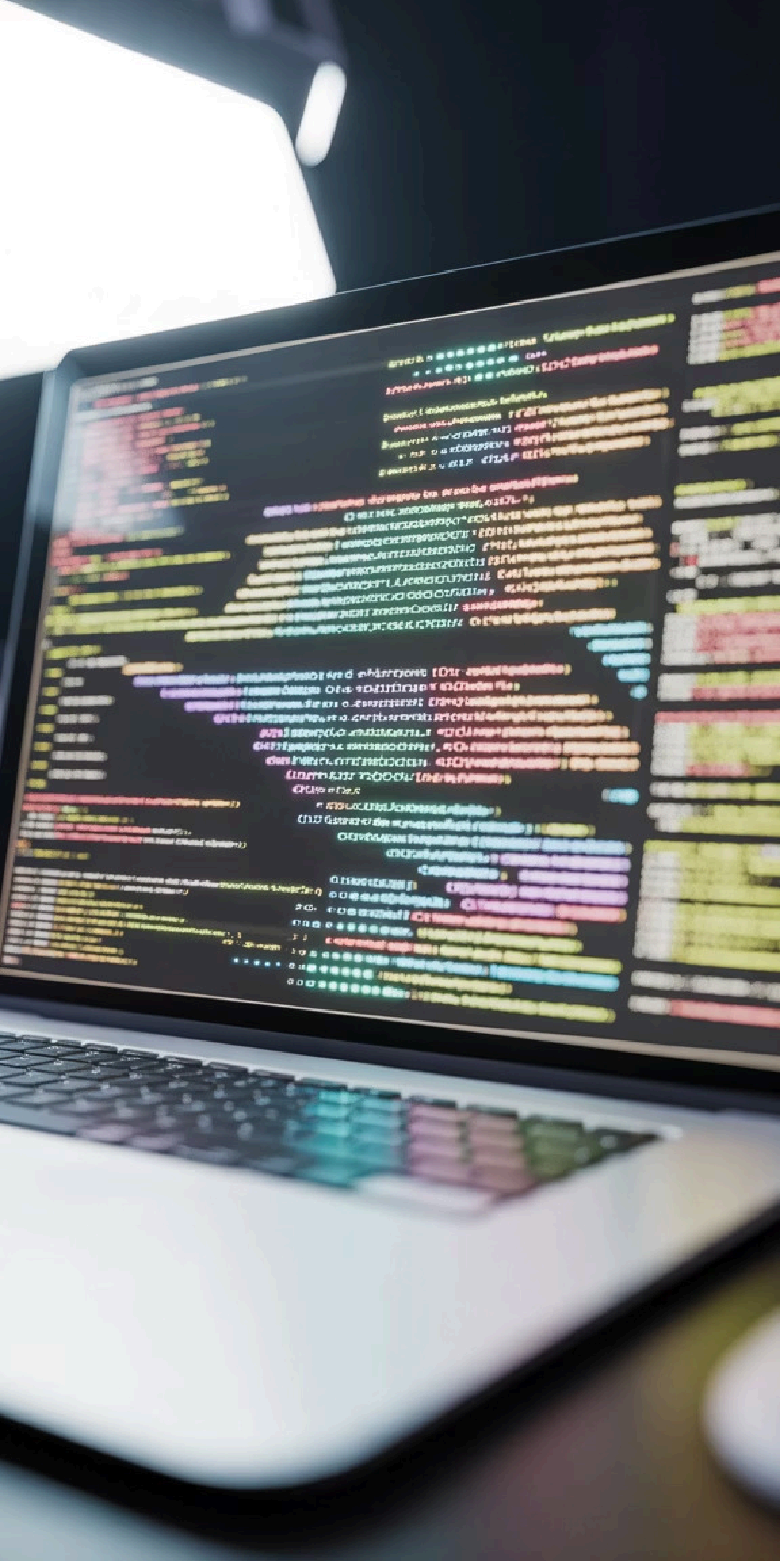


Groovy Training

Master Groovy scripting for SAP CPI integration development. From fundamentals to advanced techniques.

Variables in Groovy

Dynamic Typing

Use `def` for runtime type resolution. Flexible and reduces boilerplate in scripts.

Static Typing

Declare with explicit types like `int`, `String` for better IDE support and type safety.

```
String employeeName = "Alice Johnson"
int employeeId = 1001
double salary = 55000.0
boolean isActive = true
def department = "Finance"
def yearsOfService = 3
```

Variables can be reassigned unless declared `final`. Naming follows Java standards and is case-sensitive.

Data Types

Primitives

`int`, `double`, `boolean`, `char`

Objects

`String`, `BigDecimal` (default for floating point)

Type Checking

Use `.getClass().name` to inspect variable types

Coercion

Convert types using `as` or explicit casting

Operators

Arithmetic

`+`, `-`, `*`, `/` for calculations

Relational

`>`, `==`, `!=` for comparisons

Logical

`&&`, `||`, `!` for boolean logic

Assignment

`+=`, `-=`, `*=` for compound operations

Groovy's operator overloading maps symbols to methods internally. `==` checks value equality, `is` checks object identity.

Operator Example

Salary Calculation

Combining arithmetic and logical operators for employee compensation logic.

```
int base = 50000
int bonus = 5000
println "Total: ${base + bonus}"
println "Net: ${base - 2000}"
println "Monthly: ${base / 12}"

boolean eligible = base > 45000 && bonus >= 5000
println "Eligible? $eligible"
```

Conditionals

1

if/else

Decision making based on boolean expressions

2

switch

Supports ranges, collections, and types

3

Truthy Values

Non-null and non-zero treated as `true`

Groovy's `switch` is more flexible than Java's, evaluating types and ranges for powerful conditional logic.

Conditional Example

```
int years = 6
String performance = "Excellent"

if (years >= 5 && performance == "Excellent") {
 println "Eligible for promotion"
} else {
 println "Regular review"
}

int rating = 4
switch (rating) {
 case 5: case 4: println "Great"; break
 case 3: println "Good"; break
 default: println "Review needed"
}
```

Loops

01

for Loop

Enhanced iteration over collections and ranges directly

02

while Loop

Useful when exit condition isn't based on iteration

03

Collection Methods

`each`, `collect`, `find` as functional alternatives

04

Control Flow

Use `break` and `continue` for loop control

Loop Example

For Loop

```
def names = ["Alice", "Bob", "Carol"]
for (n in names) {
    println "Name: $n"
}
```

While Loop

```
int i = 0
while (i < names.size()) {
    println "Indexed: ${names[i]}"
    i++
}
```

Groovy introduces range syntax like `1..5` to simplify loop conditions and iteration.

Exception Handling

try

Contains code that might throw exceptions

finally

Always executes, often used for cleanup

1

2

3

catch

Handles specific exception types to avoid crashes

Groovy supports multi-catch and doesn't enforce checked exceptions like Java. Always use specific exceptions for better error handling.

Exception Example

```
try {
    def res = 10 / 0
} catch (ArithmeticException e) {
    println "Cannot divide: ${e.message}"
} finally {
    println "Always runs"
}
```

Use specific exception types rather than catching `Exception` or `Throwable` unless necessary for robust error handling.

Methods

Encapsulation

Reusable logic blocks that improve code modularity

Default Parameters

Methods support default values and omit return keyword

Named Arguments

Support named arguments using maps for clarity

Overloading

Multiple methods with same name, different parameters

Method Example

Annual Bonus Calculation

Method with default parameter demonstrating Groovy's concise syntax.

```
def annualBonus(double salary,
 double percent = 10) {
 return (salary * percent / 100).toInteger()
}

println "Bonus: " + annualBonus(60000)
println "Custom: " + annualBonus(60000, 15)
```

Closures

Anonymous code blocks that can be passed as arguments or assigned to variables. First-class citizens in Groovy.

Closure Features

Anonymous Functions

Declared with curly braces `{}`, can take parameters and return values

Scope Access

Access and modify variables defined outside their scope

Collection Methods

Used with `each`, `findAll`, `collect`

Implicit Parameters

Support `it` for single-argument use

Closure Example

```
def greet = { name -> println "Hello, $name!" }
greet("Alice")

def nums = [10, 20, 30]
def above15 = nums.findAll { it > 15 }
println "Filtered: $above15"
```

Closures enable functional programming patterns and improve readability in compact operations.

Lists

Ordered Collections

Implemented as ArrayList, hold any type, dynamically resizable

Construction

Use square brackets [] for concise list creation

Manipulation

Use <<, add, remove for element operations

Utility Methods

each, find, collect, sort for processing

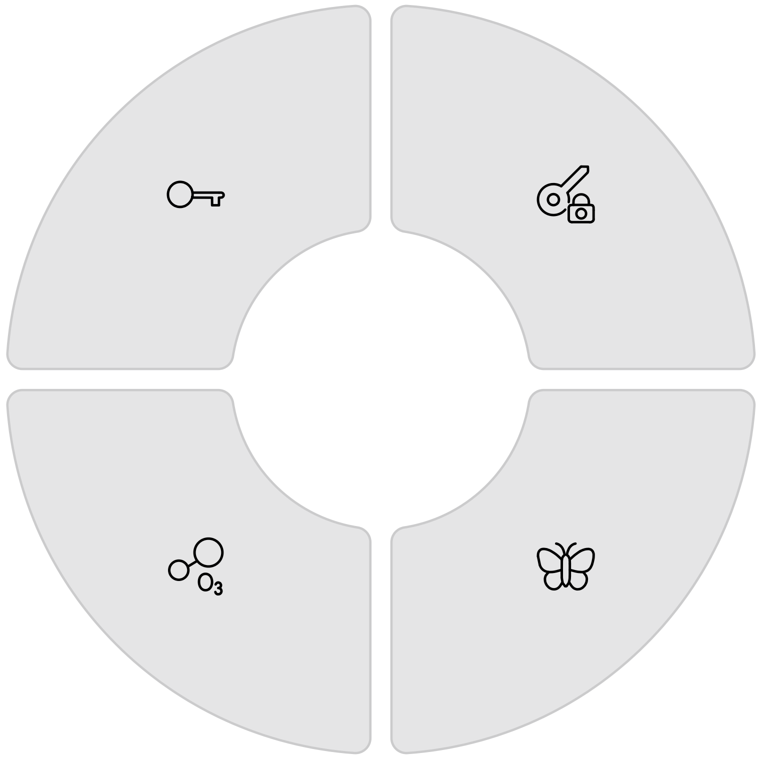
List Example

```
def emps = ["John", "Jane"]
emps << "Jim"
emps.remove("Jane")
emps.each { println "Emp: $it" }
```

Key Operations

- Add elements with `<<`
- Remove by value
- Iterate with `each`
- Access by index

Maps



Key-Value Pairs

Unordered collections using `[:]` syntax

Access Methods

Dot or bracket notation for value retrieval

Flexible Structure

Support nesting, iteration, dynamic properties

Manipulation

`find`, `collectEntries`, `groupBy` methods

Map Example

```
def emp = [id: 101, name: "Alice", dept: "HR"]
println emp.name

emp.each { k, v -> println "$k -> $v" }
```

Maps are ideal for JSON-like structures and play a vital role in message context handling in integration scenarios.

Ranges

1

Inclusive

Use `..` for inclusive ranges like `1..5`

2

Exclusive

Use `..<` for exclusive upper bound

3

Types

Works with numbers, characters, dates

4

Operations

Iterate, check membership, slice, reverse

Range Example

Numeric Range

```
def r = 1..5
r.each { println "Year: $it" }
```

Character Range

```
def letters = 'A'..'D'
println "Letters: $letters"
```

Ranges provide clean syntax in for-loops and switch statements, improving readability for consecutive values.

Classes and Objects

Class Definition

Blueprints containing fields and methods

Auto-Generation

Getters, setters, constructors, toString auto-created

Object Creation

Use `new` keyword or dynamic expansion

Encapsulation

Properties public by default, accessible via dot notation

Class Example

```
class Employee {
 String name
 double salary
 void raise(double pct) {
 salary += salary * pct / 100
 }
}

def e = new Employee(name: "Alice", salary: 50000)
e.raise(10)
println "${e.name}'s new salary: ${e.salary}"
```

Inheritance

Superclass

Base class with common functionality

Subclass

Extends parent using `extends` keyword

Override

Customize behavior with method overriding

Super Reference

Access parent class with `super` keyword

Inheritance Example

```
class Person {
   String name
   void intro() {
      println "I am $name"
   }
}

class Manager extends Person {
   String dept
   void manage() {
      println "Managing $dept"
   }
}
```

```
def m = new Manager(
   name: "Alice",
   dept: "IT"
)
m.intro()
m.manage()
```

Subclass inherits all public and protected members from superclass.

Abstract Classes

1

Template Definition

Base templates with partial implementation and required methods

2

Abstract Methods

Subclasses must implement all abstract methods

3

No Instantiation

Cannot create instances directly, only through subclasses

4

Structure Enforcement

Useful for enforcing structure in large applications

Abstract Class Example

```
abstract class Staff {
    String name
    abstract String role()
}

class Intern extends Staff {
    String role() { "Internship" }
}

def i = new Intern(name: "Jake")
println "${i.name} - ${i.role()}"
```

Abstract classes promote DRY principles and separation of concerns in object-oriented design.

XML Processing

Groovy provides powerful tools for parsing, transforming, and generating XML documents.

Parsing XML

XmlSlurper

Event-based, lazy reading, uses less memory. Preferred for large documents.

XmlParser

Builds full DOM tree in memory, enables random access.

GPath Navigation

Intuitive XPath-like navigation for accessing elements and attributes.

Object-like Access

Parsed XML behaves like Groovy objects with dot and bracket notation.

XML Parsing Example

```
import groovy.util.XmlSlurper
import groovy.util.XmlParser

def xml = "Alice"

def slurper = new XmlSlurper().parseText(xml)
println "Slurper name: ${slurper.emp.name}"

def parser = new XmlParser().parseText(xml)
println "Parser name: ${parser.emp[0].name.text()}"
```

Transforming XML

1

Parse

Load XML using XmlParser

2

Modify

Change node values, add/delete elements

3

Serialize

Use XmlUtil.serialize() to generate output

Transformations are common in SAP CPI when changing third-party payloads. Groovy's features reduce boilerplate code.

Transforming XML

1

Parse

Load XML using XmlParser

2

Modify

Change node values, add/delete elements

3

Serialize

Use XmlUtil.serialize() to generate output

Transformations are common in SAP CPI when changing third-party payloads. Groovy's features reduce boilerplate code.

01

r

Captures output as string for further processing

02

Method Names

Use method and property names to define XML structure

03

Output

Readable, properly escaped, and compact XML

XML Generation Example

```
import groovy.xml.MarkupBuilder

def xml = "Alice"
def parsed = new XmlSlurper().parseText(xml)

def w = new StringWriter()
def builder = new MarkupBuilder(w)
builder.summary {
 parsed.emp.each {
 employee(id: it.@id, it.name.text())
 }
}
println w.toString()
```

JSON Processing

Groovy's JsonSlurper makes parsing and transforming JSON simple and intuitive.

Parsing JSON

1

JsonSlurper

Parses JSON strings into Groovy Maps and Lists

2

Access Methods

Use dot or bracket notation for property access

3

Deep Nesting

Supports complex nested structures efficiently

4

Collections

JSON arrays become Lists, objects become Maps

JSON Parsing Example

```
import groovy.json.JsonSlurper

def json = '{"id":"E1","name":"Bob","skills":["Java","Groovy"]}'
def data = new JsonSlurper().parseText(json)

println "Name: ${data.name}, Skill 1: ${data.skills[0]}"
```

JSON is often used in APIs and is a key format in SAP CPI integrations for modern web services.

Transforming JSON

1

Parse

Use JsonSlurper to load JSON

2

Modify

Change values like normal Groovy objects

3

Serialize

Use JsonOutput.toJson() to convert back

4

Format

JsonOutput.prettyPrint() for readability

JSON Transform Example

```
import
groovy.json.JsonSlurper
import
groovy.json.JsonOutput

def json = '{"emps":
[{"name":"Alice","salary":500
00}]}'
def data = new
JsonSlurper().parseText(json)
```

```
data.emps.each { it.salary +=
5000 }
println
JsonOutput.prettyPrint(
 JsonOutput.toJson(data)
)
```

Combining XML Documents

Merge multiple XML documents by parsing and appending nodes. Useful for combining data from multiple sources.

```
import groovy.util.XmlParser
import groovy.xml.XmlUtil

def xml1 = "1"
def xml2 = "2"

def parser = new XmlParser()
def root1 = parser.parseText(xml1)
def root2 = parser.parseText(xml2)

root2.employee.each { root1.append(it) }
println XmlUtil.serialize(root1)
```

Sorting Collections

List Sorting

Use `sort()` or custom comparators for ordering

Map Sorting

Sort by key or value using `sort { it.key }`

Use Cases

Reporting, displaying data, normalization

Sorting Example

List

```
def numbers = [5, 1, 3, 2, 4]
println "Sorted: " + numbers.sort()
```

Map by Value

```
def employees = [
 Anna: 5000,
 Ben: 4500,
 Carl: 5500
]
def sorted = employees.sort {
 a, b -> a.value <=> b.value
}
println "Sorted: $sorted"
```

Regular Expressions

Match Operators

`==~` for entire string, `=~` for partial match

Validation

Check email, phone, and other format patterns

Search

Find patterns within strings for data extraction

Data Cleaning

Remove or replace unwanted characters

Regex Example

```
def name = "GroovyScript"
println name ==~ /Groovy.*/ // starts with
println name ==~ /.*Script/ // ends with
println name =~ /.*oo.*/ // contains 'oo'

// Email validation
def email = "user@example.com"
println email ==~ /^[\w.%+-]+@[\w.-]+\.[A-Za-z]{2,}$/

// Phone validation
def phone = "+1-202-555-0173"
println phone ==~ /^\+?\d{1,3}[- ]?\(?\d{2,4}\)?[- ]?\d{3}[- ]?\d{4}$/
```

SAP CPI Integration

Groovy scripting in SAP Cloud Platform Integration for message processing and transformation.

Message Object

Body

Payload retrieved with `getBody()`

Headers

HTTP metadata and Camel routing info

Setters

Use `setHeader()` and setProperty()

Properties

Runtime values for parameterization

Message Access Example

```
import com.sap.gateway.ip.core.customdev.util.Message

def Message processData(Message message) {
   def body = message.getBody(java.io.Reader)
   def myProp = message.getProperties().get("property_name")
   def myHeader = message.getHeaders().get("header_name")

   return message
}
```

Always use null checks when accessing optional headers or properties to prevent runtime errors.

MPL Logging

MessageLogFactory

Interface to append content to MPL

Attachments

Add strings to trace-level logs

Custom Headers

Surface key data in trace summaries

Visibility

Improve debugging and supportability

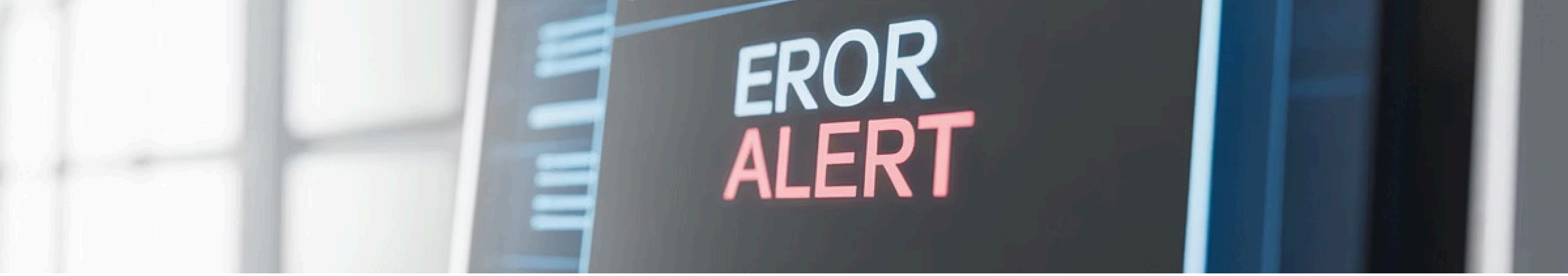
MPL Logging Example

```
import com.sap.gateway.ip.core.customdev.util.Message
import com.sap.it.api.logging.MessageLog

def Message processData(Message message) {
 def messageLog = messageLogFactory.getMessageLog(message)
 def body = message.getBody(String)

 if (messageLog != null) {
 messageLog.addAttachmentAsString("Payload", body, "text/plain")
 messageLog.addCustomHeaderProperty("MyKey", "MyValue")
 }

 return message
}
```



Throwing Errors

Interrupt Processing

Explicitly throw errors to stop flow based on custom logic

Validation

Enforce conditions that must be met before proceeding

Error Messages

Visible in CPI monitor and trace for debugging

Fallback Logic

Trigger error subprocesses or alternative paths

Error Throwing Example

```
import com.sap.gateway.ip.core.customdev.util.Message

def Message processData(Message message) {
   throw new Exception("Validation failed for input payload")
}
```

Keep exception messages readable and concise. Enrich with context to ease debugging in production environments.

Secure Parameters

1

SecureStoreService

Access credentials via ITApiFactory

2

Alias Reference

Reference by alias configured in Security Materials

3

Credential Types

`getUserCredential()` for user/password pairs

4

Security

Never log or print secure parameters

Secure Parameter Example

```
import com.sap.gateway.ip.core.customdev.util.Message
import com.sap.it.api.ITApiFactory
import com.sap.it.api.securestore.SecureStoreService

def Message processData(Message message) {
 def alias = message.getProperty("CredentialAlias")
 def secureStore = ITApiFactory.getService(SecureStoreService.class, null)
 def creds = secureStore.getUserCredential(alias)

 message.setProperty("username", creds.getUsername())
 message.setProperty("password", creds.getPassword().toString())

 return message
}
```

Value Mapping

1 **Central Definition**

Map external values to internal equivalents

2 **ValueMappingApi**

Access deployed mappings via API

3 **Reusable Logic**

Reduce hardcoding of conditionals

4 **Adaptability**

Make integrations flexible to external changes

Value Mapping Example

```
import com.sap.gateway.ip.core.customdev.util.Message
import com.sap.it.api.ITApiFactory
import com.sap.it.api.mapping.ValueMappingApi

def Message processData(Message message) {
 def vmApi = ITApiFactory.getApi(ValueMappingApi.class, null)
 def value = vmApi.getMappedValue(
 "source-agency",
 "source-id",
 "source-value",
 "target-agency",
 "target-id"
 )
 message.setProperty("mappedValue", value)
 return message
}
```

URL GET Parameters

01

CamelHttpQuery

HTTP GET parameters captured in this header

02

Parse Query String

Split on `&` and `=` to extract parameters

03

Store as Properties

Make parameters available to subsequent steps

04

Dynamic Control

Support pagination, filters, input-driven logic

GET Parameters Example

```
import com.sap.gateway.ip.core.customdev.util.Message

def Message processData(Message message) {
    def queryString = message.getHeaders().get("CamelHttpQuery")

    if (queryString) {
        queryString.split("&").each { pair ->
            def (k, v) = pair.split("=")
            message.setProperty(k, v)
        }
    }

    return message
}
```

Use URL decoding if parameter values include encoded characters. Handle null cases gracefully.

Best Practices

Null Safety

Always check for null values when accessing headers, properties, or optional data.

Security

Never log sensitive data. Use secure parameters for credentials and tokens.

Performance

Use XmlSlurper for large documents. Avoid unnecessary loops and transformations.

Testing

Test transformations thoroughly for structural and encoding correctness.

Documentation

Comment complex logic. Use descriptive variable and method names.

Error Handling

Use specific exceptions. Provide clear error messages with context.

Ready to Code!

You now have the foundation to build powerful Groovy scripts for SAP CPI integration flows. Practice these concepts and explore the extensive Groovy documentation for advanced techniques.