# Groovy

Groovy-Scripting im CMS.


- API
- Bibliothek – Übersicht
- CMS-spezifische Helfer
- Groovy Syntax und Beispiele
- Groovy – Übersicht
- Logging
- Timings
- Tips
- Verwendungsmöglichkeiten

# API

eine gekapselte Klasse mit statischen Methoden, die zusammengehörige Funktionalitäten enthältMyApi (bei `class` und `logName` muss noch angepasst werden.

```
import com.batix.Log
import com.batix.action.GroovyAction
import com.batix.modul.Customer
import com.batix.modul.SystemVariables
import com.batix.table.TableMetadata
import com.batix.tags.GroovyTag

import javax.servlet.ServletContext
import javax.servlet.http.HttpServletRequest
import javax.servlet.http.HttpServletResponse
import javax.servlet.http.HttpSession
import java.sql.Connection

class MyApi {
  private static GroovyAction.GroovyContext getContext() {
    return GroovyAction.context
  }

  private static GroovyTag getIncludeTag() {
    return context.includeTag
  }

  private static GroovyAction getAction() {
    return context.action
  }

  private static ServletContext getApplication() {
    return context.application
  }

  private static HttpServletRequest getRequest() {
    return context.request
```

```
  }

  private static HttpServletResponse getResponse() {
    return context.response
  }

  private static HttpSession getSession() {
    return context.session
  }

  private static Customer getWeb() {
    return context.web
  }

  private static SystemVariables getVariables() {
    return context.variables
  }

  private static Connection getConn() {
    return action?.connection ?: includeTag?.checkConnection()
  }

  /* Logging start */
  private static def logName = "MyApi"
  //noinspection GroovyUnusedAssignment
  private static void logD(String msg, HttpServletRequest request = null) {
Log.debug("[${logName}] ${msg}", request) }
  //noinspection GroovyUnusedAssignment
  private static void logI(String msg, HttpServletRequest request = null) {
Log.info("[${logName}] ${msg}", request) }
  //noinspection GroovyUnusedAssignment
  private static void logN(String msg, HttpServletRequest request = null) {
Log.notice("[${logName}] ${msg}", request) }
  //noinspection GroovyUnusedAssignment
  private static void logW(String msg, HttpServletRequest request = null) {
Log.warn("[${logName}] ${msg}", request) }
  //noinspection GroovyUnusedAssignment
  private static void logE(String msg, Exception ex = null, HttpServletRequest request = null)
{ Log.error("[${logName}] ${msg}", request, ex) }
  /* Logging end */
```

```
  // Beispiel: öffentliche Konstante
  public static final def ID_STATUS_OFFEN = "15FBFE52655"

  // Beispiel: öffentliche Methode
  static Map info(String vid) { return null }
}
```

# Bibliothek – Übersicht

Hier werden wiederverwendbare Code-Schnipsel und Helferlein gesammelt. Diese müssen, falls nicht anders angegeben, einfach nur rauskopiert werden und im eigenen Code (am besten ganz oben) eingefügt werden.

- API — eine gekapselte Klasse mit statischen Methoden, die zusammengehörige Funktionalitäten enthält
- BatixUtils
- Logging — um einfach Logausgaben zu erzeugen, die den Name des Scripts vorangestellt haben
- Timings — nützlich, um einzelne Code-Teile zu profilen und so herauszufinden, was genau langsam ist

# CMS-spezifische Helfer

Bestimmte CMS-Klassen wurden um Utilities zur einfacheren Verwendung erweitert.

# Importe

Einige häufig genutzte Klassen werden automatisch importiert, z.B. `java.sql.Connection`, `TableMetadata` und `BatixRecord`.

# Objekte

Dem Groovy-Script stehen automatisch folgende Objekte zur Verfügung:

- `includeTag` - Referenz zum ausführenden `bx:groovy` Tag, ansonsten `null`
  - es können hier z.B. Tag-Parameter gelesen werden:
    **Template**

    ```
    <bx:groovy includes="MyReport" format="html" />
    ```

    **Groovy-Baustein**

    ```
    String format = includeTag.getStringParameter("format")
    println("gewähltes Format: $format")
    ```

- `action` - Referenz zum ausführenden Groovy-Action, ansonsten `null`
  - kann z.B. benutzt werden, um Script-Attribute zu setzen oder das Action abzubrechen

    ```
    action.setScriptAttribute("thing", "value")
    action.cancel = true
    ```

- `application` - der `ServletContext`
- `request` - der aktuelle `HttpServletRequest`
  - z.B. um Request-Parameter auszulesen, die der Seite oder dem Action übergeben wurden

    ```
    String kid = request.getParameter("kid")
    ```

- `response` - die aktuelle `HttpServletResponse`

    - falls ein Action verwendet wird: `action.originalResponse` benutzen, um z.B. direkt Bytes zu schreiben
- `session` - die aktuelle Session, falls vorhanden
- `pageData` - enthält Infos zum Aufruf

```
String file = pageData.filename // z.B. "detail.htm"
```

- `navElement` - der aufgerufene Menüpunkt (`NavigationElement`)
- `web` - das Projekt (`Customer`)
- `variables` - die Systemeinstellungen (`SystemVariables`)
    - Das sind die pro Projekt überschreibbaren Variablen

```
String portalId = variables.getVariable("PortalID")
```

# Connection

Um einfach an eine Datenbank-Connection zu kommen, egal ob der Groovy-Baustein in einem Action oder Tag (oder sogar den Entwickler-Tools) verwendet wird, kann folgender Block verwendet werden:

```
Connection.with { conn ->
  // hier Code, der conn benutzt
}
```

Die Connection wird nach dem Block automatisch wieder geschlossen bzw. zurückgegeben.

# TableMetadata holen

Eine Instanz von `TableMetadata` lässt sich einfach zu einem Container erzeugen, in dem Entweder der Titel, der Tabellenname oder die ID des Containers benutzt wird (es wird eine `TableMetadata`-Instanz zurückgegeben):

```
// funktioniert alles gleich
def tmdKunden = TableMetadata["Kunden"]
def tmdKunden = TableMetadata["bxc_kunden"]
def tmdKunden = TableMetadata["15525037CCA"]

// auch mit Punkt-Schreibweise verfügbar
def tmdKunden = TableMetadata.Kunden
```

```
def tmdKunden = TableMetadata.bxc_kunden
def tmdKunden = TableMetadata."15525037CCA" // als String, da sonst Syntax Error wegen Ziffer am Anfang
```

# BatixRecord holen

Ein bestimmter Datensatz aus einem Container, kann durch seine ID einfach geholt werden (es wird eine `BatixRecord`-Instanz zurückgegeben):

```
def recKunde = tmdKunden["15525037CCB"]
```

Einen neuen Datensatz erzeugt man mit den üblichen Methoden:

```
def recKunde = tmdKunden.createNewRecord(conn, com.batix.util.UniqueID.createHexId(), true)
```

# Datensatz-Felder lesen / schreiben

Hat man einen `BatixRecord` kann man auf einfache Art und Weise die Datensatz-Felder lesen und ändern:

```
// Text
String name = recKunde.Name.string
recKunde.Titel.string = "Dr."

// Zahl
int alter = recKunde.Alter.number // Ganzzahl-Feld
recKunde.Alter.number = 18
double rabatt = recKunde.Rabatt.number // Dezimalzahl-Feld
recKunde.Rabatt.number = 5.83

// Datum, Datum + Uhrzeit, Uhrzeit oder Zeitraum
Date bday = recKunde.Geburtstag.date
recKunde.lastUpdate.date = new Date()

// Häkchen
boolean archived = recKunde.archiviert.state
recKunde.Newsletter.state = false
```

```
// Einzelverknüpfung (Bild, Dokument, Datensatz)
String picId = recKunde.Foto.string
String statusId = recKunde.Status.string
recKunde.Rolle.string = "15525037CCC"

// Mehrfachverknüpfung
MultipleLinkField leistungen = recKunde.Leistungen

// Untercontainer
SubListField vertraege = recKunde.Vertraege

// zum Schluss Update nicht vergessen
recKunde.updateRecordInDatabase(conn)
// oder Anlegen, falls es ein neuer Datensatz war
recKunde.createRecordInDatabase(conn, false) // false = Default-Werte der Container-Felder
übernehmen
```

> Durch die angesprochenen Probleme mit Typ-Infos, kann es passieren, dass bei Datum/Zeit-Feldern der Setter uneindeutig ist, falls `null` übergeben wird (da es dort mehrere `setDate` Methoden gibt).
>
> Falls eine Variable übergeben wird, die auch `null` sein kann, wird daher beim Setzen von Datum/Zeit Feldern empfohlen, die `setDate` Methode aufzurufen und einen expliziten Cast hinzuzufügen:

```
Date birthday = ...
recKunde.Geburtstag.setDate(birthday as Date)
```

# Mehrere Datensätze filtern

Ähnlich dem XML-Filter mit bx:containerfilter können in Groovy Datensätze gefiltert werden. Der Methode werden als erster Parameter eine Liste an Kriterien (bestehend aus einer drei-Elementigen Liste pro Kriterium mit: Feld(ern), Vergleichstyp und Vergleichswert) und als zweiter Parameter eine Map von Optionen übergeben. Das Ergebnis is ein (evtl. leeres) Array aus `BatixRecord`s.

Ab Version 2.6.8 kann bei der Kriterien-Liste jeweils ein vierter Parameter angegeben werden ( `true` oder `false`), welcher dem `required`-Attribut in der XML entspricht. Das ist nützlich, falls man direkt Sachen aus dem Request o.ä. als Vergleichswert angibt: `["Name", 1, request.getParameter("name"), true]`

(in späteren Versionen führt ein leerer Vergleichswert ohne explizite required-Angabe dann zum Abbruch)

```
BatixRecord[] recs = tmdKunden.findRecords([
  ["Leistungen", 5, 0], // Feld darf nicht leer sein
  ["Newsletter", 4, 1], // Feld muss angehakt sein
  [["Name", "Vorname"], 1, "muster"] // mindestens eins der beiden Felder muss "muster"
enthalten
], [
  orderby: "Kundennummer", // optional, Sortierungsfeld (Standard ID)
  desc: true, // optional, Sortierrichtung (Standard aufsteigend)
  index: 5, // optional, Startindex (Null-basiert)
  limit: 20, // optional, maximale Anzahl Ergebnisse
  active: true, // optional, aktive Datensätze suchen? (Standard ja)
  inactive: true // optional, inaktive Datensätze suchen? (Standard nein)
])
```

Die Zahlen entsprechen den Typen des normalen Filters. Es müssen nicht alle Parameter angegeben werden, ein kurzes Beispiel ist (in Groovy steht `[:]` für eine leere Map):

```
BatixRecord[] recs = tmdKunden.findRecords([
  ["archived", 4, 1] // alle archivierten Kunden finden
], [:])
```

Ab Version 2.7.0 können auch mehrere Sortierungs-Kriterien angegeben werden. Dabei sind folgende Formate für `orderby` möglich (`desc` wird dann ignoriert):

```
BatixRecord[] recs = tmdKunden.findRecords([], [
  orderby: [
    "Firma", // aufsteigend
    ["Name"], // aufsteigend
    ["Vorname", true] // absteigend
  ]
])
```

# Einen Datensatz filtern

Ab Version 2.7.0 ist es auch möglich nur den ersten Datensatz zu laden, anstatt alle:

```
BatixRecord rec = tmdKunden.findRecord([
  // siehe oben
], [
  // siehe oben
])
```

Rückgabewert ist entweder ein `BatixRecord` oder `null`.

# Batix-Quelltext ausführen

Will man schnell einen String evaluieren, der Batix-Tags enthält, kann diese Methode benutzt werden:

```
String code = '<bx:math>1 + 1</bx:math>'
String result = code.evaluateAsBatixSource()
```

Der Standard Mime-Typ ist text/html, um einen anderen zu benutzen (manche Tags machen unterschiedliche Sachen, je nach Mime-Typ der Seite) kann dieser als Parameter mitgegeben werden: `evaluateAsBatixSource("text/plain")`.

# Groovy Syntax und Beispiele

Nebst den hier am häufigsten genutzten Elementen gibt es noch viele weitere Funktionen und Helfer, siehe dazu auch die offizielle Doku (auf die Version achten).

# Syntax

Klammern um Parameter sowie das Semikolon am Ende der Zeile können weggelassen werden. Empfehlung: Klammern verwenden, Semikolon weglassen.

```
println "Hello, World"
println("Hello, World") // Empfehlung
println("Hello, World");
```

Typen müssen nicht explizit angegeben werden (`def` entspricht quasi `Object` in Java bzw. `var` in JavaScript). Empfehlung: Typen explizit angeben, falls die IDE sie nicht automatisch erkennt und demtentsprechend keine sinnvollen Vorschläge für Syntax-Completion macht.

```
def x = 11.8 // erst Zahl
println(x)

x = "Test" // dann String
println(x.toUpperCase())
```

Typen werden, soweit möglich automatisch umgewandelt, es müssen keine expliziten Casts verwendet werden (hier kann aber das Schlüsselwort `as` verwendet werden). Es erfolgt auch ein automatisches Boxing / Unboxing (int <--> Integer etc.).

```
def i = "5" as int
eineListe.add(12)
```

Der Gleichheits-Operator in Groovy (`==`) wird im Hintergrund zu einem `.equals()` Aufruf umgewandelt.

```
String a = "test"
String b = "te" + "st"
println(a == b) // true, in Java: a.equals(b)
```

Strings können von Anführungszeichen (`"`), Apostrophen (`'`) oder Slashes (`/`) umschlossen sein. Empfehlung: Das verwenden, was im String nicht vorkommt, um nicht escapen zu müssen (Slashes, falls Anführungszeichen und Apostrophe vorkommen bzw. für Reguläre Ausdrücke).

```
"str" == 'str' == /str/
```

Es gibt auch Mehrzeilige Strings. Ein Backslash (`\`) bewirkt, dass der Zeilenumbruch vor der ersten Zeile im Code nicht mit ausgegeben wird.

```
println("""
Zeile 1
Zeile 2
""")

println("""\
einzige Zeile""")
```

In Strings mit Anführungszeichen (einfach oder dreifach für mehrzeilige Strings) können Code-Platzhalter verwendet werden, die durch ihren Wert ersetzt werden.

```
def user = "alice"
println("Hi $user")
println("aktueller Timestamp: ${new Date().time}")
```

# Closures

Closures entsprechen in etwa den `function`s in JavaScript oder Lambda-Funktionen in Java 8. Es muss dabei nicht explizit `return` angegeben werden, der Rückgabewert des letzten Statements wird als Rückgabewert der Closure benutzt. Parameter müssen nicht typisiert werden.

```
// keine Parameter
def c1 = {
  "Hello"
}
c1()

def c2 = { name ->
  "Hello, $name!"
}
c2("World")
```

Methoden, die eine Closure als einzigen oder letzten Parameter akzeptieren, kann die Closure direkt übergeben werden (keine runden Klammern nötig).

```
"...".eachLine { line ->
  println(line.reverse())
}
```

Falls die Closure genau ein Parameter übergeben bekommt, so muss dieser nicht benannt werden, der Standardname ist "`it`".

```
"...".eachLine {
  println(it.reverse())
}
```

# Spaceship-Operator

Vergleiche (`compareTo`) erledigt der Spaceship-Operator, das vereinfacht die Implementation eines `Comparator`s.

```
println 5 <=> 10         // Java: 5.compareTo(10)
println "def" <=> "abc"  // Java: "def".compareTo("abc")
println null <=> 10      // gibt keine Exception
```

**Beispiel: System Properties filtern**

```
System.properties
  .findAll { it.key.startsWith("user") }
  .sort { x, y -> x.key <=> y.key }
  .each { println it }
```

# Get / Set

Es werden automatisch JavaBeans-Accessors erzeugt bzw. angesprochen, d.h. man muss nicht `getProp()` oder `setProp()` aufrufen, sondern kann einfach `prop` schreiben.

```
// Lesen
println(obj.thing)   // entspricht obj.getThing()
```

```
println("...".empty) // entspricht "...".isEmpty()

// Schreiben
obj.thing = null     // entspricht obj.setThing(null)
```

# Arrays / Lists / Maps / Ranges / Collections

Für Listen und Maps gibt es eine native Syntax.

**Liste**

```
def a = [1, 2, 3]  // eine ArrayList im Hintergrund
println(a[1])      // einfach auf Elemente zugreifen

println([].size()) // eine leere Liste
```

**Map**

```
def m = [language: "Groovy", version: 2.1] // eine LinkedHashMap
println m["language"]  // einfach auf Elemente zugreifen
println m.version      // einfach auf Elemente zugreifen

println([:].size())    // eine leere Map
```

Mit Ranges kann man einfach Zahlenbereiche generieren.

```
(1..10).each { println(it) }  // 1 bis 10
(1..<10).each { println(it) } // 1 bis 9
(5..-5).each { println(it) }  // absteigend
```

Außerdem können mittels Ranges teile von Strings und Collections extrahiert werden.

```
def a = [1, 2, 3, "a", "b", "c"]
println a[-2]     // b
println a[2..4]   // [3, a, b]
println a[2..1]   // [3, 2]
println a[-2..2]  // [b, a, 3]
```

```
def s = "the quick brown fox"
println s[4..9, -3..-1]   // quick fox
```

Um zu überprüfen, ob eine Collection ein Element enthält, kann der `in` Operator verwendet werden.

```
println(5 in [1, 5, 10]) // true
```

Diesen gibt es auch in der verkürzten Variante der `for`-Schleife.

```
def a = [1, 5, 10]
for (i in a) {
  println(i)
}
```

Groovy bringt viele Hilfsmethoden für Collections mit.

```
[5, 4, 8, 1, 4, 5, 2].sort().unique().reverse()   // Sortieren, Doppler entfernen, Reihenfolge
invertieren
(1..10).take(3).drop(1)                           // nur die ersten 3 Elemente nehmen und
davon dann das erste wegschmeißen
[4, 7, 1].min()                                   // Minimum finden
[cents: 5, dime: 2, quarter: 3].max { it.value }  // Maximum finden und dabei festlegen, was
verglichen werden soll pro Element
(1..100).sum()                                    // Summe bilden
["abc", "def", "abcdef"].findAll { it.startsWith("a") }  // alle Elemente finden, die einer
Bedingung entsprechen
["abc", "abcdef"].collect { it.length() } == [3, 6]      // Elemente transformieren

['a', 'b'] << [1, 2]    // [a, b, [1, 2]]   Element anhängen
['a', 'b'] + [1, 2]     // [a, b, 1, 2]     Listen zusammenführen
[1, 2, 3].join("~")   // 1~2~3     Elemente mittels Trenner zusammenführen

def a = [1, 2, 3, 4, 5]
println(a.any { it > 3 })    // true, erfüllt mindestens ein Element die Bedingung?
println(a.every { it < 5 })  // false, erfüllen alle Elemente die Bedingung?
```

Wenn man nicht immer prüfen will, ob ein Key in einer Map ist, bevor man ihn benutzt, kann man ein Default-Wert festlegen. Dieser wird dann unter dem Key automatisch in der Map angelegt, sobald das erste Mal auf ihn zugegriffen wird.

```
def map = [:].withDefault {
  new Date()
}

println map.test  // Zeit 1
sleep(2000);
println map.test  // immer noch Zeit 1
println map.test2 // Zeit 2
```

# RegEx

Reguläre Ausdrücke haben auch eine verkürzte Syntax. Slashes werden hier nur der Lesbarkeit / Identifizierbarkeit verwendet, es sind immer noch normale Strings (nicht wie in JavaScript, wo mit Slashes direkt reguläre Ausdrücke erzeugt werden).

```
// Pattern erzeugen
Pattern p = ~/\d+/
println(p.matcher("123").matches())

// Matcher erzeugen
Matcher m = "123" =~ /\d(\d)\d/
println(m.matches())
println(m.group(1))

// Direkt matchen
println("123" ==~ /\d+/) // true
```

# Null-Dereferenzierung

Um NullPointerExceptions vorzubeugen, ohne dauernd mit `if` auf `null` prüfen zu müssen, empfiehlt sich der `?.` Operator. Dieser stoppt, sobald der Ausdruck vor dem Fragezeichen `null` ist und gibt `null` zurück.

```
class Person {
  def name
}
```

```
def p
println(p?.name?.length())  // null

p = new Person()
println(p?.name?.length())  // null

p.name = "Tester"
println(p?.name?.length())  // 6
```

# Elvis-Operator

Default Werte können mit dem Elvis-Operator (`?:`) vereinfacht werden. Dies entspricht dem Ternary-If, wobei die Bedingung und der True-Teil identisch sind.

```
null ?: 5        // 5         -->  null ? null  :      5
false ?: "test"  // test      --> false ? false : "test"
4 ?: 8           // 4         -->     4 ?     4 :      8
```

# Switch

Das switch Statement wurde auch erweitert.

```
def testSwitch(val) {
  switch (val) {
    case ~/^Switch.*Groovy$/: return 'Pattern match'
    case BigInteger: return 'Class isInstance'
    case 60..90: return 'Range contains'
    case [21, 'test', 9.12]: return 'List contains'
    case 42.056: return 'Object equals'
    case { it instanceof Integer && it < 50 }:
      return 'Closure boolean'
    default: return 'Default'
  }
}
```

```
println testSwitch("Switch to Groovy")
println testSwitch(42G)
println testSwitch(70)
println testSwitch('test')
println testSwitch(42.056)
println testSwitch(20)
println testSwitch('default')
```

# JSON

Mit JSON kann sehr einfach gearbeitet werden.

### JSON lesen

```
def slurper = new groovy.json.JsonSlurper()
def json = slurper.parseText("""
{
  "name": "Mustermann",
  "ids": [5, 10, 11]
}
""")
println(json.name)   // Mustermann
println(json.ids[1]) // 10
```

> In neueren Groovy-Versionen liefert der `JsonSlurper` eine `LazyMap` zurück, die nicht serialisiert werden kann (falls etwas in der Session gespeichert werden soll). Die Variante mit `HashMap` ist noch als `JsonSlurperClassic` verfübar.

Ein JSON-String ist auch schnell aus normalen Objekten (Strings, Lists, Maps, ...) erzeugt:

### JSON schreiben

```
def builder = new groovy.json.JsonBuilder()
def obj = [
  name: "Mustermann",
  ids: [5, 10, 11]
]
builder(obj)
println(builder.toPrettyString()) // JSON von oben
```

Als Abkürzung kann die Hilfsklasse `JsonOutput` verwendet werden.

**JsonOutput**

```
import groovy.json.JsonOutput

def obj = [
  name: "Mustermann",
  ids: [5, 10, 11]
]
def jsonStr = JsonOutput.toJson(obj)
println(jsonStr) // alles auf einer Zeile, spart Bytes
println(JsonOutput.prettyPrint(jsonStr)) // lesbar
```

# XML

Für XML-Input gibt es 2 Verarbeitungsmöglichkeiten: `XMLParser` und `XMLSlurper` (siehe dazu auch die Groovy Doku).

`XmlParser` parst das komplette XML und baut eine Struktur im RAM auf, was es erlaubt Teile der XML oft und dabei schnell abzufragen (falls dem XML-Dokument Nodes aktualisiert oder hinzugefügt werden sollen, wird auch `XmlParser` empfohlen):

**XML lesen (mit XmlParser)**

```
def parser = new XmlParser()
def root = parser.parseText("""
<person>
  <access read='true' write='false' />
  <data>
    <name>Mustermann</name>
  </data>
  <data>
    <name>Musterfrau</name>
  </data>
</person>
""")
println(root.access.@write[0])     // false
println(root.data[1].name.text())  // Musterfrau
```

`XmlSlurper` hingegen wertet das XML nur partiell und on-demand aus - somit wird RAM gespart, häufige Zugriffe hingegen dauern ggf. länger (dafür können Abfragen aber etwas kürzer geschrieben werden):

**XML lesen (mit XmlSlurper)**

```
def slurper = new XmlSlurper()
def root = slurper.parseText("""
<person>
  <access read='true' write='false' />
  <data>
    <name>Mustermann</name>
  </data>
  <data>
    <name>Musterfrau</name>
  </data>
</person>
""")
println(root.access.@write)     // false
println(root.data[1].name)      // Musterfrau
```

Um XML-Dokumente programmatisch zusammenzubauen, bietet such `MarkupBuilder` an:

**XML schreiben**

```
def writer = new StringWriter()
def builder = new groovy.xml.MarkupBuilder(writer)

builder.person {
  access(read: true, write: false)
  data {
    name("Mustermann")
  }
  data {
    name("Musterfrau")
  }
}

println(writer.toString()) // XML von oben
```

Um schnell einzelne Teile einer XML zu ändern und wieder in einen String zu überführen, kann der `XmlParser` und die Hilfsklasse `XmlUtil` verwendet werden. Hier ein Beispiel, um Werte zum

Validation-Feld hinzuzufügen:

**XML parsen, bearbeiten, zurückschreiben**

```
import groovy.xml.XmlUtil

// XML parsen
def rootNode = new XmlParser().parseText("""
<result size="1">
  <field name="Kontakt_Email" resultcode="1" bin="00000001" hex="0x01" />
</result>
""")

// Attribut ändern
rootNode.@size++

// Node hinzufügen (Parameter sind Name der Node sowie Attribute als Map)
rootNode.appendNode("field", [
  "name": "Kontakt_Name",
  "resultcode": "1",
  "bin": "00000001",
  "hex":"0x01"
])

// wieder XML-String erzeugen
println(XmlUtil.serialize(rootNode))
/* Ergebnis:
<?xml version="1.0" encoding="UTF-8"?><result size="2">
  <field name="Kontakt_Email" resultcode="1" bin="00000001" hex="0x01"/>
  <field name="Kontakt_Name" resultcode="1" bin="00000001" hex="0x01"/>
</result>
 */
```

# SQL

Auch mit SQL kann einfacher gearbeitet werden.

```
java.sql.Connection conn = ...
def sql = new groovy.sql.Sql(conn)
```

```
def foo = "fruit"
sql.eachRow("SELECT * FROM food WHERE type=${foo}") {  // Parameter werden automatisch SQL-
Encoded
  println("I like ${it.name}")                         // gibt Spalte "name" aus
}
```

> Um ein automatisches SQL-Encoden zu gewährleisten, darf der Query-String nicht aus mehreren Strings zusammengebaut werden, sondern darf nur ein String mit Platzhaltern sein.

# Überladene Methoden und Typen

Da Groovy Methoden dynamisch aufruft, zur Laufzeit aber nicht mehr die genauen Typen aus dem Quellcode kennt (wegen Type Erasure etc.), kann es vorkommen, dass mehrere Methoden gefunden werden, die zum Aufruf passen.

Das ist meistens der Fall, wenn `null` übergeben wird, hier weiß Groovy nur, dass es ein `NullObject` ist, kennt aber nicht den genauen Typ:

```
String str = null
println(str.getClass().simpleName) // NullObject
```

Beispielsweise gibt es diese zwei Methoden:

```
void setDate(String str) { ... }
void setDate(Date date) { ... }
```

Erfolgt nun in Groovy der Aufruf folgendermaßen:

```
// null als Literal
obj.setDate(null)
obj.date = null

// oder als Variable
String str = null
obj.date = str
```

Dann weiß Groovy nicht, welche Methode es genau aufrufen soll, das sowohl `String` als auch `Date` zu `NullObject` passen (`null` bzw. `NullObject` kann in beides umgewandelt werden).

Frühere Groovy-Versionen haben einfach irgendeine Methode genommen, neue Versionen werfen eine Exception. Da dies nicht rückwärts-kompatibel ist, bringen CMS-Versionen, welche neuere Groovy-Versionen enthalten, stattdessen einen Fehler im Log, funktionieren aber weiterhin mit altem Code (es wird irgendeine Methode genommen).

Falls so ein Szenario auftritt, kann man Groovy aber helfen, in dem man beim Aufruf den Typ explizit als Cast hinzufügt (es muss hier die Methode aufgerufen werden, beim Property-Setter geht die Typ-Info wieder verloren):

```
obj.setDate(str as String) // OK
obj.setDate((String)str)   // OK


// obj.date = str as String // geht nicht
// obj.date = (String)str   // geht nicht
```

# Groovy – Übersicht

Seit Version 2.6.2 ist es möglich die Scriptsprache Groovy im CMS zu verwenden (sowohl in Actions als auch in Templates). Zu beachten ist, dass je CMS-Version ggf. eine andere Groovy-Version eingebunden ist - dazu die .jar Datei im lib-Verzeichnis prüfen oder in Entwickler-Tools ausführen: `println(GroovySystem.version)`.

Groovy ist eine dynamische Sprache mit vereinfachter Syntax und Funktionalitäten (ähnlich JavaScript) und vielen schon eingebauten Utilities. Außerdem wurden bestimmte Sachen des CMS erweitert, damit man sie einfach in Groovy benutzen kann (z.B. Datensätze bearbeiten).


- Verwendungsmöglichkeiten
- Groovy Syntax und Beispiele
- CMS-spezifische Helfer
- Tips
- Bibliothek – Übersicht

# Logging

Es können `logI("text")` für INFO-Meldungen etc. verwendet werden. `logE` kann als zweiter Parameter zusätzlich noch eine Exception übergeben werden.

Zur besseren Übersicht im Log wird immer der Scriptname vorangestellt.

`logName` noch anpassen

es muss noch `com.batix.Log` importiert werden

# Zum Kopieren (für Script)

```
/* Logging start */
def logName = "myScript"
//noinspection GroovyUnusedAssignment
def logD = { String msg, HttpServletRequest request = null -> Log.debug("[${logName}] ${msg}",
request) }
//noinspection GroovyUnusedAssignment
def logI = { String msg, HttpServletRequest request = null -> Log.info("[${logName}] ${msg}",
request) }
//noinspection GroovyUnusedAssignment
def logN = { String msg, HttpServletRequest request = null -> Log.notice("[${logName}]
${msg}", request) }
//noinspection GroovyUnusedAssignment
def logW = { String msg, HttpServletRequest request = null -> Log.warn("[${logName}] ${msg}",
request) }
//noinspection GroovyUnusedAssignment
def logE = { String msg, Exception ex = null, HttpServletRequest request = null ->
Log.error("[${logName}] ${msg}", request, ex) }
/* Logging end */
```

# Zum Kopieren (für Klassen)

```
/* Logging start */
private static def logName = "myScript"
//noinspection GroovyUnusedAssignment
private static void logD(String msg, HttpServletRequest request = null) {
Log.debug("[${logName}] ${msg}", request) }
//noinspection GroovyUnusedAssignment
private static void logI(String msg, HttpServletRequest request = null) {
Log.info("[${logName}] ${msg}", request) }
//noinspection GroovyUnusedAssignment
private static void logN(String msg, HttpServletRequest request = null) {
Log.notice("[${logName}] ${msg}", request) }
//noinspection GroovyUnusedAssignment
private static void logW(String msg, HttpServletRequest request = null) {
Log.warn("[${logName}] ${msg}", request) }
//noinspection GroovyUnusedAssignment
private static void logE(String msg, Exception ex = null, HttpServletRequest request = null) {
Log.error("[${logName}] ${msg}", request, ex) }
/* Logging end */
```

# Beispiel

```
def user = BxUser.findInstance(request)
if (!user) {
  logE("no user found")
  return
}

logI("found user ${user.id}")
```

# Timings

Zwischendrin immer `addTiming("kleine Info was gemacht wurde")` aufrufen.

Es können auch einzelne Durchläufe einer Schleife gemessen werden: `addTiming("Info zur Schleife", "Info zum Durchlauf")`. So werden die aufeinanderfolgende Timings, bei denen der erste Parameter identisch ist, gruppiert.

Am Ende gibt `getTimings()` eine Zusammenfassung als `String` zurück. Für Schleifen wird die Gesamtdauer sowie die Anzahl der Durchläufe und die mittlere Dauer ausgegeben. Außerdem werden die Top und Flop 3 der Durchläufe (nach Dauer) angezeigt.

# Zum Kopieren

```
/* Timings start */
def timings = []
def timingLast = new Date().time
def addTiming = { String title, String subTitle = null ->
  def duration = new Date().time - timingLast
  if (subTitle) {
    def value = [
      duration: duration,
      title: subTitle
    ]
    if (!timings || timings[-1].title != title) {
      timings << [
        title: title,
        values: []
      ]
    }
    timings[-1].values << value
    value.index = timings[-1].values.size()
  } else {
    timings << [
      duration: duration,
      title: title
```

```
    ]
  }
  timingLast = new Date().time
}
Closure<String> getTimings = {
  def res = new StringBuilder()
  res << "Timings:\n"
  def timingTotal = 0
  def emptyPrefix = " " * 27
  def maxDetails = 6
  timings.each { t ->
    if (t.values) {
      List vals = t.values.sort { a, b ->
        a.duration <=> b.duration ?:
          a.index <=> b.index
      }
      long sum = vals.sum { it.duration } as long
      double avg = sum / t.values.size()
      timingTotal += sum
      res << String.format("%8d ms (+%8d ms) %s (%dx ~%.2f ms)\n", timingTotal, sum, t.title,
t.values.size(), avg)
      if (vals.size() > maxDetails) vals = [*vals[0..(maxDetails / 2 - 1)], null, *vals[-
(maxDetails / 2)..-1]]
      vals.each { val ->
        res << emptyPrefix
        if (val) res << String.format("#%-4d %4dms %s\n", val.index, val.duration, val.title)
        else res << "[...]\n"
      }
    } else {
      timingTotal += t.duration
      res << String.format("%8d ms (+%8d ms) %s\n", timingTotal, t.duration, t.title)
    }
  }
  def lastDuration = new Date().time - timingLast
  timingTotal += lastDuration
  res << String.format("%8d ms total\n", timingTotal)
  return res.toString()
}
/* Timings end */
```

# Beispiel

```
// do stuff
addTiming("startup")
// do more stuff
addTiming("init")
// do loop stuff 1
addTiming("fetch info", "load one...")
// do loop stuff 2
addTiming("fetch info", "load two...")
// ...
// do database stuff
addTiming("finalizing")
// do final stuff
```

Erzeugt dann z.B. mit `logI(getTimings())` sowas:

```
Timings:
       1 ms (+       1 ms) startup
      94 ms (+      93 ms) init
     753 ms (+     659 ms) fetch info (7x ~94,14 ms)
                           #4      15ms load 15F53263D95
                           #5      16ms load 15F53263DA5
                           #3      17ms load 15F53263D86
                           [...]
                           #2      20ms load 15F53263D75
                           #6      78ms load 15F53263DF3
                           #1     496ms load 15F53263D61
    2598 ms (+    1845 ms) finalizing
    2605 ms total
```

# Tips

Hier gibt es lose gesammelte Tricks und Kniffe.

# Bessere Exceptions im Log

Um Stacktraces von unnötigem Groovy-Meta-Ballast zu befreien und so direkt lesbar zu machen, gibt es `StackTraceUtils.deepSanitize`. Statt die Exception direkt an `Log` o.ä. zu übergeben, einfach diese Methode drumherum packen und schon verschwinden unnötige Zeilen aus dem Stacktrace.

```
import com.batix.Log
import org.codehaus.groovy.runtime.StackTraceUtils
// ...
try {
  // ...
} catch (Exception ex) {
  Log.error("my error message", StackTraceUtils.deepSanitize(ex))
}
```

# Session

Manchmal gibt es Probleme in **Groovy**, wenn es keine Session gibt, weil standardmäßig kein Sessioncookie mehr erzeugt wird. Dadurch ist die Groovy-Variable session nun manchmal leer und muß im GroovyAction mit if (!session) session = request.session gefüllt werden und in <bx:groovy> mit session = includeTag.forceSession() Das müßte auch noch irgendwo in die Hilfe eingetragen werden.

Wenn man keine Session im **JSP** braucht sollte man oben <%@page session="false"%> hinzufügen. Sonst wird auf der Seite, auf der das JSP eingebunden ist, ein Session-Cookie erzeugt.

# Verwendungsmöglichkeiten

Groovy-Code kann in Actions (Actionbaustein "Groovy ausführen") oder in Quelltexten (bx:groovy) eingesetzt werden. Es gibt auch die Möglichkeit in den Entwickler-Tools schnell Groovy-Code auszuführen.

Um Ausgaben zu erzeugen, kann einfach `println` verwendet werden:

- Tag: der Text wird in die Seite geschrieben
- Action: der Text wird ausgegeben, falls `action.sendJSPOutput()` aufgerufen wurde, ansonsten kann auch `action.originalResponse` verwendet werden
- Entwickler-Tools: der Text wird im Output auf der Seite angezeigt

Außerdem ist es jeweils möglich zentrale Groovy-Bausteine aus dem aktuellen Projekt einzubinden (unter Dokumentvorlagen > Groovy), um Code nachzunutzen - dies ist das gleiche Prinzip wie bei Textbausteinen.