InfoFabrik gmbh

ReportServer

Script Guide 6.0

InfoFabrik gmbh

ReportServer

Script Guide 6.0

InfoFabrik GmbH, 2026

https://www.infofabrik.de/
https://www.reportserver.net/

Contents

Part I

Script Guide

Chapter 1

Introduction

The term Business Intelligence represents the capacity of a corporation to look at all business data as a whole in order to distinguish information which plays a key role in the strategic planning process and in decision making of the corporation. To achieve this all available data require consolidated preparation, for instance, in a Data Warehouse.

ReportServer serves as an interface to access these data, and the user can easily and efficiently make use of them. ReportServer offers tools to support you in your daily work establishing ready-to-print reports, or performing ad-hoc analyses.

Target Audience

In this manual we discuss ReportServer from the point of administrators that wish to extend ReportServers using scripts. This guide will also help report designers that use ReportServer script reports to create interactive reports or reports based on metadata kept by ReportServer.

Separate manuals and instructions illustrate the various aspects of ReportServer.

ReportServer Configuration Guide: Describes the installation of ReportServer as well as the basic configuration options.

ReportServer User Guide: The user guide describes ReportServer from the point of view of the ultimate user. It includes an in-depth coverage of dynamic lists (ReportServer's adhoc reporting solution), execution of reports, scheduling of reports, and much more.

ReportServer Administrator Guide: The administrator guide describes ReportServer from the point of view of administrators that are tasked with maintaining the daily operation of the reporting platform including the development of reports, managing users and permissions, monitoring the system state, and much more.

ReportServer Scripting Guide: The ReportServer scripting guide covers the scripting capabilities of ReportServer which can be used for building complex reports as well as for extending the functionality of ReportServer or performing critical maintenance tasks. It extends the introduction to these topics given in the administrator guide.

Content

This guide is based upon the ReportServer administrator's manual. We furthermore require basic programming experience in the Groovy programming language. An introduction to groovy can be found at https://groovy-lang.org/

This manual comprises the following chapters:

Part 1

Getting Started In this chapter we describe how to setup a work space to comfortably test and develop ReportServer scripts. Here we also cover the basic terminology.

Basic Script Services Introduces the GLOBALS object which provides access to services and entities for scripts.

Advanced Techniques Covers monitoring, multithreading, scheduling and other advanced techniques

Script Reporting This chapter covers the script reporting engine and script datasources.

Tapping into ReportServer Explains the ReportServer HookHandler concept that allows to extend and customize various aspects of ReportServer.

Extending the Client In this chapter we will look into various possibilities to extend the client side of ReportServer.

Part 2 - Examples

Adding additional Database Support In this chapter we explain how support for additional relational databases can be added to ReportServer.

Authentication against an Active Directory In this chapter we look into the authentication mechanisms of ReportServer and discuss how to authenticate against an Active Directory (LDAP)

Additional Report Exporters This chapter covers the basics of adding custom export formats to the various reporting engines available in ReportServer.

Please contact us in case you have questions or suggestions. We are looking forward to your requests. You can reach us via email (info@infofabrik.de) or our online forum which you can find at https://forum.reportserver.net/.

Chapter 2

Getting Started

ReportServer scripts can be developed directly from within ReportServer using the Terminal and the commands createTextFile and editTextFile. In order to take full advantage of the services offered by ReportServer to script designers it is, however, necessary to get a basic understanding of the ReportServer source code. For this we provide a javadoc documentation of the sources together with the complete source code of ReportServer Community Edition on sourceforge.

To run and develop scripts in ReportServer, login to your ReportServer and open up the terminal (CTRL+ALT+T). All scripts must be stored in the fileserver somewhere beneath the `bin` directory. Thus, to give us a temporary working directory we go to the `bin` directory by typing `cd ↙ ↳ fileserver/bin` create a new directory called `tmp` by typing `mkdir tmp` and go to our newly created directory (`cd tmp`). Next, we create our very first script by executing

```
createTextFile helloworld.groovy
```

A text editor opens in which we copy our first script:

```
return  "Hello World"
```

which simply returns the string "Hello World" upon execution. Now, execute the script via

```
exec helloworld.groovy
```

You should see the following output:

```
reportserver$ exec helloworld.groovy
Hello World
reportserver$
```

That is, when executing a script on the terminal, then the final return statement is printed on the console. If you want to print something during the execution of the script (note that all print statements are buffered until the script is successfully executed) you can use the tout object (for terminal output).

```
tout.println('Hello World')
tout.println('Hello World2')
```

To edit the file in ReportServer directly call

```
editTextFile helloworld.groovy
```

2.1 Terminology and Extension Points

ReportServer scripts can be used for a variety of tasks. You can use them to create datasources or interactive reports, they can be used as administration tools and scheduled to perform maintenance tasks. They can, however, also be used to extend ReportServer both on the client side and on the server side. In this guide we will mainly focus on the last part of ReportServer scripts. This, however, directly also provides a large insight into other applications of scripts.

In order to write scripts that extend ReportServer we need to explain a bit about how GWT works and we need to learn about several important concepts within ReportServer. ReportServer is based on the GWT framework (you can find tons of information on GWT at https://www.gwtproject.org/). GWT is a web application framework that allows you to write the entire application including the client part in Java. Before deployment the client side is then translated to JavaScript such that it can be run natively in any modern web browser. This concept has several advantages: for example, you have most of the advantages of Java such as its sophisticated debugging tools or strong types available also for the client side. The main advantage, arguably, is that GWT is able to produce highly optimized JavaScript code which greatly benefits the user experience.

Script developers should keep in mind that the source code contains both server and client side code. Although code can be shared between the server and the client this is usually not the case. As scripts are written in Groovy and interpreted and executed on the server this means that scripts can only use server side code. If you look through the projects you will find that the package structure is the following:

`net.datenwerke.(rs).(name).`

where **rs** denotes a ReportServer project and name the main package of that project. Projects that are not ReportServer specific such as for example the DwHookHandler will not have the "rs" prefix. After the name part you will usually find the following packages:

client	Contains client side code.
server	Contains servlets.
service	Contains server side code

Hooks and Services

Now that we have seen the basic source structure there are two important concepts for script designers. One is the concept of Services. Most of the functionality offered by ReportServer, such as, executing reports, managing users, scheduling and many more are exposed by Service interfaces. An example is the ReportExecutorService which exposes the functionality to execute reports. When writing scripts you will usually use services to access ReportServer functionality. Services are by convention adhering to the naming scheme: nameService. If you search for `net.datenwerke.*.service.*Service` in the sources (or javadoc) of ReportServer you will find a large number of available server side services. We have also included a special services documentation file which lists all available services together with a short description and links to the Javadoc documentation of the service.

The second important concept is that of Hooks. Hooks are extension points in ReportServer which allow you to enhance functionality or be informed on certain events (such as report x is about to be executed). As with Services Hooks are used both on the server and client side and for script writers only the server side Hooks can be used. Hooks can be recognized by interfaces or abstract classes that implement the Hook interface

The hook interface is located in net.datenwerke.hookhandler.shared.hookhandler.interfaces

An example of a Hook is the `ReportExecutionNotificationHook` which is notified throughout the various stages of report execution. To use a Hook the corresponding interface must be implemented and the implementing Hook must be registered with the HookHandlerService. Classes that instantiate a Hook are usually suffixed by Hooker (i.e., the one doing the hooking).

Entities and DTOs

Entities are objects which are persisted in the database. Most objects, such as users, reports or TeamSpaces, that you will handle as a script designer are entities. You can recognize classes that represent entities by the `@Entity` annotation. When handling entities it is important to keep in mind that changes need to be persisted. To load and store entities the JPA EntityManager is used.

Entities can also be transported to the client. However, as they are inherently server based an entity is first transformed into a so called **data transfer object** (DTO) which is the representation on the client side. The conversion between entities and DTOs is handled by the DtoService.

Chapter 3

Basics

In the previous chapter we saw that ReportServer functionality is exposed by services. In this chapter we discuss the basics of scripting within ReportServer. This includes passing parameters to scripts, accessing various services, interpreting error messages, working with entities, reading and writing text files and more.

3.1 Executing Scripts

As we have seen, scripts in ReportServer are executed via the Terminal using the `exec` command. The exec command takes as command a script and passes on any further arguments to the script. You can access command line arguments in your script via the variable `args`. Suppose we adapt our hello world script as follows:

```
package myscripts

tout.println('Hello ' + args )
```

If we now execute the script as

```
exec helloworld.groovy John
```

You will get the output

```
Hello [John]
```

The square brackets around the name, indicate that the args variable is in fact an array, or rather a groovy collection. If we again change our script to

```
package myscripts

tout.println('Hello ' + args.reverse() )
```

and call it with

```
exec helloworld.groovy John Doe
```

the output will be

```
Hello [Doe, John]
```

Return Values

Scripts have return values. The return value will be output in the terminal. You can either explicitly return from a script via the usual "return x" instruction or else the last line of your script will be interpreted as return value. Thus, we could change our above script to

```
package myscripts

'Hello ' + args.reverse()
```

to get the same effect: an output in the terminal.

3.2 Executing Scripts via URL

Besides executing scripts from the terminal, you can execute scripts by calling a specific URL. To execute a script with ID 15 you need to call

http://SERVER:PORT/reportserverbasedir/reportserver/scriptAccess?id=15

The following parameters are available

id	The script's id.
path	Alternatively to specifying an id you can specify the path of a script. E.g.: http://SERVER:PORT/reportserverbasedir/reportserver/scriptAccess?path=/bin/myscript.groovy
args	The arguments to be passed to the script.
exception	In case of an exception the server will not respond with an error message. In order to make the server respond with the actual exception set this parameter to true.
commit	Whether or not the script should be executed in commit mode.

If executing a script via URL you have access to the predefined variable `httpRequest` as well as `httpResponse` which provide access to the HTTP request and response object.

3.3 The GLOBALS Object

Every ReportServer script is initialized with a scope that contains an object called GLOBALS. This object exposes ReportServer functionality to your script. This allows you to access ReportServer services but it also provides certain functionality to scripts such as easily accessing files in the fileserver. The most common case is access of ReportServer services and to access entities.

Tip: The GLOBALS object is of type net.datenwerke.rs.scripting.service.scripting.scriptservices.GlobalsWrapper. The javadoc documentation provides an overview of the methods provided by GLOBALS: https://reportserver.net/api/latest/javadoc/net/datenwerke/rs/scripting/service/scripting/scriptservices/GlobalsWrapper.html.

Services can be retrieved via the method `getInstance()`. `getInstance` takes a class object as input and returns the corresponding service object. It is also possible to get access to a Provider object for a service. Providers can be regarded as containers which do not immediately access the service but which allow access to that particular service. This can for example be helpful when writing complex scripts and you want to give access to services to functions or classes. Providers can be accessed via the `getProvider` method which also takes a class object as input.

In the following we create a simple script that allows to search your ReportServer installation for users. For this we will use the SearchService located in package `net.datenwerke.rs.search.service.search`. The service exposes the search functionality provided by ReportServer.

Tip: In order to use auto completion and automatic import statements when editing scripts in Eclipse you must add all projects to the list of required projects. For this open the groovy project's properties (right-click on the project then choose properties) and Java Build Path. Choose the Projects tab and select all other projects.

Tip: Eclipse's auto-completion works better if you program Groovy using explicit types. That is, instead of writing

```
def var = 'string'
```

use

```
String var = 'string'
```

Create a new script in Eclipse called searchuser.groovy. We will require the SearchService from the GLOBALS object. We will then use the locate method to retrieve a list of results. As input we will simply pass on the argument array (converted to a string).

```
package myscripts

import net.datenwerke.rs.search.service.search.SearchService
import net.datenwerke.security.service.usermanager.entities.User

SearchService searchSvc = GLOBALS.getInstance(SearchService)
searchSvc.locate(User.class, args.join(" "))
```

If we add this script to our tmp folder in ReportServer and execute it via

```
exec searchuser.groovy root*
```

you will get the output

```
[User{ID=6, First name=root, Last name=root}]
```

Let us create a second user via the user manager called "Tom Rootinger". If we run our script again with root as argument we will get the same output. This is because, by default the search service looks for exact matches. If we run the script as

```
exec searchuser.groovy *root*
```

we get the expected result:

```
[User{ID=6, First name=root, Last name=root}, User{ID=6831, First name=Tom, Last
    ↪ name=Rootinger}]
```

In a next step we want to use the HistoryService (located in `net.datenwerke.gf.service.history`) to generate links for the objects found by our script. The HistoryService has a single method that takes an object and returns a list of links (objects of type `HistoryLink`). We will simply take the first link in this list (if it exists) and output it. The adapted script looks like

```
package myscripts

import net.datenwerke.gf.service.history.HistoryService
import net.datenwerke.rs.search.service.search.SearchService
import net.datenwerke.security.service.usermanager.entities.User

SearchService searchSvc = GLOBALS.getInstance(SearchService)
HistoryService historySvc = GLOBALS.getInstance(HistoryService)

searchSvc.locate(User.class, args.join(" ")).collect{
  historySvc.buildLinksFor(it).get(0)?.getLink()
}.join("\n")
```

If we run this again using `exec searchuser.groovy *root*` we get as output

```
usermgr/path:1.2.3&nonce:-1668828071
usermgr/path:1.4&nonce:475784900
```

On the one hand we are missing the server address and on the other, wouldn't it be nice to be able to actually click on the link? The server address can be accessed via the ReportServerService. Thus we could adapt our script as

```
package myscripts

import net.datenwerke.gf.service.history.HistoryService
import net.datenwerke.rs.core.service.reportserver.ReportServerService
import net.datenwerke.rs.search.service.search.SearchService
import net.datenwerke.rs.terminal.service.terminal.obj.CommandResult
import net.datenwerke.security.service.usermanager.entities.User

SearchService searchSvc = GLOBALS.getInstance(SearchService)
HistoryService historySvc = GLOBALS.getInstance(HistoryService)
String urlBase = GLOBALS.getInstance(ReportServerService).serverInfo.baseURL

searchSvc.locate(User.class, args.join(" ")).collect{
    urlBase + historySvc.buildLinksFor(it).get(0)?.link
}
```

For the second part we need to return an object of type CommandResult (in package `net.datenwerke.rs.terminal.service.terminal.obj`). The `CommandResult` object allows us to better specify how the terminal treats the result returned by the script. It is able to display lists, tables, html and what we need for our task: links. The `CommandResult` object can create links either to external pages via `addResultHyperLink()` method or to internal locations via the `addResultAnchor()` method. Following is the final script.

```
package myscripts

import net.datenwerke.gf.service.history.HistoryService
import net.datenwerke.rs.core.service.reportserver.ReportServerService
import net.datenwerke.rs.search.service.search.SearchService
import net.datenwerke.rs.terminal.service.terminal.obj.CommandResult
import net.datenwerke.security.service.usermanager.entities.User

SearchService searchSvc = GLOBALS.getInstance(SearchService)
HistoryService historySvc = GLOBALS.getInstance(HistoryService)
String urlBase = GLOBALS.getInstance(ReportServerService).serverInfo.baseURL

CommandResult result = new CommandResult()
searchSvc.locate(User.class, args.join(" ")).collect{
    result.addResultHyperLink(it.getName(), historySvc.buildLinksFor(it).get(0)?. ↙
       ↳ link )
}

return result
```

Tip: Via the `terminal/alias.cf` configuration file you can make scripts directly accessible. For example the entry

```
<entry>
  <alias>search</alias>
  <command>exec /fileserver/bin/tmp/searchuser.groovy</command>
</entry>
```

would allow you to access your searchuser script from anywhere using the command search. If you change the config file don't forget to make ReportServer reload the configuration using the `config reload` terminal command.

3.4 Working with Entities

So far we have seen how we can access services via the GLOBALS object. In the following we will see how to work with entities. Entities are stored objects such as reports, users or TeamSpaces. You can find entities by searching for classes annotated with `@Entity`. You can also find a list of all entities in our ReportServer SourceForge project https://sourceforge.net/projects/dw-rs/. Download the latest apidocs file from the src directory for this.

Without directly spelling it out, we have in our above example already worked with user objects as the SearchService returns the actual entity objects. In the following we want to show how to access a particular entity by id. Almost all entities have a unique identifier which is usually called id and which normally can be accessed via the getId() method. Ids are in most cases of type Long. Go to the user manager in the administration module and select an arbitrary user (or use your searchuser script and click on a link). The user's id is displayed in the header line of the form that allows to set the user's properties. Let us assume that id is 4.

Usually, when you work with entities you will work with JPA's EntityManager (see https://docs.oracle.com/javaee/6/api/javax/persistence/EntityManager.html). The GLOBALS ob-

ject provides access to an EntityManager by either the method getEntityManager() or as a Provider via getEntityManagerProvider(). For quick access to an entity it furthermore provides the methods `findEntity()` and `getEntitiesByType()`. In the following we want to create a simple script that displays a list of all user objects. In a first step, we only access the user object with id 4 and display its name. We name our script `userlist.groovy`.

```
package myscripts

import net.datenwerke.security.service.usermanager.entities.User

GLOBALS.findEntity(User, 6l)
```

Note the "l" behind the id. This is necessary to tell Groovy that the id is of type Long and not an integer. If you execute the script via

```
exec userlist.groovy
```

the user's name is printed. What the findEntity method returned however, was the user object. Let's have a closer look at that object. For this we can either have a look at the source of User (located in `net.datenwerke.security.service.usermanager.entities`), the javadoc of the User class or we use the terminal command desc which takes as argument an entity name and outputs a description of the entity class. Run

```
desc User
```

on the terminal. You see the various database fields that a User object stores. By convention, each field has a corresponding getter and setter. Thus the "username" property could be accessed via "getUsername()". Let us change our script to output the username.

```
package myscripts

import net.datenwerke.security.service.usermanager.entities.User

User user = GLOBALS.findEntity(User, 6l)

user.username + ' - ' + user.name
```

If we want to list the username of all users we can use the following adaption

```
package myscripts

import net.datenwerke.security.service.usermanager.entities.User;

GLOBALS.getEntitiesByType(User).collect{
  it.username
}.join("\n")
```

Writing to Entities

Suppose we want to do a bulk update, for example, synchronize our user objects with an external database. As you can imagine this is straightforward using ReportServer scripts. In the following we will add a simple comment to the description field of any group. For this, we can use a simple adaption of the above script

```
package myscripts

import net.datenwerke.security.service.usermanager.entities.Group

GLOBALS.getEntitiesByType(Group).each{
  it.description = "some description"
  tout.println("changed group: " + it.name)
}

tout.println("done")
```

We call the file `editgroups.groovy`. To test our script, first go the the user management module in the admin module and create a few groups, let's say groups A and B. If you execute the script via `exec editgroups.groovy` you should see the following output:

```
changed group: A
changed group: B
done
```

However, if you look at your groups A and B nothing has changed, that is, the description was not added. This is because, by default the database transaction spanning a script execution is rolled backed after the script is finished. In order to commit the transaction you need to supply the flag "-c" to the exec command. Call your script via

```
exec -c editgroups.groovy
```

If you now inspect the groups in the user management area you will find that, indeed, the description was added. Instead of inspecting the group in the user manager you can do this directly from the terminal via the desc command. As a third parameter the desc command takes an object which is then displayed. Thus, you could, for example, write

```
desc Group /usermanager/A
```

if your group is located directly in the root directory. Alternatively, you can specify the object via an hql (hibernate query language) query. Thus, we could also access the Group with name "A" via

```
desc Group "hql:from Group where name='A'"
```

In this way, you can also display multiple objects side by side

```
desc Group "hql:from Group"
```

To display the result in a new window, that is, the result is not directly added to the terminal window, add the -w flag to the desc command. To complete the picture you can also specify objects on the terminal via type and id. Thus, if your group has id 6, you could also write

```
desc Group id:Group:6
```

3.5 Interpreting Script Error Messages

When developing scripts ReportServer will support your debugging efforts by showing you full stack trace error messages. Lets go back to our simple `userlist` script but this time, I have included a typo:

```
package myscripts

import net.datenwerke.security.service.usermanager.entities.User;

GLOBALS.getEntitiesByType(User).collect{
  i.getUsername()
}.join("\n")
```

If we name this file "errorfile.groovy" and execute it, you will get the following error message:

```
Script execution failed.
error message: No such property: i for class: myscripts.Script22 (groovy.lang.MissingPropertyException)
script arguments:
file: errorfile.groovy (id: 6372, line 6)
line number: 6 (5)
line:    i.getUsername()
```

The error message provides the specific reason for the failure:

```
error message: No such property: i for class: myscripts.Script22 (groovy.lang.MissingPropertyException)
```

This tells us that the variable “i” is undefined. The error report also specifies the exact file and line number:

```
file: errorfile.groovy (id: 6372, line 6)
line number: 6 (5)
line:    i.getUsername()
```

By looking at line 6, we can see the origin of the failure:

```
    i.getUsername()
```

The code uses `i`, but in this context, `it` should have been the standard Groovy iterator it.

3.6 Reading and Writing to Files

Next we describe how to write to files (in the internal filesystem) from a script. Basically there are two ways to accomplish this. We have already seen how to work with persistent objects (or rather entities) such as users. A file is just such an entity:

`net.datenwerke.rs.fileserver.service.fileserver.entities.FileServerFile`

The corresponding service is the FileServerService. Thus, we could create a new file and store it. There is, however, a simpler way using the GLOBALS object. The GLOBALS object comes with a dedicated `fileService` object, that allows to easily read from and write to files.

As a first step we construct a new text file. We open the terminal and go to the fileserver and create a new temporary folder (e.g. `tmp` with `mkdir tmp`). To write some text into a new file we can simply use the following terminal command

```
echo foobar > file.txt
```

The echo command prints its arguments and the angular opening bracket forwards this output into the file `file.txt`. A single angular bracket will overwrite the contents of the file. To append text to the file use two angular brackets, for example:

```
echo more foobar >> file.txt
```

To inspect the created file you can either use the `editTextFile` command or use the "cat" command which simply dumps the text file to the terminal.

```
cat file.txt
```

Next, we are going to write a simple script that is going to emulate the cat command. We will call this script `myCat.rs`. For this create the script using

```
createTextFile myCat.groovy
```

Remember that scripts need to be located beneath the bin folder. So if you created your script outside of the bin hierarchy move it there using the `mv` command. To emulate the cat command we only need a single line as the GLOBALS object offers some helper methods to read in files.

```
GLOBALS.read(args[0])
```

The read method takes a location of a file and returns the contents of the file as a string. Similarly the write method of the GLOBALS object allows to write to a text file. Note that this will overwrite the data within the file. Thus the following script emulates the terminal command "echo TEXT > file"

```
GLOBALS.write(args[0], args[1])
```

This could then be called, for example as

```
exec -c myWrite.groovy file.txt 'This is some text'
```

Note the use of the `-c` flag to commit the changes and the use of single quotes to treat `'This ↙ ↳ is some text'` as a single argument.

The read and write method of the GLOBALS object will always return (or expect) a string. If you want to work with binary files you can use the fileService. Via the globals object you have access to all Services provided by ReportServer. Additionally, there are a couple of services specifically for working with scripts. These are accessed via the services variable of the GLOBALS object. That is, we could have written our `myCat.groovy` script also as

```
GLOBALS.services['fileService'].read(args[0])
```

In addition to the read and write methods the fileService has additionally the methods readRaw and writeRaw that allow to work directly with byte representations of the files.

Besides the fileService there are other services specifically available for script developers to help facilitate writing scripts. We will encounter them in the course of this manual. However, here is a short overview of the services available via GLOBALS.services

fileService	easy access to read and write files.
scriptService	provides methods to call scripts from within your script.
registry	A singleton registry that can be used to store and retrieve arbitrary values. The registry is held in memory and cleared on ReportServer restarts.
clientExtensionService	An interface to access client side extensions.

3.7 Nesting Scripts: Calling Scripts from Scripts

To properly structure larger scripts it can be helpful to spread functionality over several scripts. The above mentioned scriptService provides simple helper methods to call scripts from within scripts. For this, consider the following script which does nothing but provide two methods

```
def caesarEncode(k, text) {
    (text as int[]).collect { it==' ' ? ' ' : (((it & 0x1f) + k - 1).mod(26) + 1 | ↙
     ↳   it & 0xe0) as char }.join()
}
def caesarDecode(k, text) { caesarEncode(26 - k, text) }
```

Assume this is stored in a file called `lib.groovy`. For the interested reader, this is a simple implementation of the Caesar cipher found here: https://www.rosettacode.org/wiki/Caesar_cipher#Groovy (note this should not be used for actual encryption). Now, to load these helper methods we can use the following, which we store in a file called `nestingTest.groovy`

```
GLOBALS.exec('lib.groovy')

String plain = 'The quick brown fox jumps over the lazy dog'
int key = 6
String cipher = caesarEncode(key, plain)

tout.println plain
tout.println cipher
tout.println caesarDecode(key, cipher)
```

If you now run this script you should get the following output

```
reportserver$ exec nestingTest.groovy
The quick brown fox jumps over the lazy dog
Znk waoiq hxuct lud pasvy ubkx znk rgfe jum
The quick brown fox jumps over the lazy dog
```

The example above can be found here: https://github.com/infofabrik/reportserver-samples/tree/main/src/net/datenwerke/rs/samples/tools/nesting.

> Note that ReportServer uses internal script caching for performance optimization. When developing nested scripts, if you get an old script version when running, you can manually clear scripting cache with the `clearInternalScriptCache` terminal command.

Note you can use relative paths for `lib.groovy`. For example:

```
GLOBALS.exec('../crypt/libs/lib.groovy')
```

or

```
GLOBALS.exec('libs/lib.groovy')
```

Besides the simple exec method that we have used above the GLOBALS object provides a second exec method that takes as second parameter an argument string that is passed to the script.

Using classes in nested scripts

When you need to use classes in nested scripts, you can not use the

```
GLOBALS.exec('../crypt/libs/lib.groovy')
```

method explained above directly. With other words, this would not work if `B` contains a class definition:

```
GLOBALS.exec('/fileserver/bin/B.groovy')
B b = new B()
b.prepareString()
```

Groovy would try to compile your script, but cannot find `B`'s class definition during compilation time.

In this case, you can use the methods `GLOBALS.loadClass()` and `GLOBALS.loadClasses()` for loading your class definitions. Using `GLOBALS.newInstance()` allows you to create instances as shown below. Finally, you can call your instance's methods, in this example `prepareString()`.

```
def bClass = GLOBALS.loadClass('B.groovy', 'net.datenwerke.rs.samples.tools. ↙
    ↪ nesting.nestedclass.B')
def bInstance = GLOBALS.newInstance(bClass)
return bInstance.prepareString()"
```

A complete example using two levels (A.groovy creating B objects creating C objects) can be found here: https://github.com/infofabrik/reportserver-samples/tree/main/src/net/datenwerke/rs/samples/tools/nesting/nestedclass.

The following example shows how to load classes which are defined in the same `.groovy` file (B and C are defined both in `myLibraries.groovy`): https://github.com/infofabrik/reportserver-samples/tree/main/src/net/datenwerke/rs/samples/tools/nesting/multipleclass

Chapter 4

Monitoring, Multithreading, Scheduling and other Advanced Techniques

In this chapter we are covering several advanced scripting techniques such as

- how to monitor script executions (and possibly stop the execution of a script)
- how to work with multiple threads
- how to use the registry to temporarily or persistently store values and how to schedule scripts.

Let us start by looking at how ReportServer executes scripts via the terminal.

4.1 Execution of Scripts

Whenever you execute a script via the terminal command exec the script execution is wrapped into a new thread. This allows to monitor the execution and if necessary stop the script. You can display a list of the currently running scripts via the terminal command `ps`.

The following script doesn't do much, but it needs almost one minute for it:

```
import java.lang.Thread

Thread.sleep(60000)

"awake again"
```

If you run this script, the Terminal window will remain inactive during the time of execution. You can either open a second terminal window (CTRL+ALT+T), or wait for the script to return and then run it using the silent flag:

```
exec -s sleep.groovy
```

The silent flag tells ReportServer to ignore the output of the script and run it in the background.

Now, by using the command "ps" you can view executions which are presently active.

```
reportserver$ ps
ID  Date       User   Command       Thread Interrupted
1   03.05.13 16:23  3     exec -s sleep.groovy  false
```

By entering the `kill` command, you can cancel scripts. Here, first an interrupt will be sent to the script which in our case leads to sending the thread an interrupt. In our case we can interrupt the script as follows.

```
reportserver$ kill 1
```

When you call "ps" again you will see that the script was interrupted indeed. The following script, however, will not be terminated that easily. It catches any exception and thus also exceptions thrown on interrupt.

```
import java.lang.Thread

boolean slept = false;
while(! slept){
  try{
    Thread.sleep(60000)
        slept = true;
   } catch(all) {
   }
}
"done"
```

If you run this script and try to terminate it using "kill ID", you will see that the script is still listed by the "ps" command. Note that the flag "interrupted" is now set.

```
reportserver$ ps
ID  Date      User   Command       Thread Interrupted
7 03.05.13 16:23   3     exec -s sleep.groovy  true
```

To hard-terminate the execution you can enter `kill -f ID`. This will terminate the thread by `Thread.stop()`. Please keep in mind that this may have undesirable side effects. You will find a description of the `Thread.stop()` method and related issues under https://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html.

Note that scripts which are executed not via the terminal but, for example, during a script report execution or via the scheduler will not run in an extra thread. They will, thus, also not be listed by the ps command.

Starting scripts in the server thread

In some situations it might be helpful to avoid starting a script in an own thread, but to start it in the server thread. This means, however, that this script cannot be monitored and interrupted by "ps/kill". To start a script without an own thread use the `-n` flag.

4.2 Storing Values Between Scripts

When a script terminates its scope is cleared and any variables are lost. Sometimes it is helpful to store values which can then be retrieved at a later point, for example, by a different script.

ReportServer provides script developers with two mechanisms to easily store and retrieve values at a later point. The script registry is kept in memory by ReportServer at all times and, thus, allows for very fast access. However, when ReportServer is restarted the registry is cleared. To store values persistently, ReportServer offers the so called PropertiesService. Values stored via this service will be pushed to the database.

The registry is available via the GLOBALS object. Consider the following script `registryCnt.groovy`.

```
def registry = GLOBALS.services['registry']
int cnt = registry.containsKey("myCnt") ? registry.get("myCnt") : 0

cnt++;
registry.put("myCnt", cnt)

"The count is at: " + cnt
```

Executing this script multiple times will yield the following output

```
reportserver$ exec registryCnt.groovy
The count is at: 1
reportserver$ exec registryCnt.groovy
The count is at: 2
reportserver$ exec registryCnt.groovy
The count is at: 3
```

The registry implements `java.util.Map<String,Object>`. You can find detailed documentation on the various methods at https://docs.oracle.com/javase/7/docs/api/java/util/Map.html.

> Note you can list, add and modify all entries in the registry with the `registry` terminal command. You can find more information on the Admin Guide: https://reportserver.net/en/guides/admin/main/.

To store values persistently you need to use the PropertiesService (`net.datenwerke.gf.service.properties.PropertiesService`: https://reportserver.net/api/latest/javadoc/net/datenwerke/gf/service/properties/PropertiesService.html) which is a regular ReportServer service. Let us implement our same counter script as `persistentCnt.groovy`.

```
import net.datenwerke.gf.service.properties.PropertiesService

PropertiesService registry = GLOBALS.getInstance(PropertiesService)
int cnt = registry.containsKey('myCnt') ? registry.get('myCnt') as int : 0

cnt++
registry.setProperty('myCnt', cnt as String)

"The count is at: $cnt"
```

Note the `as int` and `as String`. The PropertiesService stores all its properties in form of character strings. If we execute the script, we get the following output.

```
reportserver$ exec persistentCnt.groovy
The count is at: 1
```

```
reportserver$ exec persistentCnt.groovy
The count is at: 1
reportserver$ exec persistentCnt.groovy
The count is at: 1
```

This is not quite as expected. If we have a look at the corresponding database table `RS_PROPERTY` you will see that the table is still empty. Alternatively, you can use the following terminal command

```
desc Property "hql:from Property"
```

which lists all entities of type Property. The reason is simple. We did not run the exec command in commit mode and, thus, the property change was not persisted. If we run the command again, this time with the `-c` flag, we get the following output.

```
reportserver$ exec -c persistentCnt.groovy
The count is at: 1
reportserver$ exec -c persistentCnt.groovy
The count is at: 2
reportserver$ exec -c persistentCnt.groovy
The count is at: 3
```

We can now also inspect the property using the desc command from before, which now yields:

```
d  key  version  value
1  myCnt 2   3
```

Note you can list, add and modify all entries in the properties mapping with the `properties` terminal command. You can find more information on the Admin Guide: https://reportserver.net/en/guides/admin/main/.

4.3 Multithreading in Scripts

Sometimes it can be convenient to have work done in parallel. The common way to do this in Groovy or Java is to conjure up multiple threads that do the actual work and run them in parallel. Usually you would want to have the threads controlled by a thread pool. For this ReportServer provides an easy mechanism. The following script is a simple example of how to create thread pools using ReportServer's DwAsyncService (`net.datenwerke.async.DwAsyncService`):

```
import java.util.concurrent.Future
import net.datenwerke.async.DwAsyncPool
import net.datenwerke.async.DwAsyncService
import net.datenwerke.async.configurations.*

DwAsyncService aService = GLOBALS.getInstance(DwAsyncService)
String poolName = 'myPool'

// get pool
DwAsyncPool pool = aService.initPool(poolName, new SingleThreadPoolConfig())

Runnable runA = {
  (1..10).each{
    tout.println "A says: $it"
```

```
      Thread.sleep(200)
  }
} as Runnable

Runnable runB = {
  (1..10).each{
    tout.println "B says: $it"
      Thread.sleep(200)
  }
} as Runnable

Future futureA = pool.submit(runA)
Future futureB = pool.submit(runB)

// wait for tasks
futureA.get()
futureB.get()

aService.shutdownPool(poolName)
```

The initPool method takes a name and a configuration and creates a new pool for the given configuration. If a pool with the same name already exists, it will first shutdown the old pool. If we run this script we get the following result:

```
reportserver$ exec threadExample.groovy
A says: 1
A says: 2
A says: 3
A says: 4
A says: 5
A says: 6
A says: 7
A says: 8
A says: 9
A says: 10
B says: 1
B says: 2
B says: 3
B says: 4
B says: 5
B says: 6
B says: 7
B says: 8
B says: 9
B says: 10
```

There are two things to note. First, the `println`s are not immediately pushed to the client. Currently ReportServer waits until the script terminates before it sends the result (which includes `println`s) to the client. Secondly, the script executed the two loops not in parallel, but consecutively. This is, because of our choice of thread pool. We used a SingleThreadPoolConfig, which constructs a thread pool consisting of a single thread. Besides the SingleThreadPoolConfig you can use

a FixedThreadPoolConfig which generates a pool with a fixed size. Thus, if we exchange the SingleThreadPoolConfig by

```
new FixedThreadPoolConfig(2)
```

and rerun our program, we get the following (expected) output:

```
reportserver$ exec threadExample.groovy
A says: 1
B says: 1
B says: 2
A says: 2
A says: 3
B says: 3
B says: 4
A says: 4
B says: 5
A says: 5
B says: 6
A says: 6
A says: 7
B says: 7
B says: 8
A says: 8
B says: 9
A says: 9
B says: 10
A says: 10
```

4.4 Scheduling Scripts

The ReportServer scheduler is not only used to schedule reports, but it can also be used to schedule the execution of scripts. To schedule a script use the terminal command `scheduleScript`. It comes with two commands:

list	Lists all scheduled scripts
execute	Schedules a new script

To schedule a script you can use natural language expressions. Examples are

```
scheduleScript execute myScript.groovy " " today at 15:23
scheduleScript execute myScript.groovy " " every day at 15:23
scheduleScript execute myScript.groovy " " at 23.08.2012 15:23
scheduleScript execute myScript.groovy " " every workday at 15:23 starting on ↙
    ↳ 15.03.2011 for 10 times
scheduleScript execute myScript.groovy " " every hour at 23 for 10 times
scheduleScript execute myScript.groovy " " today between 16:00 and 23:00 every 10 ↙
    ↳ minutes
scheduleScript execute myScript.groovy " " every week on monday and wednesday at ↙
    ↳ 23:12 starting on 27.09.2011 until 28.11.2012
scheduleScript execute myScript.groovy " " every month on day 2 at 12:12 starting ↙
    ↳ on 27.09.2011 11:25 for 2 times
```

The " " (quotation marks) after the script name are the scripts arguments. If we schedule our previous `persistentCnt.groovy` script we get the following output

```
scheduleScript execute persistentCnt.groovy " " every hour at 23 for 10 times
Script persistentCnt.groovy scheduled. First execution: 13.12.2013 19:23:00
```

Via the scheduleScript list command you get an overview of all currently scheduled scripts:

```
reportserver$ scheduleScript list
id scriptId name      next firetime
1  573   persistentCnt.groovy 13.12.2013 19:23:00
```

To get an overview of the next fire times you can use the scheduler command.

```
reportserver$ scheduler listFireTimes 1
13.12.2013 19:23:00
13.12.2013 20:23:00
13.12.2013 21:23:00
13.12.2013 22:23:00
13.12.2013 23:23:00
14.12.2013 19:23:00
14.12.2013 20:23:00
14.12.2013 21:23:00
14.12.2013 22:23:00
14.12.2013 23:23:00
```

Here 1 denotes the schedule entry's id. The scheduler command can also be used for scheduled reports.

To remove an entry you can use the "scheduler remove" command, for example to remove the above entry you need to run

```
reportserver$ scheduler remove 1
```

Chapter 5

Script Reporting

Besides for administrative tasks, ReportServer scripts can be used for generating reports or for providing datasources that can then be used together with any other reporting engine within ReportServer. If you are familiar with scripting and fluent in HTML and CSS you will find script reports a simple to use reporting alternative that is highly flexible. Script reports usually render to HTML which allows to create highly dynamic reports using AJAX and advanced web technologies. Naturally, you are not limited to HTML reporting but are free to produce any output format you choose, for example, PDF or even a zip archive containing multiple files. ReportServer supports you in that it provides renderers to turn HTML formatted reports into PDF and Microsoft Word (`docx`) files. Furthermore, ReportServer provides a simple to use exporter to generate Microsoft Excel documents. An example of a script report using the PDF renderer is the report documentation report that is shipped with ReportServer (see Administrators guide).

As for datasources, script datasources give you the flexibility to easily integrate any sort of data with ReportServer. A script datasource can even accumulate data from various sources, it can be used to implement a multi-platform join.

Generating Reports using Scripts. A script report consists of at least two objects: a script and a report. In addition a script report can be configured to use a datasource. In the following we are going to, step by step, build several example reports:

- The first report we are going to create is a simple static report that allows us to show the basic concepts of script reports. This includes exporting into various output formats, working with Groovy's (X)HTML builder and working with parameters and arguments. If you have read the chapter on script reports in the administrator's manual most of this should be familiar.

- The second script will explain how to work with datasources.

- The third and final example will explain how to create a dynamic report, that allows for user interaction and which uses a dynamic list as data backend.

5.1 Basic Script Reports

By default script reports are supposed to generate HTML. Thus, all we really need is to return HTML code as in the following example (note that we are using Groovy's multiline string feature here).

```
return """\
<html>
<head>
  <title>Hello World</title>
</head>
<body>
   <h1>Hello World</h1>
</body>
</html>
"""
```

Let us store this script as `/bin/reports/report1.groovy`. The next step is to define a script report. For this go to the report manager and create a new object of type script report. Provide a name and select the previously create script as script reference. After submitting the data you can execute the report as usual by simply double clicking on the item from the report manager. You should see that the report simply displays the content as defined by the HTML code. In contrast to other reports, there is no button for scheduling the report or exporting it. This is because we have not yet defined output formats for the report. For this go back to the script report in the report manager. Here you can define output Formats as a comma separated list. For example we can define

```
HTML, Plain
```

Submit the changes and reopen the report. You see that you now have two export options: HTML and Plain. However, both options produce the same, somewhat unexpected result. The browser displays the HTML code rather than the interpreted website. This is, because we have not specified the return type. For this, we need to return an object of type `net.datenwerke.rs.core.service.reportmanager.engine.CompiledReport` which besides the report's content contains information on, for example, the resulting mime type. Several predefined and simple to use objects are available:

`net.datenwerke.rs.core.service.reportmanager.engine.basereports.CompiledTextReportImpl`
: For simple text documents.

`net.datenwerke.rs.core.service.reportmanager.engine.basereports.CompiledHtmlReportImpl`
: For html documents.

`net.datenwerke.rs.core.service.reportmanager.engine.basereports.CompiledDocxReport`
: For Microsoft Word documents.

`net.datenwerke.rs.core.service.reportmanager.engine.basereports.CompiledXlsxReport`
: For Microsoft Excel documents.

`net.datenwerke.rs.core.service.reportmanager.engine.basereports.CompiledJsonReport`
: For JSON documents.

`net.datenwerke.rs.core.service.reportmanager.engine.basereports.CompiledCsvReport`
: For comma separated value documents.

`net.datenwerke.rs.core.service.reportmanager.engine.basereports.CompiledPdfReport`
: For PDF documents.

Thus, to make the browser render the output we could adapt our above script to

```
import net.datenwerke.rs.core.service.reportmanager.engine.basereports. ↙
    ↳ CompiledHtmlReportImpl

String report = """\
<html>
<head>
  <title>Hello World</title>
</head>
<body>
   <h1>Hello World</h1>
</body>
</html>
"""

return new CompiledHtmlReportImpl(report)
```

While the browser now renders the exported report properly, it also does so when using the export option **Plain**. To differentiate between the two output formats scripts have access to a predefined variable **outputFormat**. This variable contains the selected output format. Consider the following adaption

```
import net.datenwerke.rs.core.service.reportmanager.engine.basereports. ↙
    ↳ CompiledHtmlReportImpl

String report = """\
<html>
<head>
  <title>Hello World</title>
</head>
<body>
   <h1>Hello World</h1>
</body>
</html>
"""

if(outputFormat == 'plain')
  return 'Hello World'
return new CompiledHtmlReportImpl(report)
```

Note that, although the output format was defined as **Plain**, the script is given the output format in lower case and with leading and trailing whitespace removed. Now if you export the report to **Plain** you get the expected plain Hello World response, while an export to **HTML** will lead to a rendered Hello World response.

5.2 Groovy's Markup Builder

In the above example we wrote the HTML code as a plain text string. In the following example we use Groovy's Markup XML Builder.

```
import net.datenwerke.security.service.usermanager.entities.User
import net.datenwerke.security.service.authenticator.AuthenticatorService
import net.datenwerke.rs.core.service.reportmanager.engine.basereports. ↙
    ↳ CompiledHtmlReportImpl
import groovy.xml.*

User user = GLOBALS.getInstance(AuthenticatorService).currentUser

StringWriter writer = new StringWriter()

new MarkupBuilder(writer).html {
   head {
     title ( 'Hello World' )
   }
   body {
     h1("Hello ${user.firstname}")
   }
 }

return new CompiledHtmlReportImpl(writer.toString())
```

Besides using the Markup Builder we have personalized the greeting a bit. Let us add some extra styling:

```
import net.datenwerke.security.service.usermanager.entities.User
import net.datenwerke.security.service.authenticator.AuthenticatorService
import net.datenwerke.rs.core.service.reportmanager.engine.basereports. ↙
    ↳ CompiledHtmlReportImpl
import groovy.xml.*

User user = GLOBALS.getInstance(AuthenticatorService).currentUser

StringWriter writer = new StringWriter()

new MarkupBuilder(writer).html {
   head {
     title ( 'Hello World' )
   }
   body {
     h1(style: 'color: #f00', "Hello ${user.firstname}")
     p("Isn't that an easy way to create a report?")
   }
 }

return new CompiledHtmlReportImpl(writer.toString())
```

5.3 Generating PDF and Word output

Now that we have a simple styled report, let us export it not only to HTML but also to PDF and Microsoft Word. Basically, what we need to do is to output the byte code expected by a PDF reader (resp. a valid Word document). For this we can either use one of many java libraries such as Apache POI (https://poi.apache.org/). ReportServer, however, makes it easy for you to go from HTML directly to PDF and Word. For this you have access to two renderers via the predefined object renderer. First go back to the report manager and define the following output Formats

```
HTML, PDF, Word
```

Now, we need to adapt the report

```
import net.datenwerke.security.service.usermanager.entities.User
import net.datenwerke.security.service.authenticator.AuthenticatorService
import groovy.xml.*

User user = GLOBALS.getInstance(AuthenticatorService).currentUser

StringWriter writer = new StringWriter()

new MarkupBuilder(writer).html {
   head {
     title ( 'Hello World' )
   }
   body {
     h1(style: 'color: #f00', "Hello ${user.firstname}")
     p("Isn't that an easy way to create a report?")
   }
 }

if(outputFormat == 'word')
  return renderer.get("docx").render(writer.toString())
if(outputFormat  == 'pdf')
  return renderer.get("pdf").render(writer.toString())

return renderer.get("html").render(writer.toString())
```

You can now export the report to the various formats. Also note, that the HTML renderer that we used in the very last line is equivalent to using `CompiledHtmlReportImpl`.

5.4 Rendering a DOT graph as SGV in a script report

DOT graphs https://graphviz.org/doc/info/lang.html allow to define Graphviz nodes, edges, graphs, subgraphs, and clusters in a text-based manner. The resulting DOT file can be then converted into a SVG file, which can be displayed directly by most browsers.

Here: https://github.com/infofabrik/reportserver-samples/tree/main/src/net/datenwerke/rs/samples/admin/svg/scriptreport we added examples on how to render a given DOT file to SVG and show it directly in the browser as a script report.

Notice that you can insert the $mySvgString directly into your HTML without further changes.

As of ReportServer 4.7.0 you can use the "dot-svg", "dot-png" and "dot-html" renderers directly. Here you can see two examples of this: https://github.com/infofabrik/reportserver-samples/blob/main/src/net/datenwerke/rs/samples/admin/svg/scriptreport/DOT_SVG_renderer.groovy and https://github.com/infofabrik/reportserver-samples/blob/main/src/net/datenwerke/rs/samples/admin/svg/scriptreport/DOT_SVG_renderer_file.groovy.

The "dot-svg" renderer renders a given DOT file as a SVG graph, the "dot-png" renders a given DOT file as a PNG image, and the "dot-html" renders the DOT file as a SVG graph and embeds it into a HTML page.

You can use the renderers analogously as the renderers shown in Section 5.3 Generating PDF and Word output:

```
String dot = '''
digraph D {
  A -> {B, C, D} -> {F}
}
'''
return renderer.get("dot-svg").render(dot)
```

Note that the above examples are very powerful yet simple to use, as you can create the DOT file dynamically.

5.5 Rendering Markdown as HTML in a script report

Markdown is an easy-to-use markup language that is used to add formattig elements(headings, bulleted lists, URLs) to plain text(https://www.markdownguide.org/). As of ReportServer 4.7.0 markdown files in the virtual filesystem can be previewed in the corresponding tab and the "markdown-html" renderer allows you to produce a similar preview contained within a CompiledHtmlReport.

Here: https://github.com/infofabrik/reportserver-samples/blob/RS-7743/src/net/datenwerke/rs/samples/admin/markdown/scriptreport/MARKDOWN_HTML_renderer.groovy we added an example on how to render a given markdown file to HTML and show it directly in the browser as a script report.

Although most commands are supported, there are exceptions which cannot be parsed to HTML.

You can use the renderers analogously as the renderers shown in Section 5.3 Generating PDF and Word output:

```
String mdContent = '''
# h1 Heading
## Emphasis

**This is bold text**

__This is bold text__
```

```
*This is italic text*

_This is italic text_
'''
renderer.get("markdown-html").render(mdContent)
```

5.6 Running Script Reports from the Terminal

For testing it might be helpful to run the script directly from the terminal. However, if you try to do this, with our above script you get the following exception:

```
reportserver$ exec report1.rs
net.datenwerke.rs.scripting.service.scripting.exceptions.ScriptEngineException: javax.script. ↙
      ↳  ScriptException: javax.script.ScriptException: groovy.lang.MissingPropertyException: No such  ↙
      ↳  property: outputFormat for class: Script13
...
```

The problem is that we used the variable outputFormat which is supplied by the script reporting engine but if we run the script as is, this variable does not exist.

```
import net.datenwerke.security.service.usermanager.entities.User
import net.datenwerke.security.service.authenticator.AuthenticatorService
import groovy.xml.*

// define outputFormat if not already in binding
if(! binding.hasVariable('outputFormat'))
  outputFormat = null

User user = GLOBALS.getInstance(AuthenticatorService).currentUser

StringWriter writer = new StringWriter()

new MarkupBuilder(writer).html {
   head {
     title ( 'Hello World' )
   }
   body {
     h1(style: 'color: #f00', "Hello ${user.firstname}")
     p("Isn't that an easy way to create a report?")
   }
 }

if(outputFormat == 'word')
  return renderer.get("docx").render(writer.toString())
if(outputFormat  == 'pdf')
  return renderer.get("pdf").render(writer.toString())
if(outputFormat == 'html')
  return renderer.get("html").render(writer.toString())

return writer.toString()
```

The variable binding is a special groovy variable that allows you to access the available variable bindings for the current script. It is an object of type `groovy.lang.Binding`. By not giving the

outputFormat a type, that is by defining it as

```
outputFormat = null
```

instead of as

```
def outputFormat = null
```

or

```
String outputFormat = null
```

we add the variable to the binding instead of the current scope (which would be limited by the if clause. Thus, we can safely later on assume that the variable is defined. Note also that we slightly changed the final return value as also the renderer is a special object provided by the reporting engine.

5.7 Testing via Browsers

Simpler than testing via the terminal is to open the report in its own browser window by directly accessing it via the URL. For this use

```
http://SERVER:PORT/rsbasedir/reportserver/reportexport?id=REPORT_ID&format=HTML
```

More information on calling reports directly via the URL can be found in the administrator's guide.

5.8 Working with Parameters and Arguments

To conclude our first simple report example we want to show you how to access parameters and work with script arguments. The latter is easily established. If you return to the report management and select the script report you see a field for arguments. These arguments are passed to the script in the same way as command line arguments are passed to scripts. Thus, the following would simply output the arguments

```
new MarkupBuilder(writer).html {
   head {
     title ( 'Hello World' )
   }
   body {
     h1(style: 'color: #f00', "Hello ${user.firstname}")
     p("Isn't that an easy way to create a report?")
     p('Arguments: ' + args.join(','))
   }
 }
```

Parameters are handled similarly. Parameters can be accessed via the variables **parameterSet** and **parameterMap**. The first variable points to the ReportServer ParameterSet object. Other than the **parameterMap** this does not only store parameter keys and values but contains additional information on the parameter. See net.datenwerke.rs.core.service.reportmanager.parameters.ParameterSet for more details. Usually only the **parameterMap** variable is needed which stores the parameters in a

java.util.Map<String, Object>. Let us add a simple text parameter to our report and we give it the name **param**.

```
new MarkupBuilder(writer).html {
   head {
     title ( 'Hello World' )
   }
   body {
     h1(style: 'color: #f00', "Hello ${user.firstname}")
     p("Isn't that an easy way to create a report?")
     p('Arguments: ' + args.join(','))
     p('The single parameter is: ' + parameterMap['param'])
   }
 }
```

That concludes the basic example. In the next section we see how to work with datasources.

5.9 Working with Datasources

In the following section we are going to create a second script report that directly accesses a datasource. Let us create a new script called report2.groovy which we also put into /bin/reports. We will use Groovy's Sql functionality to directly access the database. Create a new script report in the report manager. As for any other report type you can select a datasource for the report. This datasource will then be given to your script via the predefined variable **connection**. In case the datasource is a relational database the **connection** variable would hold an actual database connection. Note that there is no need to close the connection as ReportServer will do that for you. The following script accesses the table T_AGG_CUSTOMER from the demo database.

```
import groovy.sql.Sql
import groovy.xml.*

StringWriter writer = new StringWriter()

new MarkupBuilder(writer).html {
   head {
     title ( 'Hello World' )
   }
   body {
     table {
     new Sql(connection).eachRow("SELECT CUS_CUSTOMERNAME, CUS_CREDITLIMIT FROM  ↙
     ↳  T_AGG_CUSTOMER") { row ->
     tr {
       td(row.CUS_CUSTOMERNAME)
            if(row.CUS_CREDITLIMIT > 100000)
                td(style:'color:#F00', row.CUS_CREDITLIMIT)
            else
           td(row.CUS_CREDITLIMIT)
     }
        }
     }
```

```
   }
 }

renderer.get('html').render(writer.toString())
```

5.10 Interactive Reports

As a final example we want to create a simple dynamic report which uses a dynamic list as its backend. For dynamic reports the logic will be written in JavaScript and the Groovy script will be mostly a container serving the JavaScript code. Create the following script as `report3.groovy` in `/bin/reports` and create a fresh script report pointing to it.

```
import groovy.sql.Sql
import groovy.xml.*

StringWriter writer = new StringWriter()

new MarkupBuilder(writer).html {
   head {
     title ( 'Dynamic World' )
     script(language: 'javascript', type: 'text/javascript', """\
      alert('Hello Dynamic World')
     """ )
   }
   body {
   }
 }

renderer.get('html').render(writer.toString())
```

If you call the report you will be greeted with a JavaScript alert message. Let us now add some content to our report. For this we will access the dynamic list `T_AGG_CUSTOMER` (from the demo package). We assume that the list has key "customer". We will use jquery to easily modify the DOM. We query the dynamic list to retrieve two attributes and build a table of the results.

```
import groovy.sql.Sql
import groovy.xml.*

StringWriter writer = new StringWriter()

new MarkupBuilder(writer).html {
   head {
     title ( "Dynamic World" )
     script(language: "javascript", type: "text/javascript", src: "https://code.jquery.com/jquery-3.7.1. ↙
      ↳  min.js", " " )
   }
   body {
     h1("Dynamic World" )
     table (id: "content", " ")
     script(type: "text/javascript") { mkp.yieldUnescaped("""
  \$.getJSON( 'http://127.0.0.1:8888/reportserver/reportexport?key=customer&c_1=CUS_CUSTOMERNAME&c_2= ↙
       ↳  CUS_CREDITLIMIT&format=json', function(data) {
     \$.each( data, function( key, val ) {
          \$( "<tr><td>" + val['CUS_CUSTOMERNAME'] + "</td><td>" + val['CUS_CREDITLIMIT'] + "</td></tr>" ↙
      ↳  ).appendTo("#content");
        });
```

```
      });
  """ ) }
   }
 }

renderer.get('html').render(writer.toString())
```

So, what have we done? First, note that self-closing script tags are likely to produce errors. Thus, we added a dummy space as content to the script tag that is loading jquery. A second change we introduced is to call `mkp.yieldUnescaped` to write our javascript code. This is to ensure that entities such as "&" are not escaped. Now, we get to the actual content retrieval. We used jquery's `getJSON` function to load data from the following URI

http://127.0.0.1:8888/reportserver/reportexport?key=customer&c_1=CUS_CUSTOMERNAME&c_2=CUS_CREDITLIMIT&format=json

Here we specified to use the report with key **customer**, and selected two columns: `CUS_CUSTOMERNAME` and `CUS_CREDITLIMIT`. Additionally, we told ReportServer to return data in the JSON format. Now all we did was to loop over the retrieved data and added an entry to our table.

So far, we have only created a static script, so let us add some dynamic elements to it. We will add an input field that allows to specify a minimum credit limit and on changes retrieve the filtered data from the server.

```
import groovy.sql.Sql
import groovy.xml.*

StringWriter writer = new StringWriter()

new MarkupBuilder(writer).html {
  head {
     title ( "Dynamic World" )
     script(language: "javascript", type: "text/javascript", src: "https://code.jquery.com/jquery-3.7.1.
      ↪ min.js", " " )
   }
   body {
     h1("Dynamic World" )
     input (id: "filter", type: "text", value: "10000")
     input (id: "btn", type: "button", value: "Retrieve Filtered Data" )
     table (id: "content", " ")
     script(type: "text/javascript") { mkp.yieldUnescaped("""
    \$("#btn").click(function(event){
      var val = \$("#filter").val();
      \$("#content").html("<tr><td>loading data</td></tr>");
      \$.getJSON( 'http://127.0.0.1:8888/reportserver/reportexport?key=customer&c_1=CUS_CUSTOMERNAME&c_2=
           ↪ CUS_CREDITLIMIT&format=json&or_2=ASC&fri_2=' + val + ' -', function(data) {
        \$("#content").empty();
        \$.each( data, function( key, val ) {
             \$( "<tr><td>" + val['CUS_CUSTOMERNAME'] + "</td><td>" + val['CUS_CREDITLIMIT'] + "</td></tr
      ↪ >").appendTo("#content");
           });
       });
    });
  """ ) }
   }
 }

renderer.get('html').render(writer.toString())
```

First, you should notice that we added a static textbox and a button. In our javascript code we added a callback to click events on the button. If the button is clicked, we first clear the content of our table (that we address by its id **content**) and read out the value of the textbox. We then perform an ajax call to URI

```
http://127.0.0.1:8888/reportserver/reportexport?key=customer&c_1=CUS_CUSTOMERNAME&
c_2=CUS_CREDITLIMIT&format=json&or_2=ASC&fri_2='+val+'-'
```

As **val** contains the value of the textfield (say 10000) this would result in the following URL

```
http://127.0.0.1:8888/reportserver/reportexport?key=customer&c_1=CUS_CUSTOMERNAME&
c_2=CUS_CREDITLIMIT&format=json&or_2=ASC&fri_2=10000-'
```

With this URL we have introduced two changes. First, we order the resulting data by column 2, that is by `CUS_CREDITLIMIT`. Secondly, we added a range filter and use the result of the filter value as a lower bound.

Chapter 6

Script Datasources

Script datasources are your swiss army knife when it comes to integrating data from various formats. Basically, what script datasources do is to produces any sort of data and output it in a table format that ReportServer understands. This data will then be loaded (and potentially cached) in ReportServer's internal database (see Administrator's guide) which then allows you to build any sort of report (for example a dynamic list) on top of it.

Say you have an XML dataset such as the following dataset taken from **socrata** (a platform for open data):

https://opendata.socrata.com/api/views/2dps-ayzy/rows.xml?accessType=DOWNLOAD

It lists the data in the following XML format:

```
<response>
  <row>
    <row _id="row-cwj5-dis8~w6z6"
      _uuid="00000000-0000-0000-2B3F-C9C31B6F54CD" _position="0"
      _address="https://opendata.socrata.com/resource/_2dps-ayzy/row-cwj5-dis8~w6z6">
      <employee_name>ABBOT, JUDITH L</employee_name>
      <office>SENATOR TOBY ANN STAVISKY</office>
      <city>FLUSHING</city>
      <employee_title>COMMUNITY LIAISON</employee_title>
      <biweekly_hourly_rate>1076.93</biweekly_hourly_rate>
      <payroll_type>SA</payroll_type>
      <pay_period>23</pay_period>
      <pay_period_begin_date>2019-02-07T00:00:00</pay_period_begin_date>
      <pay_period_end_date>2019-02-20T00:00:00</pay_period_end_date>
      <check_date>2019-03-06T00:00:00</check_date>
      <legislative_entity>SENATE EMPLOYEE</legislative_entity>
    </row>
    <row _id="row-6b26~66gt~f43k"
      _uuid="00000000-0000-0000-6047-FAD2CDBD8A36" _position="0"
      _address="https://opendata.socrata.com/resource/_2dps-ayzy/row-6b26~66gt~f43k">
      <employee_name>ABRAHAM, PRINCY A</employee_name>
      <office>MINORITY COUNSEL/PROGRAM</office>
      <city>ALBANY</city>
      <employee_title>ASSOCIATE COUNSEL</employee_title>
      <biweekly_hourly_rate>2376.93</biweekly_hourly_rate>
      <payroll_type>RA</payroll_type>
      <pay_period>23</pay_period>
      <pay_period_begin_date>2019-02-07T00:00:00</pay_period_begin_date>
      <pay_period_end_date>2019-02-20T00:00:00</pay_period_end_date>
```

```
      <check_date>2019-03-06T00:00:00</check_date>
      <legislative_entity>SENATE EMPLOYEE</legislative_entity>
    </row>
  </row>
</response>
```

In order to make this data available in ReportServer, we need to load this data using a simple script and transform it into an object of type `net.datenwerke.rs.base.service.reportengines.table.output.object.RSTableModel` which is a simple wrapper for a table. RSTableModel basically consists of a definition of a table (defining attributes) and a list of rows. So a simple table could be defined as

```
import net.datenwerke.rs.base.service.reportengines.table.output.object.*

TableDefinition td = new TableDefinition(['colA', 'colB', 'colC'], [String.class,  ↙
    ↳  Integer.class, String.class])

RSTableModel table = new RSTableModel(td)
table.addDataRow('A', 10, 'C')
table.addDataRow('foo', 42, 'bar')

return table
```

If executed, this script would produce the following output.

```
reportserver$ exec ds1.groovy
Table(definition:TableDefinition [columnNames=[colA, colB, colC], columnTypes=[ ↙
    ↳  class java.lang.String, class java.lang.Integer, class java.lang.String]], ↙
    ↳   rows: 2)
```

A script, as in the above example, is all we need to specify a script datasource. All that would be left to do is to create a new datasource (in the datasource management tree of the administrator's module) of type script datasource and select the just created script. The only other configuration choice is to specify whether the data should be cached in the internal database (and if so, for how long) or whether the script should be executed whenever a request to the dataset is made. Henceforth, the datasource can be used similar to any other relational database.

In addition, you can pass arguments to the script, which can be referred in the script with the `args` variable. E.g., refer to the following example:

```
import net.datenwerke.rs.base.service.reportengines.table.output.object.*;

TableDefinition definition = new TableDefinition(['a','b','c'],
        [String.class,String.class,String.class]);

RSTableModel model = new RSTableModel(definition);
model.addDataRow(args[0],"2","3");
model.addDataRow("4","5","6");
model.addDataRow("7","8","9");

return model;
```

The `args[0]` prints the 0th argument passed to the script. You can either pass a text, e.g."myValue",

or the value of a given report parameter, e.g. `${myParam}` for a "myParam" parameter. Note that if the value contains blank spaces, quotation marks are needed.

In the following we will explain, step by step, how the above dataset can be transformed into a RSTableModel. The first step is to create the TableDefinition. The dataset consists of 11 attributes of various types:

```
TableDefinition td = new TableDefinition(['employee_name', 'office', 'city', ' ↙
    ↳ employee_title', 'biweekly_hourly_rate', 'payroll_type', 'pay_period', ' ↙
    ↳ pay_period_begin_date', 'pay_period_end_date', 'check_date', ' ↙
    ↳ legislative_entity'],[String.class, String.class, String.class, String. ↙
    ↳ class, Float.class, String.class, Integer.class, Date.class, Date.class,  ↙
    ↳ Date.class, String.class])
```

In a next step we need to read in the XML. This task can be easily achieved using Groovy's XmlSlurper.

```
import groovy.xml.slurpersupport.GPathResult
import groovy.xml.XmlSlurper
import java.net.URLConnection

String address = "https://opendata.socrata.com/api/views/2dps-ayzy/rows.xml? ↙
    ↳ accessType=DOWNLOAD"

URLConnection connection = address.toURL().openConnection()
GPathResult feed = new XmlSlurper().parseText(connection.content.text)

return feed.size()
```

Executing the above script might take a few seconds, but the you should see the following result

```
reportserver$ exec ds1.groovy
1
```

There is exactly one root node, so the size is one. Now, all left to do is to loop over the records and add them to the table.

```
feed.row.row.each { row ->
  table.addDataRow(
    row.employee_name.text(),
    row.office.text(),
    row.city.text(),
    row.employee_title.text(),
    Float.parseFloat(row.biweekly_hourly_rate.text()),
    row.payroll_type.text(),
    Integer.parseInt(row.pay_period.text()),
    formatter.parse(row.pay_period_begin_date.text()),
    formatter.parse(row.pay_period_end_date.text()),
    formatter.parse(row.check_date.text()),
    row.legislative_entity.text(),
  )
}
```

Putting it all together, we get the following simple script:

```
import net.datenwerke.rs.base.service.reportengines.table.output.object.*
import groovy.xml.slurpersupport.GPathResult
import groovy.xml.XmlSlurper
import java.net.URLConnection
import java.util.Date
import java.text.SimpleDateFormat

String address = "https://opendata.socrata.com/api/views/2dps-ayzy/rows.xml? ↙
    ↳ accessType=DOWNLOAD"

URLConnection connection = address.toURL().openConnection()
GPathResult feed = new XmlSlurper().parseText(connection.content.text)

TableDefinition td = new TableDefinition(['employee_name', 'office', 'city', ' ↙
    ↳ employee_title', 'biweekly_hourly_rate', 'payroll_type', 'pay_period', ' ↙
    ↳ pay_period_begin_date', 'pay_period_end_date', 'check_date', ' ↙
    ↳ legislative_entity'],[String.class, String.class, String.class, String. ↙
    ↳ class, Float.class, String.class, Integer.class, Date.class, Date.class,  ↙
    ↳ Date.class, String.class])
RSTableModel table = new RSTableModel(td)
SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss")

feed.row.row.each { row ->
  table.addDataRow(
    row.employee_name.text(),
    row.office.text(),
    row.city.text(),
    row.employee_title.text(),
    Float.parseFloat(row.biweekly_hourly_rate.text()),
    row.payroll_type.text(),
    Integer.parseInt(row.pay_period.text()),
    formatter.parse(row.pay_period_begin_date.text()),
    formatter.parse(row.pay_period_end_date.text()),
    formatter.parse(row.check_date.text()),
    row.legislative_entity.text(),
  )
}
return table
```

Further, useful script datasource examples can be found in our GitHub samples project: https://github.com/infofabrik/reportserver-samples.

6.1 Using Script Datasources with Pixel-Perfect Reports

For using script datasources (csv datasources analogously) together with pixel-perfect reports, you need additional minor configuration in the given reports. The configuration is based on the following.

The data of script datasources is buffered to internal temporary tables. The query type to be used is

```
SELECT * FROM _RS_TMP_TABLENAME
```

where _RS_TMP_TABLENAME is a temporary table name assigned by ReportServer. The following replacements are available:

_RS_TMP_TABLENAME:	The name of the table
_RS_QUERY:	The basic query.

These replacements can be used in pixel-perfect reports as described by the following sections.

Using Script Datasources with Jasper Reports

You can add one of the parameters above to your Jasper report and use it inside the query. E.g., you can add a text parameter _RS_TMP_TABLENAME to your report and use the following query:

```
select * from  $P!{_RS_TMP_TABLENAME}
```

You may have to create the fields you are going to use in your report manually. Also note that you need to use $!{} instead of ${} as replacements need to be directly written into the query.

Using Script Datasources with BIRT Reports

You can add one of the parameters above to your BIRT report. The parameter can not be used directly in the query of your data set, though. Instead, you need a "beforeOpen" script in your BIRT report, which performs the replacement needed. E.g.:

```
this.queryText = "SELECT * FROM " + params["_RS_TMP_TABLENAME"].value;
```

In order to create your dataset fields and use them later in your report, you may create a dummy query similar to

```
select '' as myfield1, '' as myfield2, '' as myfield3
```

Thus, the fields "myField1", "myField2" and "myField3" can be now used in your report.

Using Script Datasources with JXLS Reports

You can use one of the parameters above in your JXLS query tag directly. E.g. for JXLS2 reports:

```
jx:area(lastCell="B2")
jx:each(items="jdbc.query('SELECT EMP_FIRSTNAME as firstname, EMP_LASTNAME as
    ↳ lastname FROM $!{_RS_TMP_TABLENAME}')" var="employee" lastCell="B2")

First name: | ${employee.firstname}
Last name: | ${employee.lastname}
```

Chapter 7

Script Datasinks

Script datasinks are your swiss army knife when it comes to sending your reports/your data to custom locations or when you need any logic or any additional files. Basically, you can use script datasinks to send your reports/your data to virtually any location you may need.

When defining a script datasource, you have the following variables available to use:

data	contains the report or the data. Depending on the type of the data, this may be a String or a byte array. Note you can always use `ReportService.createInputStream(Object)` for creating an `InputStream` out of the data object.
report	the same as the "data" variable above. Recommended is to use `data`, `report` is included for consistency.
script	the `FileServerFile` containing the script of the datasink. You can use `script.data` for accessing the data in the script, or `script.contentType` for its content-type, or `script.name` for its name. Refer to the `FileServerFile` javadocs for a complete overview.
user	the `User` executing the script datasink.
job	Information about the scheduler job when a script datasink has been scheduled. This is an instance of `ReportExecuteJob`.
datasinkConfiguration	the datasink configuration object which contains the selected filename of the report/the data. This may be accessed with `datasinkConfiguration.filename`.

An example script datasink is shown below. It creates a ZIP containing the report/the data, adds the groovy script to the ZIP and sends this per email to a given user list.

The script is available here: https://github.com/infofabrik/reportserver-samples/blob/main/src/net/datenwerke/rs/samples/tools/datasinks/scriptDatasinkPerEmail.groovy.

```
import net.datenwerke.rs.core.service.mail.MailBuilderFactory
import net.datenwerke.rs.core.service.mail.MailService
import net.datenwerke.security.service.usermanager.UserManagerService
import net.datenwerke.rs.utils.misc.MimeUtils
import net.datenwerke.rs.core.service.mail.SimpleAttachment
```

```
import net.datenwerke.rs.core.service.mail.SimpleMail
import java.nio.file.Paths

import java.time.LocalDateTime

MailBuilderFactory mailBuilder = GLOBALS.getInstance(MailBuilderFactory)
MailService mailService = GLOBALS.getInstance(MailService)
UserManagerService userService = GLOBALS.getInstance(UserManagerService)
MimeUtils mimeUtils = GLOBALS.getInstance(MimeUtils)

// the user ids. They have to exist and the ids are passed as long (L)
List<long> to = [123L]
String subject = 'Script datasink'
String content = "ReportServer script datasink ${LocalDateTime.now()}"
// name of the zip
String attachmentFilename = 'data.zip'

List<SimpleAttachment> attachments = [
   new SimpleAttachment(data, // you can also use report
   mimeUtils.getMimeTypeByExtension(datasinkConfiguration.filename),
   datasinkConfiguration.filename),
   // add the script
   new SimpleAttachment(script.data, script.contentType, script.name)
]

SimpleMail mail = mailBuilder.create(
      subject,
      content,
      to.collect{userId -> userService.getNodeById(userId)})
      .withAttachments(attachments)
      .withZippedAttachments(attachmentFilename)
      .build()

mailService.sendMail mail
```

Chapter 8

Tapping into ReportServer

ReportServer makes heavy use of a subscriber-notifier pattern which we call hooking. That is, a central HookHandlerService plays the role of registry where so called Hooks can be registered. These registered code snippets are then notified, when certain code is executed (for example, when a report is about to be executed) or to provide functionality (for example, to handle the authentication process or export a report into a specified format). In this section we explain the basics of registering scripts as Hooks and discuss some of the more common Hook interfaces. Other examples of customization using Hooks are given in the appendix.

Besides extending ReportServer via hooks, ReportServer has an event mechanism that can be used to be notified upon certain events, such as changes to entities (for example, one can be notified whenever a report is changed) or on errors.

8.1 ReportServer Hooks Basics

A ReportServer hook is simply an interface that inherits the `net.datenwerke.hookhandler.shared.hookhandler.interfaces.hook` interface. Hooks are used throughout ReportServer to provide mechanisms to easily extend the ReportServer functionality. For example, hooks can be used to register additional database drivers, to add report output formats, to allow for customized authentication (for example, LDAP) and much more. To register a hook with ReportServer you will usually use the callbackRegistry which is available via the GLOBALS object. The callbackRegistry offers two methods to register a new hook:

`attachHook()` takes as input a class object, to specify the hook interface and an object from type hook (usually called hooker) which contains the actual code. The method returns a unique name for the hook which can be used to deregister it again.

`attachHook(name)` same as before, but one additionally specifies the name as first parameter. This makes it easy to remove or update the hooker by simply accessing it via the specified name. That is, if this method is called twice with the same name, then only one hook is registered.

To deregister hooks you can use the method `detachHook(name)` which takes the name of a previously

registered hook as input. To get a list (or rather a map) of all the registered hooks use the method `getRegisteredHooks` which returns a map (name -> hook).

The easiest way to learn how to work with ReportServer hooks is to actually implement one. In the following we will create a simple hook that is notified before every report execution and denies the execution in case the request is not within the normal working hours. For this we will use the ReportExecutionNotificationHook or more precisely the (net.datenwerke.rs.core.service.reportmanager.hooks.ReportExecutionNotificationHook). Following is the basic outline of a script that registers a hook:

```
String HOOK_NAME = "THE NAME OF MY HOOK"

TypeOfHook callback = [
  /* implementation of hook */
] as TypeOfHook

GLOBALS.services.callbackRegistry.attachHook(HOOK_NAME, TypeOfHook.class, callback ↙
    ↳ )
```

In our case, this could look as follows

```
import net.datenwerke.rs.core.service.reportmanager.exceptions.*
import net.datenwerke.rs.core.service.reportmanager.hooks.*

String HOOK_NAME = "PROHIBIT_EXECUTION"

ReportExecutionNotificationHook callback = [
    notifyOfReportExecution : { report, parameterSet, user, outputFormat, configs  ↙
     ↳  ->  },
    notifyOfReportsSuccessfulExecution : { compiledReport, report, parameterSet,  ↙
     ↳  user,
outputFormat, configs -> },
    notifyOfReportsUnsuccessfulExecution : { e, report, parameterSet, user,  ↙
     ↳  outputFormat,
configs -> },
  doVetoReportExecution: { report, parameterSet, user, outputFormat, configs ->
    def cal = Calendar.instance
    def hour = cal.get(Calendar.HOUR_OF_DAY)
    if(hour > 17 || hour < 9)
      throw new ReportExecutorException("Please come back during working hours.");
    }
] as ReportExecutionNotificationHook

GLOBALS.services.callbackRegistry.attachHook(HOOK_NAME,  ↙
    ↳  ReportExecutionNotificationHook.
class, callback)
```

We have implemented the interface ReportExecutionNotificationHook which has four methods. We did this using "closure coercion" as explained here: https://docs.groovy-lang.org/latest/html/documentation/core-semantics.html#closure-coercion. In our case we wanted to stop the execution of a report execution. For this, the interface specifies that we should throw a ReportExecutorException.

All that is left is to execute the above script.

```
reportserver$ exec denyexecutionhook.groovy
PROHIBIT_EXECUTION
```

Tip: When implementing hooks you need to take care to properly implement the interface. Bugs that lead to an exception within your implementation could otherwise easily lead to unexpected behavior.

Getting a List of all Registered Hooks

It may be useful to list all the registered hooks, or all the registered hooks of a specific type. For this we can use the method "getRegisteredHooks" provided by the callbackRegistry. The following simple script simply outputs the name of all registered hooks together with the date, when the hook has been registered.

```
GLOBALS.services.callbackRegistry.getRegisteredHooks().each { key, value ->
  tout.println(key + ": " + value.getDate())
}

""
```

8.2 Registering Hooks on Start-up and on Login

On start-up and when a user logs in ReportServer executes a specially named script or all scripts in a named folder, respectively. This is the place to register your hooks in case you permanently want to adapt certain ReportServer functionality. The location of these scripts is configured in the config file `scripting/scripting.cf` (for further information refer to ReportServer Configuration Guide). The config could, for example, look like

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
   <scripting>
      <enable>true</enable>
      <restrict>
         <location>bin</location>
      </restrict>
      <startup>
         <login>fileserver/bin/onlogin.d</login>
         <rs>fileserver/bin/onstartup.d</rs>
      </startup>
   </scripting>
</configuration>
```

Assuming that `onstartup.d` is a directory, ReportServer would execute all scripts within this directory on start-up. Note that ReportServer will not recursively traverse directories.

If something goes wrong during a start-up script ReportServer will record this event in its audit log. To access the audit log simply create a dynamic list pointing to the database that hosts ReportServer. The table is `RS_AUDIT_LOG`. The action is named `STARTUP_SCRIPT_FAILED`. More information on the audit log can be found in the administrator's manual.

Should you find yourself locked out of ReportServer, you can disable any scripts via the rs.scripting.disable property in the reportserver.properties configuration file. If present and set to true, scripts will not be executed and you can thus login correctly to ReportServer.

8.3 Which Hooks can I use?

The interface `net.datenwerke.hookhandler.shared.hookhandler.interfaces.Hook` is implemented by all ReportServer Hooks. Thus, a good starting point is the Javadoc documentation of the source. In principle any such hook can be implemented. Here you can find a list of all hooks of the latest ReportServer version: `https://reportserver.net/api/latest/hooks.html`. For hooks that were especially designed to be implemented by scripts we have added an adapter class that provides a dummy implementation for all methods required by the hook. You will find the corresponding adapter in the subpackage adapter. Other than the source directly the examples in the appendix of this manual should provide a good starting point.

Chapter 9

Extending the Client

In this chapter we consider various ways to extend the client side of ReportServer. Some of these extensions will be configured via configuration files and which allow, for example, to add further information tabs to the TeamView or add further links to the module bar. Other extensions need scripting. These include adding additional report exporters, displaying messages on login, or adding custom information to the status bar.

9.1 UrlView - Incorporating Websites and More

Via the configuration file /etc/ui/urlview.cf you can add links and tabs to various locations within ReportServer. For example, you can add a new link to the module bar and specify to which URL it should point. Additionally you can restrict what you want to add to certain users or groups. A typical configuration file could like

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
   <adminviews>
      <view>
         <!-- View Configuration -->
      </view>
   </adminviews>
   <objectinfo>
      <view>
         <!-- View Configuration -->
      </view>
      <view>
         <!-- View Configuration -->
      </view>
   </objectinfo>
   <module>
      <view>
         <!-- View Configuration -->
      </view>
   </module>
</configuration>
```

The three scopes (adminviews, objectinfo and module) allow you to add additional information tabs to objects in the administration module (for example, add an additional tab to a user object displaying information on that particular user), additional tabs to objects in the teamspace and within module you can add additional links to the top module bar. The documentation report within teamspaces is by default, for example, integrated via the an objectinfo view configuration.

View configurations all share the following configuration

```
<view>
   <restrictTo><!-- can be used to restrict this to specific users/groups/ous--></ ↙
     ↳ restrictTo>
   <name>The display name</name>
   <url>http://someUrl/</url>
</view>
```

Via `restrictTo` you can specify users by either username or groups or OUs (organizational units, i.e., folders in the user tree) via IDs to include this view only for users that match one of the restrictions. The name is the name to be displayed on the tab or in the module bar. The URL denotes the URL to be loaded.

Restrictions are specified in the following form

```
<restrictTo>
   <users>username1,username2</users>
   <groups>123,234</groups>
   <ous>876</ous>
</restrictTo>
```

For adminviews and objectinfo views you can further restrict the view to specific object types. This is done by adding a `types` tag which takes a comma-separated list of fully qualified names of objects for which this view applies. For example the following restriction would restrict the (objectinfo) view to report objects within the teamspace.

```
<types>net.datenwerke.rs.tsreportarea.client.tsreportarea.dto. ↙
     ↳ TsDiskReportReferenceDto</types>
```

The following types are available for **objectinfo** views

`net.datenwerke.rs.tsreportarea.client.tsreportarea.dto.AbstractTsDiskNodeDto`
: All Objects in a Teamspace.

`net.datenwerke.rs.tsreportarea.client.tsreportarea.dto.TsDiskFolderDto`
: All folders in a Teamspace.

`net.datenwerke.rs.tsreportarea.client.tsreportarea.dto.TsDiskReportReferenceDto`
: All reports and variants in a Teamspace.

`net.datenwerke.rs.scheduleasfile.client.scheduleasfile.dto.ExecutedReportFileReferenceDto`
: All "exported reports" which were, for example, created by the scheduler.

The following types are available for **adminviews**

Objects in Report Management.
Prefix: net.datenwerke.rs.core.client.reportmanager.dto.reports.

Type	Description
AbstractReportManagerNodeDto	All objects in the report management tree.
ReportDto	reports
ReportFolderDto	folders

Objects in User Management.
Prefix net.datenwerke.security.client.usermanager.dto.

Type	Description
AbstractUserManagerNodeDto	All objects in the user management tree.
UserDto	users
GroupDto	groups
OrganisationalUnitDto	organisational units (folders)

Objects in the file server.
Prefix net.datenwerke.rs.fileserver.client.fileserver.dto.

Type	Description
AbstractFileServerNodeDto	All objects in the file server.
FileServerFolderDto	folders
FileServerFileDto	files

Objects in Datasource Management
Prefix net.datenwerke.rs.core.client.datasourcemanager.dto.

Type	Description
AbstractDatasourceManagerNodeDto	All objects in datasource management.
DatasourceFolderDto	folders
DatasourceDefinitionDto	datasources

Objects in Dadget Management
Prefix net.datenwerke.rs.dashboard.client.dashboard.dto.

Type	Description
AbstractDashboardManagerNodeDto	All objects in the dashboard tree
DashboardNodeDto	dashboards
DashboardFolderDto	folders

You can use replacements within the URL to generate dynamic URLs. The following replacements are available

`${username}` Is replaced by the current user's username.

`${type}` Is replaced by the class name of the corresponding object. This replacement is only available for objectinfo and adminviews.

`${id}` Is replaced by the ID of the corresponding object. This replacement is only available for objectinfo and adminviews.

`${reportId}` Report objects within a team space are always references. Thus, the replacement `${id}` maps to the id of the reference rather than to the id of the actual report. In order to use the actual report's id you can use the replacement `${reportId}`

ReportServer Specific URLs

Besides specifying a view via a URL ReportServer you can tell ReportServer to display the preview view of a report. For this, you need to define the URL as

```
rs:reportpreview://ID
```

where ID should be replaced by the id of the corresponding report or, for example, by the replacement `${reportId}` when defining this view for report objects in the TeamSpace.

Note that, as with any configuration file you need to run `config reload` on the terminal for changes to take effect. Furthermore you need to reload your browser to see effects as the configuration is loaded onto the client on login.

9.2 CommandResult - A Script's Result

We will now get back to ReportServer scripts and have a closer look at the object returned by a script. In theory, you can return any object from a script. For example the script

```
"Hello World"
```

returns an object of type String. In order to transport the result onto the client the terminal process wraps such a result in an object of type CommandResult (located in package `net.datenwerke.rs.terminal.service.terminal.obj`) which the client knows how to handle. If executed in the terminal the terminal will extract the string and display

```
reportserver$ exec hello.groovy
Hello World
```

This is done dynamically behind the scenes. But, you can also make this explicit and directly return a CommandResult. For this, you need to change the script to

```
import net.datenwerke.rs.terminal.service.terminal.obj.CommandResult
new CommandResult("Hello World")
```

When executing this script the result is identical to before. However, making the CommandResult explicit will give us more control over how the client treats the result. In Chapter 3 we have already seen how to add hyperlinks and anchors to CommandResults. You can furthermore add simple lists, tables and strings to structure the output on the terminal. Consider the following script

```
import net.datenwerke.rs.terminal.service.terminal.obj.CommandResult

CommandResult result = new CommandResult()
result.addResultList(['Hello','World'])
result.addResultLine('----------')
result.addResultList(['Hello','World'])

return result
```

Executing this script produces the following output

```
reportserver$ exec hello.groovy
Hello
World
----------
Hello
World
```

Besides lists and tables, you can also directly return HTML.

```
import net.datenwerke.rs.terminal.service.terminal.obj.CommandResult

CommandResult result = new CommandResult()
result.addResultHtml('<b>Hello</b> World')

return result
```

These techniques, however, are limited to format the results of a script when displayed on a terminal. The CommandResult object, on the other hand, is not limited to terminal formatting. A CommandResult object can contain one or more objects of type CommandResultExtensions which trigger certain responses on the client. For example, this allows you to display popup messages or execute arbitrary JavaScript code.

Displaying Messages

In order to display popup messages your script needs to add a `CommandResultExtension` of type `CreMessage` to the CommandResult object. Following is a simple example.

```
import net.datenwerke.rs.terminal.service.terminal.obj.*

CommandResult result = new CommandResult()

CreMessage msg = new CreMessage("Hello World")
result.addExtension(msg)

return result
```

You can further control the size of the window and also directly add HTML.

```
import net.datenwerke.rs.terminal.service.terminal.obj.*

CommandResult result = new CommandResult()

CreMessage msg = new CreMessage()
msg.setTitle("Hello")
msg.setWindowTitle("Hello")
msg.setWidth(400)
msg.setHeight(300)
msg.setHtml("<b>Hello World</b>")
result.addExtension(msg)

return result
```

Executing Custom JavaScript

The `CreJavaScript` CommandResultObject allows to specify custom JavaScript which is executed on the client.

```
import net.datenwerke.rs.terminal.service.terminal.obj.*

CommandResult result = new CommandResult()

String script = """
alert("Hello World");
"""

result.addExtension(new CreJavaScript(script))

return result
```

9.3 ClientExtensionService

So far we have seen how to customize the output within the terminal, how to display messages in popups or execute custom JavaScript. In this section we will look at the so-called ClientExtensionService that can be accessed via the GLOBALS object. The **ClientExtensionService** offers methods to insert entries to various toolbars and context menus. Each new entry comes with a callback, that is, a ReportServer script that is executed when the specified entry is activated. Depending on where you added the entry your script is provided with additional information. For example, when adding an entry to the context menu in the report management view in the admin module, your script will get the id of the corresponding report object as an argument.

One thing to keep in mind when using the ClientExtensionService is that all effects are only valid as long as the current page is not refreshed. In order to install the changes permanently you should thus add the script into the `onlogin.d` folder.

The ClientExtensionService currently offers the following methods.

`addStatusBarLabel()`	Allows to display info texts in the status bar.
`addMenuEntry()`	Allows to add entries to certain context menus.
`addToolbarEntry()`	Allows to add buttons to certain toolbars.
`clientRedirect()`	Allows to redirect the browser to a specified URI.
`addReportExportOutputFormat()`	Allows to create custom exporters. We discuss this in detail in Chapter 12.

Adding Information to the Status Bar

The first method (addStatusBarLabel) allows you to add simple info texts to the status bar. The concept is best explained by a small example.

```
def service = GLOBALS.services['clientExtensionService']

service.addStatusBarLabel("All is fine")
```

As you might have guessed, the first line obtains the ClientExtensionService from the GLOBALs object. All that is left to do is to set a status update. Besides just adding text you can also call

```
addStatusBarLabel("All is fine", "path to some icon", true)
```

where the second parameter should point to an icon and the third parameter controls whether the object is added to the left or to the right (default).

You can of course access all server objects and methods within this script. E.g., the following executes a given dynamic list and prints the report name and its number of records into the status bar.

```
import net.datenwerke.rs.base.service.reportengines.table.entities.*
import net.datenwerke.rs.base.service.reportengines.table.*
import net.datenwerke.rs.core.service.reportmanager.*
import net.datenwerke.rs.base.service.reportengines.table.entities. ↙
    ↳ TableReportVariant
import net.datenwerke.rs.core.service.reportmanager.ReportService
import net.datenwerke.rs.core.service.reportmanager.ReportExecutorService
import net.datenwerke.rs.base.service.reportengines.table.output.object. ↙
    ↳ RSTableModel
import net.datenwerke.rs.core.service.reportmanager.engine.config. ↙
    ↳ ReportExecutionConfig

ReportService reportService = GLOBALS.getRsService(ReportService.class)
ReportExecutorService reportExec = GLOBALS.getRsService(ReportExecutorService. ↙
    ↳ class)

TableReportVariant report = reportService.getReportById(2078898)

String reportName = report.getName()

RSTableModel reportCompiled = (RSTableModel) reportExec.execute(report, "RS_TABLE" ↙
    ↳ , ReportExecutionConfig.EMPTY_CONFIG)
```

```
int rowCount = reportCompiled.getRowCount()

/* prepare output */
def ces = GLOBALS.services.clientExtensionService
ces.addStatusBarLabel("Info: ${reportName} - ${rowCount} rows")
```

Adding Context Menu Entries

The ClientExtensionService allows you to add menu entries to the following context menus

Module	Description	Menu Name
Datasource Manager	The datasource manager within the admin module	`datasource:admin:tree:menu`
Report Manager	The report manager within the admin module	`reportmanager:admin:tree:menu`
File Server	The file server within the admin module	`fileserver:admin:tree:menu`
Dashboard Library	The dashboard library within the admin module	`dashboard:admin:tree:menu`
User Manager	The user manager within the admin module	`usermanager:admin:tree:menu`

To add an entry to a specific menu you need to access the menu by name (we give the menu names in the above table, for example, `usermanager:admin:tree:menu` corresponds to the context menu within the user manager tree).

The simplest way to add a menu entry is to simply call the method `addMenuEntry()` from the ClientExtensionService:

```
public void addMenuEntry(String menuName, String entryName, String scriptLocation, ↙
    ↳  String configArgument)
```

The method takes the menu name, a name for the entry, a location of the script that is to be called when the entry is activated, and an argument given to the script. In the following we want to add a menu to the user tree. We call our script that generates the entry `adddisplayinfoentry.groovy`.

```
def service = GLOBALS.services['clientExtensionService']

service.addMenuEntry("usermanager:admin:tree:menu", "display info", "fileserver/ ↙
    ↳  bin/extensions/menu/displayuserinfo.groovy", "an argument")
```

> The script must be executed before the user tree is loaded for the first time.

If you execute the above script and navigate to the user manager in the administration module and right click on any item in the tree you will see the new entry.

```
display info
```

If you activate the entry you will see an error message, since we haven't yet created a script at location

`fileserver/bin/extensions/menu/displayuserinfo.groovy`

We are now going to create this file and use the above and display a simple message outputting the object's id.

```
import net.datenwerke.rs.terminal.service.terminal.obj.*

CommandResult result = new CommandResult()

CreMessage msg = new CreMessage("ID: " + context['id'] + ", args" + args)
result.addExtension(msg)

return result
```

The important part here is the **context** object which contains a field **id** corresponding to the id of the object clicked upon. Besides the **id**, the context also contains

id	The object's id.
classname	The object's classname.
path	The object's path.

Display Conditions

Sometimes it might be inconvenient to add the menu entry to every object in the tree. That is, maybe we want to add an entry only to user objects, but not to groups or organizational units. To have full control over how the menu item is created you can directly create an object of type `AddMenuEntryExtension` located in package `net.datenwerke.rs.scripting.service.scripting.extensions`. To better control when the entry is displayed, use `DisplayConditions` (located in the same package).

```
import net.datenwerke.rs.scripting.service.scripting.extensions.*

def service = GLOBALS.services['clientExtensionService']

AddMenuEntryExtension entry = new AddMenuEntryExtension()
entry.setMenuName("usermanager:admin:tree:menu")
entry.setLabel("display info")
entry.setScriptLocation("fileserver/bin/extensions/menu/displayuserinfo.groovy")

DisplayCondition cond = new DisplayCondition("classname", "net.datenwerke.security ↙
    ↳ .client.usermanager.dto.UserDto")
entry.addDisplayCondition(cond)

service.addMenuEntry(entry)
```

Note that the for menus you can currently only compare classnames. Also note that always the base class of the DTO is used, that is, instead of `UserDtoDec` you need to use `UserDto`.

Adding Toolbar Entries

Besides adding entries to context menus, you can add buttons to various toolbars, including all toolbars within the various admin modules. The mechanism is similar to the mechanism we described above. The following toolbars support the addition of custom buttons:

Module	Description	Menu Name
Datasource Manager	The datasource manager within the admin module	`datasource:admin:view:toolbar`
Report Manager	The report manager within the admin module	`reportmanager:admin:view:toolbar`
File Server	The file server within the admin module	`fileserver:admin:view:toolbar`
Dashboard Library	The dashboard library within the admin module	`dashboard:admin:view:toolbar`
User Manager	The user manager within the admin module	`usermanager:admin:view:toolbar`
Report Executor	The report executor module. This includes the report execution by URL.	`reportexecutor:main:toolbar`
Scheduler Job List	The scheduler job list module.	`schedulerlist:main:toolbar`

Similar as above you can either use helper methods within the ClientExtensionService such as

```
public void addToolbarEntry(String toolbarName, String entryName, String entryIcon ↙
    ↳  , String scriptLocation, String arguments){
```

or alternatively create an object of type `AddToolbarEntryExtension`, also located in the package `net.datenwerke.rs.scripting.service.scripting.extensions`.

```
import net.datenwerke.rs.scripting.service.scripting.extensions.*

def service = GLOBALS.services['clientExtensionService']

AddToolbarEntryExtension entry = new AddToolbarEntryExtension()
entry.setToolbarName("usermanager:admin:view:toolbar")
entry.setLabel("display info")
entry.setIcon("path/to/icon")
entry.setScriptLocation("fileserver/bin/extensions/menu/displayuserinfo.groovy")

DisplayCondition cond = new DisplayCondition("classname", "net.datenwerke.security ↙
    ↳  .client.usermanager.dto.decorator.UserDtoDec")
entry.addDisplayCondition(cond)

service.addToolbarEntry(entry)
```

Note that with toolbar buttons you have more control over when buttons are to be displayed, as you can specify the exact type, that is, TableReportVariantDto instead of ReportVariantDto. This, for example, allows you to add a custom button only to dynamic lists, but not to jasper reports.

Redirecting the Browser to a specified URI

To redirect the browser to a specific URI, you can use the `clientExtensionService` as demonstrated in the example below:

```
def service = GLOBALS.services['clientExtensionService']
service.clientRedirect('http://my-domain:8080/ReportServer/reportserver/ ↙
    ↳ fileServerAccess?id=367350')
```

You may use any valid URI as the redirection target.

Future Additions

The possibilities offered by the ClientExtensionService are still rudimentary. We plan to extend these in future versions and are keen to hear your thoughts on what you would like to be able to do and what you are currently missing.

Chapter 10

Custom Authenticators PAMs

ReportServer comes with a flexible authentication mechanism that allows to authenticate users using almost any conceivable authentication method. In this introductory tutorial we look at how ReportServer performs user authentication and discuss how you can plug in custom authentication schemes.

10.1 Pluggable Authentication Modules

When no user is logged in, ReportServer waits for someone to ask it to start the authentication process. Authentication is handled by ReportServer's `AuthenticatorService` which is triggered, for example, when a user fills in a username and password and hits the "Login" button. This is, however, not the only way an authentication request can be triggered. For instance, on each request ReportServer attempts an authentication without additional information, for example, to implement a single-sign-on operation without the user having to ever visit the login page. Alternatively, authentication could be triggered by a custom script thereby allowing you to implement a completely separate login page containing, for example, a two-factor authentication mechanism.

> Note that ReportServer supports LDAP authentication out-of-the-box via the Ldap PAM. Details can be found in the Administration Guide.

In any case, once AuthenticatorService's authentication mechanism is triggered the following happens. The service looks for a list of installed ***PAM***s (short for ***Pluggable Authentication Modules***). Each registered PAM is asked whether the metadata provided for the authentication (e.g., username and password) is sufficient. For this, each PAM is required to respond with one of three possible responses:

1. Login failed
2. Login was successful, the corresponding user is SOME_USER
3. I can't determine anything, leave me out of the decision process.

The decision of the AuthenticatorService is then based on the combined responses of all registered PAMs. In case any PAM opted for option one (Login failed) the overall authentication will fail. The same is true, if no one opted for option two Furthermore, authentication fails, if two modules opt for a successful login but disagree on the user.

10.2 Default PAMs

What PAMs can you choose from? The default PAM settings are made in ReportServer's external configuration file `reportserver.properties`. Following is the default authenticator configuration:

```
### authenticator configuration ################################################

  # rs.authenticator.pams
  # configures the pluggable modules the authenticator uses to verify requests
  # multiple modules are separated by colon ":" characters
  # possible values are:
  #   net.datenwerke.rs.authenticator.service.pam.UserPasswordPAM
  #   net.datenwerke.rs.authenticator.service.pam.UserPasswordPAMAuthoritative
  #   net.datenwerke.rs.authenticator.service.pam.IPRestrictionPAM
  #   net.datenwerke.rs.authenticator.service.pam.EveryoneIsRootPAM
  #   net.datenwerke.rs.authenticator.cr.service.pam.ChallengeResponsePAM
  #   net.datenwerke.rs.authenticator.cr.service.pam. ↙
     ↳ ChallengeResponsePAMAuthoritative
  #   net.datenwerke.rs.authenticator.service.pam.ClientCertificateMatchEmailPAM
  #   net.datenwerke.rs.authenticator.service.pam. ↙
     ↳ ClientCertificateMatchEmailPAMAuthoritative
  #   net.datenwerke.rs.ldap.service.ldap.pam.LdapPAM
  #   net.datenwerke.rs.ldap.service.ldap.pam.LdapPAMAuthoritative
rs.authenticator.pams = net.datenwerke.rs.authenticator.service.pam. ↙
     ↳ UserPasswordPAMAuthoritative
```

The property `rs.authenticator.pams` consists of a colon (`:`) separated list of one or more PAMs. In a standard installation only a single PAM is active, namely `net.datenwerke.rs.authenticator.service.pam.UserPasswordPAMAuthoritative`. This PAM expects two tokens, a username and a password and then attempts to match these against ReportServer's database. As you can see, the default PAMs usually come in two variants, one being called ***Authoritative***. The difference between the two variants is how they handle the case where they cannot find the necessary information within the provided list of tokens. The authoritative version then denies access, while the non-authoritative version opts for option 3 (can't tell, let somebody else decide).

Further information on the available default PAMs is given in the Configuration Guide: `https://reportserver.net/en/guides/config/chapters/configfile-reportserverproperties/`

10.3 Adding Custom PAMs

While PAMs can be configured via the external configuration file, the configuration can be extended (or completely overwritten) via scripts. This allows us to bring custom authentication mechanisms into the system.

To write a custom authenticator we need to do two things:

1. Implement the net.datenwerke.security.service.authenticator.ReportServerPAM interface, and
2. hook into the authentication mechanism.

Let us look at these in turn.

The ReportServerPAM interface looks as follows

```
public interface ReportServerPAM {

   public AuthenticationResult authenticate(AuthToken[] tokens);

   public String getClientModuleName();

}
```

That is, we need to implement two methods: `authenticate` and `getClientModuleName`. The second one is the easier one, since there are currently not many options available here. This method tells ReportServer which module on the client side should handle the authentication. Here basically, the question is, do you want to use ReportServer's standard login page. If this is the case then you should return the String value "`net.datenwerke.rs.authenticator.client.login.pam.UserPasswordClientPAM`". Alternatively, if you want to use a custom login page, you can simply return `null`. Thus, a custom PAM usually takes the following form (now in Groovy).

```
import net.datenwerke.security.service.authenticator.ReportServerPAM

ReportServerPAM customPAM = [
  authenticate : { tokens -> // TODO
  },
  getClientModuleName : { return 'net.datenwerke.rs.authenticator.client.login.pam ⤶
     ↳  .UserPasswordClientPAM' }
] as ReportServerPAM
```

In case we use ReportServer's default login page, the token array given to the authenticate method will consist of a single token of type `net.datenwerke.rs.authenticator.client.login.dto.UserPasswordAuthToken` which is a simple Java bean providing the methods `getUsername()` and `getPassword()`. (You should, however, always perform proper type checking and not expect the token array to be of a specific form.) In order to choose one of the three options (deny login, allow login, don't care) the authenticate method needs to return an object of type `net.datenwerke.security.service.authenticator.AuthenticationResult`. This `AuthenticationResult` is configured via its constructor which takes two values:

```
public AuthenticationResult(boolean allowed, User user);
```

The first value defines whether or not the PAM wants to deny access. If `allowed` is set to `false`, the user will not be able to login even if all other PAMs would be ok with that. The second value given to the `AuthenticationResult` is a user object indicating the user that should be logged in. Thus, we can choose between the three options by returning `AuthenticationResults` such as the following:

```
return new AuthenticationResult(false, null) // deny access
return new AuthenticationResult(false, someUser) // deny access to someUser
return new AuthenticationResult(true, someUser) // grant access, someUser should ↙
    ↳ be logged in
return new AuthenticationResult(true, null) // don't care, somebody else should ↙
    ↳ decide
```

We added static helper methods for making this easier:

```
return AuthenticationResult.denyAccess() // deny access
return AuthenticationResult.denyAccess(someUser) // deny access to someUser
return AuthenticationResult.grantAccess(user) // grant access, someUser should be ↙
    ↳ logged in
return AuthenticationResult.dontCareAccess() // don't care, somebody else should ↙
    ↳ decide
```

Basically, this is all you need to know to write a custom authenticator. Following is an example of a fully functional (yet not really useful) authenticator. It checks whether the provided password equals "42". If so, it logs in the first super user it can find. Otherwise, it chooses option 3 (don't care).

```
import net.datenwerke.security.service.authenticator.ReportServerPAM
import net.datenwerke.rs.authenticator.client.login.dto.UserPasswordAuthToken
import net.datenwerke.security.service.authenticator.AuthenticationResult

import net.datenwerke.security.service.usermanager.UserManagerService

UserManagerService userService = GLOBALS.getInstance(UserManagerService)

ReportServerPAM customPAM = [
  authenticate : { tokens ->
   if(tokens.length == 0 || ! tokens[0] instanceof UserPasswordAuthToken)
      return AuthenticationResult.dontCareAccess() // don't care, let somebody ↙
    ↳ else decide
   if('42'.equals(tokens[0].password)){
     for(def user : userService.allUsers)
       if(user.isSuperUser())
         return AuthenticationResult.grantAccess(user) // login the super user
   }
   return AuthenticationResult.dontCareAccess() // don't care, let somebody else ↙
    ↳ decide
  },
  getClientModuleName : { return 'net.datenwerke.rs.authenticator.client.login.pam ↙
    ↳ .UserPasswordClientPAM' }
] as ReportServerPAM
```

Hooking in our custom PAM

Now that we have a custom PAM, how can we add this PAM to the list of registered PAMs? The answer is to use ReportServer's Hook infrastructure (see Chapter 8 Tapping into ReportServer for a detailed introduction). The hook we are going to implement is `net.datenwerke.security.service.authenticator.hooks.PAMHook` which is defined as follows:

```
public interface PAMHook extends Hook {
```

```
    public void beforeStaticPamConfig(LinkedHashSet pams);

    public void afterStaticPamConfig(LinkedHashSet pams);

}
```

The two methods allow us to adapt the list of registered PAMs, once before the static configuration (the loading of PAMs specified in the external `reportserver.properties` configuration file) has been done, and once after. As usual, when implementing a hook, we should instead implement the corresponding adapter (if available). Following is the combined PAM with the necessary code to hook it in. Note that we have opted to clear the registered PAMs in the `afterStaticPamConfig` method. This is because ReportServer's standard UserPasswordPAMs don't always play nice. In particular, when they find a username/password token and the username matches a given user but the password is incorrect they opt to deny authentication. As in our case the password will be "incorrect" we thus need to remove them from the list of registered PAMs.

```
import net.datenwerke.security.service.authenticator.ReportServerPAM
import net.datenwerke.rs.authenticator.client.login.dto.UserPasswordAuthToken
import net.datenwerke.security.service.authenticator.AuthenticationResult

import net.datenwerke.security.service.authenticator.hooks.PAMHook
import net.datenwerke.security.service.authenticator.hooks.adapter.PAMHookAdapter

import net.datenwerke.security.service.usermanager.UserManagerService

UserManagerService userService = GLOBALS.getInstance(UserManagerService)

ReportServerPAM customPAM = [
  authenticate : { tokens ->
   if(tokens.length == 0 || ! tokens[0] instanceof UserPasswordAuthToken)
      return AuthenticationResult.dontCareAccess() // don't care, let somebody ↙
     ↳ else decide
   if('42'.equals(tokens[0].password)){
     for(def user : userService.allUsers)
       if(user.isSuperUser())
         return AuthenticationResult.grantAccess(user) // login the super user
   }
   return AuthenticationResult.dontCareAccess() // don't care, let somebody else ↙
     ↳ decide
  },
  getClientModuleName : { return 'net.datenwerke.rs.authenticator.client.login.pam ↙
     ↳ .UserPasswordClientPAM' }
] as ReportServerPAM;

PAMHookAdapter callback = [
  afterStaticPamConfig : {pams ->
    pams.clear()
    pams.add(customPAM)
  }
] as PAMHookAdapter
```

```
GLOBALS.services.callbackRegistry.attachHook('MY_CUSTOM_AUTHENTICATOR', PAMHook,  ↙
    ↳  callback)
```

Once you've executed the script, you can no longer log in with the standard username/password combination. However, once you provide "42" as the password, you will be logged in with a super-user account.

10.4 Installing Custom Authenticators on Startup

If you've completed development of your authenticator you could place it into the `onstartup.d` folder such that whenever ReportServer is booted up your authenticator becomes active immediately. This should, however, only be done after a thorough test, as otherwise you might find yourselve locked out of ReportServer.

> Should you find yourself locked out of ReportServer, you can disable any scripts via the `rs ↙ ↳ .scripting.disable` property in the `reportserver.properties` configuration file. If present and set to true, scripts will not be executed and thus you can fallback on one of the standard PAMs.

10.5 Ignore Case for Usernames

As a final treat, here is another example. Recently the question was raised on our Community Forums https://forum.reportserver.net/ whether for the purpose of authentication usernames are case sensitive, and if so, if this could be changed. Indeed, by default, usernames in ReportServer are case sensitive. However, given the information covered in this tutorial it should not be too difficult to write a custom PAM that ignores the case of a provided username. The only missing piece is the information on how to validate a password against ReportServer's stored user passwords. For this we can use the `net.datenwerke.rs.utils.crypto.PasswordHasher` object that provides the convenience method

```
public boolean validatePassword(String hashedPassword, String cleartextPassword);
```

Following is the complete example. Note that we have opted for a slight optimization to find the user. That is, instead of looping over all users we use a single query to find the correct user.

```
import net.datenwerke.security.service.authenticator.ReportServerPAM
import net.datenwerke.rs.authenticator.client.login.dto.UserPasswordAuthToken
import net.datenwerke.security.service.authenticator.AuthenticationResult

import net.datenwerke.security.service.authenticator.hooks.PAMHook
import net.datenwerke.security.service.authenticator.hooks.adapter.PAMHookAdapter

import net.datenwerke.rs.utils.crypto.PasswordHasher

PasswordHasher passwordHasher = GLOBALS.getInstance(PasswordHasher)

ReportServerPAM customPAM = [
  authenticate : { tokens ->
```

```
   if(tokens.length == 0 || ! tokens[0] instanceof UserPasswordAuthToken)
      return AuthenticationResult.denyAccess() // don't play nice. Deny  ↙
       ↳  authentication

    try{
      def user = GLOBALS.getEntityManager()
          .createQuery('FROM User WHERE lower(username) = :name')
                 .setParameter('name', tokens[0].username.toLowerCase())
          .singleResult
      if(null != user){
        if(passwordHasher.validatePassword(user.password, tokens[0].password)){
          return AuthenticationResult.grantAccess(user) // let user pass
        }
      }

    } catch(all){
      // potential logging
    }

   return AuthenticationResult.denyAccess() // don't play nice. Deny  ↙
     ↳  authentication
  },
  getClientModuleName : { return 'net.datenwerke.rs.authenticator.client.login.pam ↙
     ↳  .UserPasswordClientPAM' }
] as ReportServerPAM

PAMHookAdapter callback = [
  afterStaticPamConfig : {pams ->
    pams.clear()
    pams.add(customPAM)
  }
] as PAMHookAdapter

GLOBALS.services.callbackRegistry.attachHook('MY_CUSTOM_AUTHENTICATOR', PAMHook,  ↙
     ↳  callback)
```

A final word of warning. ReportServer does not enforce usernames to be unique when ignoring case sensitivity. Thus, if two users are given, for example, the usernames JohnDoe and johndoe then the authentication will fail for them (the `getSingleResult()` method will throw a `NonUniqueResultException`).

Part II

Examples

Chapter 11

Adding Additional Datasources

By default ReportServer is shipped with support only for the most common database systems, although it can interact with virtually every database that offers a JDBC compliant driver. In this chapter we will show how to develop a groovy script that adds support for the Firebird database (https://www.firebirdsql.org) to ReportServer.

Although SQL is well standardized and the JDBC specification defines a vendor independent interface to the actual database driver, simply adding the database driver to the classpath does not suffice. ReportServer tries to use mostly ANSI-SQL whenever possible, but in some situations not using a vendor-specific extension bears a high performance penalty. One of these situations is the limit/offset-mechanism, that allows the application to request only a certain chunk of the resultset. Most databases offer some solution to this problem, but there is no standard approach. To manage these differences in different SQL dialects ReportServer uses a DatabaseHelper class for each supported SQL dialect.

To add support for a new datatabase, what we need to do, is to implement a **DatabaseHelper** for the database and register it with ReportServer.

> Please note that Firebird is already integrated in ReportServer out-of-the-box. In order to use Firebird with ReportServer, you just have to add the Firebird drivers to your lib directory. The example below is meant for understanding the principle.

Lets start by looking at the DatabaseHelper for the H2 database engine:

```
public class H2 extends DatabaseHelper {

 public static final String DB_NAME = "H2";
 public static final String DB_DRIVER = "org.h2.Driver";
 public static final String DB_DESCRIPTOR = "DBHelper_H2";

 @Override
 public String getDescriptor() {
  return DB_DESCRIPTOR;
 }
```

```
 @Override
 public String getDriver() {
  return DB_DRIVER;
 }

 @Override
 public String getName() {
  return DB_NAME;
 }
}
```

The three methods getDescriptor, getName and getDriver are the minimum each DatabaseHelper has to implement.

`getName()`	The text the user is shown on the front-end when selecting the database type.
`getDriver()`	The name of the JDBC driver class.
`getDescriptor()`	The key used internally to map datasources to DatabaseHelpers

To adapt this to the Firebird database, we simply change the values of the three constants

```
class Firebird extends DatabaseHelper {

 public static final String DB_NAME = "Firebird";
 public static final String DB_DRIVER = "org.firebirdsql.jdbc.FBDriver";
 public static final String DB_DESCRIPTOR = "DBHelper_Firebird";

 @Override
 public String getDescriptor() {
  return DB_DESCRIPTOR;
 }

 @Override
 public String getDriver() {
  return DB_DRIVER;
 }

 @Override
 public String getName() {
  return DB_NAME;
 }
}
```

If we added this class to ReportServer, we would already be able to execute Birt and Jasper Reports where the query is simply passed through, but using a dynamic list with a Firebird datasource would still fail.

There are basically three more changes to be made to fully support Firebird from within ReportServer.

As of ReportServer 4.2.0, the old `createDummyQuery()` method is not further used, as it now uses JDBC 4 `connection.isValid()` for this purpose. You should remove this method from your scripts if you use it.

The other two enhancements concern the earlier mentioned matter of limit and offset. ReportServers default implementation creates queries with `LIMIT` and `OFFSET` keywords at the end, so we need to provide a different implementation for Firebird.

```
@Override
 public LimitQuery getNewLimitQuery(Query nestedQuery, QueryBuilder queryBuilder)  ↙
     ↳  {
  return new LimitQuery(nestedQuery, queryBuilder){
   @Override
   public void appendToBuffer(StringBuffer buf) {
    buf.append("SELECT FIRST ");
    buf.append(queryBuilder.getLimit());
    buf.append(" * FROM (");
    nestedQuery.appendToBuffer(buf);
    buf.append(") limitQry");
   }
  }
 }

 @Override
 public OffsetQuery getNewOffsetQuery(Query nestedQuery, QueryBuilder queryBuilder ↙
     ↳  , ColumnNamingService columnNamingService) {
  return new OffsetQuery(nestedQuery, queryBuilder, columnNamingService){
   @Override
   public void appendToBuffer(StringBuffer buf) {
    buf.append("SELECT FIRST ");
    buf.append(queryBuilder.getLimit());
    buf.append(" SKIP ");
    buf.append(queryBuilder.getOffset());
    buf.append(" * FROM (");
    nestedQuery.appendToBuffer(buf);
    buf.append(") limitQry");
   }
  }
 }
```

The way ReportServer constructs dynamic queries is by basically wrapping multiple layers of SQL around each other. The LimitQuery follows the same approach. It has access to a nestedQuery and is asked to write itself into the supplied buffer, wrapping this nested query.

The final step is to register the newly created DatabaseHelper with ReportServer. This is done by implementing the net.datenwerke.rs.base.service.dbhelper.hooks.DatabaseHelperProviderHook. Following is the implementation needed to attach our custom Firebird database helper class

```
String HOOK_NAME = "DATASOURCE_HELPER_FIREBIRD"
DatabaseHelperProviderHook callback =  [
```

```
   provideDatabaseHelpers : {
    return Collections.singletonList(new Firebird());
   }
  ] as DatabaseHelperProviderHook;
GLOBALS.services.callbackRegistry.attachHook(HOOK_NAME, DatabaseHelperProviderHook ↙
     ↳  .class, callback)
```

You will find the complete script (`AddFirebirdSupport.groovy`) in the appendix as well as in the support portal for download.

Note that, in order for this to work you of course need to place the Firebird JDBC driver into your classpath. Download Jaybird from https://www.firebirdsql.org/en/jdbc-driver/ and put the files `jaybird-2.2.3.jar` and `lib/connector-api-1.5.jar` into the WEB-INF/lib subdirectory of your ReportServer installation. You might have to restart ReportServer afterwards.

Beware, that after restarting ReportServer hooks attached from the terminal will no longer be present. To automatically attach a hook on start-up put the script in the `bin/onstartup.d` directory in the fileserver.

Chapter 12

Additional Report Executors

In this chapter we go through the necessary steps to add custom additional report exporters, for example to export dynamic lists into a custom text-based format. When talking about adding custom exporting options we need to distinguish between two things. On the one hand we need the server side code to actually implement the exporting logic. This already allows you to use the exporter, for example, when executing the report via URL. If you also want to add the option to the UI we need to take additional steps. In this chapter we will discuss which steps are necessary for both the server side integration and the client side integration.

12.1 Background

Reporting engines are structured similarly and each reporting engine comes with one or more so called OutputGenerators. As usual you can add additional generators via implementing the correct Hook interface. Following is a list of reporting engines and the corresponding Hook interface needed to be implemented in order to add a custom output generator for the particular engine.

Reporting Engine:	**Dynamic Lists**
Hook Interface:	`net.datenwerke.rs.base.service.reportengines.table.hooks.TableOutputGeneratorProviderHook`
Output Generator Interface:	`net.datenwerke.rs.base.service.reportengines.table.output.generator.TableOutputGenerator`
Reporting Engine:	**JasperReports**
Hook Interface:	`net.datenwerke.rs.base.service.reportengines.jasper.hooks.JasperOutputGeneratorProviderHook`
Output Generator Interface:	`net.datenwerke.rs.base.service.reportengines.jasper.output.generator.JasperOutputGenerator`

Reporting Engine:	**Birt**
Hook Interface:	net.datenwerke.rs.birt.service.reportengine.hooks.BirtOutputGeneratorProviderHook
Output Generator Interface:	net.datenwerke.rs.birt.service.reportengine.output.generator.BirtOutputGenerator

Reporting Engine:	**Crystal**
Hook Interface:	net.datenwerke.rs.crystal.service.crystal.reportengine.hooks.CrystalOutputGeneratorProviderHook
Output Generator Interface:	net.datenwerke.rs.crystal.service.crystal.reportengine.output.generator.CrystalOutputGenerator

Reporting Engine:	**Script Reports**
Hook Interface:	net.datenwerke.rs.scriptreport.service.scriptreport.hooks.ScriptReportOutputGeneratorProvider
Output Generator Interface:	net.datenwerke.rs.scriptreport.service.scriptreport.generator.ScriptReportOutputGenerator

Reporting Engine:	**Saiku**
Hook Interface:	net.datenwerke.rs.saiku.service.saiku.reportengine.hooks.SaikuOutputGeneratorProviderHook
Output Generator Interface:	net.datenwerke.rs.saiku.service.saiku.reportengine.output.generator.SaikuOutputGenerator

Reporting Engine:	**JXLS**
Hook Interface:	net.datenwerke.rs.jxlsreport.service.jxlsreport.reportengine.hooks.JxlsOutputGeneratorProviderHook
Output Generator Interface:	net.datenwerke.rs.jxlsreport.service.jxlsreport.reportengine.output.generator.JxlsOutputGenerator

Reporting Engine:	**GridEditor**
Hook Interface:	net.datenwerke.rs.grideditor.service.grideditor.reportengine.hooks.GridEditorOutputGeneratorProviderHook
Output Generator Interface:	net.datenwerke.rs.grideditor.service.grideditor.reportengine.output.generator.GridEditorOutputGenerator

All output generators extend the interface net.datenwerke.rs.core.service.reportmanager.output.ReportOutputGenerator which specifies defines the following methods

```
String[] getFormats();

boolean isCatchAll();
```

```
CompiledReport getFormatInfo();
```

The first method (getFormats) returns an array of string constants that specify the formats this output generator recognizes. The catchAll method allows to a implement fall back mechanism, that is to implement an output generator that is called in case no other output generator recognizes the format specified by the user. The last method getFormatInfo should return an object of type net.datenwerke.rs.core.service.reportmanager.engine.CompiledReport. A compiled report is the result of an output generator, that is, it is for example a PDF document containing the finished report. The CompiledReport object contains, besides the actual report, information about the format such as the file's mime type, extension etc. The getFormatInfo() method should return an empty CompiledReport object, that is, the object does not contain any data but should specify the format. Let us look at the CompiledReport object in more detail. It is an interface specifying the following methods:

```
Object getReport();

String getMimeType();

String getFileExtension();

boolean hasData();

boolean isStringReport();
```

A CompiledReport is thus a simple bean containing information. GetReport should return the actual report (e.g., PDF document), getMimeType should return a mime type specifying the mime type of the report. GetFileExtension returns a proper file extension (e.g., pdf in case the report is of the PDF format). HasData should return true if the report contains any data, that is, if the underlying datasource provided data for the report. Finally, isStringReport specifies whether the report is of a string format (such as HTML) or of a binary format (such as PDF).

For convenience we have a default implementation of CompiledReport called CompiledReportImpl (in the same package) which takes all the data as a constructor argument. Additionally, we have CompiledReport objects for many standard file formats. These can be found in the package

`net.datenwerke.rs.core.service.reportmanager.engine.basereports`

The following objects are available

- `CompiledCsvReport`
- `CompiledDocReport`
- `CompiledDocxReport`
- `CompiledXHtmlReport`
- `CompiledJsonReport`
- `CompiledPdfReport`

- `CompiledTxtReport`
- `CompiledXlsReport`
- `CompiledXlsxReport`
- `CompiledXmlReport`

With this in mind, we can now add a simple output generator for a dynamic list (note that the corresponding object in ReportServer is called TableReport). The output generator will create a simple HTML page that displays the number of data rows in the result. For this purpose let us look at the TableOutputGenerator interface.

```
void initialize(OutputStream os, TableDefinition td, boolean withSubtotals,  ↙
    ↳ TableReport report, TableReport originalReport, CellFormatter[]  ↙
    ↳ cellFormatters, ParameterSet parameters, User user, ReportExecutionConfig ↙
    ↳ ... configs) throws IOException;

void nextRow() throws IOException;

void addField(Object field, CellFormatter cellFormatter) throws IOException;

void close() throws IOException;

CompiledReport getTableObject();

void addGroupRow(int[] subtotalIndices, Object[] subtotals, int[]  ↙
    ↳ subtotalGroupFieldIndices, Object[] subtotalGroupFieldValues, int rowSize, ↙
    ↳  CellFormatter[] cellFormatters) throws IOException;
```

The initialize method is called before the first data entry is processed. NextRow informs the generator that the current row is complete and that a new row is about to begin. AddField adds a new data cell and close is called upon completion. One thing to keep in mind is that the initialize method can be called in two modes. Either with an OutputStream or without one. In the first case the actual data is to be streamed directly into the output stream. Otherwise the report is to be generated in memory. Usually data is directly streamed and you do not need to worry about in memory generation. GetTableObject returns a CompiledReport object (without any actual data, in case of streaming) and addGroupRow is used for reports that have subtotals enabled. We will ignore this possibility here.

In order to implement our simple row counter, all we need to do is to keep track of the calls to `nextRow()`.

```
import net.datenwerke.rs.base.service.reportengines.table.output.generator.TableOutputGenerator
import net.datenwerke.rs.base.service.reportengines.table.hooks.TableOutputGeneratorProviderHook
import net.datenwerke.rs.base.service.reportengines.table.hooks.adapter. ↙
    ↳ TableOutputGeneratorProviderHookAdapter
import net.datenwerke.rs.core.service.reportmanager.engine.basereports.CompiledXHtmlReport

import java.io.OutputStream

import net.datenwerke.rs.base.service.reportengines.table.entities.Column.CellFormatter
import net.datenwerke.rs.base.service.reportengines.table.entities.TableReport
```

```
import net.datenwerke.rs.base.service.reportengines.table.output.object.TableDefinition
import net.datenwerke.rs.core.service.reportmanager.engine.CompiledReport
import net.datenwerke.rs.core.service.reportmanager.engine.config.ReportExecutionConfig
import net.datenwerke.rs.core.service.reportmanager.output.ReportOutputGenerator
import net.datenwerke.security.service.usermanager.entities.User
import net.datenwerke.rs.core.service.reportmanager.parameters.ParameterSet

String HOOK_NAME = "MY_ADDITIONAL_GENERATOR"

/* specify the generator */
class MyGenerator implements TableOutputGenerator {
  PrintWriter writer = null
  int rows = 0

  void initialize(OutputStream os, TableDefinition td, boolean withSubtotals, TableReport report,  ↙
       ↳  TableReport originalReport, CellFormatter[] cellFormatters, ParameterSet parameters, User  ↙
       ↳  user, ReportExecutionConfig... configs ){
      writer = new PrintWriter(new BufferedWriter(new OutputStreamWriter(os)));
    writer.append("<?xml version=\"1.0\" encoding=\"ISO-8859-1\" ?>" +
                "<html xmlns=\"http://www.w3.org/1999/xhtml\">" +
                "<head></head>" +
                "<body><b>Number of rows:</b>");
  }

    void nextRow(){
     rows++;
    }

    void close() {
     writer.append(" ").append((rows+1) as String).append("</body></html>");
     writer.close();
    }

    boolean supportsStreaming(){
       return true;
    }

    void addField( Object field, CellFormatter cellFormatter ){}

    CompiledXHtmlReport getTableObject() {
     return new CompiledXHtmlReport(null);
    }

    void addGroupRow (int[] subtotalIndices, Object[] subtotals, int[] subtotalGroupFieldIndices, Object ↙
      ↳  [] subtotalGroupFieldValues, int rowSize, CellFormatter[] cellFormatters ){ }

    String[] getFormats() {
      String[] formats = ['MY_CUSTOM_FORMAT']
      return formats
    }

  boolean isCatchAll() { return false; }

  CompiledXHtmlReport getFormatInfo() {
     return new CompiledXHtmlReport(null);
  }

}

/* specify provider */
TableOutputGeneratorProviderHookAdapter provider = [
  provideGenerators : { ->
    return [new MyGenerator()]
  }
```

```
] as TableOutputGeneratorProviderHookAdapter

/* plugin hook */
GLOBALS.services.callbackRegistry.attachHook(HOOK_NAME, TableOutputGeneratorProviderHook.
class, provider)
```

There are few things to explain. In order to make the new output generator available, we need to install our new generator object in the TableOutputGeneratorProviderHook which returns a list of generator objects. Note that we should return a new instance upon every call to method "provideGenerators". Otherwise one would need to ensure that the output generator is implemented in a thread safe manner.

If we execute our script (assume it is called `customDynamicListExporter.groovy`) then you should see the following terminal response.

```
reportserver$ exec generator.groovy
MY_ADDITIONAL_GENERATOR
```

You can now use your custom exporter when exporting dynamic lists, for example, if you export the dynamic list via URL:

```
http://.../reportserver/reportexport?key=customer&format=MY_CUSTOM_FORMAT
```

Note that the output generator will only be available as long as ReportServer is running. If you want this output generator added permanently you should add the script to the list of startup scripts (the startup.d folder, see Section 8.2).

12.2 Adding the Output Generator to the Client

So far we have added the new output generator, but you can only use it when exporting reports via the URL, that is, normal users would not know that the output generator exists since it doesn't show in the user interface. To change this we will use the ClientExtensionService (see Section 9.3). Following is the script needed to add the exporter as an option to the client. The ClientExtensionService offers a method "addReportExportOutputFormat which we need to supply with an instance of a report object, a string to be displayed, the actual format, and an optional path to an icon. The object instance in our case is TableReportDtoDec. Remember that each server object has a corresponding client object called ServerObjectDto. The code of most of these objects is generated automatically during the compile process and thus, in order to manually add code most objects come with a corresponding decorator. You will always find the corresponding decorator in the sub-package decorator and by convention it is called ServerObjectDtoDec, thus, in our case we deal with TableReportDtoDec.

```
import net.datenwerke.rs.base.client.reportengines.table.dto.decorator. ↙
    ↳ TableReportDtoDec

/* obtain ClientExtensionService */
def ces = GLOBALS.services['clientExtensionService']

/* register format */
ces.addReportExportOutputFormat new TableReportDtoDec(), "My Format", " ↙
    ↳ MY_CUSTOM_FORMAT", ""
```

```
""
```

Again, remember that this script needs to be run after a user has logged in and should thus be placed in the `onlogin.d` directory.

12.3 Skipping file download

Further, you can configure your output generator to skip the file download of the generated file. This may be useful if you want to export information about your report execution without having to return a file to the user. As an example, the following script tracks the number of rows in the report and saves this information to an external SQL Server database by using the Groovy Sql class. A file download would be unnecessary in this case. Install the following script into your `onstartup.d` directory.

```
import net.datenwerke.rs.base.service.reportengines.table.output.generator. ↙
    ↳ TableOutputGenerator
import net.datenwerke.rs.base.service.reportengines.table.hooks. ↙
    ↳ TableOutputGeneratorProviderHook
import net.datenwerke.rs.base.service.reportengines.table.hooks.adapter. ↙
    ↳ TableOutputGeneratorProviderHookAdapter
import net.datenwerke.rs.core.service.reportmanager.engine.basereports. ↙
    ↳ CompiledXHtmlReport

import java.io.OutputStream

import net.datenwerke.rs.base.service.reportengines.table.entities.Column. ↙
    ↳ CellFormatter
import net.datenwerke.rs.base.service.reportengines.table.entities.TableReport
import net.datenwerke.rs.base.service.reportengines.table.output.object. ↙
    ↳ TableDefinition
import net.datenwerke.rs.core.service.reportmanager.engine.CompiledReport
import net.datenwerke.rs.core.service.reportmanager.engine.config. ↙
    ↳ ReportExecutionConfig
import net.datenwerke.rs.core.service.reportmanager.output.ReportOutputGenerator
import net.datenwerke.security.service.usermanager.entities.User
import net.datenwerke.rs.core.service.reportmanager.parameters.ParameterSet
import groovy.sql.Sql

String HOOK_NAME = "MY_ADDITIONAL_GENERATOR"

/* specify the generator */
class MyGenerator implements TableOutputGenerator {

   def db = [url:'jdbc:sqlserver://IP;databaseName=myDb', user:'myUser', password ↙
     ↳ :'password', driver:'com.microsoft.sqlserver.jdbc.SQLServerDriver']
  def sql = Sql.newInstance(db.url, db.user, db.password, db.driver)

   def reportName = null

  int rows = 0
```

```
  void initialize(OutputStream os, TableDefinition td, boolean withSubtotals,  ↙
       ↳  TableReport report, TableReport originalReport, CellFormatter[]  ↙
       ↳  cellFormatters, ParameterSet parameters, User user,  ↙
       ↳  ReportExecutionConfig... configs ){
    reportName = report.name
    }

    void nextRow(){
       rows++;
    }

    void close() {
        def params = [reportName, rows+1]
        sql.execute 'insert into execution_counts (report_name, number_of_rows)  ↙
     ↳  values (?, ?)', params
    }

    boolean supportsStreaming(){
       return false;
    }

    void addField( Object field, CellFormatter cellFormatter ){}

    CompiledXHtmlReport getTableObject() {
     return new CompiledXHtmlReport("");
    }

    void addGroupRow (int[] subtotalIndices, Object[] subtotals, int[]  ↙
     ↳  subtotalGroupFieldIndices, Object[] subtotalGroupFieldValues, int rowSize, ↙
     ↳   CellFormatter[] cellFormatters ){ }

    String[] getFormats() {
       String[] formats = ['MY_CUSTOM_FORMAT']
       return formats
    }

  boolean isCatchAll() { return false; }

  CompiledXHtmlReport getFormatInfo() {
     return new CompiledXHtmlReport("");
  }

}

/* specify provider */
TableOutputGeneratorProviderHookAdapter provider = [
  provideGenerators : { ->
    return [new MyGenerator()]
  }
] as TableOutputGeneratorProviderHookAdapter
```

```
/* plugin hook */
GLOBALS.services.callbackRegistry.attachHook(HOOK_NAME,  ↙
    ↳ TableOutputGeneratorProviderHook.
class, provider)
```

In order to install this script into the client interface, you can use a similar script as in the previous example, with the difference that the `skipDownload` option is set in this case. Install the following script into your `onlogin.d` directory.

```
import net.datenwerke.rs.base.client.reportengines.table.dto.decorator. ↙
    ↳ TableReportDtoDec
import net.datenwerke.rs.scripting.service.scripting.extensions. ↙
    ↳ AddReportExportFormatProvider

import net.datenwerke.rs.base.client.reportengines.table.dto.TableReportVariantDto
import net.datenwerke.rs.base.client.reportengines.table.dto.decorator. ↙
    ↳ TableReportVariantDtoDec

/* obtain ClientExtensionService */
def ces = GLOBALS.services['clientExtensionService']

/* register format */
AddReportExportFormatProvider provider = new AddReportExportFormatProvider(new  ↙
    ↳ TableReportDtoDec(), "My Format", "MY_CUSTOM_FORMAT", "")
provider.skipDownload = true
ces.addReportExportOutputFormat provider

""
```

Chapter 13

Send To

In this chapter we explain how to add custom "send to targets" to the ***Send to...*** menu within the report executor. These can be used to, for example, upload a report to a web service.

> Note that, as of ReportServer 4.3.0, script datasinks are supported and are the recommended way to send reports to a given custom target. Scheduling is of course also supported. Details can be found in Chapter 7. Nevertheless, in case you need custom forms, you can use the SendTo functionality as explained below.

To add a custom target we need to implement `net.datenwerke.rs.core.service.sendto.hooks.SendToTargetProviderHook` which consists of three methods: `consumes`, `getId`, and `sendTo`. The `getId` method is used to define a unique identifier to identify the send to target. The `consumes` method takes as input a report object and allows to specify a target configuration. Finally, the `sendTo` method is called to execute the action. The basic outline to define a target is thus the following code snippet

```
import net.datenwerke.rs.core.service.sendto.hooks.SendToTargetProviderHook
import net.datenwerke.rs.core.service.sendto.hooks.adapter. ↙
    ↳ SendToTargetProviderHookAdapter
import net.datenwerke.rs.core.client.sendto.SendToClientConfig

String HOOK_NAME = 'MY_SEND_TO_EMAIL'

SendToTargetProviderHookAdapter callback = [
  consumes : { report ->
    SendToClientConfig config = new SendToClientConfig()
    config.title = 'Custom Send To'
    config.icon = 'wrench'
    return config
  },
  getId : { ->
    return 'aUniqueId'
  },
  sendTo : { report, values, execConfig ->
   // perform send to action
  }
```

```
] as SendToTargetProviderHookAdapter

GLOBALS.services.callbackRegistry.attachHook(HOOK_NAME, SendToTargetProviderHook,  ↙
    ↳ callback)
```

The `consumes` method takes as input the current report object and returns an object of type `SendToClientConfig` (or null, if for this report the target should not be active). In its simplest form the `SendToClientConfig` only takes a title and possibly an icon (any font-awesome icon identifier, you may find the identifiers here: `net.datenwerke.rs.theme.client.icon.BaseIcon.SEND`) which is used to populate the send-to menu on the client. If the user selects the menu item the `sendTo` method of the above hook is called. The `sendTo` method takes as input

`report`	The report object with its current configuration.
`values`	In case a form configuration is specified (we'll cover forms shortly) the values object is a map containing the selected values.
`execConfig`	Contains the execution config which should be passed on in case the report is executed.

An important point to make is that the report object is not necessarily as the stored object in the database but is configured as it was currently configured on the client side. This in particular means that the `ID` field of the report object is not set. In order to obtain the id you can use the `getOldTransientId()` method. Note that this is not the case for the report provided to the `consumes` method.

Output Format

By default the send to target does directly call the script when the user selects the option. You can specify that the user needs to configure an output format. For this call `config.selectFormat  ↙ ↳ = true` when specifying the `SendToClientConfig` object. Then, when the user selects the send-to target he or she will first be asked to specify an output format. In this case you can make your life easier when implementing the hook and override the method

```
public String sendTo(CompiledReport compiledReport, Report report,
     String format, HashMap<String, String> values,
     ReportExecutionConfig... executionConfig)
```

Here, as first parameter you are given the executed report (compiledReport). In this case we do not need to implement the other sendTo method, and an implementation could look like the following:

```
import net.datenwerke.rs.core.service.sendto.hooks.SendToTargetProviderHook
import net.datenwerke.rs.core.service.sendto.hooks.adapter.  ↙
    ↳ SendToTargetProviderHookAdapter
import net.datenwerke.rs.core.client.sendto.SendToClientConfig

String HOOK_NAME = 'MY_SEND_TO_EMAIL'

SendToTargetProviderHookAdapter callback = [
  consumes : { report ->
    SendToClientConfig config = new SendToClientConfig()
    config.title = 'Custom Send To'
```

```
    config.selectFormat = true
    return config
  },
  getId : { ->
    return 'aUniqueId'
  },
  sendTo : { compiledReport, report, format, values, execConfig ->
   // perform send to action
  }

] as SendToTargetProviderHookAdapter

GLOBALS.services.callbackRegistry.attachHook(HOOK_NAME, SendToTargetProviderHook,  ↙
     ↳  callback)
```

Form Configuration

In addition to (or in place of) having the user select an output format for the report you can specify additional form fields. That is, you can display a simple form when the send-to menu item is selected and then have the data from the form available when executing the action. This is what the parameter `values` is for. It contains a map of the form-data specified by the user. In order to configure a form you can set the "form" property of the `SendToClientConfig`:

```
SendToTargetProviderHookAdapter callback = [
  consumes : { report ->
    SendToClientConfig config = new SendToClientConfig()
    config.title = 'Custom Send To'
    config.form = '' // the form configuration via json
    return config
  },
  getId : { ->
    return 'aUniqueId'
  },
  sendTo : { report, values, execConfig ->
   // perform send to action
  }

] as SendToTargetProviderHookAdapter
```

The form is defined via JSON (https://www.json.org/). The basic outline is as follows, where "form" would take the actual form configuration. The properties width and height define the width and height of the popup window that contains the form.

```
{
        "width": 400,
        "height": 200,
        "form" : {
                // form configuration
        }
}
```

The form configuration itself consists of a definition of fields.

```
"form" : {
          "fields": [{
           // first field
   }, {
           // second field
          }]
}
```

Each field consists of an id, a type, a label and possibly a value. For example,

```
{
       "id": "email",
       "type": "string",
       "label": "Email Address",
       "value": "name@example.com"
}
```

The field id defines a unique name for the form element, label defines the field label and value allows to specify the field's content. Type, defines the type of form field. Currently supported are the types:

`string`	A text field.
`int`	An integer field.
`dd`	A dropdown list.

Putting it all together we could define a form as follows, which consists of three fields, a textfield, an integer field and a dropdown list. Note that the values for the dropdown list are provided as key-value pairs.

```
{
   "width": 400,
        "height": 200,
        "form" : {
                "labelAlign": "left",
                "fields": [{
                        "id": "email",
                        "type": "string",
                        "label": "Email Address",
                        "value": "name@example.com"
                }, {
                        "id": "int",
                        "type": "int",
                        "label": "Some integer field",
                        "value" : "15"
                }, {
                        "id" : "dropdown",
                        "type" : "dd",
                        "label" : "some list",
                        "values" : [
```

```
                                {"A" : "B"}, {"c" : "D" }, {"foo" : "bar"}
                        ],
                        "value" : "foo"
                }]
        }
}
```

If a form is specified, then the `values` parameter of the `sendTo` method consist of a `Map<String ,String>` object that contains the value (represented as string) for each form field.

Scheduling

By default any send-to target is also available as a scheduler target. You can prevent this by implementing the method `supportsScheduling`:

```
SendToTargetProviderHookAdapter callback = [
  consumes : { report ->
    SendToClientConfig config = new SendToClientConfig()
    config.title = 'Custom Send To'
    return config
  },
  getId : { ->
    return 'aUniqueId'
  },
  sendTo : { report, values, execConfig ->
   // perform send to action
  },
  supportsScheduling: { -> return false }

] as SendToTargetProviderHookAdapter
```

If you have scheduling enabled, then by default a scheduling execution is forwarded to the sendTo method of your hook. You can however further control the scheduling execution by additionally overriding the method `scheduledSendTo`.

```
SendToTargetProviderHookAdapter callback = [
  consumes : { report ->
    SendToClientConfig config = new SendToClientConfig()
    config.title = 'Custom Send To'
    return config
  },
  getId : { ->
    return 'aUniqueId'
  },
  sendTo : { report, values, execConfig ->
   // perform send to action
  },
  scheduledSendTo : { compiledReport, report, reportJob, format, values ->
   // perform action
  },
  supportsScheduling: { -> return true }
```

```
] as SendToTargetProviderHookAdapter
```

Similarly, to the `sendTo` method you have access to the report object and the configured values. However, in addition you can access the compiledReport (the scheduler executed the report according to the instructions and the first parameter contains the result) as well as to the scheduler job definition.

Complete Example

The following is a complete example that reimplements sending a report via email.

```
import net.datenwerke.rs.core.service.sendto.hooks.SendToTargetProviderHook
import net.datenwerke.rs.core.service.sendto.hooks.adapter.SendToTargetProviderHookAdapter
import net.datenwerke.rs.core.client.sendto.SendToClientConfig

import net.datenwerke.rs.core.service.mail.MailService
import net.datenwerke.rs.core.service.reportmanager.ReportExecutorService
import net.datenwerke.rs.core.service.mail.SimpleAttachment
import net.datenwerke.rs.core.service.mail.SimpleMail

String HOOK_NAME = 'MY_SEND_TO_EMAIL'

reportExec = GLOBALS.getInstance(ReportExecutorService)
mailService = GLOBALS.getInstance(MailService)

SendToTargetProviderHookAdapter callback = [
   consumes : { report ->
      SendToClientConfig config = new SendToClientConfig()
      config.title = 'Send via Custom Mail'
      config.icon = 'send'
      config.form = """
{
        "width": 400,
        "height": 180,
        "form" : {
                "labelAlign": "top",
                "fields": [{
                        "id": "email",
                        "type": "string",
                        "label": "Email Address",
                        "value": "name@example.com"
                }]
        }
}
"""
      return config
   },
   getId : {
      ->
      return 'someUniqueId'
   },
   sendTo : { report, values, execConfig ->
      def pdf = reportExec.execute(report, ReportExecutorService.OUTPUT_FORMAT_PDF, execConfig)

      // prepare for sending mail
      SimpleMail mail = mailService.newSimpleMail()
      mail.subject = 'The Report'
      mail.toRecipients = values['email']
      mail.from = 'from@reportserver.net'

      SimpleAttachment attachment = new SimpleAttachment(pdf.report, pdf.mimeType, 'filename.pdf')
      mail.setContent('Some Message', attachment)
```

```
      // send mail
      mailService.sendMail mail

      return "Send the report via mail. Config $values" as String
   }

] as SendToTargetProviderHookAdapter

GLOBALS.services.callbackRegistry.attachHook(HOOK_NAME, SendToTargetProviderHook, callback)
```

The script can be downloaded here: https://github.com/infofabrik/reportserver-samples/blob/main/src/net/datenwerke/rs/samples/tools/sendto/email/sendToEmail.groovy.

If you need scheduling, you can use the following example:

```
import net.datenwerke.rs.core.service.sendto.hooks.SendToTargetProviderHook
import net.datenwerke.rs.core.service.sendto.hooks.adapter.SendToTargetProviderHookAdapter
import net.datenwerke.rs.core.client.sendto.SendToClientConfig

import net.datenwerke.rs.core.service.mail.MailService
import net.datenwerke.rs.core.service.reportmanager.ReportExecutorService
import net.datenwerke.rs.core.service.mail.SimpleAttachment
import net.datenwerke.rs.core.service.mail.SimpleMail

String HOOK_NAME = 'MY_SEND_TO_EMAIL'

reportExec = GLOBALS.getInstance(ReportExecutorService)
mailService = GLOBALS.getInstance(MailService)

def doSendEmail(pdf, values) {
  // prepare for sending mail
  SimpleMail mail = mailService.newSimpleMail()
  mail.subject = 'The Report'
  mail.toRecipients = values['email']
  mail.from = 'from@reportserver.net'

  SimpleAttachment attachment = new SimpleAttachment(pdf.report, pdf.mimeType, 'filename.pdf')
  mail.setContent('Some Message', attachment)

  // send mail
  mailService.sendMail mail
}


SendToTargetProviderHookAdapter callback = [
   consumes : { report ->
      SendToClientConfig config = new SendToClientConfig()
      config.title = 'Send via Custom Mail'
      config.icon = 'send'
      config.form = """
{
        "width": 400,
        "height": 180,
        "form" : {
                "labelAlign": "top",
                "fields": [{
                        "id": "email",
                        "type": "string",
                        "label": "Email Address",
                        "value": "name@example.com"
                }]
        }
```

```
}
"""
      return config
   },
   getId : {
      ->
      return 'someUniqueId'
   },
   sendTo : { report, values, execConfig ->
      def pdf = reportExec.execute(report, ReportExecutorService.OUTPUT_FORMAT_PDF, execConfig)

      doSendEmail(pdf, values)

      return "Send the report via mail. Config $values" as String
   },

   scheduledSendTo: { compiledReport, report, reportJob, format, values ->
     def pdf = reportExec.execute(report, ReportExecutorService.OUTPUT_FORMAT_PDF)

     doSendEmail(pdf, values)
   }

] as SendToTargetProviderHookAdapter

GLOBALS.services.callbackRegistry.attachHook(HOOK_NAME, SendToTargetProviderHook, callback)
```

The script can be downloaded here: https://github.com/infofabrik/reportserver-samples/blob/main/src/net/datenwerke/rs/samples/tools/sendto/email/sendToEmailScheduling.groovy.

Chapter 14

Further Examples

In the following GitHub project we include some useful scripting examples. These examples aim to show how to achieve different functionalities with help of scripting.

https://github.com/infofabrik/reportserver-samples

Appendix A

AddFirebirdSupport.groovy

```
1  package databasehelper;
2
3  import net.datenwerke.rs.scripting.service.scripting.scriptservices.GlobalsWrapper;
4  import net.datenwerke.rs.base.service.dbhelper.DatabaseHelper
5  import net.datenwerke.rs.base.service.dbhelper.hooks.DatabaseHelperProviderHook
6  import net.datenwerke.rs.base.service.dbhelper.queries.LimitQuery
7  import net.datenwerke.rs.base.service.dbhelper.queries.OffsetQuery
8  import net.datenwerke.rs.base.service.dbhelper.queries.Query
9  import net.datenwerke.rs.base.service.dbhelper.querybuilder.ColumnNamingService
10 import net.datenwerke.rs.base.service.dbhelper.querybuilder.QueryBuilder
11
12
13
14 class Firebird extends DatabaseHelper {
15
16   public static final String DB_NAME = "Firebird";
17   public static final String DB_DRIVER = "org.firebirdsql.jdbc.FBDriver";
18   public static final String DB_DESCRIPTOR = "DBHelper_Firebird";
19
20   @Override
21   public String getDescriptor() {
22     return DB_DESCRIPTOR;
23   }
24
25   @Override
26   public String getDriver() {
27     return DB_DRIVER;
28   }
29
30   @Override
31   public String getName() {
32     return DB_NAME;
33   }
34
35   @Override
36   public LimitQuery getNewLimitQuery(Query nestedQuery, QueryBuilder queryBuilder) {
37     return new LimitQuery(nestedQuery, queryBuilder){
38       @Override
39       public void appendToBuffer(StringBuffer buf) {
40         buf.append("SELECT FIRST ");
41         buf.append(queryBuilder.getLimit());
42         buf.append(" * FROM (");
43         nestedQuery.appendToBuffer(buf);
44         buf.append(") limitQry");
45       }
```

```
46      }
47    }
48
49    @Override
50    public OffsetQuery getNewOffsetQuery(Query nestedQuery, QueryBuilder queryBuilder, ColumnNamingService ↙
          ↪   columnNamingService) {
51      return new OffsetQuery(nestedQuery, queryBuilder, columnNamingService){
52        @Override
53        public void appendToBuffer(StringBuffer buf) {
54          buf.append("SELECT FIRST ");
55          buf.append(queryBuilder.getLimit());
56          buf.append(" SKIP ");
57          buf.append(queryBuilder.getOffset());
58          buf.append(" * FROM (");
59          nestedQuery.appendToBuffer(buf);
60          buf.append(") limitQry");
61        }
62      }
63    }
64 }
65
66
67 def HOOK_NAME = "DATASOURCE_HELPER_FIREBIRD"
68
69 def callback =  [
70        provideDatabaseHelpers : {
71          return Collections.singletonList(new Firebird());
72        }
73      ] as DatabaseHelperProviderHook;
74
75 GLOBALS.services.callbackRegistry.attachHook(HOOK_NAME, DatabaseHelperProviderHook.class, callback)
```

Appendix B

ldapimport.groovy

The current script can be found here: https://github.com/infofabrik/reportserver-samples/blob/main/src/net/datenwerke/rs/samples/admin/ldap/ldapimport.groovy.