in the document notes

> **Note**
> Scenario:
>
> If an invoice has been successfully imported, an entry can be automatically written in the document notes.
>
> Details on the Groovy server API functions used here can be found in the specific documentation.

For this purpose, a hook function is stored for the entry point **hook_insert_exit_20**; the entry is then written by the Groovy server API.

Example

```
package com.dvelop.hooks;

//(1)
// Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the diferent hook types
import com.dvelop.d3.server.Entrypoint;
//(2)
public class D3Hooks{
//(3)
   @Entrypoint( entrypoint = "hook_insert_exit_20" )
//(4)
   @Condition( doctype = "DRECH" )
//(5)
   public int insertExit_20( D3Interface d3, Document doc, def fileDest,
def importOK,
                             User user, DocumentType docTypeShort ){
//(6)
       def returnValue = 0;
       returnValue = d3.call.note_add_string( "Hello World!", doc.id,
user.id );
//(7)
      return  0;
   } // end of insertExit_20
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.

5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called.
6. Now you can use the Groovy server API function **note_add_string** to add an entry to the document note.
7. If the action was successful, a value of **0** is returned.

Forwarding to an accounting clerk

> **Note**
> Scenario:
>
> Once an invoice has been successfully imported, it can be added directly to a group or a user's inbox.

For this purpose, a hook function is stored for the entry point " **hook_insert_exit_20**" and the document is placed in a user's inbox via a Groovy server function.

Example

```
package com.dvelop.hooks;

//(1)
// Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;
import com.dvelop.d3.server.Condition;

// Special libraries
import groovy.sql.Sql;
import java.sql.Timestamp;

//(2)
public class D3Hooks{
//(3)
   @Entrypoint( entrypoint = "hook_insert_exit_20" )
//(4)
   @Condition( doctype = "DRECH" )
//(5)
   public int insertExit_20( D3Interface d3, Document doc, def  fileDest,
def  importOK,
                             User user, DocumentType docTypeShort ){
      // Define variables ---------------------------------------------
      def       returnValue = 0;
      Timestamp currentDate = new Date().toTimestamp();
//(6)
      returnValue = d3.call.hold_file_send( user.id, "Invoice: " +
doc.getField("InvoiceNo"),
                                            doc.id, currentDate,
currentDate, false , true ,
                                            currentDate, "", "dvelop", 0,
false , false , currentDate, 0, false );
```

```
//(7)
      return 0;
   } // end of insertExit_20
} // end of D3Hooks
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called.
6. Now you can use the Groovy server API function **note_add_string** to add an entry to the document note.
7. If the action was successful, a value of **0** is returned.

Updating other documents depending on the current one

> **Note**
> Scenario:
>
> A dossier is created in the d.3 system and data from the dossier is to be transferred to other documents.

For this purpose, a hook function is stored for the entry point **hook_insert_exit_20** and the document is placed in a user's inbox via a Groovy server function.

Updating other documents depending on the import of a new document

```
//(1)
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.core.D3Interface.ArchiveInterface

import com.dvelop.d3.server.Document
import com.dvelop.d3.server.DocumentType
import com.dvelop.d3.server.User

import com.dvelop.d3.server.Condition
import com.dvelop.d3.server.Entrypoint

//------------------------------------------------------------------------
------------------
/**
*   Add function to entry point hook_insert_exit_30 for handling change in
personal data
*
*   @param   d3, doc, fileDest, importOK, user, docType --> Server definied
*   @return  integer as result value
*/
//(2)
@Entrypoint(entrypoint="hook_insert_exit_30")
@Condition( doctype =  ["PERSA"] )
public int entryUpdatePersonalData( D3Interface d3, Document docObj, def
```

```
fileDest, def importOK, User user, DocumentType docType ) {
   int retValue;
//(3)
   retValue =  updatePersData( d3, doc, user );
   return retValue;
} // end of entryUpdatePersonalData

//------------------------------------------------------------------------
------------------
/**
*   Function for updating function
*
*   @param   d3      D3Interface object
*   @param   doc     Document object
*   @param   login   current user object
*   @return  integer as result value
*/
//(4)
public int updatePersData( D3Interface d3, Document doc, User login ) {

   Document docObjRef;
//(5)
   ArchiveInterface archiveObj = d3.getArchive();

//(6)
   def currentDocId = doc.id;
   def client       = doc.field[DDF.MANDANT];    // DDF... = Database
position in ext. class
   def persNumber   = doc.field[DDF.EMPLOYEE_NO];
   def persName     = doc.field[DDF.EMPLOYEE_NAME];
//(7)
   def sqlQuery   = "SELECT .... ";
   def resultRows = d3.sql.executeAndGet( sqlQuery );
   def childDocObj;
//(8)
   resultRows.each {
      childRef = it.doku_id;
//(9)
      childDocObj = archiveObj.getDocument( childRef, login.id );
//(10)
      docObjRef.field[DDF.persName] = persName;
//(11)
      docObjRef.updateAttributes( login.id, true ); // !!!!
   }

   return  0;
} // end of updatePersDataddd
```

Comments on the individual blocks

1. Import required libraries from the **groovyhook.jar** file.
2. Deploy a function on the entry point **hook_insert_exit_20**.
3. Call the separate function for updating the personnel data.
4. Define the update function.
5. Deploy an **ArchivInterface** object for the purpose of generating the document objects.

6. The required data is read from the current document; an external class for defining global constants was used here as an example.
7. At this point, a search for further document IDs for the current personnel number could now be determined using the properties provided; this can be done using an SQL statement.
8. All recognized documents are then processed in a loop.
9. A document object is created for each document ID.
10. The new property is assigned.
11. The data is then updated; here the second parameter is very important, as when it is set to **true**, it ensures that downstream hook validation does not take place, while **false** has the opposite effect.

UpdateAttribEntry_20

Checking and adding document properties during the update

> **Note**
> Scenario:
>
> If the document property **customer number** is updated, the data should also be checked against a database and, if necessary, the remaining customer data should be added automatically. The required customer data in this example is stored in an external database and must be retrieved via an external database connection.

A hook function is saved for the entry point **hook_upd_attrib_entry_20**, which checks the customer number.

Example

```
package com.dvelop.hooks;

//(1)
// Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the different hook types
import com.dvelop.d3.server.Entrypoint;
import com.dvelop.d3.server.Condition;

// Special libraries
import groovy.sql.Sql;

//(2)
public class D3Hooks{
//(3)
   @Entrypoint( entrypoint = "hook_upd_attrib_entry_20" )
   @Condition( doctype = "DRECH" )
//(4)
   public int updateAttributeEntry_20( D3Interface d3, Document doc, User
user, DocumentType docTypeShort,
                                       DocumentType docTypeShortNew  ){
//(5)
      return getCustomerData( d3, doc );
   } // end of updateAttributeEntry_20
   public int getCustomerData( D3 d3, Document doc ){
```

```
//(6)
      def customerID = doc.field[14];
      if( customerID != "" ){
         def dbConnection = Sql.newInstance( "jdbc:sqlserver://
<ServerName>:<Port>;databaseName=<DataBaseName>",
                                             "<User>", "<Password>" );
         def sqlQuery     = "SELECT name, street, zioCode, City FROM
CustomerData WHERE customerNo = ? ";
         def sqlParams    = [ customerID ];
         def resultRows   = dbConnection.rows( sqlQuery, sqlParams );
         if( resultRows.size() == 1 ){
//(7)
            doc.field[8]  = resultRows[0].street;
            doc.field[9]  = resultRows[0].zipCode.toString(); // ATTENTION!
            doc.field[10] = resultRows[0].city;
            return 0;
         }
         else {
//(8)
            return -1;
         }
      }
      return  0;
   } // end of getCustomerData
} // end of D3Hooks
```

Comments on the individual blocks

<listitem>
Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
</listitem>
<listitem>
To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
</listitem>
<listitem>
Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
</listitem>
<listitem>
The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called.
</listitem>

1. Import the required libraries from the file **groovyhook.jar**.
2. For the hook entry points **datasets**, **validation** and **search**, an additional parameter of type **document** must always be appended at the end.
3. Validation can now be carried out within the function for a specific document type; the document type is restricted via the **pDocTypeShort.id** property.
4. Once the customer number has been determined from the document properties, the customer number can be checked against the database. In this example, a connection is made to an external Microsoft SQL database and the data is determined from this database. Details on the necessary configuration can be found in the relevant documentation.
5. As all document properties are available for both reading and writing in the context of the entry point **hook_upd_attrib_entry_20**, the additional customer data can now be added.

6. If no valid customer number could be determined, a return value not equal to **0** is returned and the processing chain is interrupted at this point.

Several hook functions in one class

How can several hook functions be combined in one class? The following example script provides one possible answer.

```
// (1) Global d.3 libraries
-------------------------------------------------
import java.sql.Timestamp;
import com.dvelop.d3.server.Condition;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.DocumentType;
import com.dvelop.d3.server.Entrypoint;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.core.D3Interface;
import groovy.sql.Sql;

//-------------------------------------------------------------------------
---
// (2)
public class D3EntryPoints{
   // (3)
   @Entrypoint( entrypoint =
"hook_insert_entry_10" ) //----------------------
   // (4)
   public int getCustomerDataForInvoice( D3Interface d3, User user,
DocumentType docTypeShort, Document doc ){
   // (5)
      if( docTypeShort.id == "DINV" ){
         d3.log.error("Hello world!");
         //return getCustomerData( d3, doc );
      }
      return 0;
   } // end of getCustomerDataForInvoice
   // (6)
   @Entrypoint( entrypoint =
"hook_insert_exit_20" ) //-----------------------
   @Condition( doctype = [ "DINV" ] )
   public int sendInvoice( D3Interface d3, Document doc, def fileDest, def
importOK, User user, DocumentType docTypeShort ){
      // Define variables ---------------------------------------
      def returnValue = 0;
      Timestamp currentDate = new Date().toTimestamp();
      // Add entry to document note
----------------------------------------
      returnValue = d3.call.note_add_string( "Hello World!", doc.id,
user.id );
      // SEND to user
-----------------------------------------------------
      returnValue = d3.call.hold_file_send( user.id, "Invoice: " +
doc.field[14], doc.id, currentDate, currentDate, false, true, currentDate,
"", "dvelop", 0, false, false, currentDate, 0, false);
   } // end of  sendInvoice

   // (7)
```

```
   @Entrypoint( entrypoint =
"hook_validate_import_entry_10" ) //--------------
   @Condition( doctype = [ "DINV" ])
   public int justDummy( D3Interface d3, User user, DocumentType
docTypeShort, Document doc ){
      return 0;
   } // end of  justDummy

   // (8)
   @Entrypoint( entrypoint =
"hook_upd_attrib_entry_20" ) //-------------------
   public int updateCustomerDataForInvoice( D3Interface d3, Document doc,
User user, DocumentType docTypeShort, DocumentType docTypeShortNew  ){
      if( docTypeShort.id == "DRECH" ){
         def oldDocOnj = d3.archive.getDocument( doc.id ); // TODO alte
inhalte
         return getCustomerData( d3, doc );
      }
      return 0;
   } // end of updateCustomerDataForInvoice

   // (9)
-----------------------------------------------------------------------
   public int getCustomerData( D3Interface d3, Document doc ){

      def customerID = doc.field[1];
      if( customerID != null && customerID != "" ){
         def dbConnection = Sql.newInstance( "jdbc:sqlserver:<ServerName>\
\<InstanceName>:0;databaseName=<DataBaseName>", "<DatabaseUser>",
"<Password>" );
         def sqlQuery   = "SELECT name, street, zipCode, city FROM
CustomerData WHERE customerNo = ? ";
         def sqlParams  = [ customerID ];
         def resultRows = dbConnection.rows( sqlQuery, sqlParams );
         //def resultRows = d3.sql.executeAndGet( sqlQuery, sqlParams );

         if( resultRows.size() == 1 ){

            doc.field["Straße"]       = resultRows[0].street.trim();
            doc.field["Postleitzahl"] =
resultRows[0].zipCode.toString(); // ATTENTION!
            doc.field[5]              = resultRows[0].city.trim();
            doc.field[1]              = resultRows[0].name.trim();

            doc.setText( 1, "Content from Groovy" );
            return 0;
         }
         else{
            return -1;
         }
      }
      return 0;
   } // end of getCustomerData
} // end of d3Hooks
```

Comments on the individual blocks

<listitem>
Import of the required libraries from the **groovyhook.jar** file.
</listitem>
<listitem>
Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
</listitem>
<listitem>
To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
</listitem>
<listitem>
The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
</listitem>

1. In this example, the document type is checked within the function using an "if" statement and a corresponding action is executed for the **invoice** document type.
2. Another example is implemented for the entry point **InsertExit_20** and the invoice document type **DRECH**. In this example, an entry is made in the document notes via the server API calls **note_add_string** when an invoice is received. In addition, the invoice is sent to an accounting clerk using the **hold_file_send** command.
3. A function is registered here for a further entry point **hook_validate_import_entry_10**, which does not yet perform any task.
4. If, for example, the customer number is changed in an invoice, the relevant customer data in the invoice must also be adjusted; for this purpose, the entry point **hook_upd_attrib_entry_20** is linked to a function. As customer data needs to be determined in several places, the function was implemented at this point.

1.6.2. Validation

> **Warning**
> These libraries must be imported in order to use validation:
>
> **Global d.3 libraries**
>
> - import com.dvelop.d3.server.core.D3Interface
>
> **Specific libraries for validation**
>
> - import com.dvelop.d3.server.Validate

Useful links

- Regular expression online tool: https://regex101.com/

Validation of an order number to a valid format

> **Note**
> Scenario:
>
> The order number should always comply with the format "two numbers-two letters-five numbers" (/[0-9]{2}-[a-zA-Z]{2}-[0-9]{5}/). Of course, you can configure this directly in the d.3 administration area, but it also serves as an example to demonstrate how validation works.

To implement this, a validation function is defined for the "order number" document property.

```
package com.dvelop.hooks;

//(1)
//Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;

// Libraries to handle the diferent hook types
import com.dvelop.d3.server.Validation;
//(2)
public class D3Validate{
//(3)
   @Validation( entrypoint = "checkOrderNumber" )
//(4)
   public int checkOrderNumber( D3Interface d3, def currentValue, Document
doc ){
//(5)
      def tmpValue  = currentValue;
      def matchFlag = ( tmpValue ==~ /[0-9]{2}-[a-zA-Z]{2}-[0-9]{5}/ );
//(6)
      return( matchFlag ? 0 : -1 );
   } // end of checkOrderNumber
}// end of D3Validate
```

You can create the same function even more compactly in Groovy.

```
package com.dvelop.hooks;

//(1)
// Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;

// Libraries to handle the diferent hook types
import com.dvelop.d3.server.Validation;
//(2)
 public class D3Validate{
//(3)
     @Validation( entrypoint = "checkOrderNumber" )
//(4)
   public int checkOrderNumber( D3Interface d3, def currentValue ){
//(6)
      return( ( currentValue ==~ /[0-9]{2}-[a-zA-Z]{2}-[0-9]{5}/ ) ? 0 : -1
);
   } // end of checkOderNumber
}// end of D3Validate
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. Optionally, you can assign the functions to specific document classes with an additional **condition** annotation.
5. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called.

The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.

6. The passed value can now be validated within the function, in the example by using a regular expression.
7. If the value corresponds to a valid value, **0** is returned, otherwise **1** is returned.

1.6.3. Datasets

This area shows the options for providing datasets as follows:

- As static list
- From internal d.3 database tables
- From external database tables
- As dynamic dependent database table
- As a translated dataset

Required libraries for the datasets

> **Warning**
> These libraries must be imported for the implementation of datasets:
>
> **Global d.3 libraries**
>
> - import com.dvelop.d3.server.core.D3Interface
> - import com.dvelop.d3.server.Document
> - Import com.dvelop.d3.server.User
> - import com.dvelop.d3.server.DocumentType
>
> **Specific libraries for the datasets**
>
> - import com.dvelop.d3.server.ValueSet
> - import com.dvelop.d3.server.RepositoryField

> **Warning**
> The sorting is taken from the script and not falsified by the client, as is the case with a JPL dataset.

Simple static dataset

> **Note**
> It does not make sense to deploy a static list via a script. The better way is using d.3 administration. But for a simple example, you can start here with a static dataset.

Simple static dataset

```
//(1) Global d.3 libraries
-----------------------------------------------------
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;

//(2) Libraries to handle the different hook types
----------------------------
```

```
import com.dvelop.d3.server.ValueSet;

//(3) Special libraries
-------------------------------------------------------
import com.dvelop.d3.server.RepositoryField;

//(4) Define the needed class
-------------------------------------------------
class SimpleValueSet{
   //(5) Combine to the value set entry point
---------------------------------
   @ValueSet( entrypoint = "customerNumbers" )
   //(6) Define the function
-------------------------------------------------
   def getCustomerNumber( D3Interface d3, RepositoryField reposField, User
user, DocumentType docType,
                          int rowNo, int validate, Document doc ){

      //(7) Define static list of customer numbers
-----------------------------
      def customerList = ["4711", "4712", "4713", "4714" ];

      //(8) Prepare List for
interaction---------------------------------------
      if( customerList.size() > 0 ){
         reposField.provideValuesForValueSet( customerList );
      }
   } // end of getCustomerNumber
}// end of SimpleValueSet
```

Comments on the individual blocks

1. Import required libraries from the **groovyhook.jar** file.
2. Import of the special library for supporting the datasets.
3. Import of a special library to return the dataset to the user.
4. Definition of a public class.
5. Registration of the entry point for the dataset.
6. Definition of the special function for determining the dataset.
7. Definition of the static dataset.
8. Deployment of the dataset for the user.

Simple static dataset from a database table

It makes more sense to determine the datasets from internal or external data sources; this is presented in these examples.

Internal d.3 database table

> **Note**
> In this example, the data is determined from an internal d.3 database table; any user input is not taken into account.

```
//(1) Global d.3 libraries
-------------------------------------------------------
Import com.dvelop.d3.server.core.D3Interface;
```

```
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;

//(2) Libraries to handle the different hook types
------------------------------
import com.dvelop.d3.server.ValueSet;

//(3) Special libraries
-------------------------------------------------------
Import com.dvelop.d3.server.RepositoryField;

//(4) Example for an internal d.3 database table
--------------------------------
class StaticValueSet{
   //(5) Define the value set entry point
--------------------------------------
   @ValueSet( entrypoint = "customerNumbers" )
   //(6) Define function for the value set
--------------------------------------
   def getCustomerNumber( D3Interface d3, RepositoryField reposField, User
user, DocumentType docType,
                          int rowNo, int validate, Document doc ){

      //(7) Prepare sql statmenet ---------------------------------------
      def sqlQuery = "SELECT customerNo FROM CustomerData ORDER BY
customerNo DESC"; // !! ATTENTION

      //(8) Execute sql statmenet ---------------------------------------
      def resultRows = d3.sql.executeAndGet( (String) sqlQuery );

      //(9) Prepare list for user interface -----------------------------
      if( resultRows.size() > 0 ){
         reposField.provideValuesForValueSet( resultRows.collect{
it.kunden_nr } );
      }
   } // end of getCustomerNumber
}// end of StaticValueSet
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Import of the special library for supporting the datasets.
3. Import of a special library to return the dataset to the user.
4. Definition of a public class.
5. Registration of the entry point for the dataset.
6. Definition of the special function for determining the dataset.
7. Definition of the SQL statement for determining the dataset.
8. Execution of the SQL statement against the internal database table, realized here via the d.3 object.
9. Display of the dataset for the user.

External database table

> **Note**
> Datasets can also be determined from external databases; this example shows a possible approach.

```
//(1) Global d.3 libraries
-------------------------------------------------------
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;

//(2) Libraries to handle the different hook types
------------------------------
import com.dvelop.d3.server.ValueSet;

//(3) Special libraries
-------------------------------------------------------------
import com.dvelop.d3.server.RepositoryField;
import groovy.sql.Sql;

//(4) Example for an external database table
-----------------------------------
class StaticValueSet{
   //(5) Define the value set entry point
---------------------------------------
   @ValueSet( entrypoint = "customerNumbers" )
   //(6) Define the function
-------------------------------------------------------
   def getCustomerNumber( D3Interface d3, RepositoryField reposField, User
user, DocumentType docType,
                          int rowNo, int validate, Document doc ){

      //(7) Prepare database Connection
-----------------------------------------
      def dbConnection = Sql.newInstance( "jdbc:sqlserver:<Server>\
\<instacneName>:0;databaseName=<DataBaseName>", "<UserName>", „<Password>"
);

      //(8) Prepare sql
statmenet-----------------------------------------------
      def sqlQuery  = "SELECT customerNo FROM CustomerData ORDER BY
customerNo DESC"; // !! ATTENTION

      //(9) Execute sql statmenet
-----------------------------------------------
      def resultRows = dbConnection.rows( (String) sqlQuery );

      //(10) Prepare lit for user interface
-----------------------------------
      if( resultRows.size() > 0 ){
         reposField.provideValuesForValueSet( resultRows.collect{
it.kunden_nr } );
```

```
      }

   } // end of getCustomerNumber
}// end of StaticValueSet
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Import of the special library for supporting the datasets.
3. Import of a special library to return the dataset to the user.
4. Definition of a public class.
5. Registration of the entry point for the dataset.
6. Definition of the special function for determining the dataset.
7. Initialization of an external JDBC database connection.
8. Definition of the SQL statement for determining the dataset.
9. Execution of the SQL statement using the internal database table, realized here via the d.3 object.
10. Display of the dataset for the user.

Dependent dynamic dataset from a database table

Deployment of a customer number list

> **Note**
> Scenario:
>
> A dataset of customer numbers from the customer database table is to be deployed for the document property **CustomerNo**. If required, this property should be dynamically limited based on the entered ZIP code range.

To implement this, a validation function is defined for the **order number** document property.

Example

```
//(1) Global d.3 libraries
-----------------------------------------------------
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the different hook types
Import com.dvelop.d3.server.ValueSet;

// Special libraries
Import com.dvelop.d3.server.RepositoryField;
//(2)
public class D3ValueSets{
   //(3)
   @ValueSet( entrypoint = "dsCustomerNumbers" )
   //(4)
   public int getCustomerNumber( D3Interface d3, RepositoryField
reposField, User user, DocumentType docType, int  rowNo, int  validate,
Document doc )
   {
      //(5)
      def  zipCode     = doc.field[9];
```

```
      def  whereClause = "";
      if( plz != "" ){
         whereClause = "WHERE zipCode LIKE '$zipCode%'";
      }
      //(6)
      def sqlQuery   = "SELECT customerNo FROM CustomerData $whereClause
ORDER BY customerNo DESC"; // !! ATTENTION
      def resultRows = d3.sql.executeAndGet( sqlQuery );
      //(7)
      // Long-Version - Collecting a list and provide list in 2 steps
-------------------------------------------------------
      def customerList = [];
      resultRows.each{
         customerList.add( it.kunden_nr );
      }
      if( customerList.size() > 0  ){
         reposField.provideValuesForValueSet( customerList );
      }

      // Short-Version Collecting a list and provide list in 2 steps
--------------------------------------------------------
      if( resultRows.size() > 0 ){
         reposField.provideValuesForValueSet( resultRows.collect{
it.kunden_nr  } );
      }

      return 0
   } // end of getCustomerNumber
} // end of D3ValueSets
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
5. To implement the task, the document property of the ZIP code is read and, if required, a **Where** component of the SQL statement is made available.
6. As the database table used was created within the d.3 database, the data can be accessed using a simple SQL implementation; see also the example for **InsertEntry_10**.
7. If data was found for the provided ZIP code range, this data can be deployed using the **provideValuesForValueSet()** function of the document property. Two different implementation methods have been included here. Only use one of the two methods.

Deployment of translated dataset

Scenario

Note

A dataset from the table **custom_articles** is to be made available. The article number is to be saved in the d.velop documents database. Depending on the language set in the browser, the articles are to be translated into English or German.

Caution

For translated datasets, care must be taken to ensure that the translations are unique in each case.

The values stored in the database are saved in the full-text database. In this example, these are the article numbers. This means that it is not possible to search for the article names (German/English) by full text.

Example

```
//(1) Global d.3 libraries
-----------------------------------------------------
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the different hook types
Import com.dvelop.d3.server.ValueSet;

// Special libraries
Import com.dvelop.d3.server.RepositoryField;
//(2)
public class D3ValueSets{
    def gArticleListTranslated = false
    //(3)
    @ValueSet(entrypoint="articleList")
    //(4)
    def getArticleList(D3Interface d3, RepositoryField repoField, User
user, DocumentType docType, Integer row_no, Integer validate, Document doc)
{
        //(6)
        String sqlQuery = "select articleNumber as id, articleName as value
from custom_articles order by value"
        def resultRows = d3.sql.executeAndGet( sqlQuery );
        //(7)
        repoField.provideValuesForValueSet( resultRows.collect{
it.id.toString() } );
        //translate value list if row count has changed
        if (gArticleListTranslated != resultRows.size())
        {
            d3.log.info("groovyHook - articleList -> translate")
            //(8)

d3.getArchive().removeTranslationFromCache("gesellschaftenList", new
Locale("de"));
            //(9)

d3.getArchive().removeTranslationFromCache("gesellschaftenList", new
```

```
Locale("en"));
            gArticleListTranslated = resultRows.size();
        } else {
            d3.log.info("groovyHook - articleList -> from cache")
        }
    }
    //10
    @ValueSetTranslation(entrypoint="articleList")
    def translateArticleList(D3Interface d3, Translation transl) {
       def sqlQuery
       //11
       if(transl.locale.language == "en"){
          sqlQuery = "select articleNumber as id, articleNameDE as value
from custom_articles"
       } else if(transl.locale.language == "en"){
          sqlQuery = "select articleNumber as id, articleNameEN as value
from custom_articles"
       } else {
          d3.log.info("no translation for language " +
transl.locale.language + " -> english translation")
          sqlQuery = "select articleNumber as id, articleNameEN as value
from custom_articles"
       }
       def resultRows = d3.sql.executeAndGet( sqlQuery );
       resultRows.each {
          //12
          transl.set(it.id.toString(), it.value);
       }
    }

} // end of D3ValueSets
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
5. To implement the task, the document property of the ZIP code is read and, if required, a **Where** component of the SQL statement is made available.
6. As the database table used was created within the d.3 database, the data can be accessed using a simple SQL implementation; see also the example for **InsertEntry_10**.
7. The values are provided to the document property via the **provideValuesForValueSet()** function.
8. The translation for **de** for German is triggered.
9. The translation for **en** for English is triggered.
10. In order to be able to use a Groovy function to translate a dataset, a Groovy annotation with the specified values is used to register the function to a dataset function, which is configured in d.3 admin.
11. An SQL statement to determine the values for the translation is created depending on the transferred language.

12. The **transl.set()** function is used to deliver the translation (value) for each value (id).

1.6.4. Document classes

> **Warning**
> These libraries must be imported for the implementation of datasets:
>
> **Global d.3 libraries**
>
> - import com.dvelop.d3.server.core.D3Interface
> - import com.dvelop.d3.server.Document
> - Import com.dvelop.d3.server.User
> - import com.dvelop.d3.server.DocumentType
>
> **Specific libraries for the document class hook**
>
> - import com.dvelop.d3.server.DocumentClass

Invoices can only be edited by certain users

> **Note**
> Scenario:
>
> Invoices should now only be able to be processed by certain users, depending on the third digit of the customer numbers. As there is no corresponding data link between the user and the customer number in the demo environment, the implementation is realized here using a static assignment via a "switch" statement, as an example.

To implement this, a new document class **CustomerInvoices** is created in the d.3 administration; the required implementation is documented in the following example.

Example

```
package com.dvelop.hooks;

//(1)
// Global d.3 libraries
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.Document;
import com.dvelop.d3.server.User;
import com.dvelop.d3.server.DocumentType;

// Libraries to handle the different hook types
import com.dvelop.d3.server.DocumentClass;
//(2)
public class D3DocumentClass{
//(3)
   @DocumentClass( entrypoint = "CustomerInvoice" )
//(4)
   pudblic int customerDocClass( D3Interface d3, String value, DocumentType
docType, String userId, Document doc ){
      // STEP 1: Get customer number
      def customerNo  = value;
      // STEP 2: Get current user
      def currentUser = userId;
      // STEP 3: Set user depending on the third place in the customer
```

```
number
      def returnFlag = false;
      def tmpValue = customerNo[2];
//(5)
      switch( tmpValue ){
         case "0":
            returnFlag = ( currentUser == "chef" );
            break;
         case "1":
            returnFlag = ( currentUser == "smith" );
            break;
         case "2":
            returnFlag = ( currentUser == "larson" );
            break;
         case "3":
            returnFlag = ( currentUser == "stark" );
            break;
         case "4":
            returnFlag = ( currentUser == "funny" );
            break;
         default:
            break;
      }
      return( returnFlag ? 1 : 0 );
   } // end of customerDocClass
} // end of D3DocumentClasss
```

Comments on the individual blocks

1. Import of the required libraries from the **groovyhook.jar** file.
2. Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts.
3. To assign a Groovy function to an entry point, a Groovy annotation with the predefined values is used to register the function for the entry point.
4. The different entry types require the parameters described in the Groovy hook documentation in the specified order. The d.3 object is always transferred first when a Groovy hook function is called. The function name does not have to match the entry point. Give a meaningful name to the function name **doSomething**.
5. In this example, a static user is now set via a switch query depending on the third value of the customer number. You can also work with restriction sets.

> **Warning**
> Please note that document class hooks should be used with great care, as the hooks have a strong impact on the performance of the overall system, depending on the function implemented.

6. Depending on the outcome of the verification, a **1** (allowed) or a **0** (denied) is returned.

1.7. Groovy API functions

As of version 8, d.3 server has a plug-in interface for API functions.

This allows you to register your own API functions developed in Java/Groovy.

Like other d.3 API functions, these can be called via the d.3 communication protocol d3fc.

> **Warning**
> Groovy API functions are not available via the d.3 web service API interface
>
> The Groovy API functions can be used in d.ecs forms via the scripting contained therein, as well as for d.velop's own projects.

Activating the plug-in interface

1. Open **d.3 admin > System settings > d. 3 config**.
2. Enter a directory in the **Java/Groovy** section for the **Java/Groovy API functions** entry.

The system will search for Groovy scripts that implement d.3 API functions in this directory and load them from this directory.

This activates the plug-in interface for API functions.

For a Java class to be loaded and registered as a d.3 API function, the Java class must be derived from the class **D3ApiCall** and implement a method **public int execute(D3Interface d3)**.

The D3Interface is then available above this.

```
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.D3ApiCall;
import com.dvelop.d3.server.User;

public class GetMyDBData extends D3ApiCall
{
    public int execute(D3Interface d3)
    {
        def import_param = d3.remote.getImportParams()
        def id_value  = import_param.get("id")
        def resultset = d3.sql.executeAndGet("SELECT column1, column2 FROM
mytable WHERE id like ?", [id_value])

        d3.remote.setExportTable( resultset )
        d3.remote.setExportParams(["number" : resultset.size()])

        return 0
    }
}
```

1.7.1. Groovy API and use in JPL

With the integration of Groovy as a server scripting language, you can create and use your own API functions.

An initial example with a call from JPL is presented below.

> **Note**
> Scenario:
>
> As an example, a function is shown here that uses SQL to read values from a table in the d.3 database and returns them.

Groovy example

SQL query as server function

```
package com.dvelop.api;
// (1)
import com.dvelop.d3.server.core.D3Interface;
import com.dvelop.d3.server.D3ApiCall;
import com.dvelop.d3.server.User;

// (2)
public class GetMyCustomerData extends D3ApiCall
{
// (3)
   public int execute( D3Interface d3 ){
      def importParams = d3.remote.getImportParams();
      def idValue      = importParams.get("id");
      def resultSet    = d3.sql.executeAndGet( """SELECT name, customerNo,
zipCode, city, street
                                                  FROM CustomerData
                                                 WHERE customerNo = ?""",
[idValue] );
// (4)
      d3.remote.setExportTable( resultset );
// (5)
      d3.remote.setExportParams(["number" : resultSet.size() ]);
      return 0;
    }
} // end of GetMyCustomerData
```

Comments on the individual blocks

<listitem>
Import of the required libraries from the **groovyhook.jar** file.
</listitem>
<listitem>
Deploy a custom class of type **public**; **public** can also be omitted in Groovy contexts. It is important that the extension **extendsD3ApiCall** is added.
</listitem>

1. The function that is now to be deployed as an API call is defined. The **getImportParams** command can also be used to read parameters that are deployed via the function call.
2. Using the function **setExportTable**, the results from the function are then also available to the calling program.

A d3FC call implemented using JPL

Example call from JPL

```
// (1)
vars lReturnValue

// (2)
call api_function("d3fc_user_set",          "dvelop")
call api_function("d3fc_password_set",      "dvelop")
call api_function("d3fc_remote_server_set", "127.0.0.1")
call api_function("d3fc_port_set",          "3400")
call api_function("d3fc_timeout_set",       "60")
call api_function("d3fc_server_set",        "B")

// (3)
```

```
// ACHTUNG ERST FUNKTION DANN PARAMETER
call api_function("d3fc_function_name_set",   "GetMyCustomerData")
call api_function("d3fc_importing_set",       "id" , "<CustomerNo>")
call api_function("d3fc_exporting_set",       "number")
call api_function("d3fc_table_set_headline",  "name customerNo zipCode City
street.","0")

// (4)
lReturnValue = api_function("d3fc_execute")

// (5)
call api_function("d3fc_exporting_get" ,"number")
vars lTableRowCount = api_single_info


// (6)
call api_log_error("SIZE :lTableRowCount  ")
call api_log_error(" :lReturnValue ")


// (7)
vars lRet, lName, lKdnr
lRet = api_function( "d3fc_first" )

// (8)
while( lRet != EOT)
{
 call api_function("d3fc_field_get","name")
 lName = api_single_info

 call api_function("d3fc_field_get","customerNo")
 lKdnr = api_single_info

 call api_log_error(":lName :lKdnr")
 lRet = api_function("d3fc_next")
}
```

Comments on the individual blocks

1. A local variable is created to hold the return value.
2. In the next step, the login parameters for the d3FC call must be specified.
3. To be able to call the function according to your own definition, the function name must be specified and the parameters passed.
4. Once all the necessary settings have been made, the custom API command can be executed using **d3fc_execute**.
5. The return values provided can be read out and transferred to local variables.
6. To document the function and the results, these are output here as an error message in the log file.
7. In the next step, the individual customer data is read out and also output in the log file.

> **Warning**
> In order for the defined API functions to be found and registered by the d.3 server, the path to the Groovy files with the created functions must be specified in the d.3 configuration for the parameter **JAVA_API_FUNCTIONS_DIR**.

1.8. Groovy-scripts

In addition to external JPL scripts, it is now also possible to execute Groovy scripts via the d.3 server interface.

In a Groovy script file, the d.3 interface is available as a field variable **d3** and can be used directly.

```
d3.log.info("Groovy-Script started!")
```

To register the type of the predefined field **d3** in a development environment so that type validations, command completions etc. can work, the following two lines should be included at the beginning of a script.

```
import com.dvelop.d3.server.Document
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.exceptions.D3Exception
D3Interface d3 = getProperty("d3");
String scriptName = getClass().getName()

d3.log.info("Groovy-Script started!")

// d.3 User which executes the update of documents
String scriptUser = "d3_groovy"
final def maxRows = 3

//Einzelne Dokumente/Akten anhand der ID
def resultSet = d3.sql.executeAndGet("select doku_id from firmen_spezifisch
where kue_dokuart = 'DTEST' and dok_dat_feld_1 = 'oldValue'", maxRows)
resultSet.each{
    try {
        d3.log.info (scriptName + "> Updating document with ID " +
it.doku_id)
                //Get document object of doc id
        Document myDoc = d3.archive.getDocument(it.doku_id, scriptUser)
                //Set new value
                myDoc.field[1] = "newValue"
                //Update metadata without triggering hook entrypoints
        myDoc.updateAttributes(scriptUser, true)
    } catch (D3Exception e) {
        d3.log.error(scriptName + "> Error updating meta data " +
e.getMessage())
    }
}
```

Using Groovy-classes and Java-libraries in scripts

The Groovy script directory "ext_groovy" and the defined Groovy Hook directories are added to the CLASSPATH when executing a script so that classes from other Groovy-scripts can be used in the executing script. Moreover, Classpath-files are also supported for scripts (file name: "<script name>.class-path") The included paths (e.g. absolute path of a JAR-file) are also added to the classpath to allow you to use these resources in the script.

Example: The JavaMail API (**javax.mail.jar**) is to be used in a script ../ext_groovy/myScript.groovy. For this purpose, the absolute path of the JAR-file is entered in a Classpath file with the same name ../ext_groovy/myScript.classpath:

Sample file content: D:\downloads\java\ext_jars\javax.mail-1.5.6.jar

Starting scripts controlled by timers

If you want a script to be started automatically and on a timer-controlled basis, rather than interactively, this can be specified as the sixth command line parameter of a server process in the d.3 process manager. The target entry in d.3 process manager will then appear similar to the following:

..\d3odbc32.exe haupt "" d3_server d3_server D3P ext_groovy/myScript.groovy

1.9. d.3 interface (D3Interface)

```
package com.dvelop.d3.server.core;

public interface D3Interface
{
    public interface ArchiveInterface         // d.3 Archiv
    public interface SqlD3Interface           // d.3 SQL Datenbank
    public interface D3RemoteInterface        // Client API
    public interface ScriptCallInterface      // Server API
    public interface ConfigInterface          // Config-Parameter
    public interface LogInterface             // Logging
    public interface HookInterface            // Hook-Eigenschaften
    public interface StorageManagerInterface  // Storagemanager

    public ArchiveInterface         getArchive();
    public SqlD3Interface           getSql();
    public D3RemoteInterface        getRemote();
    public ScriptCallInterface      getCall();
    public ConfigInterface          getconfig();
    public LogInterface             getlog();
    public HookInterface            getHook();
    public StorageManagerInterface  getStorageManager();
  }
```

> **Note**
> The D3Interface is passed by the server as the first parameter for all calls to registered Groovy functions.
>
> This means that the d.3 interface is available in all hook, API and script functions.

> **Note**
> The **javadoc** documentation for the d.3 interface is contained in the file **groovyhook-javadoc.jar**.
>
> Add this file to your Groovy project in your development environment to display a description of the classes and methods of the d.3 interface.
>
> For Eclipse, this is described in more detail on the page Creating hook projects.

1.9.1. d.3 archive (ArchiveInterface)

ArchiveInterface

```
    public interface ArchiveInterface {
        public Document             getDocument(String id, String
contextUser); //Laden Objekt vom Typ "Dokument"
        public Document             getDocument(String id); //Laden Objekt
vom Typ "Dokument"
```

```
        public DocumentType         getDocumentType(String id); //Laden
Objekt vom Typ "Dokumenttyp"
        public PredefinedValueSet   getPredefinedValueSet(String id); //
Laden Objekt vom Typ "Wertemenge"
        public RepositoryField      getRepositoryField(String id); //
Laden Objekt vom Typ "erweiterte Eigenschaft"
        public User                 getUser(String id); //Laden Objekt vom
Typ "Benutzer"
        public UserGroup            getUserGroup(String id); //Laden Objekt
vom Typ "Gruppe" geladen
        public UserOrUserGroup      getUserOrUseGroup(String id); //
Ermittlung, ob es sich beei dem Objekt um eine Gruppe oder einen Benutzer
handelt
        public AuthorizationProfile getAuthorizationProfile(String id);
//Laden Objekt vom Typ "Berechtigungsprofil"
        public void                 removeTranslationFromCache(String
entryPoint, Locale lang);    //Übersetzung einer Groovy-Hookwertemenge
        public Document             newDocument(); //Erzeugen eine neuen
Objekts vom Typ "Dokument" oder "Akte"
        public Document             importDocument(Document doc, Path
importFilePath); //Erstellen eines Dokuments
        public Document             importDocument(Document doc); //
Erstellen einer Akte
 }
```

The ArchiveInterface provides various Java objects that can be used to access the corresponding d.3 archive objects directly.

Document getDocument (String id, String contextUser)

Description: This function is used to load an object of the type "Document" or "Dossier".

Parameter:

- id: ID of the dossier or document whose object is to be loaded
- contextUser: User ID of the user in whose context the document or dossier is to be imported

Return values:

- Document object

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", Const.userHook)
```

Document getDocument(String id)

Description: This function is used to load an object of the type "Document" or "Dossier".

Parameter:

- id: ID of the dossier or document whose object is to be loaded

Return values:

- Document object

```
Document myDoc = d3.archive.getDocument("P000000001")
```

DocumentType getDocumentType(String id)

Description: This function is used to load an object of the type "Document type".

Parameter:

- id: ID of the dossier or document type whose object is to be loaded

Return values:

- Document type object

```
DocumentType myDocumentType = d3.archive.getDocumentType("DDOKU");
d3.log.info("default name of document type is
"+myDocumentType.getDefaultName());
d3.log.info("name of document type is "+myDocumentType.getName());
d3.log.info("type of document type is "+myDocumentType.getType());
if(myDocumentType.getType() == "d"){
    d3.log.info("type of this document type is 'document'");
} else if(myDocumentType.getType() == "a"){
    d3.log.info("type of document type is 'folder'");
}
```

PredefinedValueSet getPredefinedValueSet(String id)

Description: This function is used to load an object of the type "Dataset".

Parameter:

- id: ID of the dataset whose object is to be loaded

Return values:

- Dataset object

```
PredefinedValueSet myPredefinedValueSet =
d3.archive.getPredefinedValueSet("16");
d3.log.info("name of PredefinedValueSet is
"+myPredefinedValueSet.getName());
myPredefinedValueSet.values.each{
    d3.log.info("value of PredefinedValueSet -> " + it)
}
```

RepositoryField getRepositoryField(String id)

Description: This function is used to load an object of the type "Advanced property".

Parameter:

- id: ID of the advanced property whose object is to be loaded

Return values:

- Property object

```
RepositoryField repoField = d3.archive.getRepositoryField("196")
d3.log.info("repository field text "+ repoField.text)
d3.log.info("repository field name "+ repoField.name)
d3.log.info("repository field prefered field number "+
repoField.preferedFieldNumber)
d3.log.info("repository field data type "+ repoField.dataType)
```

User getUser(String id)

Description: This function is used to load an object of the type "User".

Parameter:

- id: ID of the user whose object is to be loaded

Return values:

- Dataset object

```
User myUser = d3.archive.getUser("d3_groovy");
d3.log.info("User name of " + myUser.id + " is " + myUser.getRealName())
```

UserGroup getUserGroup(String id)

Description: This function is used to load an object of the type "Group".

Parameter:

- id: ID of the group whose object is to be loaded

Return values:

- Group object

```
UserGroup myGroup = d3.archive.getUserGroup("MYTESTGROUP");
d3.log.info("Group name of " + myGroup.id + " is " + myGroup.getName())
```

UserOrUserGroup getUserOrUserGroup(String id)

Description: This function can be used to determine whether the object belonging to the ID is a group or a user.

Parameter:

- id: ID of the group or user whose object is to be loaded

Return values:

- Group or user object

```
UserOrUserGroup myUserOrGroup = d3.archive.getUserOrUserGroup("d3_groovy")
if(myUserOrGroup.isUserGroup == true){
    d3.log.info(myUserOrGroup.id + " is a group")
}
else if(myUserOrGroup.isUser == true){
    d3.log.info(myUserOrGroup.id + " is a user")
}
```

AuthorizationProfile getAuthorizationProfile(String id)

Description: This function is used to load an object of the type "Authorization profile".

Parameter:

- id: ID of the authorization profile whose object is to be loaded

Return values:

- Authorization profile object

```
AuthorizationProfile myAuthorizationProfile =
d3.archive.getAuthorizationProfile("1");
d3.log.info("name of " + myAuthorizationProfile.id + " is " +
myAuthorizationProfile.name)
```

void removeTranslationFromCache(String entryPoint, Locale lang)

Description: This function is used to (re)load the translation of a Groovy hook dataset.

Parameter:

- entryPoint: Entry point under which the dataset was registered (not the function name of the dataset)
- Locale: Language for which the translation is to be (re)loaded

Return values:

- -

```
//Lade Übersetzung für Sprache "Deutsch"
d3.getArchive().removeTranslationFromCache("meineGroovyWertemenge", new
Locale("de"));
//Lade Übersetzung für Sprache "Englisch"
d3.getArchive().removeTranslationFromCache("meineGroovyWertemenge", new
Locale("en"));

//Komplettes Beispiel für eine übersetzte Hook-Werktemenge finden Sie hier
```

Document newDocument()

Description: This function is used to create a new document or dossier object.

Parameter:

- -

Return values:

- Empty document or dossier object

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3");

// Create an new empty document
Document newDoc = d3.archive.newDocument();

// Add system properties
newDoc.type = "DA1";                                        // ID of target
document
newDoc.status = Document.DocStatus.DOC_STAT_RELEASE;        // Target state
of document
newDoc.editor = "dvelop";                                   // Handler
newDoc.setText(1, "Import per Groovy Skript");              // Comment
// erweiterte Eigenschaften zuweisen
newDoc.field[1] = "Attribute value 1 for new document";
newDoc.field[2] = "Attribute value 2 for new document";
newDoc.field[60][1] = "Multi value field 60-1 for new document";
newDoc.field[60][2] = "Multi value field 60-2 for new document";
```

Document importDocument(Document doc, Path importFilePath)

Description: This function is used to create a document with a file in d.velop documents.

Parameter:

- doc: Document object which is to be created in d.velop documents
- Path: Path of the file that is to be stored for the document.

Return values:

- Document object that was created

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document
import com.dvelop.d3.server.exceptions.D3Exception
import java.nio.file.Path
import java.nio.file.Paths

D3Interface d3 = getProperty("d3");

// Create an new empty document
Document newDoc = d3.archive.newDocument();

// Add system properties
newDoc.type = "DA1";                                        // ID of target
document
newDoc.status = Document.DocStatus.DOC_STAT_RELEASE;        // Target state
of document
newDoc.editor = "dvelop";                                   // Handler
newDoc.setText(1, "Import per Groovy Skript");              // Comment
// erweiterte Eigenschaften zuweisen
newDoc.field[1] = "Attribute value 1 for new document";
newDoc.field[2] = "Attribute value 2 for new document";
newDoc.field[60][1] = "Multi value field 60-1 for new document";
newDoc.field[60][2] = "Multi value field 60-2 for new document";

// Define file for new document
Path importFile = Paths.get("D:\\temp\\my_file.txt");
try {
   // Import the document
   newDoc = d3.archive.importDocument(newDoc, importFile);
}
catch (D3Exception e) {
   d3.log.error(e.message);
   return;
}
d3.log.info("Doc-ID of newly created document: " + newDoc.id);
```

Document importDocument(Document doc)

Description: This function creates a dossier without a file in d.velop documents.

Parameter:

- doc: Dossier object to be created in d.velop documents

Return values:

- Document object that was created

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document
import com.dvelop.d3.server.exceptions.D3Exception

D3Interface d3 = getProperty("d3");

// Create an new empty document
Document newDoc = d3.archive.newDocument();

// Add system properties
newDoc.type = "BAKT";                                        // ID of target
document
newDoc.status = Document.DocStatus.DOC_STAT_RELEASE;       // Target state
of document
newDoc.editor = "wfl_admin";                                    // Handler
newDoc.setText(1, "Folder via importDocument");// Comment

// Assign extended properties
newDoc.field[1] = "Attribute value 1 for new document";
newDoc.field[2] = "Attribute value 2 for new document";

try {
    // Import the document / create the folder
    newDoc = d3.archive.importDocument(newDoc);
}
catch (D3Exception e) {
    d3.log.error(e.message);
    return;
}

d3.log.info("Doc-ID of newly created folder: " + newDoc.id);
```

Archive objects (ArchiveObject)

ArchiveInterface

```
public interface ArchiveObject
{
    public String getId();
}
```

All d.3 archive objects are derived from this class. This means that the d.3 ID of the object can be determined in all archive objects.

```
Document doc;
doc.id         // Document ID

DocumentType docType;
docType.id     // Document type

User user;
user.id        // User ID
```

Document

Document

```
class Document extends ArchiveObject{
    public Field           getField()      // Groovy Collection Support for
reading and writing the property values
    public DocumentType    getType()
    public void            setType(DocumentType type)
    public void            setType(String typeId)
    public String          getNumber()
    public void            setNumber(String docNumber)
    public DocStatus       getStatus()
    public void            setStatus(DocStatus docStatus)
    public void            setStatus(String docStatus)
    public int             getVarnumber()
    public void            setVarnumber(int varNumber)
    public UserOrUserGroup getEditor()
    public void            setEditor(UserOrUserGroup editor)
    public void            setEditor(String editor)
    public String          getOwner()
    public String          getFilename()
    public String          getFileExtension()
    public Long            getDocSize()
    public String          getText(int lineIdx)
    public void            setText(int lineIdx, String text)
    public Timestamp       getCreated()
    public Timestamp       getLastAccess()
    public Timestamp       getLastUpdateFile()
    public Timestamp       getLastUpdateAttribute()
    public Timestamp       getLastUpdate()
    public Integer         getSignaturesRequired()
    public Integer         getColorCode()
    public void            setColorCode(Integer colorCode)
    public String          getReleaseVersionStatus()
    public boolean         getIsVerified()
    public boolean         getIsInWorkflow()
    public int             getNumericId()
    public Integer         getLastAlterationNumber()
    public Integer         getAlterationNumberReleased()
    public Integer         getCodepage()
    public String          getCaption()
    public boolean         getHasMultData()
    public Integer         getFileIdCurrentVersion()
    public Integer         getFileIdRelease()
    public Timestamp       getEndOfRetentionDate()

    public void            updateAttributes(String userId)
    public void            updateAttributes(String userId, boolean noHooks)
    public int             changeType(String docTypeId, String userId)
    public String          getPermission(String userId)
    public int             block(boolean block, String userId)
    public int             verify(String userId)
    public int             transfer(String destination, String newEditor,
String changeRemark, boolean asynchronous, int archivIndex, String userId)
    public int             addDependent(String filename, String docStatus,
String docExt, String userId)
    public int             addDependent(String filename, DocStatus
docStatus, String docExt, String userId)
```

```
    public int              deleteDependent(char docStatus, String docExt,
String userId)
    public int              deleteDependent(DocStatus docStatus, String
docExt, String userId)
    public int              startLifetime(boolean overwriteOldDate, int
lifeTimeDays)
    public int              setCacheDays(char docStatus, int archiveIndex,
int daysInCache)
    public int              checkFolderScheme(String userId);

    public DocumentVersion[] getVersions()
    public PhysicalVersion[] getPhysicalVersions()
    public DocumentNote[]    getNotes()

    public SysFields         getSysField()     // Groovy Collection Support
for reading and writing system properties
    public void              addSysField(String fieldName)
    public List              getSysValues()
    public Set               getSysFieldNames()
}
//Siehe auch verschiedene Beispiele
```

*) The Groovy collection support for "Fields" allows you to use of the Subscript Operator [] for the access to the advanced properties of a document.

Field getField()

Description: Returns the values for the advanced properties of a document.

Parameter:

- -

Return values:

- Advanced properties for a document.

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
def myField = myDoc.getField();d3.log.info("Value of docField 1 is
"+myField[1]);
d3.log.info("Value of docField 60 Row 1 is "+myField[60][1]);

#Direkter lesender Zugriff auf "Field";
d3.log.info("Value of docField 1 is "+myDoc.field[1]);
d3.log.info("Value of docField 60 Row 1 is "+myDoc.field[60][1]);

#Direkter schreibender Zugriff auf "Field";
myDoc.field[1] = "Value for docField 1";
myDoc.field[60][1] = "Value for docField 1 Row 1";
```

DocumentType getType()

Description: Returns the "Document type" object for a document.

Parameter:

- -

Return values:

- Document type of the document

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
DocumentType myType = myDoc.getType();
d3.log.info("docTypeShort of document is "+myType.id);
d3.log.info("docTypeLong of document is "+myType.getName());

#Direkter Zugriff auf "DocumentType";
d3.log.info("docTypeShort of document is "+myDoc.type.id);
d3.log.info("docTypeLong of document is "+myType.type.name);
```

void setType(DocumentType type)

Description: Sets the document type of a new object (dossier or document). The server API function "document_change_type" is recommended for changing the document/dossier type.

Parameter:

- Document type to be set

Return values:

- -

Example:

```
Document myDoc;
DocumentType myDocumentType = d3.archive.getDocumentType("DDOKU");
myDoc.setType(myDocumentType;)
```

void setType(String typeId)

Description: Sets the document type of a new object (dossier or document). The server API function "document_change_type" is recommended for changing the document/dossier type.

Parameter:

- ID of the document type to be set

Return values:

- -

Example:

```
Document myDoc;
myDoc.setType("APROJ");
```

String getNumber()

Description: Returns the document number (zeich_nr) of a document or dossier.

Parameter:

- -

Return values:

- Document number (zeich_nr) of a document or dossier

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("document number of document is "+myDoc.getNumber());
```

void setNumber(String docNumber)

Description: Sets the document number (zeich_nr) of a new object (dossier or document) or of a newly imported object in the entry point "hook_insert_entry_10".

Parameter:

- Max 30-digit character string (letters + numbers)

Return values:

- -

Example:

```
def myDoc = d3.archive.newDocument();
myDoc.setNumber("EXTERNALNUMBER10000");
```

DocStatus getStatus()

Description: Returns the document status of a document

Parameter:

- -

Return values:

- DOC_STAT_PROCESSING = Processing;
- DOC_STAT_VERIFICATION = Verification;
- DOC_STAT_RELEASE = Release ;
- DOC_STAT_ARCHIVE = Archive;

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("document status of document is "+myDoc.getStatus());
d3.log.info("document status of document is "+myDoc.status);
```

void setStatus(DocStatus docStatus)

Description: Sets the document status of a new object (dossier or document) or of a newly imported or re-versioned object in the entry point "hook_insert_entry_10" / "hook_new_version_entry". To change the document status outside the above examples, the server API function "document_transfer" is recommended.

Parameter:

- Document status
  - Processing = Document.DocStatus.DOC_STAT_PROCESSING;
  - Verification = Document.DocStatus.DOC_STAT_VERIFICATION;
  - Release = Document.DocStatus.DOC_STAT_RELEASE;
  - Archive = Document.DocStatus.DOC_STAT_ARCHIVE;

Return values:

- -

Example:

```
def myDoc = d3.archive.newDocument();
myDoc.setStatus(Document.DocStatus.DOC_STAT_RELEASE);
myDoc.status = Document.DocStatus.DOC_STAT_RELEASE;
```

void setStatus(String docStatus)
Description: Sets the document status of a new object (dossier or document) or of a newly imported or re-versioned object in the entry point "hook_insert_entry_10" / "hook_new_version_entry". To change the document status outside the above examples, the server API function "document_transfer" is recommended.

Parameter:

- Processing = Processing
- Verification = Verification;
- Release = Release;
- Archive = Archive;

Return values:

- -

Example:

```
def myDoc = d3.archive.newDocument();
myDoc.setStatus("Freigabe");
myDoc.status = "Freigabe";
```

int getVarnumber()
Description: Returns the variant number (var_nr) of a document or a dossier

Parameter:

- -

Return values:

- Variant number (var_nr) of a document or dossier

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("document variant number of document is "+myDoc.getVarnumber());
```

void setVarnumber(int varNumber)
Description: Sets the variant number (var_nr) of a new object (dossier or document) or a newly imported object in the entry point "hook_insert_entry_10".

Parameter:

- Variant number to be set

Return values:

-

Example:

```
def myDoc = d3.archive.newDocument();
myDoc.setVarnumber(1);
```

UserOrUserGroup getEditor()

Description: Returns the editor (user or group) of a document or a dossier if the document is in the Processing status.

Parameter:

- -

Return values:

- Editor of the document (user or group)

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
UserOrGroup myUserOrGroup = myDoc.getEditor();
if(myUserOrGroup == null){
    d3.log.info("document has no editor")
} else if (myUserOrGroup.getIsUser()){
    d3.log.info("editor is a user with id " + myUserOrGroup.id)
} else if (myUserOrGroup.getIsUserGroup()){
    d3.log.info("editor is a group with id " + myUserOrGroup.id)
}
```

void setEditor(UserOrUserGroup editor)

Description: Sets the editor of a new object (dossier or document) or of a newly imported or re-versioned object in the entry point "hook_insert_entry_10" / "hook_new_version_entry"; this is only necessary for the Processing status. The server API function "document_transfer" is recommended for changing the editor outside of the above examples.

Parameter:

- User object or group object

Return values:

-

Example:

```
def myDoc = d3.archive.newDocument();
UserOrUserGroup myUserOrGroup = d3.archive.getUserOrUserGroup("d3_groovy");
myDoc.setEditor(myUserOrGroup);
```

void setEditor(String editor)

Description: Sets the editor of a new object (dossier or document) or of a newly imported or re-versioned object in the entry point "hook_insert_entry_10" / "hook_new_version_entry". The server API function "document_transfer" is recommended for changing the editor outside of the above examples.

Parameter:

- User or group ID

Return values:

-

Example:

```
def myDoc = d3.archive.newDocument();
myDoc.setEditor("d3_groovy");
```

String getOwner()
Description: Returns the owner of a document or dossier. The owner is the user who last shared a document.

Parameter:

•

Return values:

• Owner of a document or dossier

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("owner of document is "+myDoc.getOwner());
```

String getFilename()
Description: Returns the file name of a document. The file name is usually the original file name of the file that was selected during document import or set via API.

Parameter:

• -

Return values:

• File name of the document or dossier

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("file name of document is "+myDoc.getFilename());
```

String getFileExtension()
Description: Returns the file extension of a document from the current version.

Parameter:

• -

Return values:

• File name of the document

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("file extension of document is "+myDoc.getFileExtension());
```

Long getDocSize()
Description: Returns the file size in bytes of a document from the current version.

Parameter:

- -

Return values:

- File size in bytes of a document from the current version.

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("file size in byte of document is "+myDoc.getDocSize());
```

String getText(int lineIdx)
Description: Returns the comment text of a document or dossier for the specified index (1-4).

Parameter:

- -

Return values:

- Comment text of a document or dossier for the specified index (1-4).

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("text 1 of document is "+myDoc.getText(1));
```

void setText(int lineIdx, String text)
Description: Sets the comment text of a document or dossier to the specified index (1-4).

Parameter:

- lineIdx = Index between 1 and 4
- text = Text to be set

Return values:

- -

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
myDoc.setText(1, "my comment for text 1");
myDoc.updateAttributes("d3_groovy");
```

Timestamp getCreated()
Description: Returns the creation date of a document or dossier.

Parameter:

-

Return values:

- Creation date of the document or dossier

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("create date of document is "+myDoc.getCreated());
```

Timestamp getLastAccess()
Description: Returns the date of the last access (preview or export) to a document file.

Parameter:

- -

Return values:

- Date of the last access (preview or export) to a file of a document

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("last access date of document is "+myDoc.getLastAccess());
```

Timestamp getLastUpdateFile()
Description: Provides the date of the last upload of a document file, regardless of the status or status transfer.

Parameter:

- -

Return values:

- Date of the last upload of a document file

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("last file update of document was "+myDoc.getLastUpdateFile());
```

Timestamp getLastUpdateAttribute()
Description: Provides the date on which the properties of a document or dossier were last updated.

Parameter:

- -

Return values:

- Date of the last update of the properties of a document or dossier.

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("last update of metadata of document was
"+myDoc.getLastUpdateAttribute());
```

Timestamp getLastUpdate()
Description: Returns the date of the last change to a document or dossier (updated properties, uploaded files, status transfer).

Parameter:

- -

Return values:

- Date of the last change to a document or dossier

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("last action date of document is "+myDoc.getLastUpdate());
```

Integer getColorCode()

Description: Returns the color code of a document or dossier

Parameter:

- -

Return values:

- 0 = Not set
- 1 – 24 = Color code

Colors from top left = 1, to bottom right = 24

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("the color code of document is "+myDoc.getColorCode());
```

void setColorCode(Integer colorCode)

Description: Sets the color code of a document or dossier

Parameter:

- 0 = No color
- 1 – 24 = Color code

Colors from top left = 1, to bottom right = 24

Return values:

- -

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
myDoc.setColorCode(2);
myDoc.updateAttributes("d3_groovy");
```

String getReleaseVersionStatus()
Description: Returns whether the status of a document or dossier is "Release blocked".

Parameter:

- -

Return values:

- f = Enable
- g = Blocked

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("release version status of document is
"+myDoc.getReleaseVersionStatus());
```

boolean getIsVerified()
Description: Returns whether the status of a document or dossier is "Verification".

Parameter:

- -

Return values:

- False = Not verified
- True = Verified

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
if(myDoc.getIsVerified() == true){
   d3.log.info("document is verified");
} else {
   d3.log.info("document is not verified");
}
```

boolean getIsInWorkflow()
Description: Returns whether the object is in an active workflow for a document or dossier.

Parameter:

- -

Return values:

- False = Not currently in a workflow
- True= Is currently in a workflow

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
if(myDoc.getIsInWorkflow() == true){
```

```
   d3.log.info("document is in a workflow");
} else {
   d3.log.info("document is not in a workflow");
}
```

int getNumericId()

Description: Returns the counter for the DocumentId for a document or dossier.

Parameter:

- -

Return values:

- Counter for the DocumentId of a document or dossier

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("numeric id of document is "+myDoc.getNumericId());
```

Integer getLastAlterationNumber()

Description: Returns the last change number of versions for a document. If, for example, a document is set to the Processing status and then transferred to the Released status without being changed, this number is incremented even if no actual changes have been made to the file.

Parameter:

- -

Return values:

- Last change number

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("last alteration number of document is
"+myDoc.getLastAlterationNumber());
```

Integer getAlterationNumberReleased()

Description: Returns the last change number of versions for a document with the status Released.

Parameter:

- -

Return values:

- Last change number in release status

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("last alteration number released of document is
"+myDoc.getAlterationNumberReleased());
```

Integer getCodepage()

Description: Returns the code page of the properties stored in the database for the document or dossier.
Note: This value is only filled if the standard code page was not used

Parameter:

- -

Return values:

- Code page of the properties for a dossier or document

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
if(myDoc.getCodepage() == 0){
    d3.log.info("codepage of document is the default codepage")
} else {
    d3.log.info("codepage of document is "+myDoc.getCodepage())
}
```

String getCaption()

Description: Provides the title or caption for a document or dossier.

Parameter:

- -

Return values:

- Title or caption for a document or dossier

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("document xxx of document is "+myDoc.getCaption());
```

boolean getHasMultData()

Description: Returns whether at least one line of a multi-value field has been filled for a document or dossier.

Parameter:

- -

Return values:

- true = At least one line of a multi-value field is filled
- false = No line of a multi-value field is filled

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
if(myDoc.getHasMultData() == true){
    d3.log.info("document has no multi metadata given")
} else {
    d3.log.info("document has multi metadata given")
}
```

Integer getFileIdCurrentVersion()

Description: Returns the ID of the file in the current version.

Parameter:

- -

Return values:

- ID of the file in the current version.

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("file id of document is "+myDoc.getFileIdCurrentVersion());
```

Integer getFileIdRelease()
Description: Returns the ID of the file in the release version.

Parameter:

- -

Return values:

- ID of the file in the release version.

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("file id in state release of document is
"+myDoc.getFileIdRelease());
```

Timestamp getEndOfRetentionDate()
Description: Provides the date at the end of the retention period of the document or dossier.

Parameter:

- -

Return values:

- Date at the end of the retention period of the document or dossier.

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
d3.log.info("retention date of document is "+myDoc.getEndOfRetentionDate());
```

void updateAttributes(String userId)
Description: Updates properties or system properties of a document or dossier

Parameter:

- userId = ID of the user who is to update the properties

Return values:

- -

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
myDoc.field[1] = "Value for field 1"
myDoc.docMultField(62, 1, "")
```

```
try{
   def retVal = myDoc.updateAttributes("d3_groovy")
   writeLog("Info", "Update successful for " + myDoc.id)
} catch (D3Exception e){
   d3.log.error("Update failed for " + myDoc.id + " with error message: " +
e.getMessage())
}
```

void updateAttributes(String userId, boolean noHooks)

Description: Updates properties or system properties of a document or dossier

Parameter:

- userId = ID of the user making the change
- noHooks = true → Entry points are not executed, false → Entry points are executed (default). This parameter has no influence on the creation of the dossier scheme

Return values:

- -

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
myDoc.field[1] = "Value for field 1"
myDoc.docMultField(62, 1, "")
try{
   def retVal = myDoc.updateAttributes("d3_groovy", true)
   writeLog("Info", "Update successful for " + myDoc.id)
} catch (D3Exception e){
   d3.log.error("Update failed for " + myDoc.id + " with error message: " +
e.getMessage())
}
```

id changeType(String docTypeId, String userId)

Description: Changes the category of a dossier or document

Parameter:

- docTypeId = ID of the document whose document type is to be changed
- userId = ID of the user making the change

Return values:

- -

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
myDoc.changeType("DTEST", "d3_groovy")
```

String getPermission(userId)

Description: Determines a user's permissions to a document

Parameter:

- ID of the user whose authorizations are to be queried

Return values:

- a1a2a3a4a5a6a7a8-b1b2b3b4b5b6b7b8-c1c2-d1d2

Parameters	Description
a1	Access to document type allowed
a2	Note file available
a3	Document links available
a4	User file available
a5	User file encrypted
a6	Released version available (1: released, 2: blocked)
a7	Last version in Verification (1: verified, 2: unverified)
a8	At least one version in Archive status
b1	Permission to display usage data
b2	Permission to change identification data
b3	Permission to change user data
b4	Permission to delete the document
b5	Permission for Transfer status
b6	Permission to review the document
b7	Permission to release/block the document
b8	Permission to perform both review steps (Verification and Release) in one step
c1	Permission to display an older version in Release
c2	Permission to release/block an older version in Release
d1	Permission to view versions in Archive status
d2	Document is in the workflow

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
```

int block()

Description: Blocking or unblocking a document

Parameter:

- block: true = block, false = unblock
- userId: ID of executing user

Return values:

- -

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
int returnValue = myDoc.block(false, "d3_groovy")
if(returnValue == 0){
   d3.log.info("success")
} else {
   //exception handling
}
```

int verify()

Description: It has been verified that the Verification status was set

Parameter:

- userId: ID of executing user

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **-1**: Syntax error, for details see d.3 logviewer
- **-2**: Function was called incorrectly
- **-3**: Unknown error, please contact d.velop Support
- **-201**: Invalid user name specified
- **-301**: User does not have permissions
- **-302**: Error when determining user permissions
- **-303**: User does not have access permissions for the document
- **-307**: User does not have permission to verify this document
- **-509**: Version in **Verification** status is already verified
- **-510**: There is no version in **Verification** status.
- **< -9500**: Error in customer-specific hook function, for details see d.velop logviewer

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
int returnValue = myDoc.verify("d3_groovy")
if(returnValue == 0){
   d3.log.info("success")
} else {
   d3.log.error("error")
   //exception handling
}
```

int transfer(String destination, String newEditor, String changeRemark, boolean asynchronous, int archivIndex, String userId)

Description:

Parameter:

- destination : Destination status Processing, Verification, Release, Archive
- newEditor: User/group name required for destination = Processing (user or group); destination = Verification (group). Mandatory field if the Processing destination status was specified for the destination parameter.
- changeRemark: Must be specified if a document is transferred from Processing to Verification and the document has been changed from a previous version in Processing or Archive status.
- asynchronous: Specifies whether the transfer is to be executed immediately or asynchronously using an asynchronous job for d.3 async(false: immediately (default); true: asynchronously by d.3 async).
- archivIndex : Index for the version to be restored in the Archive status
- userId: User who is to perform the transfer. If nothing is specified, the user from the context is used.

Return values:

- -

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
int returnValue = myDoc.transfer("Bearbeitung", "d3_groovy", "", false, 0,
"d3_groovy")
if(returnValue == 0){
```

```
   d3.log.info("success")
} else {
   d3.log.error("error")
   //exception handling
}
```

int addDependent(String filename, String docStatus, String docExt, String userId)
Description:

Parameter:

- filename
- docStatus
- docExt
- userId

Return values:

- -

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
int returnValue = myDoc.addDependent("C:/Temp/myFileOnServer.pdf",
"Freigabe", "p1", "d3_groovy")
if(returnValue == 0){
   d3.log.info("success")
} else {
   d3.log.error("error")
   //exception handling
}
```

int addDependent(String filename, DocStatus docStatus, String docExt, String userId)
Description:

Parameter:

- filename
- docStatus
- docExt
- userId

Return values:

- -

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
int returnValue = myDoc.addDependent("C:/Temp/myFileOnServer.pdf",
Document.DocStatus.DOC_STAT_RELEASE, "p1", "d3_groovy")
if(returnValue == 0){
   d3.log.info("success")
} else {
   d3.log.error("error")
   //exception handling
}
```

int deleteDependent (char docStatus, String docExt, String userId)

Description: Deletes the dependent file in the specified status

Parameter:

- docStatus: Document status of the dependent file to be deleted
  - **B** for **Processing**
  - **P** for **Verification**
  - **F** for **Release**
- docExt: Extension of the dependent file to be deleted
- userId: ID of executing user

Return values:

- -

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
int returnValue = myDoc.deleteDependent("F", "p1", "d3_groovy")
if(returnValue == 0){
   d3.log.info("success")
} else {
   d3.log.error("error")
   //exception handling
}
```

int deleteDependent (DocStatus docStatus, String docExt, String userId)

Description: Deletes the dependent file in the specified status

Parameter:

- docStatus: Document status of the dependent file to be deleted
- docExt: Extension of the dependent file to be deleted
- userId: ID of executing user

Return values:

- -

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
int returnValue =
myDoc.deleteDependent(Document.DocStatus.DOC_STAT_RELEASE, "p1",
"d3_groovy")
if(returnValue == 0){
   d3.log.info("success")
} else {
   d3.log.error("error")
   //exception handling
}
```

int startLifetime (boolean overwriteOldDate, int lifeTimeDays)

Description: Defines the lifetime of a document or dossier. Note: If the lifetime is removed or shortened, this may have no effect on the secondary storage.

Parameter:

- overwriteOldDate: true = existing value should be overwritten, false = existing value should not be overwritten
- lifeTimeDays: If you do not want to use the configured lifetime for the document type, you can specify the number of days (default: **0** - configured lifetime). If **-1** is specified, the lifetime is set to "unlimited".

Return values:

- -

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
int returnValue = myDoc.startLifetime(true, 365)
if(returnValue == 0){
   d3.log.info("success")
} else {
   d3.log.error("error")
   //exception handling
}
```

int setCacheDays (char docStatus, int archiveIndex, int daysInCache)

Description: This function sets the number of days for a document version to remain in the d.3 document tree "from today". The number of days is only relevant if the document has been or will be saved to a secondary storage.

Parameter:

- docStatus: Document status of the document
- archiveIndex: Version index, only relevant for status archive
- daysInCache: Number of days the document version is to remain in the cache

Return values:

- -

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
int returnValue = myDoc.setCacheDays("F", 0, 30)
if(returnValue == 0){
   d3.log.info("success")
} else {
   d3.log.error("error")
   //exception handling
}
```

int checkFolderScheme (String userId);

Description: Triggers the dossier scheme rules for the dossier or document

Parameter:

- userid: ID of executing user

Return values:

- -

Example:

```
Document myDoc = d3.archive.getDocument("P000000001", "d3_groovy");
int returnValue = myDoc.checkFolderScheme("d3_groovy")
if(returnValue == 0){
   d3.log.info("success")
} else {
   d3.log.error("error")
   //exception handling
}
```

Examples

Example – Setting advanced properties

```
Document doc
doc.field[1] = "Value for field 1"                    // Writing value
d3.log.info(doc.field[1])                             // Reading value

// Referencing via the title of the property
 doc.field["Name"] = "Meier"                          // Writing value
println "Name = " + doc.field["Name"]                 // Reading value

doc.docMultField(62, 1, "value for multifield in first line") // Writing
value to a multifield

println(doc.docMultFieldAsString(62, 1);); // Reading the first line
println(doc.field[62]) // Reading the value, only the first line is returned
```

> **Note**
> If a field does not exist or does not have a value, then NULL is returned.

If you want to change a property (attribute), this is possible via the updateAttributes method. This method is needed when the value cannot be assigned directly. In "entry" hooks, where the values have not yet been passed to the database, the values can be assigned directly. You can see this in the example above. If you want to make a property change in an exit hook (for example, in hook_insert_exit_30 after the import), you must execute the updateAttributes method in addition to the assignment afterwards. The method is available in two variants. You can transfer the executive user to log who made the attribute change. However, a permissions check will not take place. You can implement this check using the getPermission(String userId) method. Additionally, a boolean can be transferred to indicate whether an updatehook should be run. This is useful, for example, if you make a property change in the hook_attrib_upd_exit_20 hook and do not want the update hook to be executed again.

Example – Updating properties

```
doc.field[1] = "Value for field 1"
doc.docMultField(62, 1, "")

doc.updateAttributes("User", true)
```

The value "true" causes no update hook to be started after execution.

System properties

Using the interface for the system properties. System properties are not simple property fields assigned to a document type or a dossier. So it is not displayed in the standard client applications of d.velop. It is a self-defined, technical field that can be used to store information. The fields are managed and stored in the doc_sys_fields table and the related ones in the doc_sys_values table.

Example – Setting system properties

```
println doc.sysField["InvoiceNo"][1]                                    // Reading
value
doc.sysField["InvoiceNo"][1] = "Wert für Systemeigenschaft"             // Add or
change value / add new system property with value
doc.sysField["InvoiceNo"].add("Wert")                                   // Another
way to do this
doc.sysField["InvoiceNo"].remove(2)                                     // Delete
value

doc.addSysField("New system field")                                     // Add new
system property without value
doc.sysField["InvoiceNo"].clear()                                       // Delete
all values for this property

println doc.sysFieldNames                                               // Go
through all system properties
println doc.sysField["InvoiceNo"].sysValues                             // Go
though all values of one system property
println doc.sysValues                                                   // Go
through all values of all system properties
```

To ensure that the values are actually saved, the updateAttributes method must still be executed.

Example – Updating system properties

```
D3Interface d3 = getProperty("d3");

String sDocIDToManip = "P000000001";

try {
   d3.log.info("hinzufuegen SysField: " + sDocIDToManip)

   Document doc = null;
   doc = d3.archive.getDocument(sDocIDToManip, "Administr2");

   doc.addSysField("myField")

   doc.sysField["myField"][1] = "Wert fuer Systemeigenschaft"
   doc.sysField["myField"].add("Ein Wert für myField")
   doc.updateAttributes("Administr2")

   d3.log.info(doc.getSysField())
   d3.log.error("doc.sysField[myField].sysValues =" +
doc.sysField["myField"].sysValues)

   d3.log.info("Sysfield erfolgreich gesetzt")

} catch (Exception ex){
          d3.log.error("Fehler beim setzen des Sysfields: " +
ex.getMessage());
}
```

Example – Importing documents

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document
import com.dvelop.d3.server.exceptions.D3Exception
```

```
import java.nio.file.Path
import java.nio.file.Paths

D3Interface d3 = getProperty("d3");

// Create an new empty document
Document newDoc = d3.archive.newDocument();

// Add system properties
newDoc.type = "DA1";                                        // ID of target
document
newDoc.status = Document.DocStatus.DOC_STAT_RELEASE;        // Target state
of document
newDoc.editor = "dvelop";                                   // Handler
newDoc.setText(1, "Import per Groovy Skript");              // Comment
// erweiterte Eigenschaften zuweisen
newDoc.field[1] = "Attribute value 1 for new document";
newDoc.field[2] = "Attribute value 2 for new document";
newDoc.field[60][1] = "Multi value field 60-1 for new document";
newDoc.field[60][2] = "Multi value field 60-2 for new document";

// Define file for new document
Path importFile = Paths.get("D:\\temp\\my_file.txt");
try {
   // Import the document
   newDoc = d3.archive.importDocument(newDoc, importFile);
}
catch (D3Exception e) {
   d3.log.error(e.message);
   return;
}
d3.log.info("Doc-ID of newly created document: " + newDoc.id);
```

Example – Importing dossiers

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document
import com.dvelop.d3.server.exceptions.D3Exception

D3Interface d3 = getProperty("d3");

// Create an new empty document
Document newDoc = d3.archive.newDocument();

// Add system properties
newDoc.type = "BAKT";                                        // ID of target
document
newDoc.status = Document.DocStatus.DOC_STAT_RELEASE;        // Target state
of document
newDoc.editor = "wfl_admin";                                   // Handler
newDoc.setText(1, "Folder via importDocument");// Comment

// Assign extended properties
newDoc.field[1] = "Attribute value 1 for new document";
newDoc.field[2] = "Attribute value 2 for new document";
```

```
try {
    // Import the document / create the folder
    newDoc = d3.archive.importDocument(newDoc);
}
catch (D3Exception e) {
    d3.log.error(e.message);
    return;
}

d3.log.info("Doc-ID of newly created folder: " + newDoc.id);
```

Document versions (DocumentVersion)

DocumentVersion

```
public class DocumentVersion extends ArchiveObject
{
    public enum Category {
          DOC_VERS_REGULAR,
          DOC_VERS_OVERWRITTEN,
          DOC_VERS_BLOCKED,
          DOC_VERS_REPLACED,
          DOC_VERS_DELETED,
          DOC_VERS_ERASED,
          DOC_VERS_MIGRATED,
          DOC_VERS_BOOKED,
          INVALID_CATEGORY
        };

    public String           getDocId()
    public boolean          isCurrentVersion()
    public boolean          hasStatus()
    public DocStatus        getStatus()
    public boolean          hasHistStatus()
    public DocStatus        getHistStatus()
    public boolean          hasFileId()
    public Integer          getFileId()
    public boolean          hasHistFileId()
    public Integer          getHistFileId()
    public Category         getCategory()
    public String           getChangeReason()
    public String           getDeleteReason()
    public PhysicalVersion  getPhysicalVersion()
    public String           getCreator()
    public Timestamp        getCreateDate()
    public String           getReleaseUser()
    public Timestamp        getReleaseDate()
    public String           getBlockUser()
    public Timestamp        getBlockDate()
    public String           getArchiveUser()
    public Timestamp        getArchiveDate()
    public String           getVerifier()
    public Timestamp        getVerifyDate()
    public String           getDeleter()
    public Timestamp        getDeleteDate()
    public Double           getExternalVersionId()
}
```

Example for accessing the versions of a document.

```
import com.dvelop.d3.server.Document
import com.dvelop.d3.server.DocumentVersion
import com.dvelop.d3.server.PhysicalVersion
import com.dvelop.d3.server.DependentFile

// Get document object
Document myDoc = d3.archive.getDocument("LV00004228", Const.userHook)
// Go / interate through all versions
for (version in myDoc.versions)
{
    // Get the properties of the current Version
    d3.log.info("- version id " + version.id)
    d3.log.info("- category of version is " +
version.getCategory().toString())

    if(version.getCategory() == DocumentVersion.Category.DOC_VERS_REGULAR){
        if(version.isCurrentVersion() == true){
            d3.log.info("- this version is the current version")
        }
        if(version.hasStatus() == true){
            d3.log.info("- status of version is " + version.getStatus())
        }
        if(version.hasHistStatus() == true){
            d3.log.info("- history status of version is " +
version.getHistStatus())
        }
        if(version.hasFileId() == true){
            d3.log.info("- file id of version is " + version.getFileId())
        }
        if(version.hasHistFileId() == true){
            d3.log.info("- history file id of version is " +
version.getHistFileId())
        }

        d3.log.info("-- change reason of this version is " +
version.getChangeReason())
        d3.log.info("-- delete reason of this version is " +
version.getDeleteReason())
        d3.log.info("-- creator of this version is " + version.getCreator())
        d3.log.info("-- create date of this version is " +
version.getCreateDate())
        d3.log.info("-- release user of this version is " +
version.getReleaseUser())
        d3.log.info("-- release date of this version is " +
version.getReleaseDate())
        d3.log.info("-- archive user of this version is " +
version.getArchiveUser())
        d3.log.info("-- archive date of this version is " +
version.getArchiveDate())
        d3.log.info("-- verify user of this version is " +
version.getVerifier())
        d3.log.info("-- verify date of this version is " +
version.getVerifyDate())
        d3.log.info("-- delete user of this version is " +
```

```
version.getDeleter())
        d3.log.info("-- delete date of this version is " +
version.getDeleteDate())
        d3.log.info("-- external version id of this version is " +
version.getExternalVersionId())

        // When the current Version has a physical file attached, get access
        if (version.physicalVersion)
        {
            // Get the properties of the file
            d3.log.info("--- file id of this version is " +
version.physicalVersion.getFileId())
            d3.log.info("--- doc id of this version is " +
version.physicalVersion.getDocId())
            d3.log.info("--- status of this version is " +
version.physicalVersion.getStatus())
            d3.log.info("--- file extension of this version is " +
version.physicalVersion.getFileExtension())
            d3.log.info("--- file format of this version is " +
version.physicalVersion.getFileFormat())
            d3.log.info("--- file size of this version is " +
version.physicalVersion.getFileSize())
            d3.log.info("--- file localisation of this version is " +
version.physicalVersion.getFileLocalisation())
            d3.log.info("--- d3 hash of this version is " +
version.physicalVersion.getD3Hash())
            d3.log.info("--- file hash of this version is " +
version.physicalVersion.getFileHash())

            // Get all properties of depending files
            for (dependentFile in version.physicalVersion.dependentFiles)
            {
                // Get the properties of each depending file
                d3.log.info("---- file extesion of dependend file of this
version is " + dependentFile.getExtension())
                d3.log.info("---- doc id of dependend file of this version
is " + dependentFile.getDocId())
                d3.log.info("---- file id of dependend file of this version
is " + dependentFile.getFileId())
                d3.log.info("---- file format of dependend file of this
version is " + dependentFile.getFileFormat())
                d3.log.info("---- file localisation of dependend file of
this version is " + dependentFile.getFileLocalisation())
                d3.log.info("---- file size in byte of dependend file of
this version is " + dependentFile.getFileSize())
                d3.log.info("---- file size in kilobyte of dependend file
of this version is " + dependentFile.getFileSizeInKb())
                d3.log.info("---- d3 hash of dependend file of this version
is " + dependentFile.getD3Hash())
                d3.log.info("---- create date of dependend file of this
version is " + dependentFile.getProcDate())
                d3.log.info("---- export flag of dependend file of this
version is " + dependentFile.getFlagStorageExport())
                d3.log.info("---- export date of dependend file of this
version is " + dependentFile.getStorageExportDate())
```

```
                d3.log.info("---- file hash of dependend file of this
version is " + dependentFile.getD3FileHash())
                // ..
            }
        }
    } else if(version.getCategory() ==
DocumentVersion.Category.DOC_VERS_BLOCKED){
        d3.log.info("block user of this version is " +
version.getBlockUser())
        d3.log.info("block date of this version is " +
version.getBlockDate())
    } else if(version.getCategory() ==
DocumentVersion.Category.DOC_VERS_DELETED){
        d3.log.info("block user of this version is " +
version.getBlockUser())
        d3.log.info("block date of this version is " +
version.getBlockDate())
    } else {
        d3.log.info(version.getCategory())
    }
}
```

File versions (PhysicalVersion)

PhysicalVersion

```
public final class PhysicalVersion extends ArchiveObject
{
    enum FileLocalisation
          FILE_LOC_NO_FILE,
          FILE_LOC_DISK_ONLY,
          FILE_LOC_STORAGE_ONLY,
          FILE_LOC_DISK_AND_STORAGE,
          FILE_LOC_PROXY_PLACEHOLDER,
          FILE_LOC_PROXY_PENDING
        };
    public Integer            getFileId()
    public String             getDocId()
    public Document.DocStatus getStatus()
    public String             getFileExtension()
    public String             getFileFormat()
    public Long               getFileSize()
    public FileLocalisation   getFileLocalisation()
    public String             getD3Hash()
    public String             getFileHash()
    public SignatureInfo[]    getSignatureInfos()
    public DependentFile[]    getDependentFiles()
}
```

For an example, see Document versions (DocumentVersion)

Dependent files (DependentFile)

DependentFile

```
public class DependentFile extends ArchiveObject
{
    public String           getExtension()
```

```
    public String           getDocId()
    public Integer          getFileId()
    public String           getFileFormat()
    public FileLocalisation getFileLocalisation()
    public Long             getFileSize()
    public Integer          getFileSizeInKb()
    public String           getD3Hash()
    public Timestamp        getProcDate()
    public String           getExternalMedium()
    public String           getExpired()
    public String           getFlagStorageExport()
    public Timestamp        getStorageExportDate()
    public String           getD3FileHash()
 }
```

For an example, see Document versions (DocumentVersion)

Signatures (SignatureInfo)

SignatureInfo

```
public class SignatureInfo  extends ArchiveObject
{
    enum SigContentType
          SIG_DETACHED, SIG_EMBEDDED, SIG_ATTACHED,
SIG_EMBEDDED_AND_DETACHED,
          SIG_INVALID_CONTENT_TYPE;
        };
    public String         getExtension()
    public SigContentType getContent()
    public Integer        getReachedNumber()
    public Integer        getRequestedNumber()
    public Integer        getReachedLevel()
    public Integer        getRequestedLevel()
    public String         getTodo()
    public String         getDone()
    public String         getRemark()
}
```

System properties (DocumentSysValue)

DocumentSysValue

```
public class DocumentSysValue extends ArchiveObject
{
    public String         getDocId()
    public Integer        getFieldId()
    public String         getFieldName()
    public Integer        getFieldIndex()
    public void           setFieldIndex(int fieldIndex)
    public String         getFieldValue()
    public void           setFieldValue(String value)
    public String         toString()
}
```

Example

```
D3Interface d3 = getProperty("d3");
```

```
try {
    //Retrieve document object
    Document doc = d3.archive.getDocument("P000000001", "d3_groovy");
    //adding sysField "myField" to document
    doc.addSysField("myField")
    //adding / changing value for sysField "myField" at index position 1
    doc.sysField["myField"][1] = "value for sysField"
    //adding value for sysField "myField" at new free index position
    doc.sysField["myField"].add("another value for myField")
    //update document object
    doc.updateAttributes("d3_groovy")

    //Listing all SysValues asssined to document object
    doc.sysValues.eachWithIndex(){ it, row ->
        d3.log.info("doc id of sysField = " + it.getDocId() + " in row " +
row)
        d3.log.info("id of sysField = " + it.getFieldId() + " in row " +
row)
        d3.log.info("name of sysField = " + it.getFieldName() + " in row "
+ row)
        d3.log.info("idx of sysField = " + it.getFieldIndex() + " in row "
+ row)
        d3.log.info("value of sysField = " + it.getFieldValue() + " in row
" + row)
        d3.log.info("just print all values for sysField = " + it.toString()
+ " in row " + row)

    }
    d3.log.info("just print all values for document = " +
doc.sysValues.toString())

} catch (Exception ex){
    d3.log.error("error updating document object: " + ex.getMessage());
}
```

Notes (DocumentNote)

DocumentVersion

```
public class DocumentNote extends ArchiveObject
{
    public String     getMessage()
    public User       getUser()
    public Timestamp  getDateCreated()
}
```

Script example for outputting the notes of a document.

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document
import com.dvelop.d3.server.User
import com.dvelop.d3.server.DocumentNote
import java.text.SimpleDateFormat

D3Interface d3 = getProperty("d3")

Document doc = d3.archive.getDocument("P000000123")
```

```
println "Notes form the document <" + doc.id + "> :"

doc.notes.each  { note ->
   String noteCreated = new SimpleDateFormat("dd.MM.yyyy
HH:mm:ss").format(note.dateCreated);
   println "  User:" + note.user.realName + " Created: " + noteCreated + "
Note: " + note.message
}
```

Document type (DocumentType)

DocumentType

```
class DocumentType extends ArchiveObject
{
    public DocumentTypeAttribute getfield()    // Groovy Collection Support
für den Zugriff auf die Attribute der Dokumentart
    public String   getDefaultName()
    public String   getName()
    public String   getType()
    public boolean  getIsFolder()
    public boolean  getIsExportedToStorage()
    public boolean  getIsSystemDocType()
    public boolean  getIsDummyType()
    public boolean  getIsMultiType()
    public boolean  getIsTemplateType()
    public String   getArchiveId()
    public String   getDsearchCorpus()
    public int      getFieldNoByName(String attribName)
}
```

Example of accessing the properties of a document type.

```
import com.dvelop.d3.server.Document
import com.dvelop.d3.server.DocumentType
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.RepositoryField

DocumentType myDocumentType = d3.archive.getDocumentType("DDOKU");
d3.log.info("default name of document type is
"+myDocumentType.getDefaultName());
d3.log.info("name of document type is "+myDocumentType.getName());
d3.log.info("type of document type is "+myDocumentType.getType());
if(myDocumentType.getType() == "d"){
    d3.log.info("type of this document type is 'document'");
} else if(myDocumentType.getType() == "a"){
    d3.log.info("type of document type is 'folder'");
}
if(myDocumentType.getIsFolder() == true){
    d3.log.info("document type is a folder");
} else {
    d3.log.info("document type is a document");
}
if(myDocumentType.getIsExportedToStorage() == true){
    d3.log.info("document type is configured to be exported to a sec.
storage");
```

```
}
if(myDocumentType.getIsSystemDocType() == true){
    d3.log.info("document type is a system document type");
}
if(myDocumentType.getIsDummyType() == true){
    d3.log.info("document type is a dummy document type");
}
d3.log.info("archive id document type is "+myDocumentType.getArchiveId());
d3.log.info("d.3 search corpus of document type is
"+myDocumentType.getDsearchCorpus());
d3.log.info("field number of the property 'Dokumentname' of document type
is "+myDocumentType.getFieldNoByName("Dokumentname"));

// Go / interate through all fields
for (int i = 1; i <=89; i++) {
    try {
        RepositoryField repoField =
myDocumentType.field[i].getRepositoryField()

        d3.log.info("field infos for number "+ i)
        d3.log.info("repository text "+ repoField.text)
        d3.log.info("repository name "+ repoField.name)
        d3.log.info("repository prefered field number "+
repoField.preferedFieldNumber)
        d3.log.info("repository data type "+ repoField.dataType)
        d3.log.info("repository has a plausibility hook hunction assigned
"+ repoField.hasPlausibilityHookFunction())
        d3.log.info("repository has a values providing hook function
assigned "+ repoField.hasValuesProvidingHookFunction())
        d3.log.info("repository has a predifined value list assigned "+
repoField.hasPredefinedValues())

        if(myDocumentType.field[i].getIsMandatory() == true){
            d3.log.info("field is mandatory")
        } else {
            d3.log.info("field is not mandatory")
        }

        if(myDocumentType.field[i].getIsModifiable() == true){
            d3.log.info("field is modifiable")
        } else {
            d3.log.info("field is not modifiable")
        }

        if(myDocumentType.field[i].getIsHidden() == true){
            d3.log.info("field is hidden")
        } else {
            d3.log.info("field is not hidden")
        }

        if(myDocumentType.field[i].getViewInSearchMask() == true){
            d3.log.info("field is visibile in search mask")
        } else {
            d3.log.info("field is not visibile in search mask")
        }
```

```
        if(myDocumentType.field[i].getViewInImportMask() == true){
            d3.log.info("field is visibile in import mask")
        } else {
            d3.log.info("field is not visibile in import mask")
        }

        if(myDocumentType.field[i].getViewInResultSet() == true){
            d3.log.info("field is visibile in result set")
        } else {
            d3.log.info("field is not visibile in result set")
        }
    } catch (Exception ex){
        d3.log.info("field with number ${i} does not exist")
    }
}
```

Properties of a document type (DocumentTypeAttribute)

DocumentTypeAttribute

```
public final class DocumentTypeAttribute extends ArchiveObject
{
    public RepositoryField getRepositoryField()
    public RepositoryField getRepoField()
    public boolean         getIsMandatory()
    public boolean         getIsModifiable()
    public boolean         getIsHidden()
    public boolean         getViewInSearchMask()
    public boolean         getViewInImportMask()
    public boolean         getViewInResultSet()
}
```

User

User

```
public final class User extends ArchiveObject
{
    public enum MemberType {DIRECT, RECURSIVE};

    public String                getLongName()
    public String                getRealName()
    public String                getEmail()
    public String                getPhone()
    public String                getPlant()
    public String                getDepartment()
    public boolean               getIsCheckedOut()
    public boolean               getIsSysUser()
    public String                getCheckoutText()
    public String                getOptField(int idx)
    public String                getLdapDN()
    public UserGroup[]           getGroups()
    public boolean               isMemberOfGroup(String groupId)     //
check of direct member (MemberType.DIRECT)
    public boolean               isMemberOfGroup(String groupId,
MemberType memberType)
```

```
    public AuthorizationProfile[] getAuthorizationProfiles()
    public boolean                hasAuthorizationProfile(String profileId)
    public DocumentType[]         getDocTypes()
 }
```

Example of use in a Groovy script

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.User
import com.dvelop.d3.server.UserGroup
import com.dvelop.d3.server.AuthorizationProfile

// Get a user object
def user = d3.archive.getUser("dvelop")

// Reading some user properties
println "Die EMail-Adresse des Benutzers " + user.realName + " (" + user.id
+ ") lautet: " + user.email

// Get all groups, with the current user as a member
user.getGroups().each { group ->
   println group.id + " - " + group.name
}

// Check if user is member of a specific group
if(user.isMemberOfGroup("myGroup") == true){
   d3.log.info(user.id+" is a member of myGroup");
}

// Get all right profiles, with the current user as a member
user.getAuthorizationProfiles().each { profile ->
   println profile.id + " - " + profile.name
}

// Get all document types, to which the user has access to
user.getDocTypes().each { docType ->
   println docType.id + " - " + docType.name
}
```

User groups (UserGroup/UserOrUserGroup)

User / UserOrUserGroup

```
public final class UserOrUserGroup extends ArchiveObject
{
    public String      getDefaultName()
    public String      getLongName()
    public String      getDisplayName()
    public User        getUser()
    public UserGroup   getUserGroup()
}


public final class UserGroup extends ArchiveObject
{
    public enum MemberType {DIRECT, RECURSIVE};
```

```
    public String                 getDefaultName()
    public String                 getName()
    public UserOrUserGroup[]      getMembers()
    public UserOrUserGroup[]      getMembers(MemberType memberType)
    public boolean                isMemberOfGroup (String parentGroupId)
    public boolean                isMemberOfGroup (String parentGroupId,
MemberType memberType)
    public AuthorizationProfile[] getAuthorizationProfiles()
    public boolean                hasAuthorizationProfile(String profileId)
}
```

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.UserGroup
import com.dvelop.d3.server.UserOrUserGroup
import com.dvelop.d3.server.AuthorizationProfile

// Get user object
def userGroup = d3.archive.getUserGroup("GroupId");

println "Show group information for <" + userGroup.name + "> ";

// Get all users and groupy, which are members of the current Group
userGroup.getMembers(UserGroup.MemberType.RECURSIVE).each { userOrGroup ->
   println userOrGroup.id + " - " + userOrGroup.defaultName
}

// Get all right profiles with the current group
userGroup.getAuthorizationProfiles().each { profile ->
   println profile.id + " - " + profile.name
}
```

Datasets (PredefinedValueSet)

```
public final class PredefinedValueSet extends ArchiveObject
{
    public String  getName()
    public String  getDataType()
    public String  getSortFlag()
    public String  getValues(int valIdx)
}
```

Example of creating a dataset using the hook function Datasets

Property fields (RepositoryField)

RepositoryField

```
public final class RepositoryField extends ArchiveObject
{
    public String             getName()
    public String             getText()
    public String             getDataType()
    public boolean            getHasPredefinedValues()
    public Integer            getPreferedFieldNumber()
    public PredefinedValueSet getPredefinedValueSet()
    public boolean            getHasPlausibilityHookFunction()
    public boolean            getHasValuesProvidingHookFunction()
```

```
    public void                provideValuesForValueSet(List<Object> values)
}
```

Example of creating a dataset using the hook function Datasets

Authorization profile (AuthorizationProfile)

AuthorizationProfile

```
public final class AuthorizationProfile extends ArchiveObject
{
    public String getName()
}
```

1.9.2. d.3 SQL database (SqlD3Interface)

The d.3 SQL interface provides easy, Groovy-compliant access to the native database interface of the d.3 server.

No JDBC driver is required.

SqlD3Interface

```
    public interface SqlD3Interface {
        public int execute(String query, List<Object> params) throws
Exception;
        public int execute(String query) throws Exception;
        public GroovyRowResult firstRow(String sqlquery, List<Object>
params) throws Exception;
        public GroovyRowResult firstRow(String query) throws Exception;
        public List<GroovyRowResult> executeAndGet(String query,
List<Object> params) throws Exception;
        public List<GroovyRowResult> executeAndGet(String query,
List<Object> params, int maxRows) throws Exception;
        public List<GroovyRowResult> executeAndGet(String query,
List<Object> params, int offset, int maxRows) throws Exception;
        public List<GroovyRowResult> executeAndGet(String query) throws
Exception;
        public List<GroovyRowResult> executeAndGet(String query, int
maxRows) throws Exception;
        public List<GroovyRowResult> executeAndGet(String query, int
offset, int maxRows) throws Exception;
    }
```

Modifier and type	Method and description	Use case

int	**execute(String query, List<Object> params)**
	
	Execute an SQL command with bind variables (e.g. an **insert** or **update** command)
	
	The **execute()** method returns the number of affected lines after successful execution.
		List<String> queryParam = []
		
		//SQL statement
		
		String **sqlStamm = "update custom_customer set customerType = ? where customerType = ?"**
		
		//Set value for 1st bind variable (**customerType** = new value)
		
		queryParam.add("A")
		
		//Set value for 2nd bind variable **(customerType** = old value)
		
		queryParam.add("C")
		
		//Execute SQL statement
		
		int sqlResult = d3.sql.execute(sqlStamm, queryParam)
		
		//The number of affected lines is in the return value
		
		d3.log.info**(sqlResult + "affected lines")**
		
		Note: Only useful for insert and update statements.

int	**execute(String query)**
	
	Execute an SQL command without bind variables (e.g. an **insert** or **update** command)
	
	The **execute()** method returns the number of affected lines after successful execution.
		//SQL statement
		
		String **sqlQuery = "update custom_customer set customerType = 'A' where customerType = 'C'"**
		
		//Execute SQL statement
		
		int sqlResult = d3.sql.execute(sqlQuery)
		
		//The number of affected lines is in the return value
		
		d3.log.info**(sqlResult + "affected lines")**
		
		Note: Only useful for insert and update statements.
		
		Note: The use of bind variables is recommended for performance and security reasons.

GroovyRowResult	**firstRow(String sqlquery, List<Object> params)**
	
	Execute an SQL-SELECT-command with a Bind-variable.
	
	Only the first line of the result set is retrieved and returned.
		List<String> bindParam = []
		
		//SQL statement
		
		String **sqlQuery = "select customerName from custom_customer where customerNo = ?"**
		
		//Set value for 1st bind variable **(customerNo** = value)
		
		bindParam.add(10000)
		
		//Execute SQL statement
		
		GroovyRowResult resultFirstRow = d3.sql.firstRow(sqlQuery, bindParam)
		
		if(resultFirstRow.size() == 1){
		
		//There was (at least) one entry found
		
		d3.log.info**("Name = "+resultFirstRow.customerName)**
		
		} else {
		
		//No entry was found
		
		d3.log.info**("No entry for customerNo = 10000")**
		
		}
		
		Note: Only useful if it is ensured that only one result is returned/required. Even if the SQL statement returns several results, the number of lines is (**resultFirstRow.size() = 1**).

Modifier and type	Method and description	Use case

GroovyRowResult	**firstRow(String query)**
	
	Execute an SQL-SELECT-command without a Bind-variable.
	
	Only the first line of the result set is retrieved and returned.
		//SQL statement
		
		String **sqlQuery = "select customerName from custom_customer where customerNo = 10000"**
		
		//Execute SQL statement
		
		GroovyRowResult resultFirstRow = d3.sql.firstRow(sqlQuery)
		
		if(resultFirstRow.size() == 1){
		
		//There was (at least) one entry found
		
		d3.log.info**("Name = "+resultFirstRow.customerName)**
		
		} else {
		
		//No entry was found
		
		d3.log.info**("No entry for customerNo = 10000")**
		
		}
		
		Note: Only useful if it is ensured that only one result is returned/required. Even if the SQL statement returns several results, the number of lines is **(resultFirstRow.size() = 1**).
		
		Note: The use of bind variables is recommended for performance and security reasons.

List<GroovyRowResult>	**executeAndGet(String query, List<Object> params)**
	
	Execute an SQL-SELECT-command with a Bind-variable.
	
	All lines of the result set are retrieved and returned.
		List<String> bindParam = []
		
		//SQL statement
		
		String **sqlQuery = "SELECT * FROM firmen_spezifisch where kue_dokuart = ? and dok_dat_feld_1 = ?"**
		
		//Set value for 1st bind variable (**kue_dokuart** = value)
		
		bindParam.add("DDOKU")
		
		//Set value for 2nd bind variable **(database field_40** = value)
		
		bindParam.add("Credit")
		
		//Execute SQL statement
		
		List<GroovyRowResult> sqlResult = d3.sql.executeAndGet(sqlQuery, bindParam)
		
		//The number of affected lines is in the object **sqlResult**
		
		d3.log.info**(sqlResult.size() + "affected lines")**
		
		//Validate return
		
		if(sqlResult != null) {
		
		if(sqlResult.size() > 0) {
		
		//Scroll list if there is more than one affected line
		
		sqlResult.each{
		
		d3.log.info**("DokuID: "+it.doku_id+"/database field_2: "+it.database field_2)**
		
		}
		
		} else {
		
		//No line was found
		
		d3.log.info**("No results were found")**
		
		}
		
		}

Modifier and type	Method and description	Use case

List<GroovyRowResult>	**executeAndGet(String query, List<Object> params, int maxRows)**
	
	Execute an SQL-SELECT-command with a Bind-variable.
	
	The first maxRows lines of the result set are retrieved and returned.
		List<String> bindParam = []
		
		//SQL statement
		
		String **sqlQuery = "SELECT * FROM firmen_spezifisch where kue_dokuart = ? and dok_dat_feld_1 = ?"**
		
		//Set value for 1st bind variable (**kue_dokuart** = value)
		
		bindParam.add("DDOKU")
		
		//Set value for 2nd bind variable **(database field_40** = value)
		
		bindParam.add("Credit")
		
		//Execute SQL statement, maximum 100 are returned
		
		List<GroovyRowResult> sqlResult = d3.sql.executeAndGet(sqlQuery, bindParam, 100)
		
		//The number of affected lines is in the object **sqlResult**
		
		d3.log.info**(sqlResult.size() + "affected lines")**
		
		//Validate return
		
		if(sqlResult != null) {
		
		if(sqlResult.size() > 0) {
		
		//Scroll list if there is more than one affected line
		
		sqlResult.each{
		
		d3.log.info**("DokuID: "+it.doku_id+"/database field_2: "+it.database field_2)**
		
		}
		
		} else {
		
		//No line was found
		
		d3.log.info**("No results were found")**
		
		}
		
		}

List<GroovyRowResult>	**executeAndGet(String query, List<Object> params, int offset, int maxRows)** Execute an SQL-SELECT-command with a Bind-variable. Beginning with **offset**, maxRows lines of the result set are retrieved and returned.	Offset or paging is not currently supported

Modifier and type	Method and description	Use case

List<GroovyRowResult>	**executeAndGet(String query)**
	
	Execute an SQL-SELECT-command without a Bind-variable.
	
	All lines of the result set are retrieved and returned.
		//SQL statement
		
		String **sqlQuery = "SELECT * FROM firmen_spezifisch where kue_dokuart = 'DDOKU' and dok_dat_feld_1 = 'Audi'"**
		
		//Execute SQL statement
		
		List<GroovyRowResult> sqlResult = d3.sql.executeAndGet(sqlQuery)
		
		//The number of affected lines is in the object **sqlResult**
		
		d3.log.info(sqlResult.size() + "affected lines")
		
		//Validate return
		
		if(sqlResult != null) {
		
		if(sqlResult.size() > 0) {
		
		//Scroll list if there is more than one affected line
		
		sqlResult.each{
		
		d3.log.info("DokuID: "+it.doku_id+"/database field_2: "+it.database field_2)
		
		}
		
		} else {
		
		//No line was found
		
		d3.log.info("No results were found")
		
		}
		
		}
		
		Note: The use of bind variables is recommended for performance and security reasons.

List<GroovyRowResult>	**executeAndGet(String query, int maxRows)**
	
	Execute an SQL-SELECT-command without a Bind-variable.
	
	The first maxRows lines of the result set are retrieved and returned.
		//SQL statement
		
		String **sqlQuery = "SELECT * FROM firmen_spezifisch where kue_dokuart = 'DDOKU' and dok_dat_feld_1 = 'Audi'"**
		
		//Execute SQL statement, maximum 100 are returned
		
		List<GroovyRowResult> sqlResult = d3.sql.executeAndGet(sqlQuery, 10)
		
		//The number of affected lines is in the object **sqlResult**
		
		d3.log.info(sqlResult.size() + "affected lines")
		
		//Validate return
		
		if(sqlResult != null) {
		
		if(sqlResult.size() > 0) {
		
		//Scroll list if there is more than one affected line
		
		sqlResult.each{
		
		d3.log.info("DokuID: "+it.doku_id+"/database field_2: "+it.database field_2)
		
		}
		
		} else {
		
		//No line was found
		
		d3.log.info("No results were found")
		
		}
		
		}
		
		Note: The use of bind variables is recommended for performance and security reasons.

Modifier and type	Method and description	Use case
List<GroovyRowResult>	**executeAndGet(String query, int offset, int maxRows)** Execute an SQL-SELECT-command without a Bind-variable. Beginning with **offset**, maxRows lines of the result set are retrieved and returned.	Offset or paging is not currently supported

1.9.3. Client API (D3RemoteInterface)

D3RemoteInterface

```
    public interface D3RemoteInterface {
        // Parameter
        public Map<String, Object>          getImportParams();
        public void                         setExportParams(Map<String,
Object> exportParams);

        // Table
        public Iterable<Map<String,Object>> getImportTable(String[]
columnNames);
        public void
setExportTable(Iterable<Map<String,Object>> exportTable);
        // File (table binary)
        public ByteBuffer                   getImportBytes();
        public void                         setExportBytes(ByteBuffer
exportBytes);

        public void                         setReturnCode(int retCode);

        // d3fc header information
        public String                       getLanguage();
        public String                       getFunctionName();
        public String                       getUserName();
        public String                       getVersion();
        public String                       getServerId();
        public String                       getSourceIpAddress();
 }
```

The interface "D3RemoteInterface" can be used to implement your own d3fc API functions via Groovy (see chapter Plug-in interface for API functions).

The context information of a d3fc function call (d3fc header) can also be accessed. This means that if a hook function is executed in the context of a d.3 API function, information from the API call can be accessed.

Modifier and type	Method and description
Map<String, Object>	getImportParams() Receive the list of import parameters from the caller. Key of the map = name of the parameter Value of the map = value of the parameter

Modifier and type	Method and description
void	setExportParams(Map<String, Object> exportParams) Fill the list with the return values. Key in the map = name of the parameter Value in the map = value of the parameter
Iterable<Map<String,Object>>	getImportTable(String[] columnNames) Accept all values from the d3fc import table of the column names specified in the columnNames list. The column name is the key in the returned map.
void	setExportTable(Iterable<Map<String,Object>> exportTable) Sends the transferred list of maps as a d3fc export table. The column name is always the key in the map.
ByteBuffer	getImportBytes() Receive a binary object from the caller. (e.g. a file)
void	setExportBytes(ByteBuffer exportBytes) Return a binary object
void	setReturnCode(int retCode) Set the return value of the API function. If the method is not called, the default value "0" (success) is returned.
`String`	`getLanguage()` Language abbreviation with which the current API function was called
`String`	`getFunctionName()` The function name of the current API function
`String`	`getUserName()` d.3 user who called the current API function
`String`	`getVersion()` Client version information of the current API call
`String`	`getServerId()` Abbreviation of the d.3ecm repository of the current API call (e.g. 'P')
`String`	`getSourceIpAddress()` Client IP address of the caller of the current API function

```
import com.dvelop.d3.server.Document
import com.dvelop.d3.server.DocumentType
import com.dvelop.d3.server.Entrypoint
import com.dvelop.d3.server.User
import com.dvelop.d3.server.core.D3Interface

public class Test{
   @Entrypoint( entrypoint = "hook_insert_entry_10" )
//-----------------------
   public int showAppId( D3Interface d3, User user, DocumentType
docTypeShort, Document doc ){

      d3.log.error( "---------->>>>>>>" + d3.remote.getVersion());
      def appID = d3.remote.getVersion()[0..2];
      d3.log.error( "APP-ID---------->>>>>>>" + appID);
      return 0;
   } // end of showAppId
}// end of Test
```

> **Note**
> d3fc calls from a Groovy hook function are currently not intended. Local methods should be preferred to calls via the d.3 interface.

1.9.4. Server API functions (ScriptCallInterface)

ScriptCallInterface functions

The ScriptCallInterface methods apply to the JPL file functions from the d.3 server scripting API (old name before Version 8: d.3 server API). In this documentation you will find all these functions with parameter and return values. The global variables described are not supported.

ScriptCallInterface

```
public interface ScriptCallInterface {
        // Documents
        public int document_change_type ( String doc_type_short, String
doc_id, String user_name);
        public int document_delete (String reason, boolean
del_from_each_status, boolean del_file_always, String doc_id, String
user_name, boolean del_privileged);
        public String document_get_permission (String doc_id, String
user_name);
        public int document_block (boolean block, String user_name, String
doc_id);
        public int document_verify (String user_name, String doc_id);
        public int document_transfer (String destination, String
new_editor, String change_remark, boolean asynchronous, int archiv_index,
String user_name, String doc_id);
        public int document_publish_for_web (boolean publish, String
doc_id, String user_name);
        // -- Notes
        public int note_add_file(String file_name, String doc_id, String
user_id);
        public int note_add_string(String line, String doc_id, String
user_id);
        //  Links
        public String[] link_get_parents (String doc_id, String user_name);
        public String[] link_get_children (String doc_id, String user_name);
        public int      link_documents (String doc_id_parent, String
doc_id_child, String user_name, boolean folder_definition, boolean
use_folder_plan);
        public boolean  link_exists (String doc_id_parent, String
doc_id_child, boolean test_vice_versa);
        public int      link_delete (String doc_id_parent, String
doc_id_child, String user_name);
        //  dependent files
        public int document_dependent_add (String filename, String
doc_status, String doc_ext, String doc_id, String user_name);
        public int document_dependent_delete (String doc_status, String
doc_ext, String doc_id, String user_name);
        public int document_start_lifetime (String doc_id, boolean
overwrite_old_date, int life_time_days);
        public int document_set_cache_days (String doc_id, String
doc_status, int archive_index, int days_in_cache);
```

```
        public int folder_create (Document doc);
        public String document_get_file_path (String doc_id, String
doc_status, int archive_index, String user_group, String dependent_ext);
        public int document_send_to_dsearch (String doc_id, String
ocr_file, int version, String dsearch_corpus, boolean use_existent_ocr_file,
                String doc_status, int archive_index, String user_name);
        public int restore_from_jukebox (String doc_status, int
archiv_index, String doc_id, String user_name);
        public int restore_from_jukebox (String doc_status, int
archiv_index, String doc_id, String user_id, String category, String
no_validation, boolean to_cache_only);
        public int restore_from_history (int aktion_id, String doc_id,
String user_name);
        //  Right inheritance:
        public int add_inherit_doc_rights (String doc_id, String granter,
String grantee, String right_flags, Timestamp tstamp_expire);
        public int remove_inherit_doc_rights (String doc_id, String
user_name, String grantee);
        // Inbox / resubmission
        public int hold_file_send (String recipient, String notice, String
doc_id, Timestamp tstamp_acknowledge, Timestamp tstamp_remember,
                boolean expand_groups, boolean ignore_checkout, Timestamp
date_activate, String type, String sender, int chain_id, boolean
remove_immediately,
                boolean inherit_class_rule, Timestamp inherit_class_tstamp,
int inherit_class_right, boolean check_write_access);
        public String[] hold_file_find (String recipient, String doc_id,
String user_name);
        public int      hold_file_delete (long chain_id, Byte
sent_received, boolean workflow_only, String recipient, String doc_id,
String user_name);
        // Workflow:
        public int workpath_end_document (String doc_id, String user_name,
boolean delete_jobs);
        public int workpath_start_document (String wfl_id, String doc_id,
String user_name);
        public int workpath_go_to_next_step (int exit_value, String
next_step_id, String doc_id, String user_name);
        // Authorization profiles:
        public String[] roll_get_names ();
        public String[] roll_get_users (long roll_id, String roll_name);
        // TIFF / PDF functions
        public int document_render (String source, String destination, int
render_option, boolean ocr, boolean asynchronous,
                boolean replace_doc, boolean overwrite, String doc_id,
String user_name, String doc_status, int archiv_index, String prio);
        public int tiff_concat (String source, String destination, String
source_rdl, String destination_rdl);
        public int document_render_wfl_prot (String doc_id, String
user_name);
        // Object properties
        public int object_property_set (String property_name, String
object_id, int object_class_id, String property_value);
        // Lock token
        public int lock_token_acquire (String object_id, String
```

```
object_name, String token, String object_info, int ttl, String user_name);
        public int lock_token_release (String object_id, String
object_name, String token);
        // Restriction sets:
        public int d3set_add_filter (String user_name, String doc_id,
String set_name, String object_id, String filter, boolean overwrite);
        public int d3set_remove_filter (String user_name, String doc_id,
String set_name, String object_id, String filter);
        public int d3set_remove_set (String user_name, String doc_id,
String set_name, String object_id);
        // Async jobs
        public int d3async_job_open(String doc_id_ref, String job_type,
String user_name);
        public int d3async_job_set_attribute(String attr_name, String
attr_value, int attr_type);
        public int d3async_job_set_lin002_attribute(String attr_type,
String attr_name, String attr_value);
        public int d3async_job_close();
        // Various / Miscellaneous
        public int send_email (String recipient, String notice,
Timestamp date, String doc_id, String user_name, String body_file, String
mail_format, boolean attach,
                               String doc_status, int archiv_index, String
attach_abh, String attach_file);
     }
```

```
def callScriptFunction(D3Interface d3, Document doc, reposField, def user)
{
    def sqlQuery  = "SELECT note FROM notes_table WHERE doc_id = ?";
    def resultRows = d3.sql.executeAndGet(sqlQuery, [doc.id]);

    resultRows.each
    {
       d3.log.info("Add note $it.note to document $doc.caption ");
       d3.call.note_add_string(it.note, doc.id, user.id);
    }
}
```

Below you will find descriptions, parameters and return values for the functions:

document_change_type

Description: This function changes the document type.

Parameter:

- **doc_type_short** (mandatory field): Short name of the new document type
- **doc_id** (mandatory field): ID of the document whose document type is to be changed
- **user_name**: Name of the user making the changes

Return values:

- **0**: Everything OK
- **-1**: Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-3**: Unknown error, please contact d.velop Support
- **-101**: Either document ID not passed or context not set

- **-104**: Document type short name not specified
- **-201**: Invalid user name specified
- **-205**: Invalid document type short name specified
- **-301**: Invalid document ID specified or user does not have permissions
- **-302**: Error when determining user permissions
- **-303**: User does not have access permissions for the document

Example:

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String doc_type_short = "DRECH";
String doc_id = "P000000001"
String user_name = "d3_groovy";

def error = d3.call.document_change_type(doc_type_short, doc_id, user_name)
if (error){
  d3.log.info("$error while changing document type")
}
else {
  d3.log.info("Changing document type successful!")
}
```

document_delete

Description: This function deletes documents from d.3.

Parameter:

- **reason**: Reason for wipe, necessary for Release/Archive version. Mandatory field for **Release**/**Archive** status, else optional.
- **del_from_each_status** (optional): Delete from all statuses: **false** = only the current version, **true** = all versions, default: **false**
- **del_file_always** (optional): Delete document file regardless of status: **false** = only delete document files in Processing/Verification, **true** = delete document file in any case, default: **false**
- **doc_id** (mandatory field): ID of document to be deleted: If nothing is passed, the context ID is used.
- **user_name** (optional): User for whom the document is to be deleted: if nothing is specified, the executing user is used.
- **del_privileged** (optional): If set to **true** , the executing user's corresponding license or permission for privileged deletion is verified. If the permission does not exist, the errors **319** and **320** appear.

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-3**: Unknown error, please contact d.velop Support
- **-101**: Either document ID not passed or context not set
- **-120**: Reason for wipe not specified
- **-201**: Invalid user name specified
- **-301**: Invalid document ID specified or user does not have permissions
- **-302**: Error when determining user permissions

- **-303**: User does not have access permissions for the document
- **-311**: User does not have permission to delete the document
- **-319**: No license to delete privileges (new in version 7.0)
- **-320**: User does not have permission to delete privileges (new in version 7.0)
- **< -9500**: Error in customer-specific hook function, please see d.3 logviewer for details

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String reason = "Wrong file uploaded";
boolean del_from_each_status = true;
boolean del_file_always = false;
String doc_id = "P000000001";
String user_name = "d3_groovy";
boolean del_privileged = false;

def error = d3.call.document_delete(reason, del_from_each_status,
del_file_always, doc_id, user_name, del_privileged)
if (error){
  d3.log.info("$error while deleting document")
}
else {
  d3.log.info("Deleting document successful!")
}
```

document_get_permission

Description: This function defines a user's permissions to query a document. The specification is written into the **api_single_info** in the form described in the API call **SearchDocument(doc_permission)**: **1**: Yes, **0**: No.

Parameter:

- **doc_id** (mandatory field): ID of the document whose permissions are to be changed. If nothing is specified, the context ID is used. Either as a parameter or context **api_doc_id**.
- **user_name** (optional): User whose permissions are to be queried. if nothing is specified, the executing user is used.

Return values: The feature specifies a user's query permissions for a document. The permissions are returned as a string: **a1a2a3a4a5a6a7a8-b1b2b3b4b5b6b7b8-c1c2-d1d2**.

Parameters	Description
a1	Access to document type allowed
a2	Note file available
a3	Document links available
a4	User file available
a5	User file encrypted
a6	Released version available (**1**: released, **2**: locked)
a7	Last version in Verification (**1**: verified, **2**: unverified)
a8	At least one version in **Archive** status
b1	Permission to display usage data
b2	Permission to change identification data
b3	Permission to change user data
b4	Permission to delete the document

Parameters	Description
b5	Permission for **Transfer** status
b6	Permission to review the document
b7	Permission to release/block the document
b8	Permission to perform both review steps (**Verification** and **Release**) in one step
c1	Permission to display an older version in Release
c2	Permission to release/block an older version in Release
d1	Permission to view versions in **Archive** status
d2	Document is in the workflow

- **1** Panther syntax error, for details please see d.3 logviewer
- **-3**: User does not exist

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String doc_id = "P000000001";
String user_name = "d3_groovy";

String returnValue = d3.call.document_get_permission(doc_id, user_name)
if (returnValue.length() >= 45){
  d3.log.info("Current permissions for user $user_name are $returnValue")
  if(returnValue.substring(0,1) == "1"){
        d3.log.info("Access to document type allowed")
  } else {
    d3.log.info("Access to document type not allowed")
  }
}
else {
  d3.log.info("$error while getting document permission")
}
```

document_block

Description: This function blocks or unblocks a document.

Parameter:

- **block** (optional):
  - **true**: Unblock document
  - **false**: Block document (default)
- **user_name** (optional): User blocking or unblocking document. If nothing is specified, the user from the context is used.
- **doc_id** (mandatory field): Document to be transferred. If nothing is specified, the ID from the context is used.

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-3**: Unknown error, please contact d.velop Support

- **-101**: Either document ID not passed or context not set
- **-201**: Invalid user name specified
- **-301**: Invalid document ID specified or user does not have permissions
- **-302**: Error when determining user permissions
- **-303**: User does not have access permissions for the document
- **-306**: The user does not have permission to block or unblock this document.
- **-506**: There is no released version.
- **-507**: Version is already blocked in Release
- **-508**: Version is not blocked in Release
- **< -9500**: Error in customer-specific hook function, please see d.3 logviewer for details

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

boolean block = true;
String user_name = "d3_groovy";
String doc_id = "P000000001";

def error = d3.call.document_block(block, user_name, doc_id)
if (error){
  d3.log.info("$error while blocking document ")
}
else {
  d3.log.info("Blocking document successful!")
}
```

document_verify

Description: This function moves a document from **Verification, unverified** status to **Verification, verified** status.

Parameter:

- **user_name** (optional): User who is to perform the verification. If nothing is specified, the user from the context is used.
- **doc_id** (mandatory field): Document to be transferred. If nothing is specified, the ID from the context is used. Either as a parameter or context **api_doc_id**.

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-3**: Unknown error, please contact d.velop Support
- **-101**: Either document ID not passed or context not set
- **-201**: Invalid user name specified
- **-301**: Invalid document ID specified or user does not have permissions
- **-302**: Error when determining user permissions
- **-303**: User does not have access permissions for the document
- **-307**: User does not have permission to verify this document
- **-509**: Version in **Verification** status is already verified
- **-510**: There is no version in **Verification** status.

- **< -9500**: Error in customer-specific hook function, please see d.3 logviewer for details

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String user_name = "d3_groovy";
String doc_id = "P000000001";

def error = d3.call.document_verify(user_name, doc_id)
if (error){
  d3.log.info("$error while verifying document ")
}
else {
  d3.log.info("Verifying document successful!")
}
```

document_transfer

Description: This function performs a status transfer. The current version is used unless an archived version is available and an **archiv_index** is specified.

Parameter:

- **destination** (mandatory field): Destination status **Processing**, **Verification**, **Release**, **Archive**
- **new_editor**: User/group name required for destination = **Processing** (user or group); destination = **Verification** (group). Mandatory field if the **Processing** destination status was specified for the **destination** parameter.
- **change_remark**: Must be specified if a document is transferred from Processing to Verification and the document has been changed from a previous version in **Processing** or **Archive** status.
- **asynchronous** (optional): Specifies whether the transfer is to be executed immediately or asynchronously using an asynchronous job for d.3 async**(false**: immediately (default); **true**: asynchronously by d.3 async).
- **archiv_index** (optional): Index for the version to be restored in the **Archive** status
- **user_name** (optional): User who is to perform the transfer. If nothing is specified, the user from the context is used.
- **doc_id** (mandatory field): Document to be transferred

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-3**: Unknown error, please contact d.velop Support
- **-101**: Either document ID not passed or context not set
- **-113**: Group name not specified
- **-113**: No longer appears with status verification
- **-114**: Destination status not specified
- **-115**: Archive index not specified
- **-116**: Processing comment not specified
- **-201**: Invalid user name specified
- **-213**: Invalid group name specified
- **-214**: Invalid destination status specified
- **-301**: Invalid document ID specified or user does not have permissions

- **-302**: Error when determining user permissions
- **-303**: User does not have access permissions for the document
- **-308**: User does not have permission to release this document
- **-309**: User does not have permission to transfer the document
- **-310**: User does not have permission to view older versions in the **Archive** status
- **-511**: This transfer is not allowed
- **-512**: There is a newer version. To overwrite the version, it must be moved to the **Processing** status.
- **-513**: Error when creating the job
- **-514**: Error when comparing the old and new version of the document file
- **< -9500**: Error in customer-specific hook function, **hook_transfer_<name of hook function>**

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String destination = "Bearbeitung"
String new_editor = "myUser"
String change_remark = ""
boolean asynchronous = false;
int archiv_index = 0;
String user_name = "d3_groovy";
String doc_id  = "P000000001";

def error = d3.call.document_transfer(destination, new_editor,
change_remark, asynchronous, archiv_index, user_name, doc_id)
if (error){
  d3.log.info("$error while transferring document ")
}
else {
  d3.log.info("Transferring document successful!")
}
```

document_publish_for_web

Description: Marks a document as published on the Web.

Parameter:

- **publish**:
  - **0**: Mark document as not published on the Web
  - **1**: Mark document as published on the Web (default)
- **doc_id**: Document to be published
- **user_name**: Publishing user

Return values:

- **0**: Everything OK
- **-1**: Panther syntax error, for details please see d.3 logviewer
- **-614**: Document type not defined for web publication

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

boolean publish = true;
```

```
String doc_id  = "P000000001";
String user_name = "d3_groovy";

def error = d3.call.document_publish_for_web(publish, doc_id, user_name)
if (error){
  d3.log.info("$error while publishing document for web")
}
else {
  d3.log.info("Publishing document for web successful!")
}
```

note_add_file

Parameter:

- **file_name** (mandatory field): Name of file whose content is to be attached to the note
- **doc_id** (mandatory field): ID of the document whose note file is to be changed. If nothing is specified, the context ID is used. Either as a parameter or context **api_doc_id**.
- **user_id** (optional): User who the note is to be written for. if nothing is specified, the executing user is used.

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-3**: Unknown error, please contact d.velop Support
- **-101**: Either document ID not passed or context not set
- **-150**: File name not specified
- **-201**: Invalid user name specified
- **-250**: File not found
- **-301**: Invalid document ID specified or user does not have permissions
- **-302**: Error when determining user permissions
- **-303**: User does not have access permissions for the document
- **-401**: The file could not be encrypted/decrypted.
- **-402**: Encrypted/decrypted file could not be renamed correctly
- **-403**: Encrypted/decrypted temporary file could not be deleted
- **< -9500**: Error in customer-specific hook function

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String file_name = "C:\\temp\\myNote.txt";
String user_name = "d3_groovy";
String doc_id  = "P000000001";

def error = d3.call.note_add_file(file_name, doc_id, user_name)
if (error){
  d3.log.info("$error while adding note")
}
else {
  d3.log.info("Adding note successful!")
}
```

note_add_string

Description: This function attaches a file's content to the document's note file.

Parameter:

- **line** (mandatory field): Line that is to be attached to the note
- **doc_id** (mandatory field): ID of the document whose note file is to be changed. If nothing is specified, the context ID is used. Either as a parameter or context **api_doc_id**.
- **user_id** (optional): User who the note is to be written for. if nothing is specified, the executing user is used.

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-3**: Unknown error, please contact d.velop Support
- **-101**: Either document ID not passed or context not set
- **-111**: No line specified to insert
- **-201**: Invalid user name specified
- **-301**: Invalid document ID specified or user does not have permissions
- **-302**: Error when determining user permissions
- **-303**: User does not have access permissions for the document
- **-401**: The file could not be encrypted/decrypted.
- **-402**: Encrypted/decrypted file could not be renamed correctly
- **-403**: Encrypted/decrypted temporary file could not be deleted
- **< -9500**: Error in customer-specific hook function, **hook_transfer_<name of hook function>**

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String line = "myNote";
String user_name = "d3_groovy";
String doc_id = "P000000001";

def error = d3.call.note_add_string(line, doc_id, user_name)
if (error){
  d3.log.info("$error while adding note")
}
else {
  d3.log.info("Adding notesuccessful!")
}
```

link_get_parents

Description: This function determines the parent documents and returns the relevant document IDs.

Parameter:

- **doc_id**: ID of the document whose permissions are to be queried
- **user_name**: User who the links are to be determined for

Return values:

Array of document IDs

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

String doc_id = "P000000001";
String user_name = "d3_groovy";

String[] parents = d3.call.link_get_parents(doc_id, user_name);
for(String parent : parents){
  d3.log.info("link " + parent + "/" + doc_id)
  def error = d3.call.link_delete(parent, doc_id, user_name);
  if (error){
    d3.log.info("$error while deleting link")
  }
  else {
    d3.log.info("Deleting link successful!")
  }
}
```

link_get_children

Description: This function determines the child documents and returns the relevant document IDs.

Parameter:

- **doc_id**: ID of the document whose child documents are to be determined
- **user_name**: User who the links are to be determined for

Return values:

Array of document IDs

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String doc_id = "P000000001";
String user_name = "d3_groovy";

String[] children = d3.call.link_get_children(doc_id, user_name);
for(String child : children){
  d3.log.info("link " + child + "/" + doc_id)
  def error = d3.call.link_delete(doc_id, child, user_name);
  if (error){
    d3.log.info("$error while deleting link")
  }
  else {
    d3.log.info("Deleting link successful!")
  }
}
```

link_documents

Description: This function links two documents.

Parameter:

- **doc_id_parent**: ID of the parent document
- **doc_id_child**: ID of the child document
- **user_name**: User who the documents are to be linked for
- **folder_definition**: **true**: Create dossier if this does not exist
- **use_folder_plan**: If **doc_id_parent** is not specified; **true**: Use dossier creation for the document

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-3**: Unknown error, please contact d.velop Support
- **-101**: Either document ID not passed or context not set
- **-201**: Invalid user name specified
- **-219**: Parent and child document ID are identical
- **-301**: Invalid document ID specified or user does not have permissions
- **-302**: Error when determining user permissions
- **-303**: User does not have access permissions for the document
- **-312**: User does not have permission to link/unlink the documents
- **-520**: The documents are already linked.
- **-521**: The documents are already linked reversely.
- **< -9500**: Error in customer-specific hook function **hook_link**

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String doc_id_parent = "P000000001"
String doc_id_child = "P000000002"
String user_name = "d3_groovy"
Boolean folder_definition = false
Boolean use_folder_plan = false

def error = d3.call.link_documents(doc_id_parent, doc_id_child, user_name,
folder_definition, use_folder_plan)
if (error){
  d3.log.info("$error while linking documents")
}
else {
  d3.log.info("Linking documents successful!")
}
```

link_exists

Description: This function checks whether two documents are linked.

Parameter:

- **doc_id_parent**: ID of the parent document
- **doc_id_child**: ID of the child document
- **test_vice_versa**: Only **false** is currently supported

Return values:

- **False**: Documents are not linked
- **True**: Documents are linked
- **> 2**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-101**: Either document ID not passed or context not set for parent or child document
- **-219**: Parent and child document ID are identical

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String doc_id_parent = "P000000001"
String doc_id_child = "P000000002"
Boolean test_vice_versa = true

def error = d3.call.link_exists(doc_id_parent, doc_id_child,
test_vice_versa)
if (error == true){
  d3.log.info("The documents are linked")
} else if (error == false){
  d3.log.info("The documents are not linked")
} else {
  d3.log.info("$error while checking link existence")
}
```

link_delete

Description: This function deletes the link between a parent and child document.

Parameter:

- **doc_id_parent**: ID of the parent document
- **doc_id_child**: ID of the child document
- **user_name**: User who the link is to be determined for

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-3**: Unknown error, please contact d.velop Support
- **-101**: Either document ID not passed or context not set for parent or child document
- **-201**: Invalid user name specified
- **-219**: Parent and child document ID are identical
- **-301**: Invalid document ID specified or user does not have permissions
- **-302**: Error when determining user permissions
- **-303**: User does not have access permissions for the document
- **-312**: User does not have permission to link/unlink the documents
- **-522**: The documents are not linked in this way.
- **< -9500**: Error in customer-specific hook function **hook_unlink**

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document
```

```
D3Interface d3 = getProperty("d3")

String doc_id = "P000000001";
String user_name = "d3_groovy";

String[] children = d3.call.link_get_children(doc_id, user_name);
for(String child : children){
  d3.log.info("link " + child + "/" + doc_id)
  def error = d3.call.link_delete(doc_id, child, user_name);
  if (error){
    d3.log.info("$error while deleting link")
  }
  else {
    d3.log.info("Deleting link successful!")
  }
}
```

document_dependent_add

Description: Saves a dependent document or overwrites an existing dependent document.

Parameter:

- **filename**: File path for new dependent file
- **doc_status**: Status of the dependent document to be saved (**B** for **Processing**, **P** for **Verification**, **F** for **Release**); default: Latest version
- **doc_ext**: Dependent file extension (default: file name extension)
- **doc_id**: Master document ID
- **user_name**: Name of executing user

Return values:

- **0**: Everything OK
- **-1**: Interpreter syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-101**: No document ID specified
- **-150**: File name not specified
- **-214**: Invalid status specified (must be **B**, **P** or **F**)
- **-250**: File not found
- **-313**: User may not change document files
- **-350**: Revoke access to files
- **-555**: Error when renaming the file
- **-556**: Error when copying the file
- **-617**: Error when querying the current status in the database
- **-670**: This function can no longer save a signature file (**.s1**) for a document.

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String filename = "C:/temp/depDoc.pdf"
String doc_status = "F"
String doc_ext = "P1"
String doc_id = "P000000001"
```

```
String user_name = "d3_groovy"

def error = d3.call.document_dependent_add(filename, doc_status, doc_ext,
doc_id, user_name)
if (error){
  d3.log.info("$error while adding dependent document")
}
else {
  d3.log.info("Adding dependent document successful!")
}
```

document_dependent_delete

Description: This function deletes a dependent document.

Parameter:

- **doc_status**: Status of the dependent document to be deleted
  - Possible statuses:
    - **B** for **Processing**
    - **P** for **Verification**
    - **F** for **Release**
  - Default: current version
- **doc_ext**: Data extension of the dependent file
- **doc_id**: Document ID of the dependent document
- **user_name**: Name of executing user

Return values:

- **0**: Everything OK
- **< 0**: Database error
- **1** Syntax error in interpreter, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-101**: No document ID specified
- **-139**: No extension of the dependent file (**doc_ext**) specified
- **-214**: Invalid status specified (must be **B**, **P** or **F**)
- **-313** User may not change document files
- **-350**: Revoke access to files
- **-617** Error when querying the current status in the database

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String doc_status = "F"
String doc_ext = "P1"
String doc_id = "P000000001"
String user_name = "d3_groovy"

def error = d3.call.document_dependent_delete(doc_status, doc_ext, doc_id,
user_name)
if (error){
  d3.log.info("$error while deleting dependent document")
}
else {
```

```
  d3.log.info("Deleting dependent document successful!")
}
```

document_start_lifetime

Description: The function defines the lifespan for a document.

Note: If you import a new document version, the previously defined duration is reset and must be set again.

Note: If secondary storage is used, the files for the document on the secondary storage medium cannot be reduced or removed.

Parameter:

- **doc_id**: Document ID
- **overwrite_old_date**: Overwrites the current value (default: **true**)
- **life_time_days**: If the lifetime configured for the document type is not to be used, you can specify the number of days (default: **0** – configured lifetime, **-1** – unlimited retention period).

Return values:

- **0**: Everything OK
- **-1**: Syntax error in interpreter, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-621**: It is not intended to change the document type's lifetime and specify **0** for the **overwrite_old_date** parameter.
- **-622**: Database error, for details please see d.3 logviewer

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String doc_id = "P000000001"
boolean overwrite_old_date = true
int life_time_days = 3650

def error = d3.call.document_start_lifetime(doc_id, overwrite_old_date,
life_time_days)
if (error){
  d3.log.info("$error while starting lifetime")
}
else {
  d3.log.info("Starting lifetime successful!")
}
```

document_set_cache_days

Description: This function sets the number of days for a document version to remain in the d.3 document tree "from today". The number of days is only relevant if the document has been or will be saved to a secondary storage.

Parameter:

- **doc_id**: Document ID
- **doc_status**: Document status
  - Possible statuses:

    - **Release** (**Fr**)
    - **Archive**
  - Default: **Release** (**Fr**)
- **archive_index** (mandatory field): Index of the archive version (mandatory for the **Archive** status); is not considered with Release
- **days_in_cache** (mandatory field): Number of days the document version is to remain in the cache

Return values:

- **0**: Everything OK
- **-1**: Syntax error in interpreter, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-115**: No index or no archive version specified for the **Archive** status
- **-223**: Invalid status specified (must be **F** or **A**)
- **-623**: Number of days the document version is to remain in the d.3 document tree missing
- **-624**: Document version does not exist or is already deleted in the d.3 document tree

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String doc_id = "P000000001"
String doc_status = "Fr"
int archive_index = 0
int days_in_cache = 10

def error = d3.call.document_set_cache_days(doc_id, doc_status,
archive_index, days_in_cache)
if (error){
  d3.log.info("$error setting cache days")
}
else {
  d3.log.info("Setting cache days successful!")
}
```

folder_create

Description: This function is used to create new dossiers.

Parameter:

doc: Document item with all properties for the new dossier

Return values:

- **0**: Everything OK
- **-548**: Error when creating the dossier, for details please see d.3 logviewer

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

def doc = d3.archive.newDocument();

doc.type = "APERS"
```

```
doc.status = Document.DocStatus.DOC_STAT_RELEASE
doc.editor = "d3_groovy"
doc.setText(1, "Comment text row 1")
doc.setNumber("myNumber2")

doc.field[1] = "folder_create - Attrib 1"
doc.field[2] = "folder_create - Attrib 2"
doc.field[4] = "folder_create - Attrib 4"
...
doc.field[60][1] = "folder_create - Attrib 60-1"
...

def error = d3.call.folder_create(doc)
if (error) {
  d3.log.info("$error while folder creation!")
}
else {
  d3.log.info("Folder creation successful!")
}
```

document_register_dependent (Deprecated)

Description: This function registers a document's dependent files.

Parameter:

- **doc_id**: Original document which the dependent documents are to be registered to
- **archive_index**: The archive version index is required for the **Archive** status
- **status**: Document status (**Processing**, **Release**, **Verification** or **Archive**)
- **user_group**: When passed, this parameter precedes the actual document editor specified in **doc_id**. The parameter is only for the **Processing** and **Verification** status.

Return values:

- **0**: Everything OK
- **-1**: Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly

document_get_file_path

Description: This function returns a document's full path in the d.3 document tree.

Parameter:

- **doc_id**: Original document which the path is to be determined for
- **doc_status**: Document status (**B** for **Processing**, **P** for **Verification**, **F** for **Release**)
- **archive_index**: Index of the archive version (mandatory for the **Archive** status)
- **user_group**: When passed, this parameter precedes the actual document editor specified in **doc_id**. The parameter is only for the **Processing** and **Verification** status.
- **dependent_ext**: If it is a dependent file, specify the file extension.

Return values:

- **0**: Everything OK
- **-1**: Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-115**: No index or no archive version specified for the **Archive** status
- **-126**: Either document ID not passed or context not set

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

String doc_id = "P000000001"
String doc_status = "F"
int archive_index = 0
String user_group = null
String dependent_ext = ""

def pathOfDocument = d3.call.document_get_file_path(doc_id, doc_status,
archive_index, user_group, dependent_ext)
if (pathOfDocument){
  d3.log.info("Getting path successful! -> " + pathOfDocument)
}
```

document_send_to_dsearch

Description: This function sends OCR information for a document version to d.3 search.

Parameter:

- **doc_id**: Document that is to be updated in d.3 search
- **ocr_file**: OCR file with the OCR information. If properties are to be transferred or **use_existent_orc_file** is enabled, the parameter can be left empty.
- **version**: Version of the document to be changed. If no version is specified, the latest version is used. The constants do not have to be passed for note files.
- **dsearch_corpus**: Corpus which the document is to be passed to. You must set the parameter if an existing document is to be passed to another corpus or new corpus.
- **use_existent_ocr_file**: Set the parameter to use the existing OCR files. You can pass existing documents to the d.3 search again using the parameter, for example to restore them, for a new full-text machine (default: **false**).
- **doc_status**: Status of the document version to be updated. If no version is specified, the latest version of the document is used (**B** for **Processing**, **P** for **Verification**, **F** for **Release**).
- **archive_index**: Archive index of archive version. The parameter only has to be passed if the **Archive** value is set for the **doc_status** parameter.
- **user_name**: Name of executing user

Return values:

- **0**: Everything OK
- **-1**: Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-101**: No document ID specified
- **-313** User may not change document files
- **-660**: no value for **DSEARCH_SUPPORT**
- **-661**: Error when creating the OCR directory
- **-662**: Error when copying the OCR file
- **-663**: d.3 search reports an error, please see d.3 logviewer for the error number

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String doc_id = "P000000001"
String ocr_file = null
int version = 0
```

```
String dsearch_corpus = ""
boolean use_existent_ocr_file = false
String doc_status = "B"
int archive_index = 0
String user_name = "d3_groovy"

def error = d3.call.document_send_to_dsearch(doc_id, ocr_file, version,
dsearch_corpus, use_existent_ocr_file, doc_status, archive_index, user_name)
if (error){
  d3.log.info("$error while sending to d.3 search")
}
else {
  d3.log.info("Sending to d.3 search successful!")
}
```

restore_from_jukebox

Description: This function restores a file from the secondary storage.

Parameter:

- **doc_status**: Status of the document version to be restored (**F** for **Release**, **A** for **Archive**)
- **archive_index**: Archive index of archive version. The parameter only has to be passed if the **Archive** value is set for the **doc_status** parameter.
- **doc_id**: Document to be restored from the secondary storage
- **user_id** : ID of executing user

Return values:

- **0**: Everything OK
- **-1**: Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-3**: Unknown error
- **-101**: No document ID specified
- **-126**: Either document ID not passed or context not set
- **-201**: Invalid user ID
- **-223**: Invalid document status
- **-301**: Invalid document ID or missing permissions
- **-302**: Error when retrieving the user permission
- **-303**: User does not have permissions
- **-310**: User does not have permissions for archive version
- **-506**: No public version available
- **-531**: No archive version available
- **-532**: Document path cannot be determined
- **-533**: File is already in the document tree, restore not necessary
- **-534**: d.ecs storage manager did not deliver the file
- **-556**: Error when copying the file

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String doc_status = "Freigabe"
int archiv_index = 0
String doc_id = "P000000001"
```

```
String user_name = "d3_groovy"

def error = d3.call.restore_from_jukebox(doc_status,archiv_index, doc_id,
user_name)
if (error){
  d3.log.info("$error while restoring document")
}
else {
  d3.log.info("Restoring document successful!")
}
```

restore_from_history

Description: This function restores a file from the history.

Parameter:

- **aktion_id**: ID of the action that deleted the document. If nothing is specified, use the highest action number used for this document.
- **doc_id**: Document to be restored from the secondary storage
- **user_id** : ID of executing user

Return values:

- **0**: Everything OK
- **-1**: Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

int aktion_id = 0;
String doc_id = "P000000001"
String user_name = "d3_groovy"

def error = d3.call.restore_from_history(aktion_id, doc_id, user_name)

if (error){
  d3.log.info("$error while restoring document")
} else {
  d3.log.info("Restoring document successful!")
  d3.log.info("Now trigger folder plan for document")
  Document myDoc = d3.archive.getDocument(doc_id, user_name)
  myDoc.checkFolderScheme(user_name)
}
```

add_inherit_doc_rights

Description: This function passes on permissions for a document.

Parameter:

- **doc_id**: Document which permissions are to be granted for
- **granter**: User who passes on the permissions
- **grantee**: User who inherits the permissions
- **right_flags**:

- Position 1:
  - **1**: Read permissions
  - **2**: Write permissions (also includes read permissions)
- Position 2:
  - **1**: Permissions expire after the time stamp
  - **2**: Permissions expire after acknowledgment
- Position 3: **1**: Recursive permissions (pertains to all child documents linked to the document)
- **tstamp_expire**: Time stamp for expiration, only with correct flag

Return values:

- **0**: Everything OK
- **3**: Missing parameter
- **120**: User passing on or inheriting permissions does not exist
- **450**: User passing on permissions does not have permission to pass them on
- **451**: User passing on permissions does not have a document
- **452**: User passing on permissions already has permissions for the document
- **453**: Permissions for the document cannot be passed on as the inheriting user does not have permissions for the document type

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document
import java.sql.Timestamp;

D3Interface d3 = getProperty("d3")

String doc_id = "P000000001"
String granter = "user1"
String grantee = "user2"
String right_flags = 211 //Inherit write rights for a given period of time.
For given document and all linked children
Timestamp tstamp_expire = new Date().plus(30).toTimestamp() // 30 days

def error = d3.call.add_inherit_doc_rights(doc_id, granter, grantee,
right_flags, tstamp_expire)
if (error){
  d3.log.info("$error while inheriting doc rights")
}
else {
  d3.log.info("Inheriting doc rights successful!")
}
```

remove_inherit_doc_rights

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String doc_id = "P000000001"
String granter = "user01"
String grantee = "user02"

def error = d3.call.remove_inherit_doc_rights(doc_id, granter, grantee)
if (error){
  d3.log.info("$error while removing inherited doc rights")
```

```
}
else {
  d3.log.info("Removing inherited doc rights successful!")
}
```

hold_file_send

Description: This function sends a document to a d.3 user's mailbox.

Parameter:

- **recipient** (mandatory field): Recipient of the resubmission (d.3 user or d.3 group). Separate several recipients with a semicolon.
- **notice** (optional): Text for the subject
- **doc_id**: ID of the document being sent
- **tstamp_acknowledge** (optional): Time stamp by when the entries are to be acknowledged
- **tstamp_remember** (optional): Time stamp for when a reminder is to be sent
- **expand_groups** (optional): If you enable this parameter, users are triggered and informed individually (default: **false**).
- **ignore_checkout** (optional): With this parameter you set whether a document is to be sent, even if the recipient is logged out.
  - Possible values:
    - **false**: The document is not sent
    - **true**: The document is still sent
  - Default: **false**
- **date_activate** (optional): Date on which the resubmission is to be sent
- **type** (optional): Type of resubmission (**W** for Workflow)
- **sender**: Sender's user name
- **chain_id** (optional): Resubmission chain ID
- **remove_immediately**: If the value is **true**, the resubmission is marked immediately after it is sent, so an entry does not appear under the sent resubmissions.
- **inherit_class_rule**: Expiry of permission after the time stamp (in this case **inherit_class_tstamp** must be set). If the value is **false**, the permissions expire after acknowledgment in the mailbox.
- **inherit_class_tstamp**: Time of expiration for the forwarded permissions as time stamp
- **inherit_class_right**:
  - Possible values:
    - **1**: Read permissions
    - **2**: Write permissions (also includes read permissions)
- **check_write_access**: If the value is **true**, it is checked whether the recipient has write permissions for the document.

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-3**: Undefined error, please contact d.velop Support
- **-101**: Either document ID not passed or context not set
- **-110**: Recipient not specified
- **-201**: Invalid user name specified
- **-207**: Invalid recipient specified
- **-208**: Invalid date specified
- **-209**: Invalid chain ID specified

- **-212**: Invalid document ID specified
- **-244**: Invalid rule or permission specified
- **-301**: Invalid document ID specified or user does not have permissions for the document
- **-302**: Error when determining the user permissions
- **-303**: User does not have access permissions for the document
- **-315**: No group member has access permissions for the document that is to be sent.
- **-316**: Recipient does not have any access permissions but the sender can grant permissions
- **-317**: Sender does not have write permissions. The permissions cannot be transferred to the recipient.
- **-318**: Sender does not have write permissions. The permissions cannot be passed on.
- **-450**: User does not have permissions to inherit
- **-451**: User does not have the appropriate permissions
- **-452**: Recipient already has the permissions that are to be inherited
- **-453**: No document classes or permission profiles are assigned to the user
- **-454**: Recipient needs permissions for the document type that is to be inherited
- **-455**: Temporary document class has already been assigned to the user
- **-503**: Recipient is logged out

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document
import java.sql.Timestamp;

D3Interface d3 = getProperty("d3")

String recipient = "user01"
String notice = "Please check"
String doc_id = "P000000001"
Timestamp tstamp_acknowledge = new Date().plus(10).toTimestamp() // 10 days
Timestamp tstamp_remember = new Date().plus(30).toTimestamp() // 30 days
boolean expand_groups = false
boolean ignore_checkout = true
Timestamp date_activate = null // Now
String type = ""
String sender = "d3_groovy"
int chain_id = 0
boolean remove_immediately = false
boolean inherit_class_rule = false
Timestamp inherit_class_tstamp = null;
int inherit_class_right = 0
boolean check_write_access = false

def error = d3.call.hold_file_send(recipient, notice, doc_id,
tstamp_acknowledge, tstamp_remember, expand_groups, ignore_checkout,
date_activate, type, sender, chain_id, remove_immediately,
inherit_class_rule, inherit_class_tstamp, inherit_class_right,
check_write_access)
if (error){
  d3.log.info("$error while sending hold file")
}
else {
  d3.log.info("Sending hold file successful!")
}
```

hold_file_find

Description: This function searches for users who have the document in their resubmission

Parameter:

- **recipient** (mandatory field): Recipient of the resubmission (d.3 user or d.3 group). Separate several recipients with a semicolon.
- **doc_id**: ID of the document to be sent
- **user_name**: User in whose name the search is to be executed

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-3**: Undefined error, please contact d.velop Support
- **-101**: Either document ID not passed or context not set
- **-201**: Invalid user name specified
- **-301**: Invalid document ID specified or user does not have permissions for the document
- **-302**: Error when determining the user permissions
- **-303**: User does not have access permissions for the document

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String recipient = null //get all hold_file_recipients of document
String doc_id = "P000000001"
String user_name = "d3_groovy"

String[] hold_file_recipients = d3.call.hold_file_find(recipient, doc_id,
user_name)
if (hold_file_recipients){
  d3.log.info("Finding hold files successful!")
  for(String hold_file_recipient : hold_file_recipients){
    d3.log.info("hold file recipient -> " + hold_file_recipient)
  }
}
```

hold_file_delete

Description: This function acknowledges received reminders and deletes sent reminders.

Parameter:

- **chain_id** (mandatory field): ID of the resubmission chain which the entry belongs to
- **sent_received** (optional):
  - Possible values:
    - **1**: Sent or received resubmission file
    - **2**: Resubmission
  - Default: **2**:
- **workflow_only** (optional): If the value is **0** or **1**, the resubmission workflow is acknowledged (only if **sent_received** has the value **2**).
- **recipient** (optional): Recipient (only if **sent_received** has the value **1** or the recipient is a group)
- **doc_id**: User in whose name the search is to be executed
- **user_name**: User in whose name the deletion or acknowledgement is to be executed

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-3**: Undefined error, please contact d.velop Support
- **-101**: Either document ID not passed or context not set
- **-110**: Recipient not specified
- **-124**: No chain ID specified
- **-528**: Error when deleting the sender list (wrong chain ID)
- **-529**: Error when deleting the recipient list (wrong chain ID)

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

long chain_id = 15
Byte sent_received = 2
boolean workflow_only = false
String recipient = "user01"
String doc_id = "P000000001"
String user_name = "d3_groovy"

def error = d3.call.hold_file_delete(chain_id, sent_received,
workflow_only, recipient, doc_id, user_name)
if (error){
  d3.log.info("$error while sending hold file")
}
else {
  d3.log.info("Sending hold file successful!")
}
```

workpath_end_document

Description: This function removes the document from a workflow.

Parameter:

- **doc_id**: Document ID, if nothing is specified
- **user_name**: User who the search is to be executed for
- **delete_jobs**:
  - Possible values:
    - **false**: Workflow jobs that are still in the asynchronous job queue are not removed.
    - **true**: All relevant workflow jobs that are still in the asynchronous job queue are removed.
  - Default: **false**

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document
```

```
D3Interface d3 = getProperty("d3")

String doc_id = "P000000001"
String user_name = "d3_groovy"
boolean delete_jobs = false

def error = d3.call.workpath_end_document(doc_id, user_name, delete_jobs)
if (error){
  d3.log.info("$error while ending workpath")
} else {
  d3.log.info("Ending workpath successful!")
}
```

workpath_start_document

Description: This function inserts the document in the workflow.

Parameter:

- **wfl_id**: ID for the workflow where the document is to be inserted
- **doc_id**: Document ID
- **user_name**: User to be searched for. If no value is specified, the executing user is used.

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String wfl_id = "398t9mgq983q44jg"
String doc_id = "P000000001"
String user_name = "d3_groovy"

def error = d3.call.workpath_start_document(wfl_id, doc_id, user_name)
if (error){
  d3.log.info("$error while starting workpath")
} else {
  d3.log.info("Starting workpath successful!")
}
```

workpath_go_to_next_step

Description: This function moves the document in the workflow either by specifying a decision value or the next step.

Parameter:

- **exit_value**: Exit with which the step is to be completed
  - 1: Still in start item
  - 2: TRUE
  - 3: FALSE
  - 100: Manual move

- **next_step_id**: ID of the step with which the workflow is to be continued This parameter is needed for **exit_value** = 100.
- **doc_id**: Document ID.
- **user_name**: User in whose name the action is to be executed.

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly

```
mport com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

int exit_value = 100
String next_step_id = ""
String doc_id = "P000000001"
String user_name = "d3_groovy"

def error = d3.call.workpath_go_to_next_step(exit_value, next_step_id,
doc_id, user_name)
if (error){
  d3.log.info("$error while going to next workpath step")
} else {
  d3.log.info("Going to next workpath step successful!")
}
```

roll_get_names

Description: This function queries all role names.

Parameter: None

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

def role_names = d3.call.roll_get_names()
if (role_names){
  d3.log.info("Getting role names successful!")
  for(String role_name : role_names){
    d3.log.info("role_name -> " + role_name)
  }
}
```

roll_get_users

Description: This function displays the user who the role is assigned to.

Parameter:

- **roll_id**: Role ID, optional
- **rollname**: Role name, optional

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-118**: Either no role ID or role name specified
- **-217**: Invalid role ID or name specified

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

Long roll_id = 6
String roll_name = ""

String[] assigned_users = d3.call.roll_get_users(roll_id, roll_name)
if (assigned_users){
  d3.log.info("Getting assigned users successful!")
  for(String assigned_user : assigned_users){
    d3.log.info("assigned_user -> " + assigned_user)
  }
}
```

document_render

Description: Generates TIFF, PDF, OCR

Parameter:

- **source**: Source file; required if the doc_id (last version) document is not to be rendered
- **destination** : Destination file (mandatory field), if the rendered file is not to be saved as a dependent t1 or p1 file for doc_id.
- **render_option**:
  - 0: No creation of a render file, e.g. only create the OCR file
  - 1: Create a TIFF file as render file (default)
  - 2: Create a PDF file as render file
  - 3: Create a TIFF and PDF file as render file
- **ocr**:
  - **false:** Do not extract index keywords (preset)
  - **true**: Extract index keywords
- **asynchronous**: Only the value **true** is possible
- **replace_doc**:
  - **false:** Generated TIFF/PDF does not replace the source document in d.velop documents (default)
  - **true**: Generated TIFF/PDF replaces the source document in d.velop documents
- **overwrite**:
  - **false:** Do not replace destination file if it already exists (default)
  - **true**: Replace destination file if it already exists
- **doc_id**: Document ID
- **user_name**: User in whose name the command is to be executed.

- **doc_status**: Status of the version to be rendered. If nothing is specified, the latest version is used (**B** for **Processing, P** for **Verification, F** for **Release, A** for **Archive**).
- **archiv_index** : Index for the version in the status archive that is to be rendered (only necessary for **=A/Archive**)
- **prio**: Valid values are **low, normal** (default), **high**

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-118**: Either no role ID or role name specified
- **-217**: Invalid role ID or name specified

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String source = null
String destination = null
int render_option = 2
boolean ocr = true
boolean asynchronous = false
boolean replace_doc = false
boolean overwrite = true
String doc_id = "P000000001"
String user_name = "d3_groovy"
String doc_status = null
int archiv_index = 0
String prio = "Low"

def error = d3.call.document_render(source, destination, render_option,
ocr, asynchronous, replace_doc, overwrite, doc_id, user_name, doc_status,
archiv_index, prio)
if (error){
  d3.log.info("$error while sending to d.ecs rendition service")
}
else {
  d3.log.info("Sending to d.ecs rendition service successful!")
}
```

tiff_concat

Description: This function attaches a TIFF document to another document.

Parameter:

- **source**: TIFF source file
- **destination** : TIFF destination file
- **source_rdl**: Source file for comments, optional
- **destination_rdl**: Destination file for comments, optional

Return values:

- **0**: Everything OK

- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-3:** Unknown error, please contact d.velop Support.
- **-151**: Source file name not specified
- **-152**: Destination file name not specified
- **-559**: Error when attaching the TIFF files
- **-560**: Number of target pages different to the total source pages
- **-561**: TIFF source file does not exist
- **-562**: TIFF source file is not supported (possibly corrupted or wrong format)
- **-563**: TIFF destination file cannot be opened
- **-564**: Comment source file is not supported (possibly corrupted or wrong format)
- **-565**: Comment source file contains unassigned pages

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String source = "C:\\temp\\myTiff1.tif"
String destination = "C:\\temp\\myTiff2.tif"
String source_rdl = ""
String destination_rdl = ""

def error = d3.call.document_render(source, destination, render_option,
source_rdl, destination_rdl)
if (error){
  d3.log.info("$error while concating tif files")
}
else {
  d3.log.info("Concating tif files successful!")
}
```

document_render_wfl_prot

Description: Transfers the workflow logs to a document ID and creates a dependent W1 file. If a W1 file already exists, the function does not execute an action (return 1). If a new TIFF file is to be submitted, for example because another log was added for the document ID, you have to remove the dependent W1 file.

Parameter:

- **doc_id**: Document whose workflow logs are to be submitted
- **user_name**: Name of executing user

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")
```

```
String doc_id = "P000000001"
String  user_name = "d3_groovy"

def error = d3.call.document_render_wfl_prot(doc)
if (error){
  d3.log.info("$error while rendering wfl protocol")
} else {
  d3.log.info("Rendering wfl protocol successful!")
}
```

object_property_set

Description: Sets the property values for items. If you set a zero string for the value, the property value for the item is deleted.

Parameter:

- **property_name**: Internal name of the item property
- **object_id**: Item name (e.g. user name, group name dependent of **object_class_id)**
- **object_class_id**: Item class ID
  - **1**: User
  - **2**: Groups
  - **3**: Organizational unit
  - **4**: Activity profiles
  - **5**: Records
  - **6**: Repository fields
  - **7**: Document types
  - **8**: Document classes
  - **9**: Permission profiles
  - **10:** Sets
  - **11:** Workflows
- **object_info**: Property value

Return values:

- **0**: Everything OK
- **-1**: Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String property_name = "myProperty"
String object_id = "d3_groovy"
int object_class_id = 1
String object_info = "myValue"

def error = d3.call.object_property_set(property_name, object_id,
object_class_id, object_info)
if (error){
  d3.log.info("$error while setting object property")
} else {
  d3.log.info("Setting object property successful!")
}
```

d3set_add_filter

Description: Adds a filter to an existing permission set. If the allocation for permission sets (set_name, object_id) does not exist, the allocation is created.

Parameter:

- **user_name**: d.3 user's name in whose name the change is to be executed.
- **doc_id**: Document ID of the permission set document. If the document ID is set, then **set_name** and **object_id** can be left empty.
- **set_name**: Name of the permission set.
- **object_id**: User name or group identifier. If the parameter is empty, it searches for a set without allocation.
- **filter**: Filter values that are to be added to the set. You can separate several filter values by a semicolon (;). The maximum filter length is 150 characters.

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String user_name = "d3_groovy"
String doc_id = ""
String set_name = "myRestrictionSet"
String object_id = "user01"
String filter = "myValue1;myValue2"
boolean overwrite = true

def error = d3.call.d3set_add_filter(user_name, doc_id, set_name,
object_id, filter, overwrite)
if (error){
  d3.log.info("$error while adding filter")
} else {
  d3.log.info("Adding filter successful!")
}
```

d3set_remove_filter

Description: Removes a filter from an existing permission set.

Parameter:

- **user_name**: d.3 user's name in whose name the change is to be executed.
- **doc_id**: Document ID of the permission set document. If the document ID is set, then **set_name** and **object_id** can be left empty.
- **set_name**: Name of the permission set
- **object_id**: User name or group identifier. If the parameter is empty, it searches for a set without allocation.
- **filter**: Filter values that are to be added to the set. You can separate several filter values by a semicolon (;). The maximum filter length is 150 characters.

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String user_name = "d3_groovy"
String doc_id = ""
String set_name = "myRestrictionSet"
String object_id = "user01"
String filter = "myValue1"

def error = d3.call.d3set_remove_filter(user_name, doc_id, set_name,
object_id, filter)
if (error){
  d3.log.info("$error while removing filter")
} else {
  d3.log.info("Removing filter successful!")
}
```

d3set_remove_set

Description: Removes an existing permission set.

Parameter:

- **user_name**: d.3 user's name in whose name the change is to be executed.
- **doc_id**: Document ID of the permission set document. If the document ID is set, then **set_name** and **object_id** can be left empty.
- **set_name**: Name of the permission sets.
- **object_id**: User name or group identifier. If the parameter is empty, it searches for a set without allocation.

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String user_name = "d3_groovy"
String doc_id = ""
String set_name = "myRestrictionSet"
String object_id = "user01"

def error = d3.call.d3set_remove_set(user_name, doc_id, set_name, object_id)
if (error){
  d3.log.info("$error while removing set")
```

```
} else {
  d3.log.info("Removing set successful!")
}
```

d3async_job_open

Description: Opens a new d.3 async job

Parameter:

- **doc_id_ref**: Document ID of the relevant document
- **job_type**: Type of job
- **user_name**: d.3 user's name in whose name the change is to be executed.

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String doc_id_ref = "P000000001"
String job_type = "CUJ001"
String user_name = "d3_groovy"

d3.call.d3async_job_open( doc_id_ref ,  job_type, user_name);
d3.call.d3async_job_set_attribute("hook_function", "myJplFunction", 0);
d3.call.d3async_job_set_attribute("custom_job_par[1]", "valueForParam1", 0);
d3.call.d3async_job_set_attribute("custom_job_par[2]", "valueForParam2", 0);
d3.call.d3async_job_set_attribute("custom_job_par[3]", "valueForParam3", 0);
d3.call.d3async_job_close();
```

d3async_job_set_attribute

Description: Sets the parameter of the previously opened d.3 async job.

Parameter:

- **attr_name**: Attribute name
- **attr_value**: Attribute value
- **attr_type**: Attribute type

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")
```

```
String doc_id_ref = "P000000001"
String job_type = "CUJ001"
String user_name = "d3_groovy"

d3.call.d3async_job_open( doc_id_ref ,  job_type, user_name);
d3.call.d3async_job_set_attribute("hook_function", "myJplFunction", 0);
d3.call.d3async_job_set_attribute("custom_job_par[1]", "valueForParam1", 0);
d3.call.d3async_job_set_attribute("custom_job_par[2]", "valueForParam2", 0);
d3.call.d3async_job_set_attribute("custom_job_par[3]", "valueForParam3", 0);
d3.call.d3async_job_close();
```

d3async_job_set_lin002_attribute

```
currently not supported
```

d3async_job_close

Description: Closes d.3 async jobs

Parameter: None

Return values:

- **0**: Everything OK
- **> 0**: Database error
- **1** Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document

D3Interface d3 = getProperty("d3")

String doc_id_ref = "P000000001"
String job_type = "CUJ001"
String user_name = "d3_groovy"

d3.call.d3async_job_open( doc_id_ref ,  job_type, user_name);
d3.call.d3async_job_set_attribute("hook_function", "myJplFunction", 0);
d3.call.d3async_job_set_attribute("custom_job_par[1]", "valueForParam1", 0);
d3.call.d3async_job_set_attribute("custom_job_par[2]", "valueForParam2", 0);
d3.call.d3async_job_set_attribute("custom_job_par[3]", "valueForParam3", 0);
d3.call.d3async_job_close();
```

send_email

Description: This function allows you to send e-mails.

Parameter:

- **recipient**: Recipient of the e-mail (d.3 user name or d.3 group name)
- **notice**: Subject
- **date**: Send date (may also be in the future), optional
- **doc_id**: ID of the document to be sent as a link, optional
- **user_name**: Sender of the e-mail (d.3 user name)
- **body_file**: Name and path of the file that contains the text for the body
- **mail_format**: HTML format, else: Text format (default)
- **attach**: Send document as attachment (**0** (default) or **1**).

- **doc_status**: Attach version in doc_status status.
- **archiv_index** : Index for the archive version to be attached.
- **attach_abh**: List of dependent files (e.g. "T1,P1")
- **attach_file**: List of files to be attached (e.g., "C:\file1.jpg,c:\file2.txt")

Return values:

- **0**: Everything OK
- **-1**: Panther syntax error, for details please see d.3 logviewer
- **-2**: Function was called incorrectly
- **-3**: Unknown error, please contact d.velop Support.
- **-110**: No recipient specified
- **-201**: Invalid user name specified
- **-301**: Invalid document ID specified or user does not have rights
- **-302**: Error when determining user rights
- **-303**: User does not have access rights to the document
- **-540**: SMTP_SUPPORT is not enabled
- **-541**: SMTP DLL is not available
- **-542**: Error when sending the e-mail

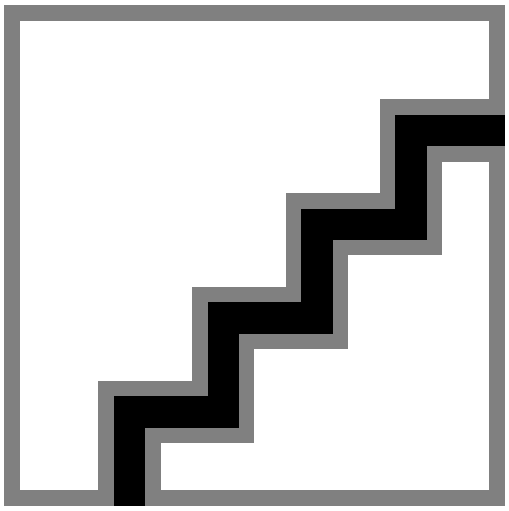
```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.Document
import java.sql.Timestamp;

D3Interface d3 = getProperty("d3")

String recipient = "dvelop"
String notice = "Please check this document"
Timestamp date = null
String doc_id = "P000000001"
String user_name = "d3_groovy"
String body_file = "<html>Please check this document</html>"
String mail_format = "html"
boolean attach = true
String doc_status = ""
int archiv_index = 0
String attach_abh = ""
String attach_file = ""

def error = d3.call.send_email (recipient, notice, date, doc_id, user_name,
body_file, mail_format, attach, doc_status, archiv_index, attach_abh,
attach_file)
if (error){
  d3.log.info(error + " while sending mail")
} else {
  d3.log.info("Sending mail successful! ->"+error)
}
```

1.9.5. Config parameter (ConfigInterface)

ConfigInterface

```
    public interface ConfigInterface {
        public String value(String paramName);
        public String value(String paramName, Integer paramIndex);
    }
```

You can use the ConfigInterface to query all d.3 config parameters. Possible parameter names are all parameters that are displayed in d.3 config.



```
import com.dvelop.d3.server.core.D3Interface
D3Interface d3 = getProperty("d3");

// Getting the ID of the DBMS and writing it to the log file
String dbServer = d3.config.value("db_server");
d3.log.info( "Database: ${dbServer}");

// Get the first hostimport directory
String hostimpDir1 = d3.config.value("HOSTIMP_IMPORT_DIR", 1)
d3.log.info( "Hostimport directory: ${hostimpDir1}");

// Get the d.3 serverId of d.3 server process
String serverId = d3.config.value("d3fc_server_id")
d3.log.info("serverId " + serverId)

// Set variables depending on the serverId
String myTechnicalUser
if(serverId == "A"){
        myTechnicalUser = "techUser1"
}else if(serverId == "B"){
        myTechnicalUser = "techUser2"
} else {
        d3.log.error("Unknown server id " + serverId)
}
d3.log.info("Using user " + myTechnicalUser)
```

1.9.6. Logging (LogInterface)

LogInterface

```
    public interface LogInterface extends GroovyLogInterface{
        public void critical(Object msg);
        public void error(Object msg);
        public void warn(Object msg);
        public void info(Object msg);
        public void debug(Object msg);
        public boolean isDebugEnabled();
```

```
        public void message(Object msg, int logLevel);
    }
```

> **Note**
> Furthermore the Groovy **println()** method is mapped to **LogInterface.info()**, so that it can simply be written into the d.3 log at any point in the Groovy code via **println()**.

Example

```
import com.dvelop.d3.server.core.D3Interface

D3Interface d3 = getProperty("d3");

d3.log.critical("My critical message")
d3.log.error("My error message")
d3.log.warn("My warn message")
d3.log.info("My info message")
d3.log.debug("My debug message")

if(d3.log.isDebugEnabled()){
  d3.log.debug("My debug message")
} else {
  //no message
}

d3.log.message("My critical message", 0)
d3.log.message("My error message", 1)
d3.log.message("My start/stop message", 2)
d3.log.message("My warn message", 3)
d3.log.message("My warn message", 4)
d3.log.message("My warn message", 5)
d3.log.message("My info message", 6)
d3.log.message("My debug message", 7)
d3.log.message("My debug message", 8)
d3.log.message("My debug message", 9)
```

1.9.7. Hook properties (HookInterface)

Some d.3 entry points have specific properties that can be changed in the corresponding hook function. This hook properties interface is used to read and change these properties.

If such properties exist, they are specified in the description of the d.3 entry point. Examples of this are the render options of "hook_rendition_entry_20" or the e-mail properties of "hook_send_email_entry_20".

Calling the methods in a hook function:

- Reading a property value: d3.hook.getProperty("property name")
- Changing a property value: d3.hook.setProperty("property name", "property value")

For multi-line properties accordingly:

- Read the first value of a multi-value property: d3.hook.getProperty("property name", 1)
- Change the second value of a multi-value property: d3.hook.setProperty("property name", 2, "property value")

HookInterface

```
    public interface HookInterface {
        public String getProperty (String propName);
        public String getProperty (String propName, int propIndex);
        public void   setProperty (String propName, String propValue);
        public void   setProperty (String propName, int propIndex, String
propValue);
    }
```

1.9.8. Error handling (D3Exception)

D3Exception

```
package com.dvelop.d3.server.exceptions;

public class D3Exception     extends RuntimeException

    public class AmbitiousResultException          extends D3Exception
    public class ConnectionError                   extends D3Exception
    public class GroovyAPIFunctionException        extends D3Exception
    public class GroovyAPIFunctionRuntimeException extends D3Exception
    public class GroovyHookException               extends D3Exception
    public class GroovyHookRuntimeException        extends D3Exception
    public class InvalidDateFormatException        extends D3Exception
    public class InvalidFormatException            extends D3Exception
    public class InvalidInputException             extends D3Exception
    public class InvalidParameterException         extends D3Exception
    public class ObjectNotFoundByIdException       extends D3Exception
    public class ReferenceToUnknownObjectException extends D3Exception
    public class SQLException                      extends D3Exception
    public class TimeoutException                  extends D3Exception
    public class UnconvertableException            extends D3Exception
    public class NullValueException                extends D3Exception
```

Example

```
import com.dvelop.d3.server.core.D3Interface
import com.dvelop.d3.server.exceptions.D3Exception
import com.dvelop.d3.server.Document;
D3Interface d3 = getProperty("d3");

//D3Exception
try{
   Document tmpDoc = d3.archive.getDocument("P000000001",
"thisUserDoesNotExist")
} catch(D3Exception ex){
   d3.log.error('Error "Failed to find user" ' + ex.getMessage())
}

//General exception
try{
   def arr = new int[3];
   arr[5] = 5;
} catch(Exception ex){
   d3.log.error('Error "Index out of bounds" ' + ex.getMessage())
}
```

1.9.9. Storage manager

StorageManagerInterface

```
    public interface StorageManagerInterface {
        public void addFileToReload(String docId, int fileId);
        public void addFileToReload(String docId, int fileId, String
dependentExtension);
        public void addFileToReload(String fileName);      // Not document
related file

        public void addFileToReload(Document doc);         // All files for
the document
        public void addFileToReload(PhysicalVersion physVers);
        public void addFileToReload(DependentFile depFile);

        public void setNumberOfFilesPerJob(int value);
        public void setSMThreadsToUse(int value);
        public void setReloadToCachedDocs(boolean value);
        public void setMoveFilesToDocsDir(boolean value);

        public String getReloadPrefix();
        public int reloadFiles();
    }
```