SAP PRESS

SAP PRESS E-Bites

Developing Groovy Scripts for SAP® Cloud Platform Integration

Vadim Klimov
Eng Swee Yeoh

Rheinwerk Publishing

Vadim Klimov, Eng Swee Yeoh

Developing Groovy Scripts for SAP® Cloud Platform Integration

Rheinwerk Publishing

What You'll Learn

Create custom integration patterns for SAP Cloud Platform Integration with Groovy! Start by getting a handle on the language before dancing your way into rapid development, from iFlow simulations to dependency management with IntelliJ IDEA. Use Groovy scripts to transform data formats and test your code before deployment. Finish your performance by exploring SAP Cloud Platform Integration's underlying frameworks and using Groovy to access and develop with them!

1 Introduction

The typical digital landscape in most organizations has evolved drastically over the past few decades, from traditional mainframe systems, to monolithic enterprise applications, to the current era of ubiquitous mobile devices, microservices, and hybrid environments. Alongside this transformation, there has been a shift in organizations from using products from a single software vendor to a more heterogeneous best-of-breed approach. It isn't uncommon for applications and services from multiple vendors to be combined within an organization's landscape to meet its unique needs and requirements.

On one hand, this allows organizations to tailor-make their digital landscapes to boost their competitive advantage. On the other hand, such a mix of systems, applications, and services has created a more complex environment from an integration perspective. Multiple systems using different protocols spread across myriad vendors require a modern, robust, and feature-rich integration platform to connect it all together seamlessly.

Enter *SAP Cloud Platform Integration*, one of the core services within the SAP Cloud Platform Integration Suite. It's SAP's de facto solution for integrating processes across organizations' landscapes comprising SAP and non-SAP solutions.

SAP Cloud Platform Integration is an integration-platform-as-a-service (iPaaS) that is designed to meet the integration needs of today's complex digital landscapes. As a service of SAP Cloud Platform, it's available to organizations in both consumption-based and subscription-based licensing models, and in both Neo and Cloud Foundry environments.

SAP Cloud Platform Integration provides a feature-rich experience across its many components. On the frontend, the web user interface (UI) is a browser-based interface used for discovery, design, and administration of integration artifacts. Interfaces are modeled as integration flows (iFlows) in a graphical-based modeling environment. A wide range of iFlow steps are available for developers to implement various enterprise integration patterns (EIPs) such as routing, message transformation, and persistence steps. SAP Cloud Platform Integration also has a wide selection of adapters that provide technical connectivity to a majority of modern protocols. Additionally, it integrates seamlessly with SAP Cloud Platform Open Connectors, leveraging its capability to connect to more than 100 third-party cloud applications in a harmonized manner.

On the runtime environment, SAP Cloud Platform Integration uses the Apache Camel integration framework and OSGi frameworks. These technologies form the backbone for running robust, enterprise-grade interfaces that power an organization's mission-critical business processes.

While the UI and the runtime environment provide a comprehensive out-of-the-box experience, SAP Cloud Platform Integration further provides a rich set of OData application programming interfaces (APIs) that allow an extension of its functionality, enabling the development of complementary third-party solutions.

In this section, we begin by looking at Groovy's position in SAP Cloud Platform Integration developments. Subsequently, we'll briefly look at the most common use cases for Groovy scripts in SAP Cloud Platform Integration. We conclude with a road map of the exciting content that we've prepared for you as you journey through this E-Bite.

1.1 Groovy in SAP Cloud Platform Integration

Of the many iFlow steps available in SAP Cloud Platform Integration, a prominent one is the *scripting* step. It allows the developer to write a script, which is a custom coding block performing functionality that isn't available in any of the out-of-the-box steps. SAP Cloud Platform Integration supports scripting in two programming languages, *Groovy* and *JavaScript*.

While the term "script" may imply that these steps can contain only basic or simple functionalities, the opposite is true. Both Groovy and JavaScript are modern, full-fledged programming languages that are also used in the development of complex, standalone applications, services, and frameworks. Therefore, the possible functionalities within such scripting steps are only limited by your creativity and imagination.

The decision to choose one programming language over another is a highly subjective one, ranging from organizational guidelines to an individual's preference and experience. While we don't discriminate against the use of JavaScript in SAP Cloud Platform Integration, our personal preference and professional recommendation is Groovy, hence this E-Bite. This recommendation is based on the following premises:

- Groovy is a Java syntax-compatible Java Virtual Machine (JVM) language and therefore integrates seamlessly with the underlying components of SAP Cloud Platform Integration because they are mostly Java-based.
- The majority of integration platforms are Java-based, for example, solutions from MuleSoft, IBM, TIBCO, and even SAP Process Orchestration. Developers coming from these platforms to SAP Cloud Platform Integration will find a smoother learning curve for their transition from Java to Groovy.

This E-Bite will enable developers new to SAP Cloud Platform Integration and/or Groovy to quickly become proficient in Groovy scripts development.

1.2 Use Cases for Groovy Script

With the changing needs of integration, an integration platform should be able to cater to requirements ranging from the simplest to the most complex. This is where the power of Groovy really shines by enabling the development of custom functionality across many different use cases.

Following are some of the common use cases for Groovy scripts in SAP Cloud Platform Integration:

- Reading and/or setting values in a message's header or property based on custom logic
- Performing custom logging and error handling
- Extending transformation in message mapping with custom functions
- Transforming messages across various data formats and message structures in either text or binary representations

The reality is that the use of Groovy is pervasive in SAP Cloud Platform Integration. A developer that is well versed in developing with Groovy will be positioned well to tackle a multitude of integration requirements.

1.3 Road Map Ahead

Over the course of this E-Bite, we'll look into many aspects of Groovy, particularly regarding developing Groovy scripts in SAP Cloud Platform Integration.

For beginners, the first few sections will focus on basics to establish a solid foundation for Groovy script development. You'll first learn the basics of Groovy in an agnostic manner, and then we'll provide you with productivity tips to enable rapid Groovy development. Subsequently, you'll learn how to develop transformations for some of the more common use cases in SAP Cloud Platform Integration.

In the later sections, we'll turn to more advanced topics that are beneficial even for experienced developers, bringing their Groovy development skills to the next level. We'll look into techniques to enable testing Groovy scripts in a local environment, which can significantly increase the speed of development and testing. We'll then explore the underlying components of SAP Cloud Platform Integration and how to unleash its capabilities using Groovy scripts. And, finally, we'll finish by providing some further resources for you to continue your journey.

Note that due to the compact nature of an E-Bite, it isn't meant to be a comprehensive exposition into all aspects of the topics covered. The main goal is to provide an overview of concepts relevant to developing Groovy scripts in SAP Cloud Platform Integration, coupled with practical examples and productivity tips.

We encourage you to try out the many examples found throughout this E-Bite and even enter some of the code listings manually instead of copying and pasting them. This is especially true when working with IntelliJ IDEA as the local development environment, where you can really experience its power and ease of use. So, experiment, learn, have fun, and, even if you encounter difficulties, try to resolve them as part and parcel of the learning process. Don't worry if you aren't able to resolve them: a complementary package containing all the main examples in working order accompanies this E-Bite.

Note about Versions

In the fast-changing world of software, today's version of a software component could easily be out of date tomorrow. With this in mind, it's likely that some of the software components mentioned in this E-Bite will have newer versions compared to the versions at the time of writing. Table 1.1 lists the key software components that you'll encounter throughout this E-Bite and their corresponding versions at the time of writing.

Software Component	Version
Java Development Kit (JDK)	1.8.0
Groovy Development Kit (GDK)	2.4.12
IntelliJ IDEA Community Edition	2020.1
SAP Cloud Platform Integration Script API	1.36.1
Apache Camel	2.17.4
Spock Framework	1.3-groovy-2.4

Table 1.1 Versions of Key Software Components at the Time of Writing

2 Basics of Groovy

This section describes some fundamental practices of developing Groovy scripts that are found in the vast majority of scripts developed for SAP Cloud Platform Integration. Good understanding of these practices is essential for efficient Groovy development.

The section starts from the de facto example—the Hello World script—that illustrates key parts of the basic Groovy script in SAP Cloud Platform Integration. After that, we'll set SAP Cloud Platform Integration aside and focus on pure SAP Cloud Platform Integration–agnostic Groovy development, where we highlight the most prominent Groovy language aspects. The remaining sections address operations with the most commonly used and commonly faced objects, that is, strings and collections. We also devote attention to using regular expressions in the context of Groovy development and focus on shortcuts that Groovy provides for operations with search patterns. Fluent usage of these objects is a prerequisite for subsequent sections of this E-Bite.

2.1 Hello Groovy World

As this E-Bite is about Groovy development in the SAP Cloud Platform Integration context, the very basic and frequently used introductory Hello World example will take the shape of a Groovy script that is added to the iFlow. To keep the example simple for now, the SAP Cloud Platform Integration web UI is used to perform all development activities.

This E-Bite doesn't go into details of usage of the SAP Cloud Platform Integration web UI, as we assume you're familiar with the tool and its main UI components. You also can consult the corresponding SAP Cloud Platform Integration product documentation at the SAP Help Portal (https://help.sap.com/viewer/product/CLOUD_INTEGRATION/Cloud/en-US/).

Integration Flow Used in the Hello World Example

To create the iFlow used in our example, follow these steps:

1. Access the SAP Cloud Platform Integration web UI using a web browser. For this, use the SAP Cloud Platform Integration tenant URL (the URL ends with */itspaces*) in the web browser.
2. Navigate to the **Design** view (the **Design** section that can be accessed from the menu on the left in the web UI).
3. Create a new content package, or go to an existing package.
4. Create a new iFlow in the package.
5. In the integration flow editor, switch to edit mode, and configure the HTTPS sender adapter channel so that the iFlow can be invoked by sending a message over HTTPS.
6. Add the Groovy Script step.

After these steps are complete, the iFlow will look similar to the one shown in Figure 2.1.

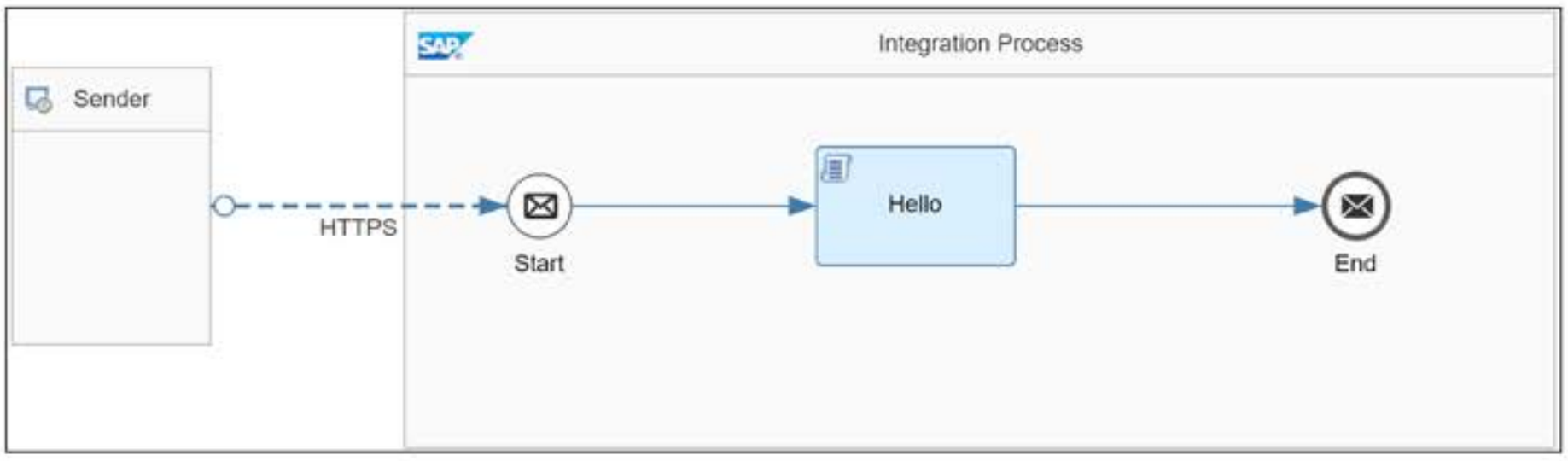


Figure 2.1 iFlow Used in the Hello World Example

Warning on CSRF Protection

By default, the HTTPS sender adapter of SAP Cloud Platform Integration enables Cross-Site Request Forgery (CSRF) protection. CSRF attacks commonly target state changing requests, such as those sent using HTTP methods `POST`, `PUT`, `PATCH`, and `DELETE`. With that in mind, it's strongly encouraged to consider enforcement of CSRF protection for production-grade integration scenarios. Invocation of a CSRF-protected end point requires a valid CSRF token to be preemptively fetched by the caller and presented in the corresponding HTTP header field (`X-CSRF-Token`) of such requests.

In this example, the HTTP method `POST` is used when sending requests to the iFlow and testing the described technique. To simplify the illustration, we deliberately disabled CSRF protection of incoming requests by clearing the **CSRF Protected** checkbox on the **Connection** tab of the HTTPS sender adapter channel configuration in the integration flow editor.

Note on Authentication Method

In the described iFlow and in other iFlows that will be introduced later in the E-Bite, for simplicity's sake, we use the basic authentication method in the HTTPS sender adapter channel configuration. Thus, a valid user name and password must be provided in the corresponding HTTP header field (*Authorization*) of requests that are sent to such an iFlow for requests to be authenticated and authorized successfully by SAP Cloud Platform Integration and for the iFlow to get triggered.

Next, create a new script and maintain the script function that is illustrated in Figure 2.2.

Figure 2.2 Hello World Groovy Script Function

The script function reads a request message body that is expected to hold a user name and then produces a response message whose payload contains the welcome message for the submitted user name, that is, the text `Hello, {username}`. For simplicity, the script function doesn't implement validation of the input message payload; for example, it doesn't check the payload for being null or empty, and it expects that the user name is always submitted in the request message payload in plain text.

After the iFlow has been saved and deployed to the runtime, the Hello World example can be tested by sending the HTTP `POST` request to the end point of the iFlow. The end point that has been registered for the deployed iFlow can be found from the **Operations** view (**Monitor** section) of the SAP Cloud Platform Integration web UI.

Throughout the E-Bite, we use Postman (https://www.postman.com/) as an HTTP client. The tool is widely used in HTTP API testing and provides comprehensive functionality for HTTP request creation and documentation. Any other relevant HTTP client tools—tools with command-line interfaces (CLIs) and graphical UIs—can be used instead of Postman as well. A sample request message and the corresponding response message that was returned by the iFlow to Postman are shown in Figure 2.3.

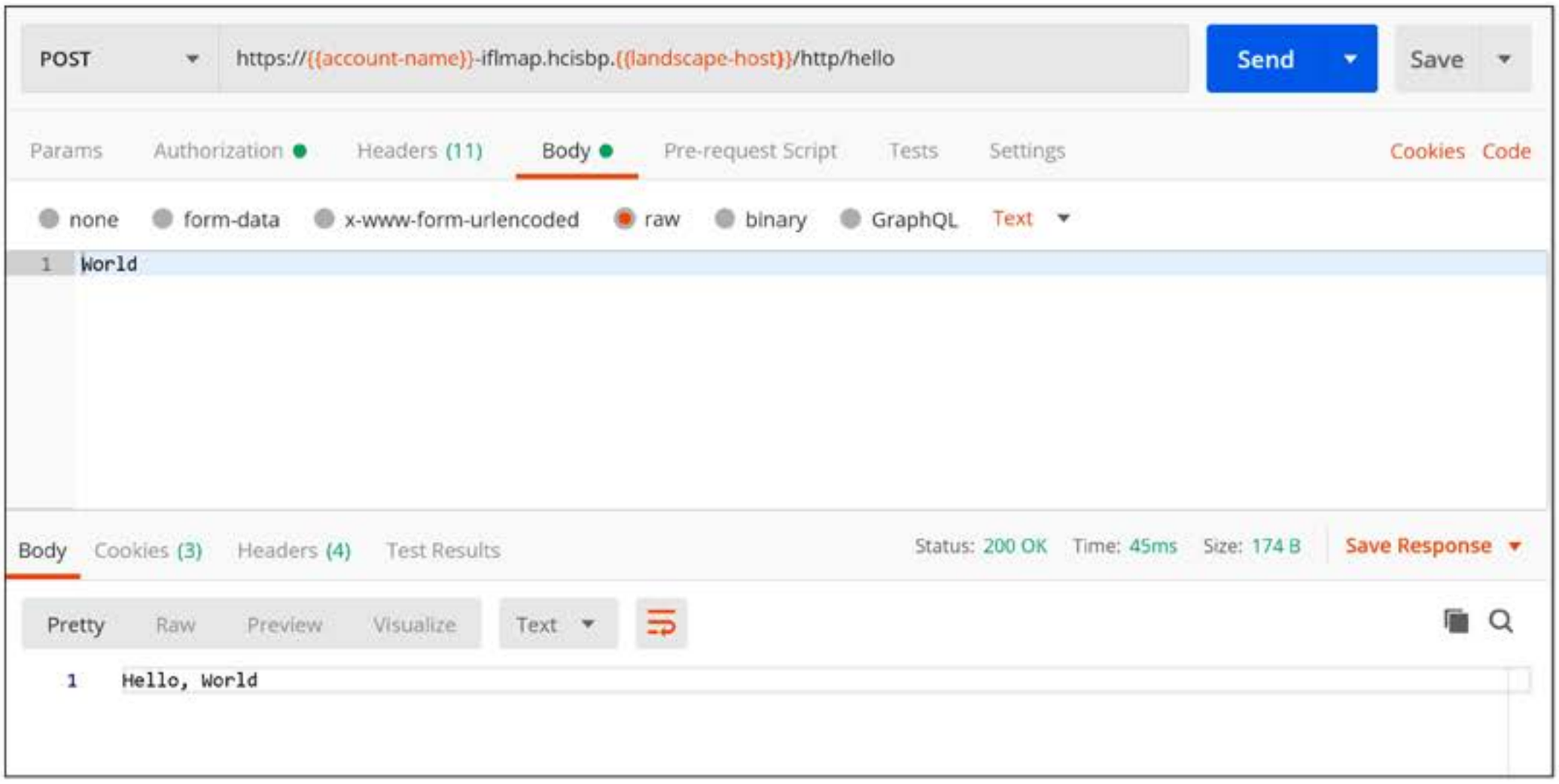


Figure 2.3 Hello World Request and Response Example Using Postman

In spite of its simplicity, the developed Groovy script function used in the Hello World example is a valuable starting point for our subsequent exploration.

Anatomy of the Groovy Script Function in SAP Cloud Platform Integration

Let's take a quick look at key ingredients of the *script function declaration*:

- **Function name**
  By default, the name is `processData`, but it can be changed to any valid name (the name will conform to requirements for Java identifiers) to better reflect the purpose of the function. If the function name isn't the default, the custom function name has to be specified in the Groovy script step configuration (in the **Script Function** field on the **Processing** tab), so that the script processor becomes aware of which function will be invoked at runtime.
- **Function parameters**
  The only parameter that is passed to the function is the instance of

`com.sap.gateway.ip.core.customdev.util.Message`. This is the representation of the input message of the function that is further consumed and used from within the function. Each message has a body (a payload), headers, and optional attachments that can be accessed and modified by the function.

- **Function return type**
  The function has to return the instance of `com.sap.gateway.ip.core.customdev.util.Message`. This is the representation of the output message of the function, a result of transformations that the function has performed with the input message. In the SAP standard template, the function definition uses `def Message` to define the return type. Strictly speaking, simultaneous use of dynamic typing (the keyword `def`) and static typing (`Message`) in the return type definition is redundant, but to stay consistent with the SAP documentation, the observed style of the return type declaration is used throughout this E-Bite.

The *script function implementation* is a combination of statements that form the function body and implement logic that is executed when the function is invoked. In the Hello World example, the function implements the following sequence of actions:

1. Access a message body (of the input message) and convert it into a `String` by calling the `getBody(Class)` method of the `message` object.
2. Set the augmented message body (for the output message) that contains a text—concatenation of *Hello* and the retrieved user name—by calling the `setBody(Object)` method of the `message` object.
3. Return the augmented message, so that it can be used by subsequent message flow steps (in any) or returned by the iFlow.

Tip

In the SAP documentation, the recommendation for using the `getBody(Class)` method of the `message` object is to pass a fully qualified class name to the input parameter, for example, `getBody(java.lang.String)`. In this E-Bite, we deviate

from the recommendation and use just the simple class name (e.g., `getBody(String)`) to achieve a more compact format. Following are the reasons behind this approach:

- An `import` statement should be used instead of using inline fully qualified class names.
- Certain packages are imported in Groovy by default, so an explicit `import` statement isn't necessary for them. Some examples are `java.lang.*`, `java.io.*`, `java.util.*`, `groovy.lang.*`, and `groovy.util.*`.

You might also come across the following examples of the code in the documentation:

```
def body = message.getBody(java.lang.String) as String
```

The use of Groovy coercion (e.g., `as String`) is redundant here and can be omitted because the return type of `message.getBody(String)` is already a `String` object due to the underlying type conversion.

Camel, which is the integration framework used by SAP Cloud Platform Integration, is payload agnostic, meaning it supports any type of data that can be stored in the message payload. In the Hello World example, we use plain text in the payload of both input and output messages and convert it into a plain string object. In later sections, you'll get to know different types of strings that are supported in Groovy besides plain strings. In Section 4, for example, you'll learn about some of the most widely used data formats (based both on streams of characters and bytes) and how transformations can be performed with them using Groovy scripts. Transformations of the message in general and the message payload in particular are key areas where Groovy scripts are often used in SAP Cloud Platform Integration. This knowledge will be complemented in Section 6 with the overview of the Camel type conversion system and how some types can be converted into another using native capabilities of the Camel framework.

Similar to getting and setting a message body, it's also possible to manipulate (read and write) message headers and attachments using corresponding methods that are provided by the `Message` interface. There are also some additional objects that aren't a part of the message but are closely

associated with the message and its processing, and they can be accessed by the function, for example, exchange properties, message processing log (MPL) of the message that is used to store MPL properties, and MPL attachments for the corresponding message. In Section 6, you'll learn how you can access some other useful objects that aren't accessible using the public API of the `Message` interface, such as a Camel exchange and CamelContext (a part of the Camel integration framework).

Implementation of the script function entirely depends on the function's purpose and can use SAP Cloud Platform Integration APIs and third-party libraries. The complexity of the function can also vary, but it's highly recommended to consider the single responsibility principle and modularization when designing and developing functions so that the complexity of each function individually remains within reasonable limits. It's important to keep function implementation consistent, clear, and concise while remaining mindful of relevant error and exception handling, performance and resources consumption (e.g., memory footprint, CPU intensiveness, operations that require I/O resources management), and security aspects.

Groovy Console

The preceding Hello World example is the only one in this section to make use of SAP Cloud Platform Integration. As topics covered later in this section are generic and aren't specific to SAP Cloud Platform Integration, and we won't develop entire script functions, we'll evaluate Groovy expressions and execute Groovy statements and scripts locally to simplify things. The Groovy library distribution includes two tools to help with that: the Groovy Shell CLI tool, and the Groovy Console GUI tool.

You need to set up and prepare the local environment first by downloading and installing two components:

- **Java Development Kit (JDK)**
  Download Oracle JDK 1.8.0 that implements Java 8 to stay in sync with the Java version that is used in SAP Cloud Platform Integration (https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html).

In Section 3, you'll see how to identify Groovy and Java versions that are currently used by SAP Cloud Platform Integration.

- **Groovy binaries bundle**
  Download the Groovy binaries bundle from the Groovy site (https://groovy.apache.org/download.html). You must use the distribution type that includes Groovy binaries, such as the Groovy binary distribution (contains only binaries) or the complete Groovy SDK bundle (contains binaries, source code, and documentation).

To test that the local Groovy environment is set up and ready to use, navigate to the directory that contains Groovy binaries (alternatively, add the corresponding path to the `PATH` environment variable), and execute the command `groovyConsole` in the terminal. You'll see the main window of the Groovy Console: Groovy statements are entered in the input area (top part of the main window in the vertical orientation; left in the horizontal orientation), and after they are executed, the result of their execution appears in the output area (bottom part of the main window in vertical orientation; right in horizontal orientation), as illustrated in Figure 2.4.

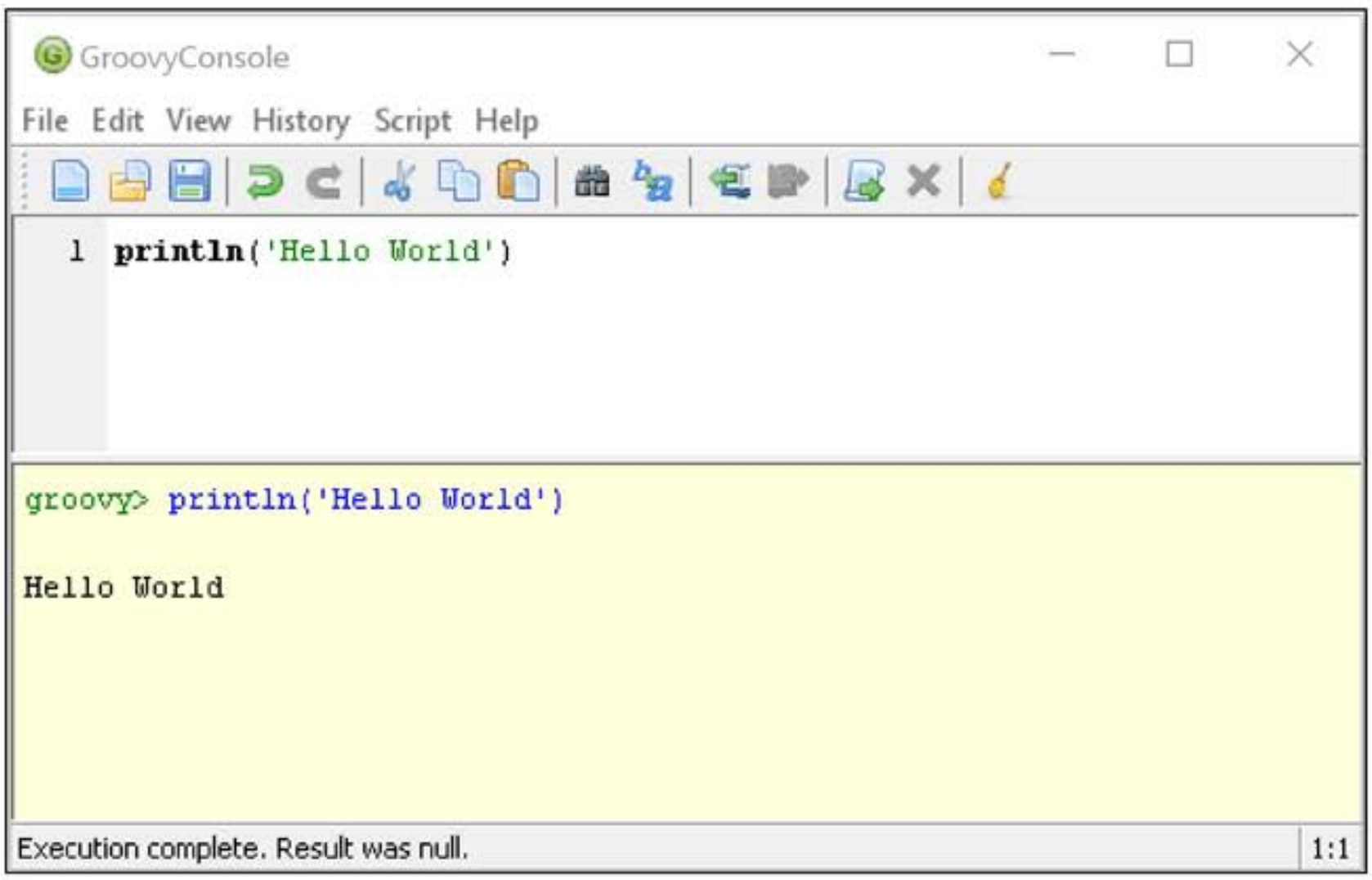


Figure 2.4 Groovy Console Main Window

Alternatively, you can use the online Groovy web console at https://groovy-console.appspot.com/. In this case, there is no need to install any components on a local machine, and all examples can be executed purely using the web browser.

In following sections, you'll learn about the more advanced tools to develop and test Groovy scripts. Although those tools provide integrated access to the Groovy Console, we'll use the standalone Groovy Console illustrated in Figure 2.4 until you've been introduced to those tools.

2.2 Groovy Style Guide

The Groovy language originated from and was influenced by the Java language (as well as by a few other languages, e.g., Python, Ruby, and Smalltalk), so Groovy integrates smoothly with the Java ecosystem. As a result, Groovy has a shallow learning curve and is relatively easy to learn for Java developers; in fact, a large number of Groovy developers tend to have prior Java development experience. The high level of interoperability with Java that Groovy offers makes it possible for Java developers to experience soft onboarding, starting their journey in the Groovy world from pure Java style development and using already-familiar Java APIs, followed by progressively moving toward more concise and "Groovier" syntax and APIs.

The peril of such gradual transition is a fuzzy boundary that the developer might experience between Java and Groovy at the beginning, which can tempt developers to use Java style when developing Groovy applications and scripts and consequently not to use the full extent of Groovy's power. For this reason, we recommend any Java developer to become familiar with the Groovy style guide (https://groovy-lang.org/style-guide.html) as it consolidates most prominent features of the Groovy language and looks at those features from a Java perspective. Just as the Groovy language specification defines language syntax and semantics, the Groovy style guide features the most remarkable and substantial differences between Java and Groovy development styles.

Although the Groovy style guide is an invaluable reference, approach it sensibly. Organizations or developer teams might want to define their own set of stylistic rules and principles based on the Groovy style guide without following it unequivocally. This is an acceptable alternative as long as the style includes considerations for best practices and doesn't conflict with them, and the style is consistent, clear, and natural to developers and maintainers of the developed code base.

Following are several rules defined in the Groovy style guide that are worth applying to practice or at least considering when developing in Groovy:

- Semicolons used to denote the end of the statement in Java can be omitted in Groovy.
- Classes are first-class citizens; as a result, the `.class` suffix can be omitted in expressions. For example, the expression `message.getBody(String)` was used in the earlier example instead of a lengthier Java-style equivalent `message.getBody(String.class)`.
- Both *static (strict) typing* (e.g., `String text = 'Hello, World'`) and *dynamic (optional) typing* (e.g., `def text = 'Hello, World'`) are supported by Groovy, so developers can decide which style (or even a combination of both) is more appropriate for them. For better source code readability and coherence, use static typing for APIs that are public or intended to be shared and used across different modules and developments, as this will help to ensure that the API contract is strict and clear, reducing the probability of incorrect types usage when invoking corresponding exposed APIs.
- *Closures* exemplify functional programming style and represent an anonymous block of code that can take input parameters (arguments) and return a value that can be assigned to a variable. When using closures, closure parameters can be defined in two ways: explicitly and implicitly. Consider the definition of the explicit parameter list (using the *arrow operator* (->)), especially when defining multiple parameters to make their usage in closure statements clearer and more readable. In simpler cases, it's sufficient to use an implicit parameter; when the closure doesn't define explicit parameters, it always defines an implicit parameter named `it` that can be used in closure statements.

- The *equality operator* (==) in Groovy acts similarly to `Object.equals(Object)` in Java and isn't the same as the equality operator in Java. In addition, the equality operator in Groovy also checks whether compared objects are `null` or not, so its usage eliminates the need for the explicit check of objects for null, which helps to avoid `NullPointerException` during comparison operations. As a result, the comparison of objects (e.g., string objects) in Groovy can be expressed in a more elegant way.
- The *Groovy truth* (or Boolean resolution) has broader coverage as compared to its equivalent in Java because Groovy supports evaluation of a larger range of expression types in Boolean tests, as all objects in Groovy can be coerced to a Boolean value. Knowledge of what expressions evaluate to true in Boolean tests can help in composing more compact expressions that are used in various checks, such as in conditions in `if` or `switch` statements, or when working with regular expressions. For example, a frequently used expression to check if the string object isn't null and isn't empty—`if (text !=null && text.length() > 0)`—can be compacted in Groovy to `if (text)`. Moreover, Groovy truth behavior can be customized via the implementation of the `asBoolean()` method of objects involved in the Boolean test.
- Many operations with data structures, such as with collections and maps, are natively supported by Groovy. You'll see some of them in action later in this section to learn how their usage can make source code more compact.

The overwhelming majority of examples and code listings that you'll see in this E-Bite deviate from the Groovy style guide in the following aspects:

- The `return` keyword is used to emphasize the object that is returned by the function.
- Parentheses aren't omitted for top-level expressions to retain the consistent look of method calls.
- Getter and setter methods are used over field-like access notation to emphasize access to properties.

2.3 Groovy String

The *string* is the fundamental and essential object that is almost inevitable in any development, and Groovy scripts are no exception. Strings are used to represent sequences of characters (or text literals) and are frequently seen when working with a variety of objects. This section provides a brief overview of some techniques and operations that are applicable to strings in the context of Groovy development, starting from creating basic string literals and progressing toward manipulation with string content and usage of more advanced string representations that can help make operations with strings more compact thanks to some Groovy shortcuts. Because strings are among the most frequently used objects in any Groovy development, it's important to have not only a solid knowledge of operations with strings but also their performance aspects. Therefore, we'll share some performance tips to consider and pitfalls to avoid when working with strings at scale.

String Classes

Groovy provides two core classes that are used to manage string literals:

- **Plain string: `java.lang.String`**
  The class is inherited from Java and represents character strings. Besides Java-native methods, string objects are extended with additional methods to simplify manipulations with the string literal value and type coercion of string objects to other relevant object types in the Groovy environment (corresponding additional methods are defined in the class `org.codehaus.groovy.runtime.StringGroovyMethods`).
- **Interpolated string: `groovy.lang.GString`**
  Also known as a *Groovy string*, the class has been introduced in Groovy to enrich string capabilities with advanced features, namely, capability to embed placeholders (expressions) into the string value. Expressions are evaluated and placeholders are replaced with corresponding values at runtime.

When assigning a value to a plain string, the assigned string literal will be surrounded by single quotation marks, as follows:

```
def text = 'Hello, World'
```

Groovy also supports multiline plain strings. Triple single quotation marks are used to surround such string literals, as follows:

```
def text = '''Hello,
World'''
```

When working with interpolated strings, a string literal will be surrounded by double quotation marks (for multiline interpolated strings, triple double quotation marks are used), and embedded expressions will be preceded with the dollar sign ($) and surrounded by curly braces ({}) (for unambiguous dotted expressions, curly braces are optional and can be omitted). There is an alternative notation for interpolated strings called *slashy strings*, which is more commonly seen when working with search patterns in regular expressions because of an additional escaping feature that is provided in that notation. You'll learn more about slashy strings in the next section when we look at regular expressions.

Not all string literals surrounded by double quotation marks are Groovy strings. For dynamic typing, the Groovy runtime recognizes whether the string literal contains expressions and requires interpolation or not; if the literal is a plain string and doesn't contain any expressions, even if it's surrounded by double quotation marks, the corresponding object will be initialized as `String`, not `GString`.

Placeholders Supported in Interpolated Strings

Groovy supports a variety of placeholders that can be used in interpolated strings. The following are most commonly encountered in Groovy developments:

- Expressions to access objects and their properties, as follows:

  ```
  def world = 'World'
  def text = "Hello, ${world}"
  ```

If you execute these statements in the Groovy Console, you'll get the `Hello, World` output.

- Expressions to invoke methods or chains of methods (both instance methods and static methods are supported), as follows:

  ```
  def text = "Hello, ${System.getProperty('user.name')}"
  ```

  The Groovy Console result will look similar to the following (the displayed user name will be different and will reflect the current user): `Hello, CPI`.

Using interpolated strings makes code more concise, and classic string concatenation techniques can be replaced with more compact expressions. For example, the following string concatenation statement follows Java style:

```
String world = "World"
String text = "Hello, " + world
```

It can be transformed into the following in Groovy style:

```
def world = 'World'
def text = "Hello, ${world}"
```

The example can be evolved and developed, showing that Groovy excels in—among other things—simplification of operations with strings, which also is reflected in template engines' capabilities that use interpolated string features heavily.

Although there is no technical restriction on the complexity of expressions that can appear in placeholders, placeholders in interpolated strings should be kept simple for better readability and maintainability.

We've skipped describing and demonstrating some other common and basic operations that can be performed with strings, such as string split and substring, find, replace, and string compare operations, because we assume you'll already know those if you have prior development experience in Java or Groovy. If that isn't the case, we encourage you to become familiar with the documentation for `String` and `GString` classes and practice with methods that are exposed for them.

Performance Tips

Performance is an important aspect that will be taken into consideration when working with strings, especially when operating with character sequences and transforming them at scale (e.g., when processing large payloads).

String objects—unlike Groovy strings—are immutable, which means that the string object doesn't change its internal state once initialized. Because of this trait, you should avoid using string objects for frequently changed string literals; instead, use corresponding `StringBuilder` and `StringBuffer` classes. Let's take a look at the example in Listing 2.1.

```
def text = ''
10.times { text += "Line ${it}\r\n" }
println(text)
```

Listing 2.1 Basic Example of String Literals Concatenation Using the String Built-In Functionality

This implementation is suboptimal from a performance perspective, as it causes the creation of several string objects in memory that are consecutively concatenated. A more efficient implementation replaces frequent string concatenation with the builder and avoids creation of multiple intermediate string objects, as shown in Listing 2.2.

```
StringBuilder sb = new StringBuilder()
10.times { sb.append("Line ${it}\r\n") }
def text = sb.toString()
```

Listing 2.2 More Efficient Example of String Literals Concatenation Using Intermediate StringBuilder

In the context of scripts development for SAP Cloud Platform Integration, this pattern can be applied when constructing the output message payload in an iterative manner and composing it from smaller pieces that are assembled together. The intermediate character sequence operations in the transformation can use a `StringBuilder` object whose string representation is only retrieved on a final stage.

Tip

As evident from the Groovy language documentation, Groovy syntax offers more concise alternative expressions for commonly used operations. For example, instead of the Java style `StringBuilder.append(Object)` statement, we could have used the *left shift operator* (`<<`) to append values to the `StringBuilder` object, that is, replacing `sb.append("Line ${it}\r\n")` with `sb << "Line ${it}\r\n"` in Listing 2.2. This would have made the code more compact and retained its readability. Such convenient shortcuts can be found in multiple areas and are worth applying in practice.

2.4 Regular Expressions

Regular expressions (*regex*) are frequently encountered when certain search patterns must be applied to character sequences. While the subjects of regular expressions and their syntax, as well as implementations of regular expressions engines in a broader sense, are beyond the scope of this E-Bite, we'll take a glance at how Groovy can simplify operations that involve regular expressions. Groovy capabilities regarding regular expressions are primarily based on the Java Regular Expressions API (the package `java.util.regex`) but are extended with a number of shortcuts and additional operators that make operations with regular expressions and their usage more convenient because they are supported by Groovy at the language level.

Let's illustrate some concepts and techniques of working with regular expressions on the example of phone number validation. Although different countries use various conventions to record phone numbers, an international standard called ITU-T E.164 is generally followed by countries and organizations. According to this standard, the international phone number is composed of a 1- to 3-digit country code prefixed with the plus sign (+), and followed by the national (significant) number. Overall length of the phone number, excluding the plus sign, can be up to 15 digits and is normally not fewer than 7 digits. It's also possible to see single space characters in phone numbers that are used to delimit certain numeric blocks such as

national destination codes. Examples of valid international phone numbers are +6012 345 6789, +7 812 1234567, +48 12 345 67 89, and +44 1223 123456.

In contrast, following are examples of invalid international phone numbers:

- +6012 ABC 6789 (letters aren't permitted)
- +7(812)123-45-67 (brackets and dashes aren't permitted)
- +48 12 345 67 89 12345 (length of the phone number can't exceed 15 characters)
- 44 1223 123456 (country code isn't prefixed with the plus sign)

The corresponding code snippet is shown in Listing 2.3, and we'll go through its most significant parts directly after.

```
def phoneNumber = '+44 1223 123456'
def pattern = ~/^\+(?:\d\u0020?){6,14}\d$/
def matcher = pattern.matcher(phoneNumber)
println(matcher.matches())
```

Listing 2.3 Regular Expression to Validate Phone Numbers at Work

From this code, you'll get the result `true` in the console, which indicates that the submitted phone number conforms to the international phone number notation. If you now replace the value of the variable `phoneNumber` with one of the invalid examples, such as +6012 ABC 6789 or +7(812)123-45-67, and re-execute this code snippet, you'll get `false`— as the result because the submitted input isn't a valid international phone number anymore.

A search pattern lies at the heart of any regular expression. In Groovy, the `Pattern` object can be created using the *pattern operator* (~). The pattern operator is commonly used in conjunction with a *slashy string*, that is, a string literal surrounded with opening and closing forward slashes (/). A slashy string can be thought of as another notation for an interpolated string that not only supports features and properties of the interpolated string discussed in the previous section but also eliminates the need for an explicit escaping of backslashes contained in the string literal. This means

you can still use placeholders in slashy strings in a similar way as described earlier with interpolated strings.

An example illustrated previously in Listing 2.3 creates a pattern that is used further to perform validation. A brief decomposition of the pattern and a summary of its integral parts are provided in Table 2.1.

Regular Expression Part	Description
`^\+`	Begin with the plus sign (+).
`(?:`	Beginning of noncapturing group.
`\d`	Match digits. Here, the JDK predefined character class for digits `[0-9]` is used, `\d`.
`\u0020?`	Match optional single space character. Here, the space character is specified with reference to its Unicode code point, `\u0020`.
`){6,14}`	End of noncapturing group, which can be repeated between 6 and 14 times (6 to 14 occurrences are expected).
`\d$`	End with the digit.

Table 2.1 Decomposition of a Regular Expression to Validate Phone Numbers

Tip

The JDK documentation for the `java.util.regex.Pattern` class contains comprehensive reference on syntax that can be used when building regular expression patterns in Java/Groovy.

After the `Pattern` object has been created, you create the corresponding `Matcher` object and use it to validate the submitted input string.

Groovy provides several shortcuts that can make operations with regular expressions more concise, such as the *find operator* (`=~`) and the *match operator* (`==~`). Let's see how you can make the example more compact using the latter of these:

```
def phoneNumber = '+44 1223 123456'
println(phoneNumber ==~ /^\+(?:\d\u0020?){6,14}\d$/)
```

Note how three statements—creation of `Pattern` and `Matcher` objects and the subsequent `Matcher.matches()` call—were combined into the single statement using the match operator that is used to examine the input string for a strict match against the pattern.

The regular expressions technique is a very powerful and flexible instrument in the developer's toolbox, and you'll come across it in many search-related applications. For example, we'll get back to regular expressions in the next section, and you'll see how they can be put in context of collections and used when performing filter operations over elements of collections.

2.5 Working with Collections

Collections—representations of groups of homogeneous objects—are essential elements that are encountered in almost every Groovy script in SAP Cloud Platform Integration, reading elements from various groups or creating new groups of elements are common requirements. Collections in Groovy are based on the Java collections framework, which consists of interfaces that define abstractions for collection types, classes that provide concrete implementations of corresponding interfaces, and algorithms that provide methods to access and manipulate elements in collections. The collections framework includes several interfaces, among which, the most commonly used in Groovy scripts in SAP Cloud Platform Integration are the following:

- `java.util.List`
  This ordered collection allows duplicate elements.
- `java.util.Set`
  This collection contains no duplicate elements.

Groovy complements the Java Collections API with additional methods and techniques that simplify initialization and manipulations with collections. Let's look at some commonly used methods that are part of the JDK and enhancements that Groovy offers on top of them to work with collections.

Initializing and Accessing Lists

To create a new collection (e.g., a new `List`) and put some elements into it during the initialization, you can use a Java style notations, or you can use Groovy style to make the declaration and initialization more concise:

```
def languages = ['Java', 'JavaScript', 'Groovy']
```

Obviously, you don't necessarily need to put elements into the collection at its initialization; you can declare the collection and put some values into it at a later time. Moreover, Groovy provides multiple ways to express that operation, as well as a number of ways to access list elements by their index. Both are shown in Listing 2.4.

```
def languages  = []

languages.push('Java')
languages.add('JavaScript')
languages << 'Groovy'

println(languages.get(2))
println(languages[2])
```

Listing 2.4 Techniques to Add Elements to a Collection and Access Them

The latter statements reflecting printing accessed elements of the collection will lead to the same result in the console: `Groovy`.

Groovy introduces some additional features for collections that can turn into helpful shortcuts. For example, it's possible to invoke a method or access a property on all elements of the collection using the *spread-dot operator* (`*.`). Let's use the earlier illustrated initialized collection that was introduced in Listing 2.4 and print all string literals of elements of the collection in uppercase:

```
languages*.toUpperCase()
```

The following appears as the result in the console: `[JAVA, JAVASCRIPT, GROOVY]`.

Note that string literals in Groovy can be considered as special kinds of collections; correspondingly, characters within string literals can be accessed similar to elements in the collection:

```
def text = 'Hello, World'
text[7..11]
```

The console result will contain "elements" in the 8th to 12th position of the given string literal (note that positional access to collection elements is zero based; i.e., the first element in the collection has an index of 0): `World`.

Initializing and Accessing Maps

Another important interface is `java.util.Map`, which allows for the grouping of key-value pairs (or mapping of keys to values) because a map can't contain duplicate keys. Strictly speaking, a map isn't a true collection, but it provides collection views that make it possible to manipulate the contents of maps as collections, in particular, `Map.entrySet()` to get a set view of key-value pairs contained in the map, `Map.keySet()` to get a set view of keys contained in the map, and `Map.values()` to get a collection view of values contained in the map.

You can define a map and put elements into it during initialization, as follows:

```
def languages = [java: 'Java', js: 'JavaScript', gsh: 'Groovy']
```

You can also declare the map, put some elements into it using separate statements in multiple ways, and access the map elements using different techniques, as illustrated in Listing 2.5.

```
def languages  = [:]

languages.put('java', 'Java')
languages.js = 'JavaScript'
languages << [gsh: 'Groovy']
```

```
println(languages.get('gsh'))
println(languages['gsh'])
println(languages.gsh)
```

Listing 2.5 Techniques to Add Elements to a Map and Access Them

All three alternative ways to access elements of the map will produce the same output to the console once executed: `Groovy`.

Java provides a comprehensive number of methods to access, search, and manipulate elements in collections and maps, and Groovy enhances and extends those methods with additional methods that make operations with collections and maps even more feature-rich and comfortable. We strongly recommend that you become familiar with the corresponding classes and methods provided by the GDK for collections and maps.

Iterating Over Collections

Iterating over collections is commonly required not only to access specific elements of the collection but also to filter elements, search for them, and execute certain operations over them.

You can iterate over a collection in multiple ways, for example, the classic Java *for loop* and its more modern *enhanced for loop* (*for-each loop*) version introduced in Java 5 (both are based on using a `for` statement), `Iterator` object and `Iterable.forEach(Consumer)` method introduced in Java 8, Stream API that exemplifies functional programming style and also introduced in Java 8, and Groovy-specific shortcut `Iterable.each(Closure)` based on using closures that adheres to the functional programming paradigm. The optimal techniques that naturally fit in the Groovy philosophy are those based on using closures and the Stream API (as you'll see, these two techniques can even be combined), as they provide a compact, elegant method of iteration and conform to modern development styles in general.

Let's now use the list created earlier in this section and apply those selected iteration techniques to practice:

```
def languages = ['Java', 'JavaScript', 'Groovy']
```

To start with, list the list entries. In this example, you use an enhanced version of `Iterable.each(Closure)` that provides access to indices of collection elements and uses explicit closure parameters:

```
languages.eachWithIndex { language, index ->
  println("${index}. ${language}")
}
```

In the console output, you'll get a numbered enumeration of elements contained in the list:

```
0. Java
1. JavaScript
2. Groovy
```

Next, let's see how you can apply iteration techniques in the context of solving a specific task, in this case, performing a search operation for all languages (elements in the collection) that start with *Java* and printing the corresponding found language names in uppercase and pipe-delimited to the console. You can implement the operation of searching for required elements and placing them into a new list that contains search results in multiple ways (note that the following examples are using implicit closure parameters):

- **Basic form:**
  ```
  def result = languages.findAll { it.startsWith('Java') }
  ```
- **Alternative using a regular expression:**
  ```
  def result = languages.findAll { it ==~ /^Java.*/ }
  ```
- **Alternative using the `grep(Object)` method instead of `findAll(Closure)`:**
  ```
  def result = languages.grep(~/^Java.*/)
  ```

Finally, search results are uppercased and pipe-delimited while being joined into a string result:

```
result*.toUpperCase().join('|')
```

In all three cases, regardless of the chosen implementation approach, you'll get the same result in the console: `JAVA|JAVASCRIPT`.

While the Collections API primarily focuses on data structures and techniques to access that data, the Stream API is focused on operations and computations that can be performed on data. Stream on its own doesn't store data; instead, it gets data from the data source (collection being one such data source) and applies operations that form a stream pipeline to a sequence of elements obtained from the data source.

Let's implement the same search logic as illustrated earlier, but this time, using the Stream API. In the example illustrated in Listing 2.6, the sequential stream is used.

```
import java.util.stream.Collectors
def languages = ['Java', 'JavaScript', 'Groovy']
def result = languages.stream()
                      .filter { it.startsWith('Java') }
                      .map { it.toUpperCase() }
                      .collect(Collectors.joining('|'))
```

Listing 2.6 Usage of the Stream API for Iteration over the Collection and Searching for Elements

Similar to earlier run examples, you'll get the same result in the console this time: `JAVA|JAVASCRIPT`.

3 Rapid Groovy Development

Often, when using Groovy Script steps in developed iFlows in SAP Cloud Platform Integration, the following steps are carried out:

1. Create script files in the iFlow, and implement required functions or import existing script files to the iFlow as static resources.
2. Use them in Groovy script steps.
3. Save the iFlow at design time.
4. Deploy the iFlow to the runtime.
5. Trigger the iFlow, and test the script function.

Subsequent changes to the script function require code corrections in created/imported script files and redeployment of the iFlow. This process involves additional manual efforts and takes time, and it has to be repeated every time an amendment—even a minor one—needs to be introduced to the developed and tested script function.

In this section, you'll learn the iFlow simulation tool technique for testing Groovy scripts in SAP Cloud Platform Integration. This is a built-in functionality of SAP Cloud Platform Integration that enables rapid testing of message processing steps at design time even before the iFlow gets deployed to the runtime. The described technique can significantly improve the developer experience and reduce the time required for Groovy script debugging and iterative script corrections.

Later in the section, we illustrate how a local integrated development environment (IDE) can be used for development of Groovy scripts through the example of IntelliJ IDEA and how this can boost developer productivity. Because scripts commonly use functionality provided by external libraries, we'll pay special attention to dependencies management in IntelliJ IDEA.

3.1 Integration Flow Simulation Tool

The *iFlow simulation tool* is a part of the standard functionality of SAP Cloud Platform Integration and enables integration developers to test the iFlow, a subset of the iFlow, or even the specific message processing step contained in the iFlow at design time, without the need to deploy the iFlow and trace it at runtime. With this tool, you can submit input to a simulation—in particular, to define input message body, headers, exchange properties—making it possible to run a simulation of message processing steps with some relevant and representative input data.

The tool supports simulation for a Groovy script step, which makes it very useful for integration developers when testing Groovy scripts that have been added to the developed iFlow. Of course, the scope of application of the iFlow simulation tool goes beyond testing Groovy scripts in SAP Cloud Platform Integration and covers a variety of other message processing steps, but for the purposes of this section, we'll illustrate it using the example of a Groovy script step. Let's get back to the iFlow used in the Hello World example in Section 2.1 to see how the simulation mode can facilitate testing of a Groovy script step and the corresponding Groovy function it invokes.

Access the iFlow simulation tool by opening the iFlow in the integration flow editor from the **Workspace** view of SAP Cloud Platform Integration, and then switch it on by enabling the **Simulation Mode** toggle switch. After the simulation mode has been enabled, you'll see an additional toolbar that appears at the editor's canvas and contains controls to run and clear a simulation. The simulation toolbar and switch toggle are shown in Figure 3.1.

Figure 3.1 iFlow Simulation Tool: Toolbar and Switch Toggle

Next, it's necessary to add simulation start and end points; these points define the group of iFlow message processing steps that will be executed (or for some steps, mocked) during the simulation run. To add start and end points, click on the connector lines between relevant message processing steps, that is, the connector line before the first step that is to be tested in simulation, for the start point, and the connector line after the last step, for the end point. A single step or multiple steps can be used in the simulation. SAP Cloud Platform Integration product documentation can be consulted further to see restrictions that are applicable to some message processing step types.

In the context of our demonstration, click on the connector line that goes to the Groovy script step, and choose the **Simulation Start Point** option from the palette that appears. In a similar manner, click on the connector line that comes from the Groovy script step, and choose the **Simulation End Point** option from the palette. At this point, the canvas of the integration flow editor will look similar to the one shown in Figure 3.2.

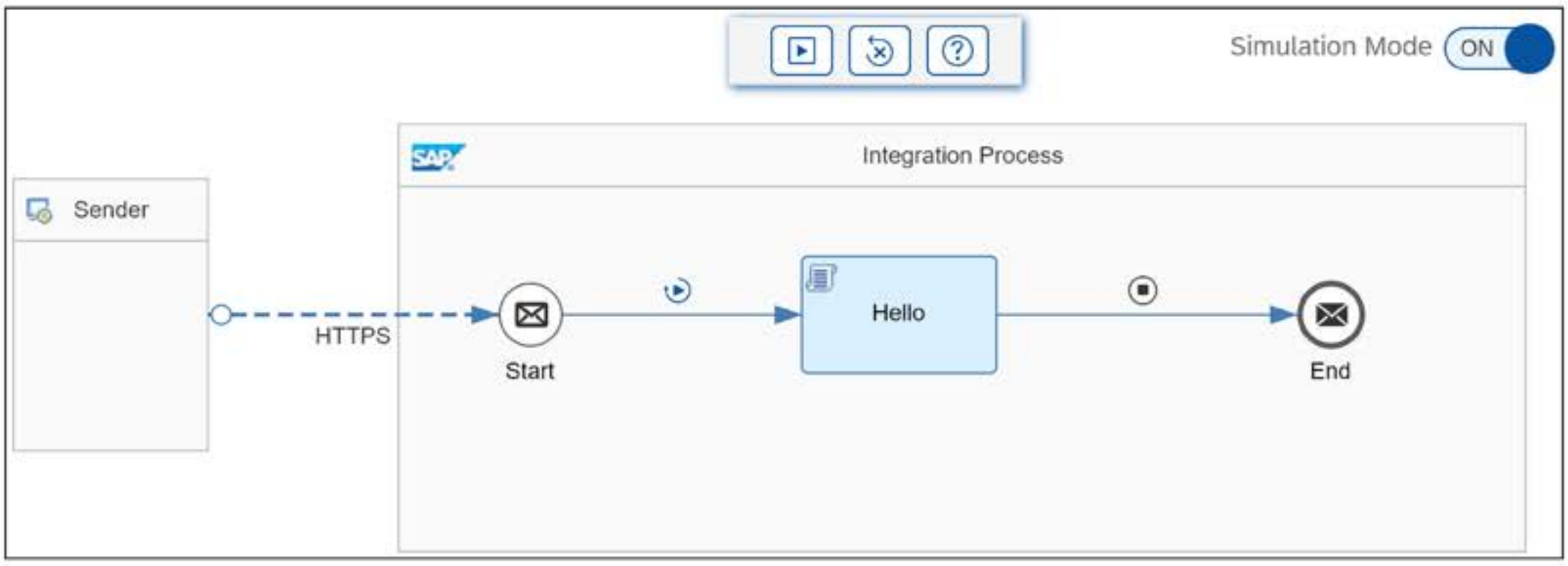


Figure 3.2 Defined Start and End Points of the Simulation

After simulation start and end points have been added, you can define input for the simulation. Click on the earlier added simulation start point, and this will open an **Add Simulation Input** window where message body and headers, and exchange properties can be added or deleted. We're going to run the simulation for a sample input that has been previously submitted in the HTTP request; therefore, to stay in line with that example, provide the input "World" in the **Body** field, as shown in Figure 3.3, and save it by clicking the **OK** button. We don't set any message headers or exchange properties as an input for the demonstrated simulation as they are irrelevant to the Hello World example and won't be used by tested flow logic, although we could have done so if required in more complex cases.

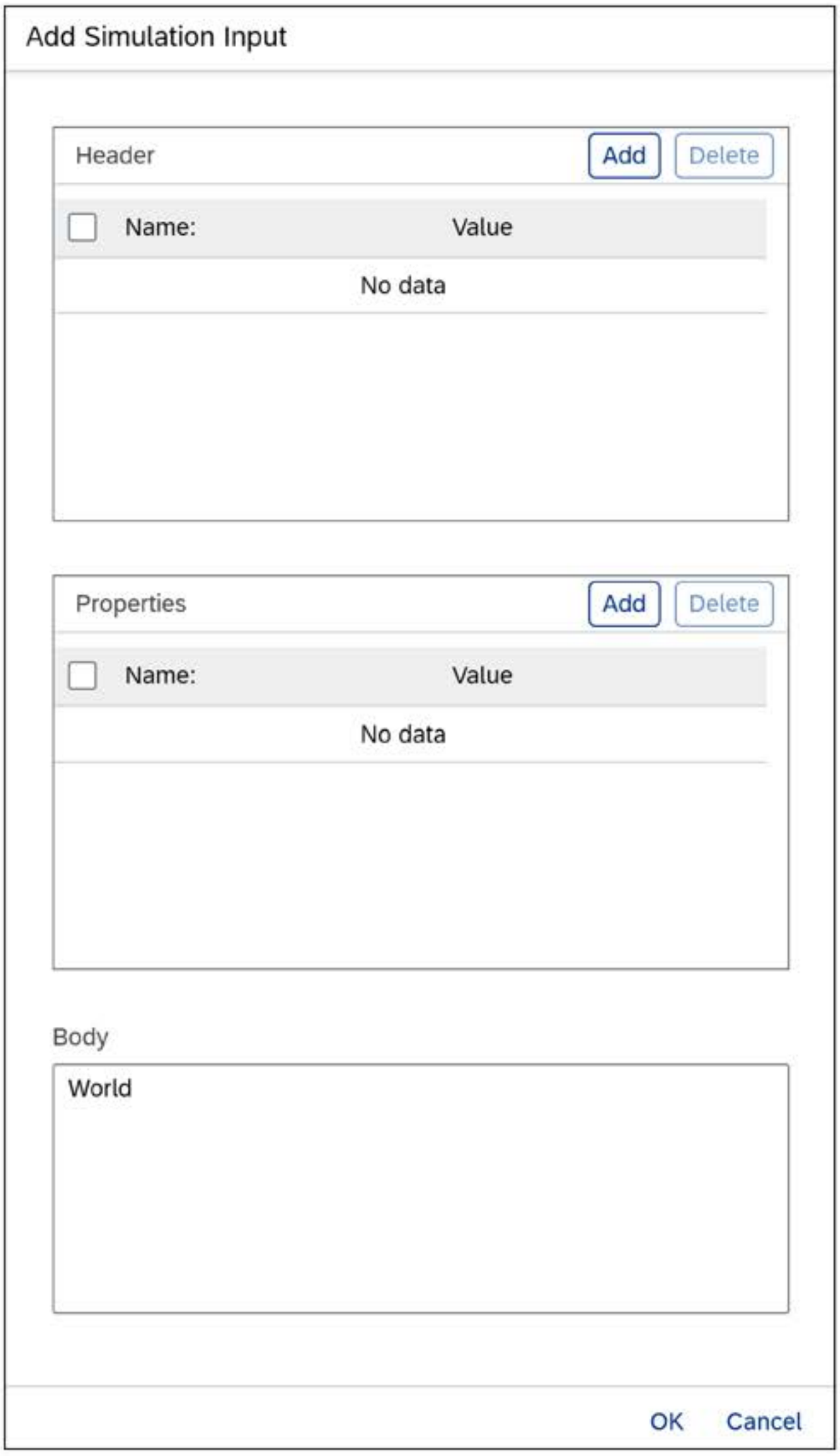


Figure 3.3 Simulation Input

You define start and end points of the simulation and define an input that will be used when the simulation will be run. Now run the simulation by clicking on the **Run Simulation** button in the simulation toolbar. After this is

done and the simulation run completes, the canvas appearance will indicate the outcome of the simulation run (color-coded icons that reflect success or failure), as well as the content of the message at the beginning of the simulation run and after each message processing step that was executed by the simulation (that enables step-by-step message content inspection) until the end point or up to the step that issues an error. Message content (and error details if the simulation run fails) can be accessed by clicking on the corresponding message envelope icons that appear above the connector lines that connect message processing steps and then exploring the **Message Content** view, as shown in Figure 3.4.

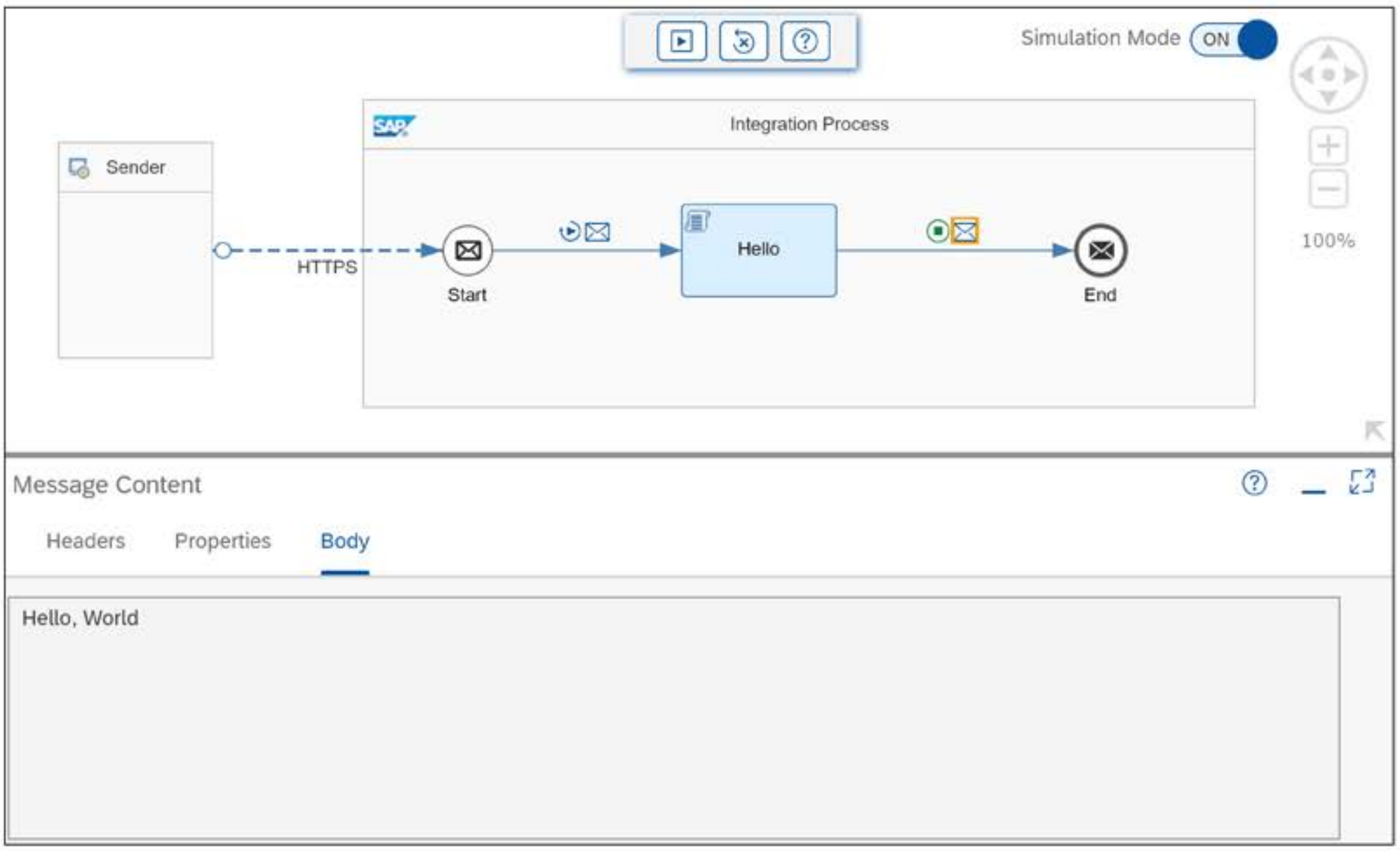


Figure 3.4 Simulation Run and Its Output

As you can see, the output of the Groovy script run in a simulation mode is identical to what you've observed in a response message when we tested the iFlow in Section 2.1.

3.2 Local Development Using an IDE

Following discontinuation of the *designer for SAP HANA Cloud Integration* feature for Eclipse and the corresponding **Integration Designer** perspective in 2018, integration developers are now encouraged to use the SAP Cloud Platform Integration web UI for development of integration artifacts and their assembly into content packages. The SAP Cloud Platform Integration web UI comes with the code editor that can be used to display and edit source code for Groovy scripts. Although the code editor provides basic capabilities for Groovy scripts development and is continuously enhanced and improved, currently it lacks some features that can make Groovy development more productive, such as project structure configuration, sophisticated code navigation, comprehensive code find and replace features, code completion, code refactoring, organization of imports, code inspection, integration with version control systems, dependency management systems, static code analysis tools, and testing frameworks to name a few.

That being said, note that the code editor provided as a part of the SAP Cloud Platform Integration web UI isn't the only tool that integration developers can use when developing Groovy scripts. Scripts can be developed outside of the SAP Cloud Platform Integration web UI and saved in appropriate Groovy script files (with extension *.groovy* or *.gsh*), and script files can be added (imported) as resources of the iFlow using the **Resources** tab of the integration flow editor and then used in Groovy script steps of the iFlow afterwards.

Development of Groovy scripts can be conducted in a vast number of tools that vary from simple text editors (e.g., Notepad), advanced text editors (e.g., Notepad++, Sublime Text, TextMate, Atom, or Vim), and code editors (e.g., Visual Studio Code) to feature-rich IDEs such as IntelliJ IDEA, Eclipse, or NetBeans IDE. Although lightweight text editors are convenient when introducing minor amendments to simple Groovy scripts, complex Groovy development is more comfortable and productive when using IDEs because they provide more capabilities in the area of code editing and refactoring, debugging, and integration with development infrastructure tools. Most, if

not all, modern IDEs provide support for Groovy features either natively and out of the box or with the help of appropriate plug-ins, which makes it possible for developers to pick the IDE that best fits their needs and preferences.

The IDE of choice for Groovy development is *IntelliJ IDEA* (https://www.jetbrains.com/idea/), thanks to its native support for Groovy, a comprehensive list of integrated tools, and a progressive and growing ecosystem that offers a multitude of plug-ins to extend out-of-the-box features of IntelliJ IDEA. IntelliJ IDEA is developed by JetBrains and is available for all major desktop platforms (Windows, Mac, Linux) in two flavors—a commercial Ultimate Edition and a free and open-source Community Edition. IntelliJ IDEA Ultimate Edition is bundled with features that are demanded in web and enterprise development, whereas IntelliJ IDEA Community Edition comes with a subset of features. Nevertheless, IntelliJ IDEA Community Edition offers all essential tools that are required for productive Groovy development, so it has been selected as the IDE for Groovy scripts development and will be used throughout the E-Bite.

We won't describe in detail the installation process of IntelliJ IDEA nor basic concepts of its usage. Instead, we assume you're familiar with the core features and capabilities of IntelliJ IDEA and will focus on how IntelliJ IDEA (exemplified by IntelliJ IDEA Community Edition) can be used for Groovy scripts development for SAP Cloud Platform Integration.

Now that the added value of a local IDE usage has been defined and the IDE has been selected, you can create a sample project and go through its structure. In this section, we set the scene with the initial structure of the project that will be progressed and enhanced in the following sections. Projects in IntelliJ IDEA help organize source code, required libraries and external dependencies, tests, and certain settings in such a way that they can be managed consistently. We're going to create a new project from scratch and set it up in a way that enables you to develop Groovy scripts for SAP Cloud Platform Integration. In particular, we'll build up the required minimal project structure and add the necessary dependencies. After the project is set

up, we'll progress with the creation of a sample Groovy script that implements the Hello World function that was used in Section 2.1. Given that some project setup steps are common and have the potential to be reused across multiple development projects, you'll also learn how to use custom templates to facilitate future development and accelerate creation of an empty development project that is ready for use.

Identify and Download Required Java and Groovy Library Versions

Before you start up IntelliJ IDEA and create the project, it's essential to identify the Java version and the Groovy library version that you'll need later.

Note on Groovy Version

It's important to ensure that the Groovy library version that is used in SAP Cloud Platform Integration is in sync with those used in the project in IntelliJ IDEA. Currently, SAP Cloud Platform Integration doesn't use the latest available Groovy version, so some language statements and APIs that are a part of the latest Groovy version are missing and not supported by SAP Cloud Platform Integration. Consequently, if IntelliJ IDEA happens to use a newer version of libraries, locally developed Groovy scripts that use recently introduced language statements and GDK APIs are likely to fail at the SAP Cloud Platform Integration runtime. This aspect is not only critical during development of Groovy scripts, but also, as you'll see in Section 5, it might significantly impact local testing, as, among other factors, we'll aim execution of tested Groovy scripts in a representative local environment and its runtime to qualify test results as trustful.

A version of the Groovy library that is used in runtime can be identified with the help of the standard GDK API by calling the `getVersion()` method of the `groovy.lang.GroovySystem` class. At the time of writing, the Groovy library version 2.4.12 is used in SAP Cloud Platform Integration.

Similar to identifying the Groovy library version currently used in SAP Cloud Platform Integration, you can also determine the Java version (Java 1.8.0, i.e., Java 8) by getting the value of the Java system property `java.version` by calling the `getProperty(String)` method of the `java.lang.System` class.

A simple form of a script function to output Groovy and Java versions is shown in Listing 3.1. Similar to other Groovy scripts that you find in this E-Bite, you can use the iFlow simulation tool to rapidly execute it and accelerate your work.

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

def Message processData(Message message) {
 StringBuilder sb = new StringBuilder()
 sb << "Groovy: ${GroovySystem.getVersion()}\r\n"
 sb << "Java: ${System.getProperty('java.version')}\r\n"
 message.setBody(sb.toString())
 return message
}
```

Listing 3.1 Script Function to Output Groovy and Java Libraries Versions

The Groovy library version 2.4.12 can be downloaded from https://archive.apache.org/dist/groovy/2.4.12/distribution/. Make sure the distribution that includes binaries (binary or SDK distributions) is downloaded.

Note that the binaries of earlier Groovy versions (versions before 2.5) used to be distributed in two flavors:

- **"Fat JAR" (the binary is named groovy-all-{version}.jar)**
  This single bundle contains the Groovy core and all modules except optional modules. This distribution option offers easier dependency management because only a single dependency must be added to the project, but the imported bundle will contain some modules that likely won't be used in the project and will remain redundant.
- **Collection of a Groovy core (the binary named groovy-{version}.jar) and individual modules (binaries are named groovy-{module}-{version}.jar)**
  Each Groovy module is packaged into its own binary and is available as a distinct dependency. This distribution option enables you to add only the required and used Groovy modules to the project, but it requires a good understanding of the available modules to make reasonable and correct choices regarding specifically required modules.

From Groovy version 2.5 onward, the former distribution option (a single bundle) has been deprecated, and only the latter option (individual modules) remains.

Although either of the two options can be used for Groovy version 2.4.12, here we use the latter option to make the future transition to version 2.5 or newer versions simpler and reduce adjustments to the project that will be associated with that transition. You can download the Groovy library from the Apache repository and unzip the files into the local directory *D:\lib\groovy-2.4.12*.

Create Groovy Project

Now you're ready to create the Groovy project. Because the project will be used for Groovy development, we'll create a Groovy project with the help of the project wizard. To do so, launch IntelliJ IDEA, and follow these steps:

1. If there are no opened projects, IntelliJ IDEA presents a **Welcome** screen on which you select **Create New Project**. If IntelliJ IDEA has already been in use and there are any opened projects, the earlier created project will be opened instead. If this happens, choose **File • New • Project** in the menu bar.

2. In the **New Project** window of the Project Wizard, select **Groovy** from a list of available types of projects. Depending on the plug-ins that have been installed and enabled in IntelliJ IDEA, the list of available project types and additional libraries and frameworks that can be used in them may vary. If you don't find **Groovy** in the list of available types of projects, check the list of installed plug-ins (from the previous **Welcome** screen, select **Configure • Plugins**, or from the IntelliJ IDEA main window, navigate to **File • Settings**, and go to **Plugins**), and ensure that the *Groovy* plug-in is enabled; if it's disabled, please enable it. If the plug-in had been disabled previously and had to be enabled, don't forget to restart IntelliJ IDEA for the changes to take effect.

3. Select the required JDK in the **Project SDK** dropdown, and select the required Groovy library in the **Groovy library** dropdown. If the dropdown

lists are empty or don't contain the required versions, use buttons to the right of the dropdown lists and select the locations of the required JDK and the Groovy library (that has been downloaded earlier). Ensure that the correct versions of Java and Groovy libraries have been selected—Java 1.8.0 and Groovy 2.4.12. At this step, the **New Project** window will be similar to what is shown in Figure 3.5. Click the **Next** button to proceed.

4. Provide the project name, for example, "sap-press-cpi-groovy", in the **Project name** field, and select the preferred location of the project, such as **D:\workspace\sap-press-cpi-groovy**, in the **Project location** field. Leave the default values in additional settings; those values are derived from the project name and location and are optimal for projects that include a single module. Click the **Finish** button.

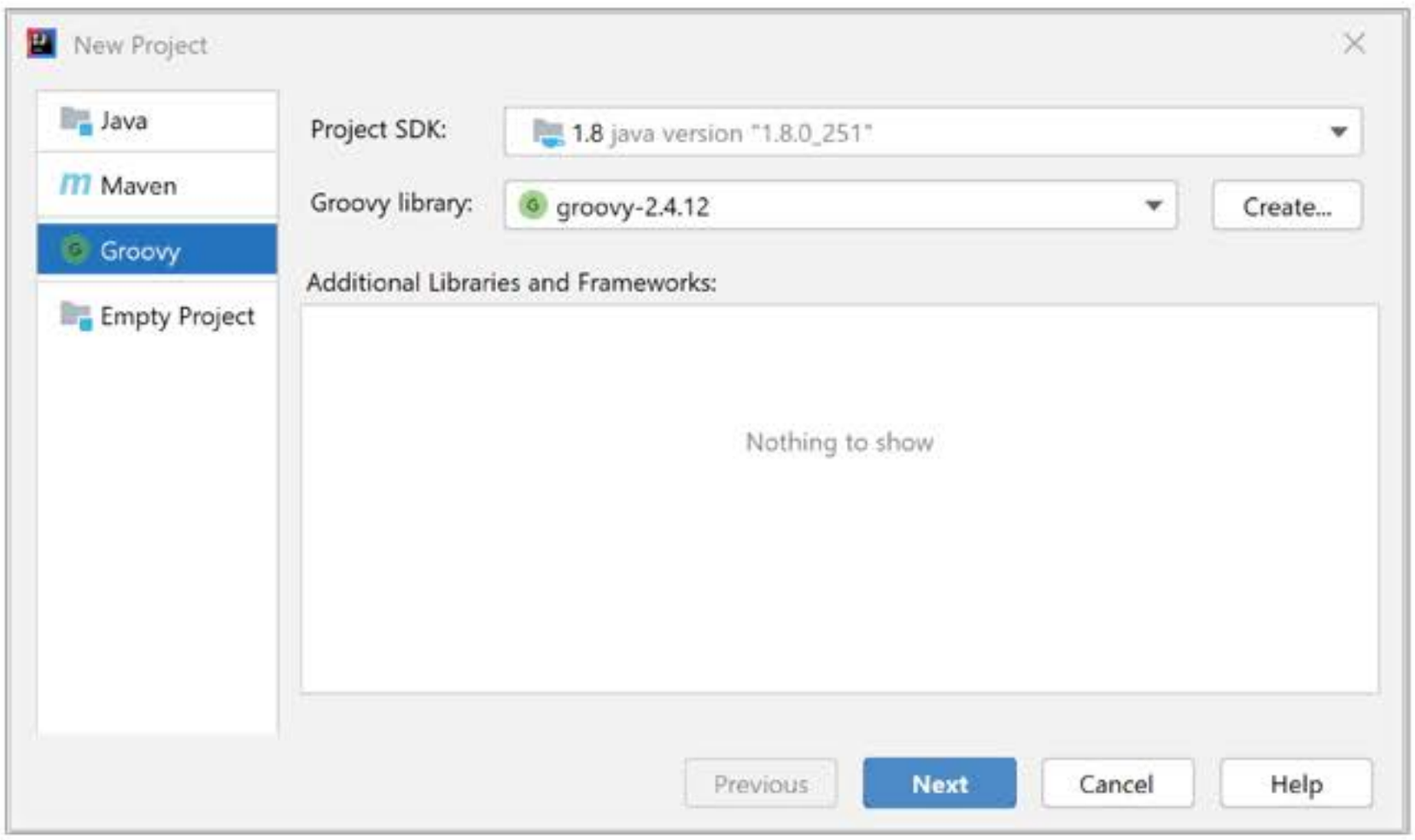


Figure 3.5 New Project Window of the Project Wizard

A new Groovy project is created and opened in the IntelliJ IDEA main window.

Modify Project Directory Structure

After the Groovy project has been created, the next step is to accommodate its directory structure to your needs. By default, the project includes a single

module that is named after the project and that contains a single source directory: *src*.

Using multiple modules within the single project usually makes sense when artifacts contained in different modules are developed using different programming languages, they are logically separated (e.g., client side and server side), or they have to be compiled using different settings. However, given the entire project is created only for Groovy scripts without compilation, assembly, and packaging for Groovy scripts developed for SAP Cloud Platform Integration, we don't need those features here, so the single module within the project is fine for now.

However, we do need to adjust the directory structure. Following the directory hierarchy and naming conventions that are commonly used in development projects, create an *src/main/groovy* subdirectory, and define it as the only source root. Generally speaking, directories that are defined as source roots usually follow the naming convention *src/main/{language}*, but given we don't intend to use the project for development using multiple programming languages, we don't need any other directories as source roots. At this step, it's important to ensure that the directory *src* doesn't remain a source root. Although the impact of it being a source root will be transparent in this section because we aren't going to create any other subdirectories under the directory *src* now, it will make a difference in Section 5 when additional directories used to store artifacts required for local testing are introduced under the *src* directory.

To modify the project structure:

1. In the menu bar, choose **File • Project Structure**.
2. In the **Project Structure** window, go to **Project Settings • Modules**.
3. Select the earlier created **sap-press-cpi-groovy** module, and navigate to the **Sources** tab if not already there.
4. Select the **src** directory, and click **Sources** in **Mark as** options to deselect it as a source root. Alternatively, right-click on the **src** directory, and deselect the **Sources** item from the context menu.

5. Right-click on the **src** directory, and select the **New Folder** item from the context menu.
6. In the **New Folder** popup window, enter a name for the newly created folder as "main/groovy". Click the **OK** button.
7. Expand the directory hierarchy under the **src** directory. Note that the required subdirectories—*main* and *groovy*—have been created.
8. Select the **groovy** directory, and click **Sources** in **Mark as** options to select it as a source root. Alternatively, right-click on the **groovy** directory, and select the **Sources** item from the context menu. Figure 3.6 illustrates how the project structure will look like after these modifications.
9. Click the **OK** button to apply all changes.

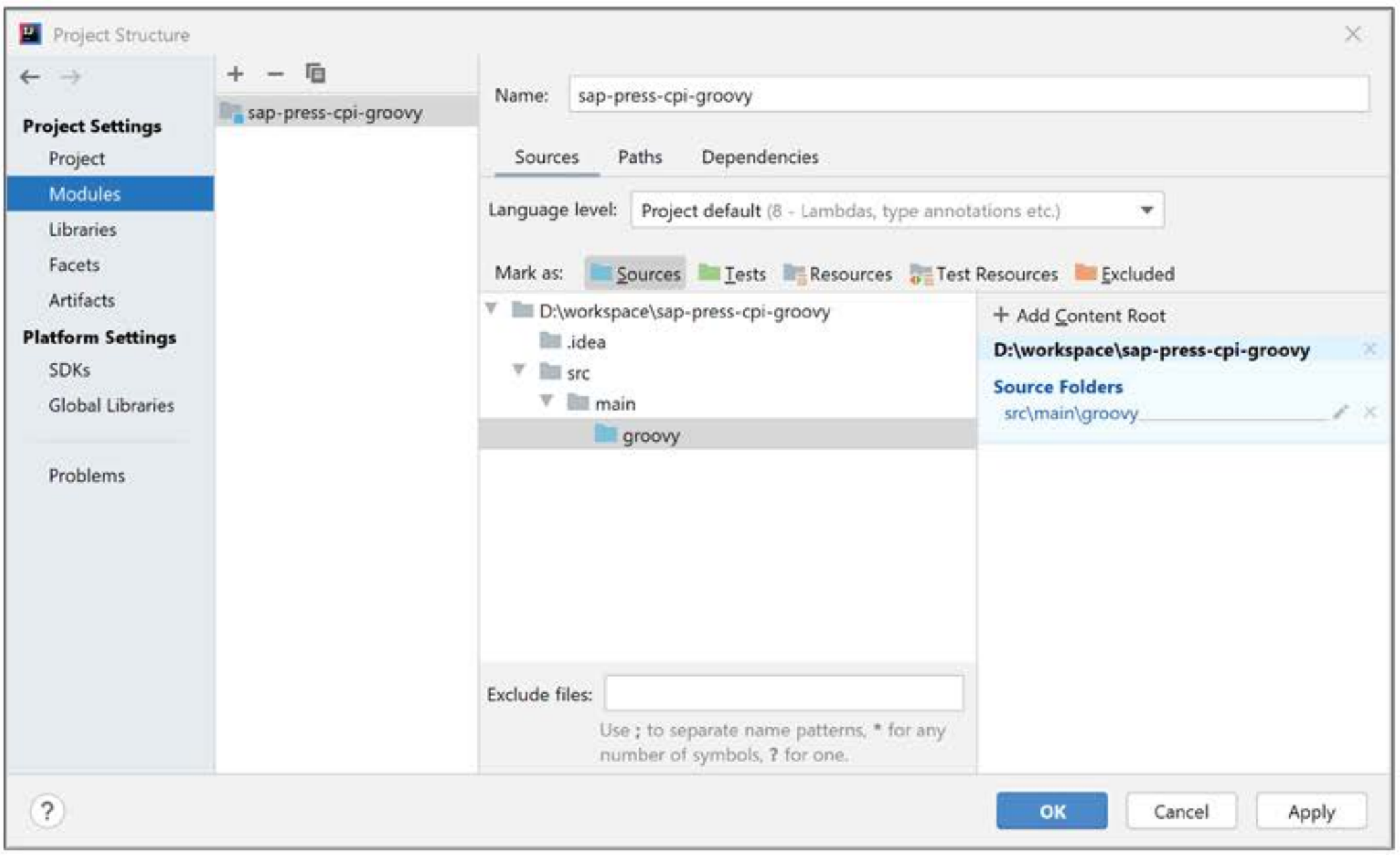


Figure 3.6 Groovy Project Structure

The modified and enhanced project directory structure will also be reflected in the **Project** view of the opened project in the IntelliJ IDEA main window.

Add Dependencies: External Libraries

Next, you need to add the required dependencies (external libraries) that will be used by the developed Groovy script. The complete list of dependencies highly depends on the specific script's functionality and varies depending on what libraries have to be used to accomplish that task. However, every Groovy script that is intended to be used in iFlows in SAP Cloud Platform Integration has at least one dependency that is essential for the project—the library that is provided by SAP and that defines the Script API. The Script API contains definition of the `com.sap.gateway.ip.core.customdev.util.Message` interface that is fundamental to any such Groovy script.

In this section, we walk through the basic steps of adding the Script API library as a global library in IntelliJ IDEA, which will allow us to move forward with the Groovy script development. In the next section, we'll provide more precise insight into what features the built-in dependency management mechanism of IntelliJ IDEA offers.

The Script API library can be downloaded from the SAP Cloud Platform Integration Tools site (https://tools.hana.ondemand.com/#cloudintegration): go to the **Using Script API** section to find a download link to a binary file that contains a current version of the Script API library. Download the Script API library (currently the latest binary is *cloud.integration.script.apis-1.36.1.jar*) to the local directory *D:\lib\sap\cpi*.

The next step is to create the library and add the downloaded binary file to it, as follows:

1. In the menu bar, choose **File • Project Structure**.
2. In the **Project Structure** window, go to **Platform Settings • Global Libraries**.
3. You can see the Groovy library that was added earlier during the project creation. Click on the plus icon (**New Global Library**) above the list of libraries, and select the **Java** item from the context menu.
4. In the **Select Library Files** popup window, navigate to the **D:\lib\sap\cpi** directory, and select the **cloud.integration.script.apis-1.36.1.jar** file. Click the **OK** button.

5. In the **Choose Modules** popup window, confirm the addition of the library to the *sap-press-cpi-groovy* module by clicking the **OK** button.
6. The created library inherited the name of the added dependency—*cloud.integration.script.apis-1.36.1*. Rename the library to a more meaningful and readable name, for example, *sap-cpi-script-api*. Replace the existing value **cloud.integration.script.apis-1.36.1** with "sap-cpi-script-api" in the **Name** field. Figure 3.7 shows how the configuration of global libraries will look. Click the **OK** button.

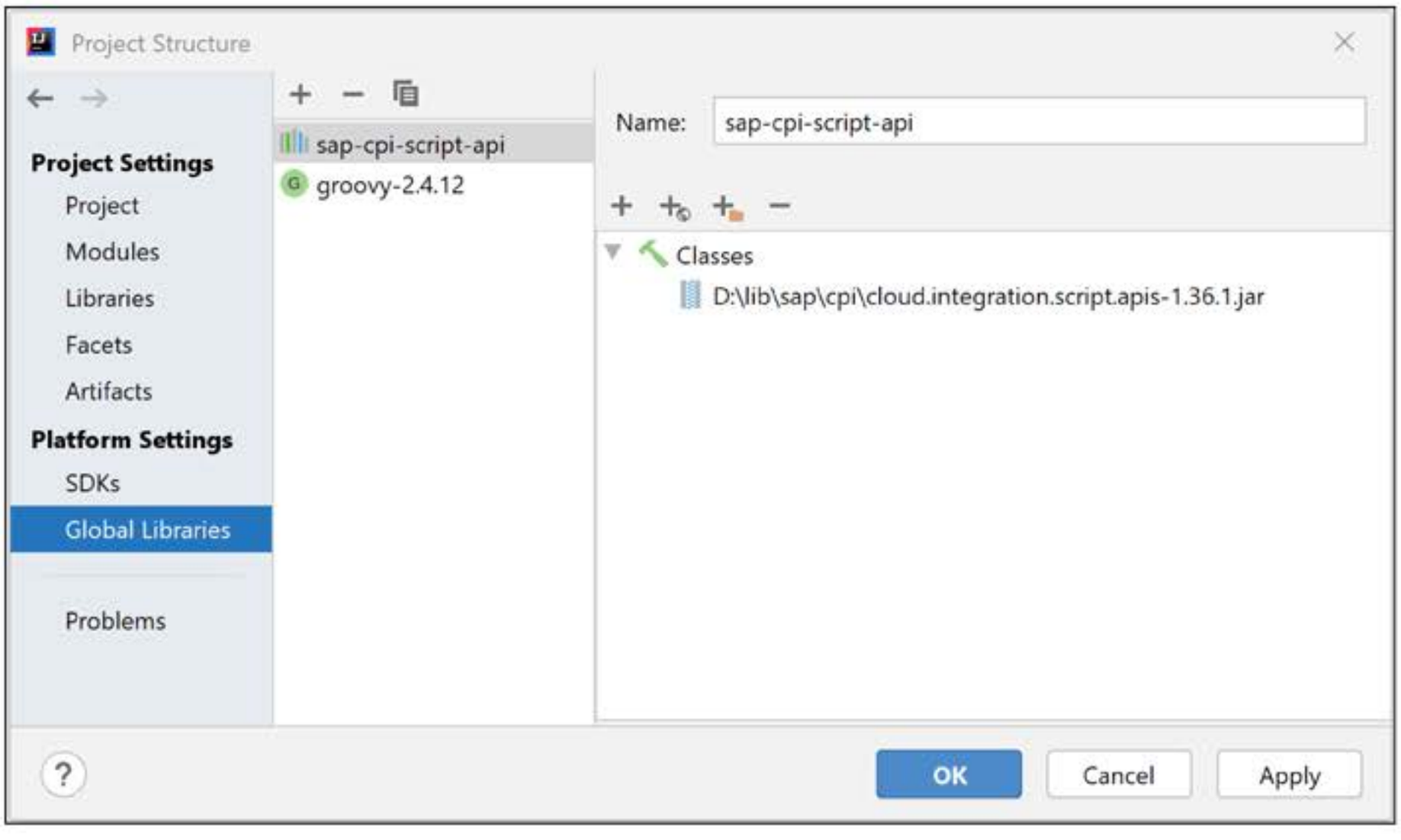


Figure 3.7 SAP Cloud Platform Integration Script API Library Added to the Global Library

Create SAP Cloud Platform Integration Groovy Project Template (Optional)

You might want to use the created project as a template for future projects that will relate to the development of Groovy scripts for SAP Cloud Platform Integration, for example, not to repeat steps to modify project directory structure and add required minimal dependencies. To save the project as a template, follow these steps:

1. In the menu bar, choose **Tools • Save Project as Template.**
2. In the **Save Project As Template** window, provide a meaningful name for the template—for example, "Groovy for CPI"—in the **Name** field, and click the **OK** button to save the project template.

The new project template will now become available in the **User-defined** section of the Project Wizard when creating new projects, as shown in Figure 3.8.

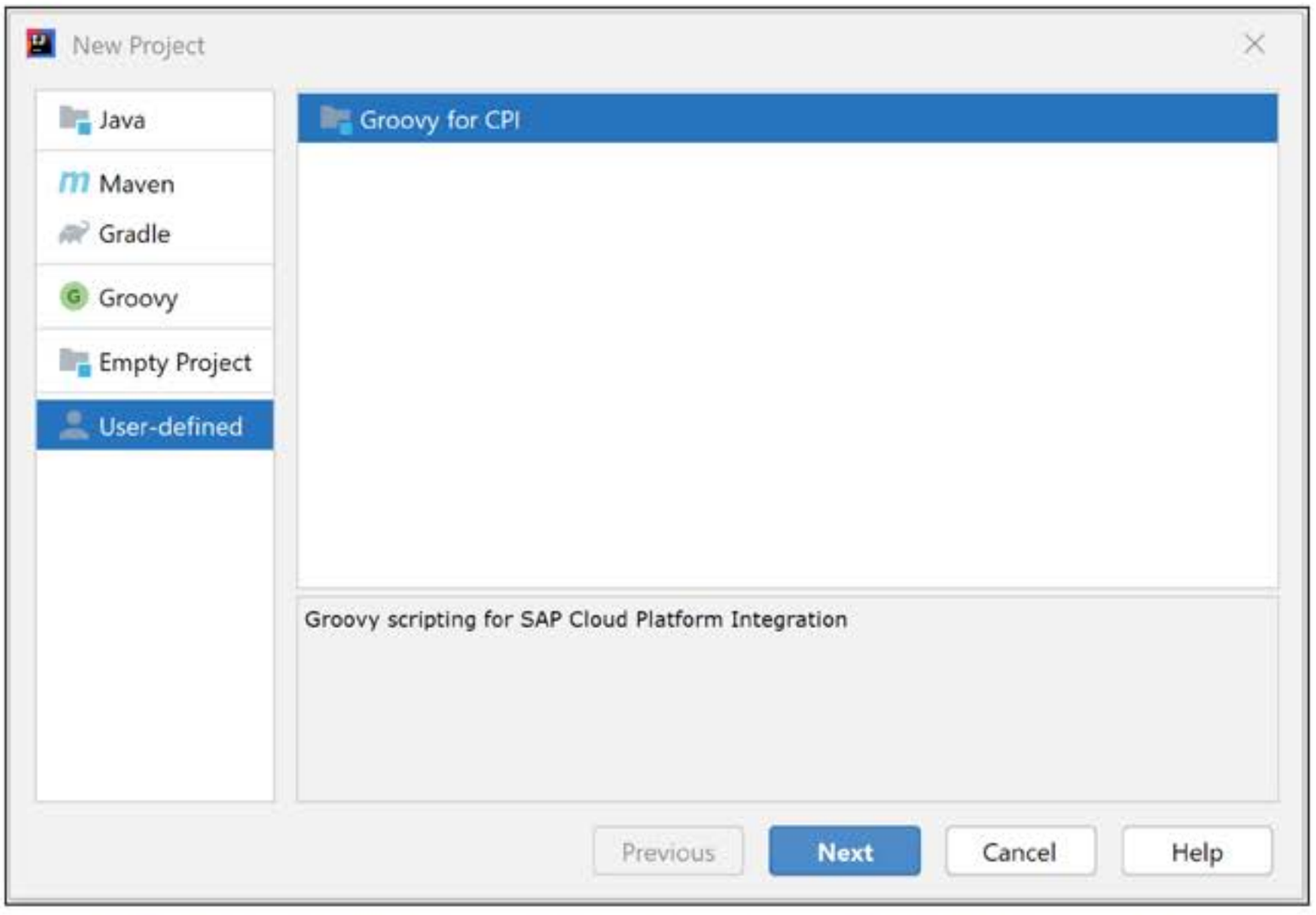


Figure 3.8 Custom Project Template for Groovy for SAP Cloud Platform Integration in the Project Wizard

Create Groovy Script

By now, you've settled the initial project settings, project structure, and project dependencies, and you're ready to create your first Groovy script in IntelliJ IDEA. For the sake of simplicity, you'll create it directly in the *src/main/groovy* directory.

To create a new Groovy script, follow these steps:

1. In the **Project** view, expand the project structure down to the *src/main/groovy* directory, and select that directory.
2. In the menu bar, navigate to **File • New**, and select the **Groovy Script** item. Alternatively, right-click on the **src/main/groovy** directory, and select **New • Groovy Script** item from the context menu.
3. Enter the name of the created Groovy script file, for example, **HelloWorld**, ensure that the **Groovy Script** type is selected, and press Enter.

You can now proceed with script development using the code editor by populating it with the code snippet that was used earlier, as demonstrated in Figure 3.9.

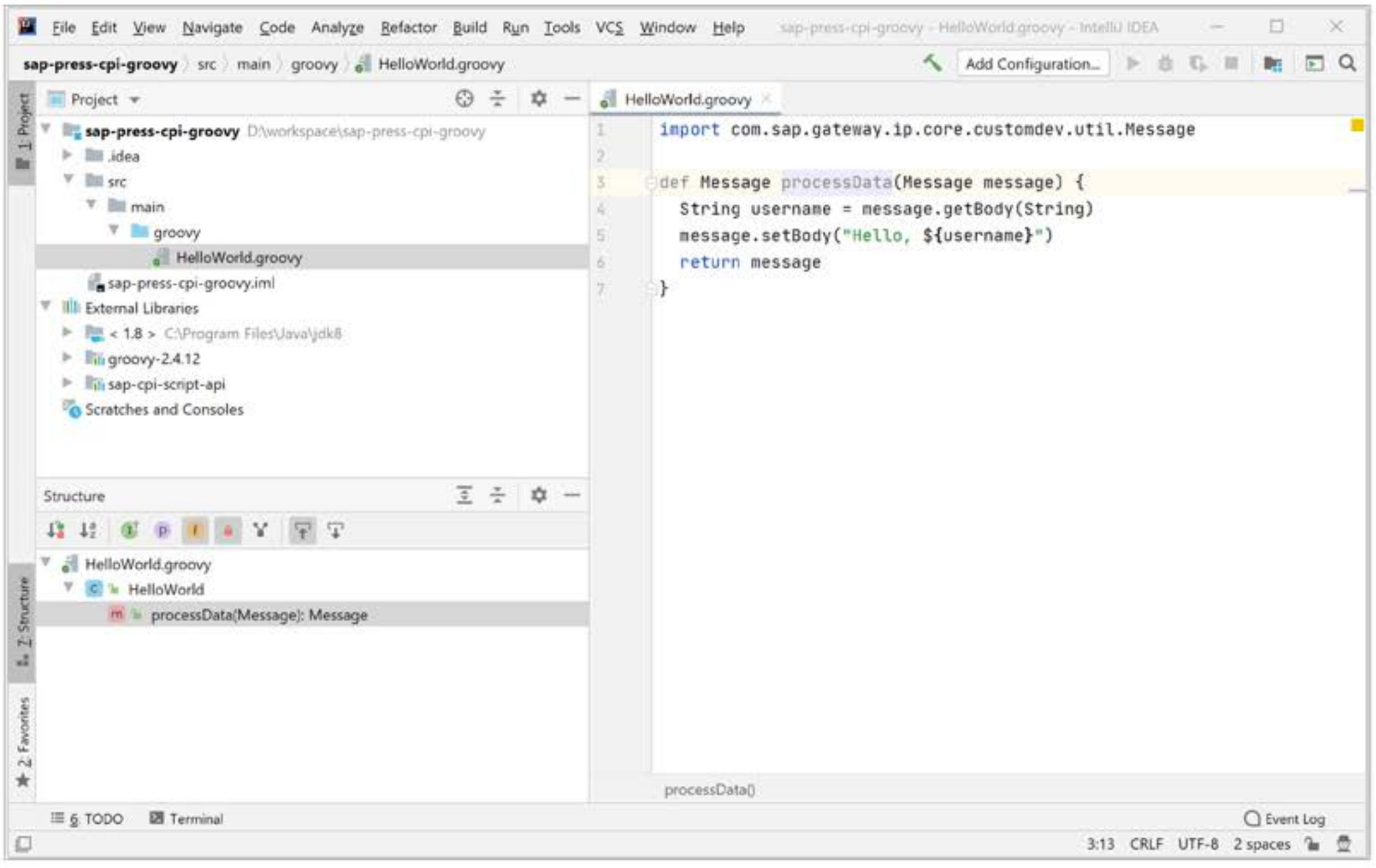


Figure 3.9 Groovy Script for SAP Cloud Platform Integration in the Code Editor

After the script is finished, the corresponding script's file remains persisted in the location of the project and can be added to iFlows' resources as a script resource using the SAP Cloud Platform Integration web UI.

Create SAP Cloud Platform Integration Groovy Script File Template (Optional)

The overwhelming majority of Groovy script development for SAP Cloud Platform Integration begins with a boilerplate code of the script, which is the `processData` function (it can have a different name, but `processData` is the default) that is defined with the single input parameter of the type `Message` and that returns an instance of the (transformed) `Message`. To avoid copying and pasting the same code snippet again and again, you can automate the generation of the function definition with the help of file templates. Let's create a new custom file template that is based on the standard *Groovy Script* template and that can be used further when creating new Groovy scripts for SAP Cloud Platform Integration. To access file templates and create a new template, follow these steps:

1. In the menu bar, choose **File • Settings.**
2. In the **Settings** window, go to **Editor • File and Code Templates.**
3. On the **File** tab, click on the button with the plus icon (**Create Template**), provide a meaningful name for the template—for example, "Groovy Script for CPI"—in the **Name** field, enter "groovy" in the **Extension** field as the template will be used for Groovy script files, enter the code snippet that the template will fill in when a new file is created with that template, and click the **OK** button to save the file template.
4. File templates can apply to the entire workspace (**Default**) or only to the specific project (**Project**).Create the file template at the workspace level by leaving the **Default** value in the **Scheme** dropdown list, so that the file template can be reused in other projects (note: this behavior/scope can be adjusted). After completion of these steps, the file template will look like Figure 3.10.

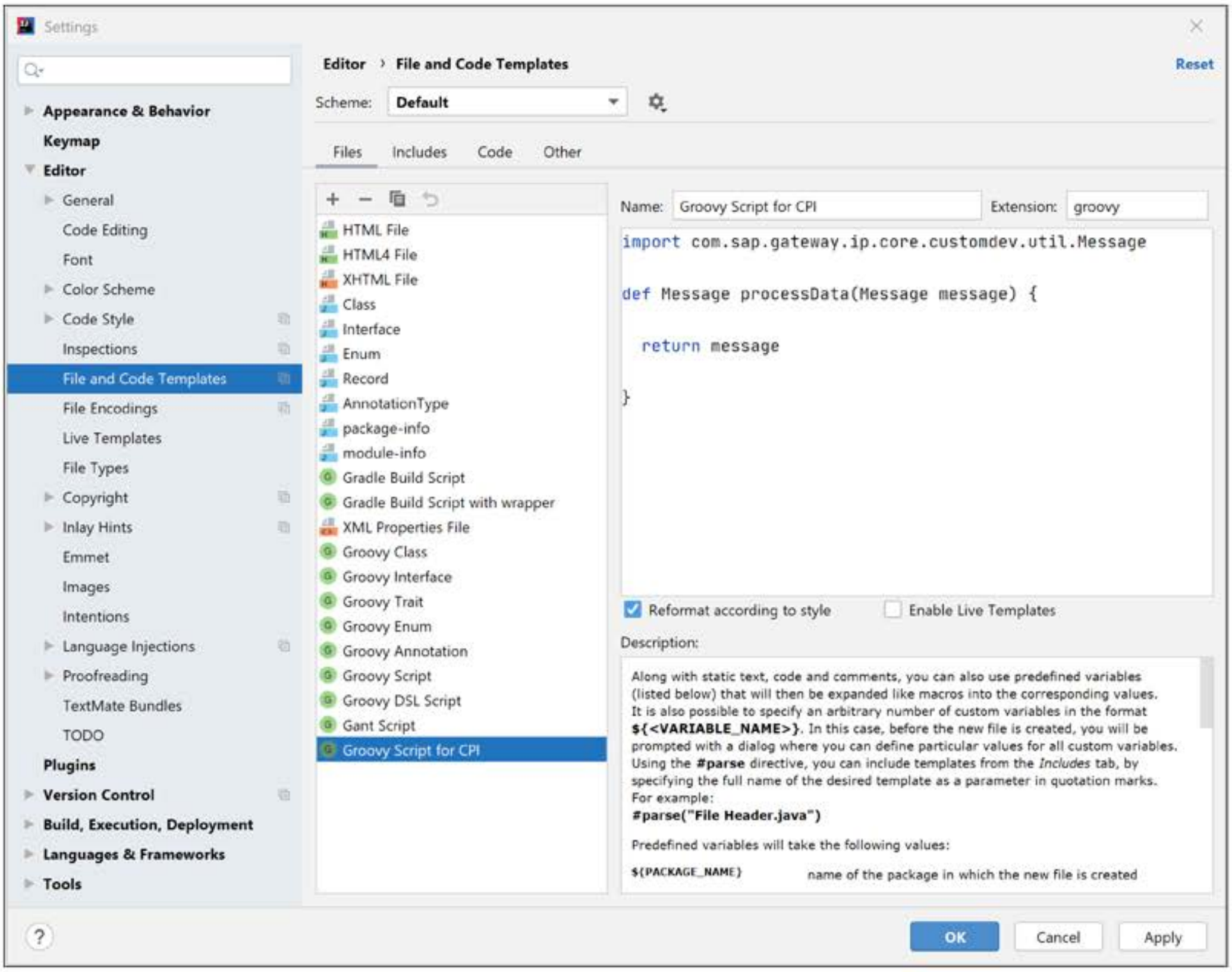


Figure 3.10 Custom File Template for Groovy Scripts for SAP Cloud Platform Integration

This example provides a basic template for the script function. Depending on specific requirements, you can extend and enhance it, for example, by adding optional package declarations (if your organization uses development packages to organize Groovy scripts for SAP Cloud Platform Integration), inserting comment blocks into the template (if GroovyDoc tags or other comment markup is used), and so on.

After the file template has been created, it will become accessible from the list of available artifact types when creating new files in the project (e.g., when choosing **File • New**), as illustrated in Figure 3.11.

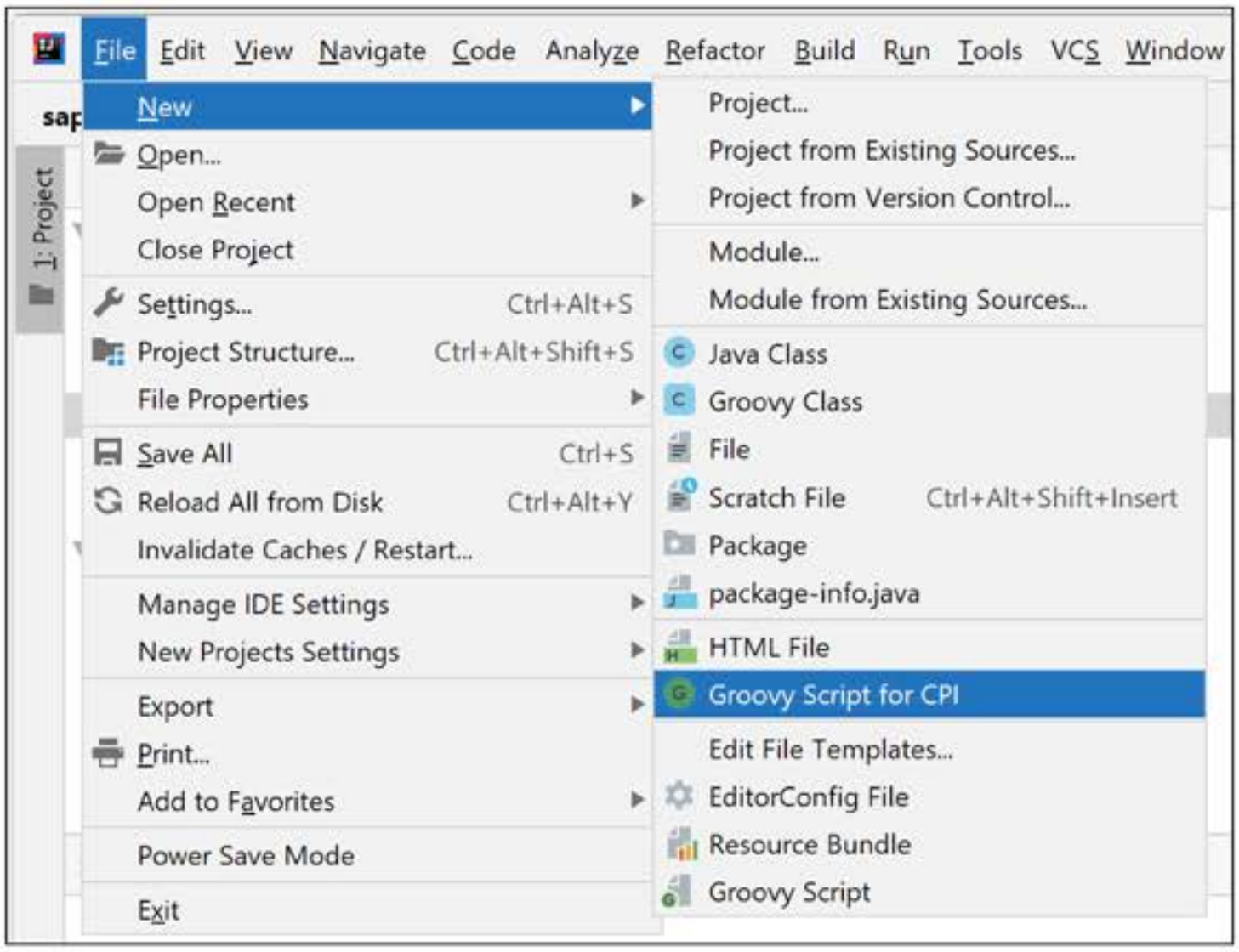


Figure 3.11 Custom File Template for Groovy Script for CPI in the New File Creation Menu

Tip

Project templates and file templates can be exported and imported, which makes it possible to share them. To export project and/or file templates, follow these steps:

1. In the menu bar, choose **File • Manage IDE Settings • Export Settings.**
2. In the **Export Settings** window, clear all settings items except **Project templates** item (to export project templates) and/or the **File templates (schemes)** item (to export file templates).
3. Provide the location for the ZIP file that will contain exported settings, and click the **OK** button.

As a result, the generated ZIP file will contain project and/or custom file templates (depending on the selection of items). This ZIP file can be shared and imported in IntelliJ IDEA installations on other developer machines.

Although the IntelliJ IDEA code editor of is powerful and alone increases developer productivity significantly (we've already highlighted some of its key features at the beginning of this section), IntelliJ IDEA and tools that can

be integrated with it enrich the toolbox of the integration developer with a lot of useful capabilities that can increase not only productivity but also development quality. We highly encourage you to become familiar with and explore features that IntelliJ IDEA and its relevant plug-ins offer.

3.3 Dependency Management in IntelliJ IDEA

IntelliJ IDEA already comes with built-in tools that can be used to manage project dependencies. In addition, it provides integration with specialized dependency management and build tools such as Maven and Gradle, which are widely used in Java development projects. To support the latter option, IntelliJ IDEA has corresponding plug-ins that enable you to create and change build scripts, execute Maven goals or Gradle tasks, and inspect results of their execution using views that enrich the IntelliJ JDEA UI. Although this E-Bite doesn't aim to provide in-depth description either Maven or Gradle tools, in this section, we'll focus on tools that are provided by IntelliJ IDEA out of the box.

Libraries

In IntelliJ IDEA, individual dependencies are grouped and managed in libraries, where a single library can contain one or multiple files. For example, if there are multiple dependencies that together form a logical group and that will be managed as a single unit, instead of adding them as separate external dependencies, you can create a single library and add all logically related dependencies into it. Such grouping doesn't modify or bundle dependencies in their original locations, but it's used within IntelliJ IDEA to organize dependencies and simplify their further management.

Libraries can be added in IntelliJ IDEA on three levels, which depend on the scope of library usage and impact the reusability aspect of libraries:

- **Global level**
  The library can be reused in multiple projects.

- **Project level**
  The library can be reused in multiple modules of one project, but it isn't accessible to other projects.
- **Module level**
  The library can only be used in one module, but it isn't accessible to other modules of the same project or to other projects.

IntelliJ IDEA offers two built-in methods to create and maintain global and project libraries:

- **Maintain the library using a local Java library**
  The library includes files that are located on a local machine.
- **Maintain the library using a Maven dependency**
  Dependencies are fetched from configured Maven repositories.

Usage of Maven repositories is highly recommended for dependencies that are already maintained in repositories and that can be searched and fetched from there. It eliminates the need for a manual download of specific binary files, allows central maintenance of dependencies, and simplifies dependencies' version upgrade or downgrade in the project. If the dependency hasn't been published to an accessible Maven repository but can be accessed using other means (downloaded from the site/web repository, polled from the file server, etc.), then it can be saved locally and added as a local Java library. Given that currently the Script API library that we needed to add to project dependencies in the previous section, isn't available in common public Maven repositories, we added the dependency for the Script API library using a local Java library.

We've already walked through the process of creating a global library using a local Java library when we needed to add the dependency for the Script API earlier. The project library is created similarly, except that in the **Project Structure** window, global libraries are maintained in the **Platform Settings • Global Libraries** section, whereas project libraries are maintained in the **Project Settings • Libraries** section.

When creating the library using a local Java library, it's possible to add not only specific files but also entire directories that contain required

dependencies' files. The ability to add directories to the library is helpful if several files are part of the library and are located in the same directory; however, it's recommended to check the content of the directory in advance and ensure that it doesn't contain an excessive number of unwanted files that won't be included in the library.

Tip

The Groovy library that was added earlier during the Groovy project creation using the Project Wizard alternatively could have been added like any other library; that is, we could have skipped the step of creating the Groovy library and its assignment to the project in the Project Wizard by leaving the corresponding field empty, which would have led to creating the Groovy project without the Groovy library. We could have then added the Groovy library to the project using the technique described here.

To create a library using a Maven dependency, follow these steps:

1. In the menu bar, navigate to **File • Project Structure** window.
2. Go to the **Libraries** or **Global Libraries** section (depending on what library you create—project or global).
3. Click on the plus icon (**New Project Library** or **New Global Library**, depending on whether you're creating a project or global library), and select the **From Maven** item.
4. In the **Download Library from Maven Repository** popup screen, you can search for required dependencies using keywords or naming conventions that are commonly accepted in Java dependency management systems and artifact repositories, such as combination of group ID, artifact ID and version, using the following pattern: *{group ID}:{artifact ID}:{version}*. For example, for the Groovy library version 2.4.12 that is distributed as a fat JAR, the corresponding "coordinates" are *org.codehaus.groovy:groovy-all:2.4.12*. Figure 3.12 exemplifies the search for the Groovy library used in the project.

   If the downloaded dependency relies on other dependencies (called *transitive dependencies* for the searched dependency), you can fetch the

entire bundle of all required dependencies (in a *dependency tree*) by selecting the **Transitive dependencies** checkbox (that is selected by default). You not only can download transitive dependencies alongside the downloaded library but also exclude specific transitive dependencies from the fetched list of known transitive dependencies of the library afterwards when editing the created library's configuration. This feature is helpful when external libraries and certain transitive dependencies that are all added to the project run into conflict with each other and when only one version of conflicting dependencies will remain among the dependencies while others will be excluded. Besides binary files of dependencies, it's possible to download supplementary files that are associated with the dependency—such as source files and Javadocs—if they are present in the repository.

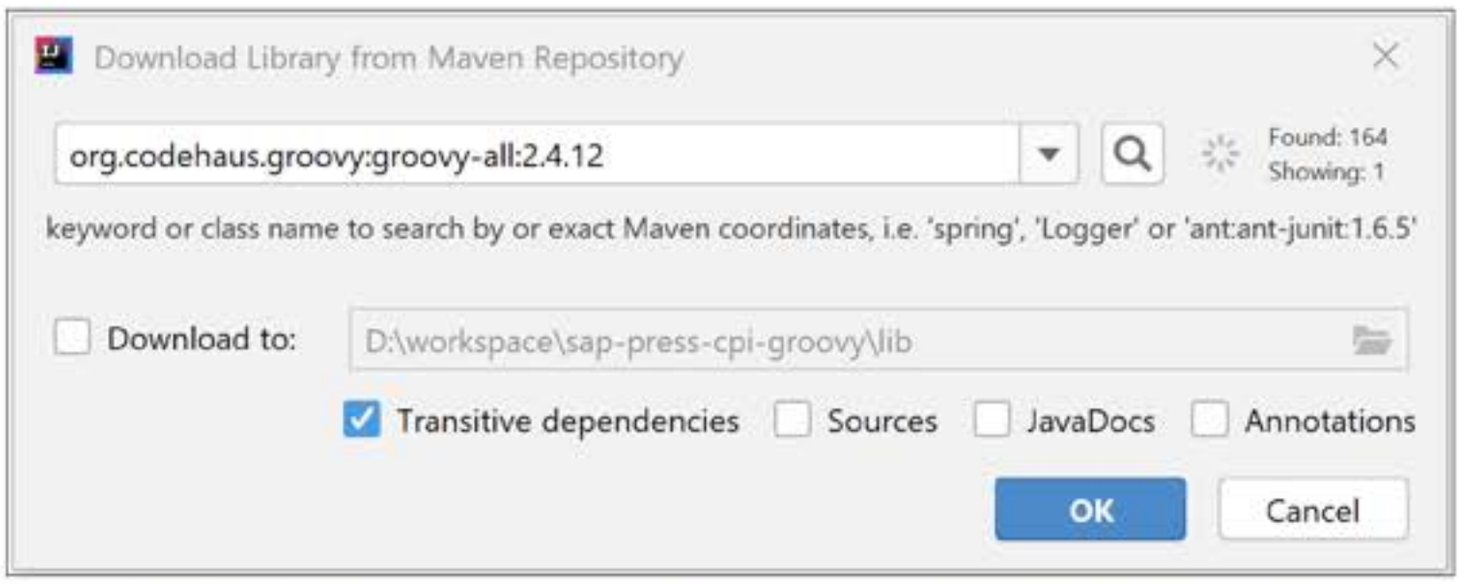


Figure 3.12 Searching for Maven Dependencies When Creating the Library

5. After the required dependency is found and selected, click the **OK** button to create the library with that dependency.

Note on List of Repositories

You can modify the list of repositories where IntelliJ IDEA searches for Maven dependencies and from where it fetches found dependencies, for example, by adding a custom repository to that list. To access the list of repositories, follow these steps:

1. In the menu bar, choose **File • Settings.**
2. In the **Settings** window, choose **Build, Execution, Deployment • Remote Jar Repositories.**

If the library was created as the global library, it can be copied to the project libraries. Note that the library will be *copied*, and after the operation is finished, the original global library will remain, and a project library will be created as a copy of it. If the library was created as the local library, it can be moved to global libraries. Note that the library will be *moved*, and after the operation is finished, only the global library will exist. Corresponding operations are accessible from the context menu of the library in the **Project Structure** window.

Module Dependencies

Before dependencies contained in the library can be used in developed scripts, it's necessary to add them to the module, where scripts are located. Right-click on the required library, choose the **Add to Modules** item from the context menu, and then select the module(s) to which the library will be added.

To check which libraries have been added to the module:

1. Navigate to the **Project Structure** window.
2. Go to **Project Settings** · **Modules**.
3. Select the required module—for example, the earlier created *sap-press-cpi-groovy* module—and navigate to the **Dependencies** tab.
4. As illustrated in Figure 3.13, you can see which libraries have been added to the module and add module-specific libraries, delete earlier added dependencies, or change their order.

Warning on Module Dependencies

Dependencies that are added to the module and used in developed Groovy scripts for SAP Cloud Platform Integration don't get added to the iFlow automatically after the corresponding script file is added to resources of the iFlow. Therefore, if the Groovy script relies on dependencies that aren't part of SAP Cloud Platform Integration (i.e., they aren't already available at runtime), the binary files of such dependencies must be manually added to the iFlow as resources of type **Archive** on the **Resources** tab of the integration flow editor to avoid errors at runtime.

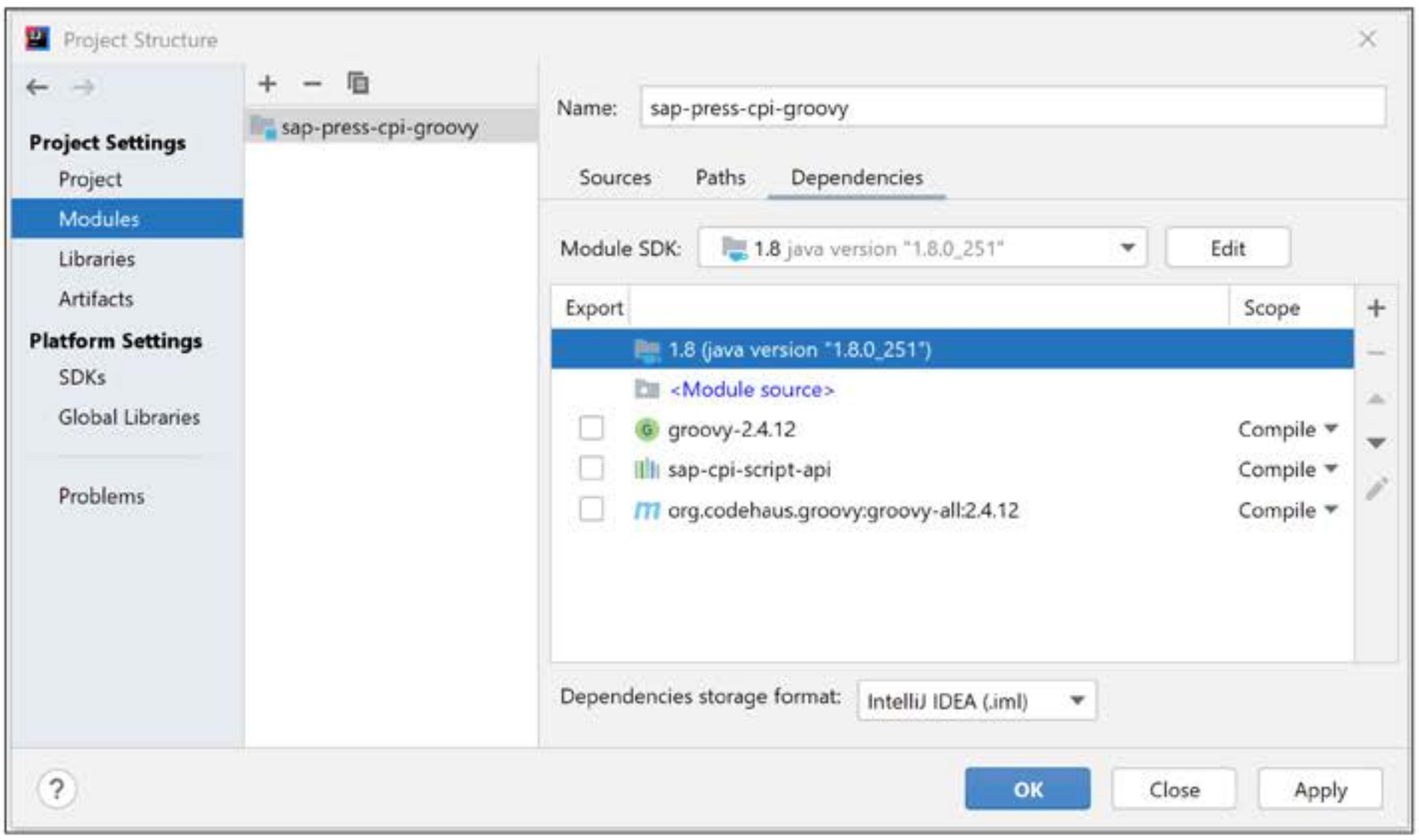


Figure 3.13 Module Dependencies

Dependency Hell

Projects that contain a large number of dependencies, especially those with a complex and highly nested structure of transitive dependencies, eventually may suffer from problems that are collectively known as *dependency hell*—an unwanted state of project dependencies when the consolidated list of the entire dependency tree includes conflicting versions of dependencies (commonly originating from different versions of the same library used as transitive dependencies in required higher level dependencies) and a large number of binary files that comprise the entire set of declared or determined dependencies. Because optimization of dependencies and resolution of dependency conflicts can become nontrivial tasks, even with the ability to visualize the dependency tree via specialized dependency management tools, you should be mindful about dependencies and clean up any unused ones.

Although Groovy scripts for SAP Cloud Platform Integration don't normally suffer from dependency hell as it's uncommon for them to require a significant number of dependencies, you should pay attention to dependencies

and their imports used in your projects. You can analyze dependencies for the given script(s) using the **Dependency Viewer** (choose **Analyze • Dependencies**). Subsequently, class navigation (choose **Navigate • Class** for the selected class or interface) can be used to determine which dependency contains the used class. Both tools are illustrated in Figure 3.14.

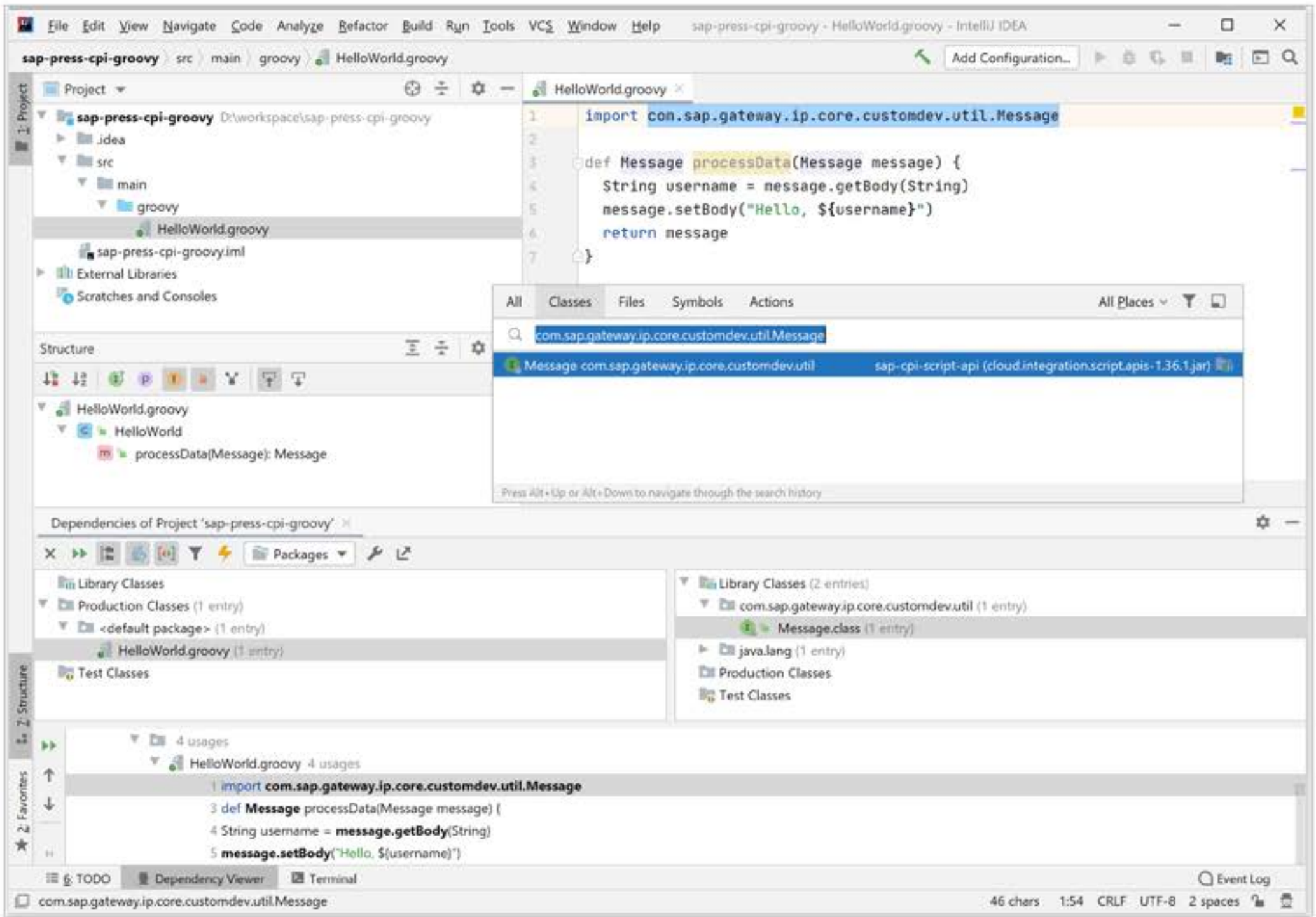


Figure 3.14 Dependency Viewer (Bottom) and Class Finder (Popup in the Middle)

4 Transformation Using Groovy Scripts

One of the key functionalities of SAP Cloud Platform Integration is transformation of data format and structure. Following are the various options to perform such transformation:

- Graphical message mapping
- Extensible Stylesheet Language Transformation (XSLT)
- Groovy or JavaScript scripts
- Content modifier

Of the four options, the scripting approach is the most robust and flexible option as it allows you to use the full capabilities of a programming language to transform any type of data format and message structure. As such, this enables you to handle the simplest to the most complex transformation requirements.

In this section, you'll learn how to develop Groovy scripts for a few use cases of transformation. We'll first focus on Extensible Markup Language (XML) and JavaScript Object Notation (JSON), two of the most popular data formats used for integration. For both, you'll see how to read (parse) and compose (generate) message structures using these formats. During parsing, you deserialize the data from its XML/JSON format into Groovy objects, allowing you to perform further manipulation of the data natively in Groovy. The opposite occurs during generation in which the Groovy objects are serialized into specific XML/JSON formats. While it's possible to interchange between any combination of input and output data formats, these examples will only show XML-to-XML transformation and JSON-to-JSON transformation.

Then, we'll cover how to handle other text and binary data formats exemplified by comma-separated values (CSV; for a text format) and PDF (for a binary format), and show the role of the binary-to-text encoding technique and how it can be applied in transformations.

The section concludes with an insight into how Groovy functions can be used in graphical message mappings to implement custom functions that accompany standard mapping functions. This is yet another commonly seen use case where Groovy development can help solve diverse requirements in the area of message payload transformation.

To benefit from IntelliJ IDEA, which was introduced in the previous section, we encourage you to develop examples that you'll find in this section using the IDE. This allows you to become familiar with the IDE and benefit from its native support for Groovy.

4.1 XML

When it comes to integration, Extensible Markup Language (XML) has been and still is a popular data format. Groovy has native capabilities for XML processing that allow you to parse and generate XML documents easily. Let's look into XML processing in Groovy in more detail, starting with parsing and generating sample XML documents. After you're familiar with parsing and building techniques, we'll bring them together and see how some sample conversion logic can be implemented for an XML-to-XML transformation using Groovy capabilities.

Parsing XML Document

Groovy comes with the following classes that are normally used for parsing XML documents:

- `groovy.util.XmlSlurper`
- `groovy.util.XmlParser`

Both are based on Simple API for XML (SAX), which has a lower memory footprint compared to Document Object Model (DOM) parsers that are often used in Java. Although both can parse, update, and transform an XML document, one of the key differences is that `XmlSlurper` evaluates the structure lazily, so any updates aren't available until the whole XML document is parsed again. Therefore, the recommended usages are as follows:

- **`XmlSlurper`**
  Use for read-only parsing of XML documents.
- **`XmlParser`**
  Use for parsing and in-place manipulation of XML documents.

In the following examples, we'll focus only on using `XmlSlurper` to illustrate the parsing of XML documents.

`XmlSlurper` comes with various methods for parsing an XML document. While the `parseText(String)` method (requires a `String` object as an input) is commonly used, we recommend using the overloaded `parse(Object)` method with a `Reader` object as an input. In the context of SAP Cloud Platform Integration, an XML document contained within the message body is typically an `InputStream` object. Converting this to a `String` object requires allocation of additional memory, which doesn't scale well when working with large payloads. Instead, using a `Reader` avoids additional memory allocation as it's just a reference to the payload object, and thus allows the XML document to be streamed during parsing. The following code snippet shows how `XmlSlurper` is used to parse the input XML document via a `Reader` object:

```
Reader reader = message.getBody(Reader)
def rootNode = new XmlSlurper().parse(reader)
```

Tip

The technique to use a `Reader` to stream the input payload is also applicable to other character-based data such as JSON or plain text.

GPath

`XmlSlurper` returns instances of `groovy.util.slurpersupport.GPathResult` when parsing XML documents. `GPathResult` allows you to conveniently access the XML data using *GPath*, a path expression language built into Groovy. It's similar to *XPath*, but it can be applied not only to XML but JSON as well as Plain Old Java Object (POJO) classes. Using GPath expressions, you can easily access nested structures using dot notation.

Listing 4.1 illustrates accessing data in an XML document using a sample XML document.

```
<Order>
    <Header>
        <OrderNumber>ORD60001</OrderNumber>
        <Date>20190218</Date>
    </Header>
    <Item>
        <ItemNumber>10</ItemNumber>
        <MaterialNumber>MT70001</MaterialNumber>
        <Quantity>57</Quantity>
        <Valid>true</Valid>
    </Item>
    <Item>
        <ItemNumber>20</ItemNumber>
        <MaterialNumber>MT80001</MaterialNumber>
        <Quantity>28</Quantity>
        <Valid>false</Valid>
    </Item>
</Order>
```

Listing 4.1 Content of Sample Input XML Document

Listing 4.2 shows how you can access the following data using GPath expressions:

- `OrderNumber` on the header section
- First `Quantity` of the item section

```
Reader reader = message.getBody(Reader)
GPathResult Order = new XmlSlurper().parse(reader)
GPathResult orderNo = Order.Header.OrderNumber
GPathResult firstQty = Order.Item[0].Quantity
```

Listing 4.2 Usage of GPath Expressions to Access XML Document Elements

Both `orderNo` and `firstQty` are instances of `GPathResult`, which provides various methods to access the text of the node as a `String` or other types, such as an integer, `BigDecimal`, and so on. The following code snippet shows how you can further extract the order number as a `String` using the `text()` method and the quantity as an integer using the `toInteger()` method:

```
def orderNoAsText = orderNo.text()
def firstQtyAsInt = firstQty.toInteger()
```

Generating XML Documents

For generating XML documents using Groovy, we'll use `groovy.xml.MarkupBuilder`. `MarkupBuilder` provides a WYSIWYG approach to generating the output XML tree, which makes it easier compared to using DOM in Java.

To instantiate `MarkupBuilder`, you need an instance of a `Writer` object to write the output to a stream. Using the `MarkupBuilder` instance, you can start generating the XML tree. Listing 4.3 shows the generation of a two-level XML tree with `PurchaseOrder` as the root node with an `ID` as a child field. Finally, you use the `toString()` method of the `Writer` to retrieve the output XML document.

```
import groovy.xml.MarkupBuilder

Writer writer = new StringWriter()
def builder = new MarkupBuilder(writer)
builder.PurchaseOrder {
  'ID' 'ORD6001'
}
String output = writer.toString()
```

Listing 4.3 Generating XML Documents Using MarkupBuilder

When you execute Listing 4.3 in the Groovy Console, the result will be the following XML document:

```
<PurchaseOrder>
  <ID>ORD6001</ID>
</PurchaseOrder>
```

> **Tip**
>
> Groovy also provides a streaming approach to generating XML documents with `groovy.xml.StreamingMarkupBuilder`, which uses a similar approach to build the XML tree. `StreamingMarkupBuilder` is useful when generating large XML

documents due to the streaming nature, as well as producing compact payloads that aren't pretty-printed.

XML-to-XML Transformation

Now that you've learned how to parse and generate XML documents, you can combine these techniques to perform an XML-to-XML transformation in an SAP Cloud Platform Integration Groovy script.

The transformation example used here has the following requirements.

- Transform input XML structure to a different output XML structure.
- Generate output XML with indentation set to four spaces.
- Transform header date format.
- Filter only line items that are valid.
- Pad the item number with leading zeros.

Listing 4.4 shows the Groovy script performing this transformation using the input XML content from Listing 4.1. Following are the key aspects of the Groovy script.

- Input XML is parsed using `XmlSlurper` (with a `Reader` object).
- `MarkupBuilder` is instantiated using an `IndentPrinter` to set the indentation to four spaces.
- The output is defined by generating the XML tree.
- For direct assignment of the source node to target node, it's assigned without using the `text()` method, for example, `Order.Header.OrderNumber` to `ID`.
- The content of the source node is converted to `String` using the `text()` method when it's used for further data manipulation, such as date transformation.
- `LocalDate` and `DateTimeFormatter` are used to transform the date format.

- GPathResult's findAll(Closure) method is used to filter the line item by providing a closure with the filter criteria, for example, items with the Valid field equal to *true*.
- The each(Closure) method is used to iterate through all the items that were filtered.
- Leading zeros are added to ItemNumber using the padLeft(Number, CharSequence) method.

```
import com.sap.gateway.ip.core.customdev.util.Message
import groovy.xml.MarkupBuilder
import java.time.LocalDate
import java.time.format.DateTimeFormatter

def Message processData(Message message) {
  Reader reader = message.getBody(Reader)
  def Order = new XmlSlurper().parse(reader)
  Writer writer = new StringWriter()
  def indentPrinter = new IndentPrinter(writer, '    ')
  def builder = new MarkupBuilder(indentPrinter)

  builder.PurchaseOrder {
    'Header' {
      'ID' Order.Header.OrderNumber
      'DocumentDate' LocalDate.parse(Order.Header.Date.text(), Date
TimeFormatter.ofPattern('yyyyMMdd')).format(DateTimeFormatter.ofPat
tern('yyyy-MM-dd'))
    }
  def items = Order.Item.findAll { it.Valid.text() == 'true' }
  items.each { item ->
    'Item' {
      'ItemNumber' item.ItemNumber.text().padLeft(3, '0')
      'ProductCode' item.MaterialNumber
      'Quantity' item.Quantity
    }
  }
}

message.setBody(writer.toString())
```

```
  return message
}
```

Listing 4.4 XML-to-XML Transformation Using Groovy Script

To test this transformation, add the script provided in Listing 4.4, which was developed in IntelliJ IDEA, to the iFlow. You can then either test it at design time using the iFlow simulation tool or at runtime by first deploying the iFlow and then sending an HTTP request with the relevant payload to its end point. When you submit the input XML from Listing 4.1 and apply the transformation from Listing 4.4 to it, the message body after script execution will contain the output XML shown in Listing 4.5. Later, in Section 5, we'll look at how you can perform testing of such transformations locally on IntelliJ IDEA.

```
<PurchaseOrder>
    <Header>
        <ID>ORD60001</ID>
        <DocumentDate>2019-02-18</DocumentDate>
    </Header>
    <Item>
        <ItemNumber>010</ItemNumber>
        <ProductCode>MT70001</ProductCode>
        <Quantity>57</Quantity>
    </Item>
</PurchaseOrder>
```

Listing 4.5 Content of Sample Output XML Document

Note

From Groovy 3.0.x onward, refactoring has been done, and various classes in package `groovy.util` have been deprecated, such as `XmlSlurper`, `XmlParser`, and `GPathResult`. When the Groovy version in SAP Cloud Platform Integration is updated to 3.0.x in the future, the recommendation is to use the corresponding same classes in the `groovy.xml` package.

4.2 JSON

Now that you're familiar with the techniques that can be used to parse and produce payloads in an XML format, let's expand on that and discuss the transformations that involve another popular data format—JavaScript Object Notation (JSON).

In the following sections, let's reuse the example from the previous section and demonstrate how an equivalent input in a JSON format, which is provided in Listing 4.6, can be parsed and transformed.

```
{
    "Order": {
        "Header": {
            "OrderNumber": "ORD60001",
            "Date": "20190218"
        },
        "Items": [
            {
                "ItemNumber": "10",
                "MaterialNumber": "MT70001",
                "Quantity": 57,
                "Valid": true
            },
            {
                "ItemNumber": "20",
                "MaterialNumber": "MT80001",
                "Quantity": 28,
                "Valid": false
            }
        ]
    }
}
```

Listing 4.6 Content of a Sample Input JSON Document

Parsing a JSON Document

Similar to the `XmlSlurper` used to parse input in an XML format, Groovy provides a relevant `groovy.json.JsonSlurper` class that is accompanied with several types of parser implementations to parse input in a JSON format. It comes with the overloaded `parse(Object)` method that can be used to parse a JSON data structure from various sources, as well as the method `parseText(String)` to parse a JSON data structure contained in a `String`.

Although the API of `JsonSlurper` that is used to parse data structures looks very similar to the API of `XmlSlurper`, the underlying parsing mechanics are very different between these two: while `XmlSlurper` parses the XML data structure into `GPathResult` instances, `JsonSlurper` parses the JSON data structure into Groovy collections (`Map` for JSON objects and `List` for JSON arrays) and primitive types (`String`, `Integer`, `Boolean`, etc.). This drives a difference in methods to access parsed data structures' elements—as it will be illustrated later in this section, you can use dot notation (dotted expressions) to access elements of the parsed JSON data structure without calling methods such as `text()` or `toInteger()` to access corresponding elements' values.

In addition, in contrast to XML documents that must have a single root element (node), JSON documents can have one or multiple elements at the upmost level. As a consequence, `XmlSlurper` returns a reference to the root node of the parsed XML data structure, and `JsonSlurper` returns a reference to the entire parsed JSON data structure as an entry point.

Generating JSON Documents

To produce JSON documents, Groovy offers several options:

- `groovy.json.JsonOutput` is used for simple serialization of a Groovy object into a `String` that contains a compacted or pretty-printed JSON representation.
- `groovy.json.JsonGenerator` is used for customized serialization of a Groovy object into a JSON representation. The `JsonGenerator` instance

can be configured using various settings that are provided by `JsonGenerator.Options` builder, such as to exclude certain fields, configure date format, or use custom converters.

- `groovy.json.JsonBuilder` (and its streaming-enabled variation `groovy.json.StreamingJsonBuilder`) is used for advanced generation of JSON documents. It follows principles and syntax that are common for other builders, for example, `MarkupBuilder`, described earlier.

`JsonOutput` and `JsonGenerator` allow serialization to a `String`, whereas `JsonBuilder` also supports serialization to a `Writer`. `JsonOutput` and `JsonBuilder` can be used to serialize JSON data structures into not only a compacted but also a pretty-printed `String`.

In this section, we use `JsonBuilder` because it provides feature-rich and comprehensive functionality to generate the output JSON document. In addition, it will parse the earlier illustrated input JSON document and will apply the same transformation logic as demonstrated in the previous section to generate the output JSON document.

Note on Groovy Version and JsonGenerator Availability

At the time of writing, SAP Cloud Platform Integration used Groovy version 2.4.12; given that `JsonGenerator` was introduced later in Groovy version 2.5.0, this option isn't available at SAP Cloud Platform Integration runtime yet.

JSON-to-JSON Transformation

For demonstration purposes, we parse and generate JSON documents in the following example. Listing 4.7 illustrates the Groovy script that implements a corresponding transformation.

Following are the key aspects of the Groovy script:

- The message body that contains a JSON data structure is accessed with the help of a `Reader` and is parsed using `JsonSlurper`.
- `JsonBuilder` is instantiated and further used for generation of the output JSON data structure.

- Header fields are generated using direct assignment of the source field value for the `ID` field and converted date format using `LocalDate` and `DateTimeFormatter` for the `DocumentDate` field. We use dotted expressions to access corresponding fields and their values in the input data structure.
- The `findAll(Closure)` method is used to filter line items, where the item's `Valid` field's value equals to *true*.
- The `collect(Closure)` method is further used to iterate through the collection of filtered line items, so that corresponding line items can be generated in the output data structure.
- Because you want to ensure that the `Items` element in the output data structure is represented as an array, use square brackets ([ ]) to make that explicit to the JSON generator. Corresponding line items' fields are generated using Groovy's map literal syntax.
- Leading zeros are added to the `ItemNumber` field's value using the `padLeft(Number, CharSequence)` method so that the length of the output field is three characters.
- The output JSON data structure is passed to the body of the message. Here, use the `toPrettyString()` method of `JsonBuilder` to get a pretty-printed JSON document.

```
import com.sap.gateway.ip.core.customdev.util.Message
import groovy.json.JsonBuilder
import groovy.json.JsonSlurper
import java.time.LocalDate
import java.time.format.DateTimeFormatter

def Message processData(Message message) {
  Reader reader = message.getBody(Reader)
  def input = new JsonSlurper().parse(reader)

  def builder = new JsonBuilder()
  builder.PurchaseOrder {
    'Header' {
      'ID' input.Order.Header.OrderNumber
```

```
    'DocumentDate' LocalDate.parse(input.Order.Header.Date,
    DateTimeFormatter.ofPattern('yyyyMMdd'))
      .format(DateTimeFormatter.ofPattern('yyyy-MM-dd'))
}
def items = input.Order.Items.findAll { item -> item.Valid }
'Items' items.collect { item ->
    [
        'ItemNumber' : item.ItemNumber.padLeft(3, '0'),
        'ProductCode': item.MaterialNumber,
        'Quantity'   : item.Quantity
    ]
}

    message.setBody(builder.toPrettyString())
    return message
}
```

Listing 4.7 JSON-to-JSON Transformation Using Groovy Script

The script can be tested in a similar way to how you performed testing of the script that implemented XML-to-XML transformation in the previous section. To test the script, use the sample input JSON document contained in Listing 4.6 and expect the transformed document provided in Listing 4.8 to be produced.

```
{
    "PurchaseOrder": {
        "Header": {
        "ID": "ORD60001",
        "DocumentDate": "2019-02-18"
    },
    "Items": [
        {
            "ItemNumber": "010",
            "ProductCode": "MT70001",
            "Quantity": 57
        }
    ]
```

```
        }
}
```

Listing 4.8 Content of Sample Output JSON Document

Note on Compacted and Pretty-Printed Versions

The output produced in this example is a pretty-printed representation of a JSON document. A compacted (or minified) JSON document is normally smaller in size as new line characters and indents are removed from the output; for that reason a compacted version is preferred when transmitting large payloads. Nevertheless, we use a pretty-printed version here and in subsequent sections because it's more readable and convenient when performing expected and actual payloads comparison, for example, when testing Groovy scripts and transformations implemented using them.

4.3 Other Data Formats

Although XML and JSON are examples of predominant data formats that are commonly encountered in integration scenarios, as it was mentioned earlier, SAP Cloud Platform Integration is based on the integration framework that is payload agnostic and that supports any type of data that can be stored in the message payload.

Message payload—regardless of its data format—can be processed as a raw byte stream, but if payload content can be presented as a character stream, this opens possibilities for using a variety of techniques and tools to parse and generate corresponding data structures in a more convenient way. Therefore, when working with different data formats, it's essential to determine if those are character (text) or binary formats.

Next, we'll explore whether there are any libraries that can be used to manipulate data in those formats. For some data formats, Groovy provides native support, whereas for others, appropriate third-party libraries can be used for parsing and generation of relevant payloads. In the latter case, archives that contain corresponding libraries need to be imported to the iFlow as resources to allow resolution of required dependencies at runtime.

For text data formats, it's also worth verifying the character set (a combination of supported characters) and character encoding (a combination of representations of supported characters into sequences of bytes) that will be used because the same character can be encoded using different sequences of bytes. By default, SAP Cloud Platform Integration uses character encoding UTF-8, but this can be overwritten in the iFlow with the help of a corresponding header or a property, that is, `CamelCharsetName`. Moreover, the character encoding can be set programmatically in the Groovy script when reading character data from an input stream using `Reader` or writing character data to an output stream using `Writer` when the encoding is different from the platform's default.

Let's now take a look at a few examples to see how data in some other formats can be processed. For consistency, we'll reuse the earlier provided example of input in a JSON format provided in Listing 4.6. We'll focus on the document's line items data contained in the input payload and will use the same transformation logic as described earlier. Therefore, we'll omit the document's header data to keep further illustrations more compact.

We'll walk through examples to see how transformation logic can be implemented using several third-party libraries, and we encourage you to test them, which you can do similar to the testing of XML-to-XML and JSON-to-JSON transformations earlier in this section. Be mindful about representation of binary data (e.g., a PDF document), as some tools might not present it in the way and form that can be easily digested unless and until data is saved to a file of the appropriate format or rendered in the tool that is capable of processing a corresponding data format. As an alternative, you might want to combine transformation to a binary data format with the binary-to-text encoding technique that will help to obtain an output in text format and process (decode, save, display content) it further.

Text Data Format Example: Comma-Separated Values

A CSV format is popular when it comes to representation of tabular data. SAP Cloud Platform Integration already comes with standard converter

steps that can be used to convert payload in a CSV format into an XML or a JSON format and to convert payload in an XML or a JSON format into a CSV format, but let's see how we can manipulate data in a CSV format using Groovy.

There is no built-in support for parsing or producing documents in a CSV format in Groovy using a relevant abstraction layer, but third-party libraries can be used to compensate for that. In the following example, we'll use the Apache Commons CSV library (http://commons.apache.org/proper/commons-csv/) that can be used for reading from and writing to CSV and that provides extensive customizing options to set the format of a parsed CSV input or a produced CSV output. The library is available in public artifact repositories such as Maven Central and can be searched there and fetched from there. For example, for version 1.8, which is the latest at the time of writing, the corresponding dependency is *org.apache.commons:commons-csv:1.8*.

Let's see how a CSV output can be produced for line items of the transformed document. For this, use line items of the parsed input document, and add a header line to the output.

Listing 4.9 illustrates a Groovy script that generates a corresponding CSV output to a `StringWriter`, which is further converted to a `String` representation using the `toString()` method of a `StringWriter` and set to a message body.

```
import com.sap.gateway.ip.core.customdev.util.Message
import groovy.json.JsonSlurper
import org.apache.commons.csv.CSVFormat
import org.apache.commons.csv.CSVPrinter

def Message processData(Message message) {
  Reader reader = message.getBody(Reader)
  def input = new JsonSlurper().parse(reader)

  def items = input.Order.Items.findAll { it.Valid }
```

```
Writer writer = new StringWriter()
String[] headers = ['ItemNumber', 'ProductCode', 'Quantity']
CSVFormat format = CSVFormat.DEFAULT.withHeader(headers)
CSVPrinter csv = new CSVPrinter(writer, format)

items.each { item ->
  csv.printRecord(
      item.ItemNumber.padLeft(3, '0'),
      item.MaterialNumber,
      item.Quantity
  )
}

  message.setBody(writer.toString())
  return message
}
```

Listing 4.9 JSON-to-CSV Transformation Using Groovy Script

Binary Data Format Example: PDF

Let's now make use of some binary data format, for example, the portable document format (PDF), which is a popular choice for electronic documents. Groovy doesn't provide out-of-the-box support for processing PDF documents, but there are third-party libraries that can be used, such as iText PDF (https://itextpdf.com/) and Apache PDFBox (https://pdfbox.apache.org/). The following example uses the iText PDF library, which consists of multiple modules that can be downloaded together in a bundle or individually. Per general recommendations from the vendor, the minimally required modules are `kernel`, `io`, and `layout`. Required modules can be searched in public artifact repositories. For example, for version 7.1.11, which is the latest version at the time of writing, add the following dependency and its transitive dependencies: *com.itextpdf:layout:7.1.11*.

Note and ensure that its transitive dependencies are also added, so that relevant and required `kernel` and `io` modules are fetched and added alongside the `layout` module. When developing using IntelliJ IDEA and maintaining the library as a Maven dependency, transitive dependencies can be fetched

together with the downloaded library. If the library is downloaded manually, make sure that all three modules (`layout`, `kernel`, and `io`) are downloaded. Because these libraries aren't provided by the runtime of SAP Cloud Platform Integration, they need to be added as resources of the iFlow before the iFlow can be tested in SAP Cloud Platform Integration.

We'll generate a PDF document that will contain a table with transformed line items but no sophisticated formatting options or other added elements—paragraphs with text, images. In addition, we won't adjust page layout to keep the example compact, but note that the library provides feature-rich capabilities to add various text and image elements to the created PDF document, to customize the document's pages, and to read the PDF document and extract data from it.

Listing 4.10 illustrates a Groovy script that generates a PDF formatted output to a `ByteArrayOutputStream`, which is further converted to a byte array representation using the `toByteArray()` method and set to a message body.

```
import com.itextpdf.kernel.pdf.PdfDocument
import com.itextpdf.kernel.pdf.PdfWriter
import com.itextpdf.layout.Document
import com.itextpdf.layout.element.Table
import com.itextpdf.layout.property.UnitValue
import com.sap.gateway.ip.core.customdev.util.Message
import groovy.json.JsonSlurper

def Message processData(Message message) {
Reader reader = message.getBody(Reader)
def input = new JsonSlurper().parse(reader)

def items = input.Order.Items.findAll { it.Valid }

OutputStream outstream = new ByteArrayOutputStream()
PdfWriter writer = new PdfWriter(outstream)
PdfDocument pdf = new PdfDocument(writer)
Document document = new Document(pdf)

Table table = new Table(UnitValue.createPercentArray(3))
```

```
table.useAllAvailableWidth()

table.addHeaderCell('Item number')
table.addHeaderCell('Product code')
table.addHeaderCell('Quantity')

items.each { item ->
  table.addCell(item.ItemNumber.padLeft(3, '0') as String)
  table.addCell(item.MaterialNumber as String)
  table.addCell(item.Quantity as String)
}
  document.add(table)
  document.close()

  message.setBody(outstream.toByteArray())
  return message
}
```

Listing 4.10 JSON-to-PDF Transformation Using Groovy Script

Binary-to-Text Encoding

In our earlier examples, we produced data in either text or binary format. There are certain use cases in which it's required to combine binary and text data within a single message. For example, you may have to transmit a message that contains document data in a machine-readable structured data format (e.g., an XML or JSON format) that is accompanied with a human-readable original image of the document (e.g., scanned signed or certified printout of the document in a JPG or PNG format).

Such a requirement can be fulfilled in two ways:

- **Message with attachments**
  Such parts of content can be processed as separate objects within a message, where one part of the content (e.g., data in a structured data format) is placed in the message payload, and others (e.g., a binary image) are contained in message attachments. A message can have several attachments, and each attachment can contain data in a different data format. Message

attachments can be accessed using corresponding getter and setter methods of a `Message` that are a part of the Script API of SAP Cloud Platform Integration, and content from individual attachments can be further retrieved or set by iterating through a collection of attachments or accessing the specific attachment by name.

Specifically, you can use the `getAttachments()` method of the `Message` interface to retrieve a map of named attachment objects, and for each available attachment, you can then retrieve its content that is wrapped into an instance of `javax.activation.DataHandler`. Similarly, it's possible to add attachments to the message by creating a map of `DataHandler` objects and adding them to a message using the `setAttachments(Map)` method of the `Message` interface.

- **Payload with binary to text encoded data**
  Binary data is converted and processed in the form of a character sequence and is embedded into payloads in text data formats.

While the concept and principles of developing transformations applied to data that is stored in attachments don't differ much from those applied to the message payload as described earlier, let's delve a little deeper into the binary-to-text encoding technique.

Groovy—as well as many other programming languages—natively supports encoding of binary data into plain text (into an ASCII string format). Several binary-to-text encoding schemes can be used for this purpose, and one of the most commonly used today is Base64.

SAP Cloud Platform Integration provides standard functionality that can be used to add Base64 `Encoder` and `Decoder` steps to encode the entire message content to Base64 format or decode it from Base64 format, respectively.

Here, in contrast, our intention isn't to demonstrate how the entire content can be encoded or decoded, but to illustrate how this technique can be applied to parts of content. Java comes with the relevant class—`java.util.Base64`—that can be used to instantiate `Encoder` and `Decoder` and execute binary to text encoding or decoding operations using the Base64

scheme. With the help of the `org.codehaus.groovy.runtime.EncodingGroovyMethods` class, Groovy extends several traditional classes with additional methods to simplify Base64 encoding and decoding operations: corresponding encoding methods can be used to encode a byte array to a `Writable` object that holds text (which is a text output of Base64-encoded binary data), and the inverse operation of decoding can be applied to a `String` to decode it back to a byte array. These methods are neutral to a source of binary data—as far as binary data can be read from the source into a byte array, this technique can be applied.

Let's assume that within the transformation in a Groovy script, a byte array was initialized (let it be `byte[] imageBinary`) and populated with some binary data—it can be, for example, in an image of a scanned document that was sent by a sender system or fetched from some data source. Using Base64 encoder, you can encode that binary data and obtain a `String` with the encoded representation of binary data:

```
String imageBase64 = imageBinary.encodeBase64().toString()
```

Output to a `String` is provided just for demonstration purposes here. We could have used some other `Writable` object as an alternative; for example, we could have issued an output to a `Writer`. What's important here is that we now get a `String` whose value can be used in subsequent transformation steps; for instance, it can be used as a part of the generated output data structure in text format (e.g., assigned as a value to some field in the produced JSON or XML formatted output). In such a way, it's possible to generate an overall output that contains document data in a structured data format and embed encoded binary data into the same output structure. Some data formats provide a specific data type to emphasize and highlight data content and semantics for elements that are designated for Base64 string storage. For example, XML Schema Definition (XSD), used to describe XML documents, defines a dedicated built-in primitive data type—`base64Binary`—for this purpose. In other data formats where such a dedicated data type is missing, elements of a `String` type are commonly used to store Base64 strings—that are, for example, relevant for JSON.

This example demonstrates how binary data can be encoded to a string, but it's also possible to make an opposite operation. Given some binary data has been encoded into a string, sent to the iFlow in SAP Cloud Platform Integration, and is processed and transformed by a Groovy script, you can parse corresponding input in text data format, retrieve a string that contains Base64-encoded data, and decode it to a byte array. Let's consider the earlier example that demonstrated how you can apply Base64 encoding to a binary input, and see how you can implement the inverse operation and decode earlier encoded binary data back to a byte array from a `String` that stores Base64-encoded data (variable `imageBase64`):

```
byte[] imageBinary = imageBase64.decodeBase64()
```

After a byte array that stores a binary representation of an image is obtained, the required subsequent steps can be applied to it, such as setting it to a message payload, adding it as an attachment, or storing it further using some other available and relevant technique.

4.4 Custom Functions in Message Mapping

Previously in this section, we looked at how Groovy scripts can be developed and used to implement the entire transformation logic for the processed message. In the described examples, Groovy scripts are required to parse the input payload, apply transformation logic, and create an output payload of the message. This approach offers a great level of flexibility and can be used to implement transformations of any level of complexity. In contrast, there are use cases when the required transformation logic is relatively simple. For these, you can consider using graphical message mappings instead of transformations implemented entirely using Groovy scripts.

Note that message mappings are applied to XML messages. For XML-to-XML transformations that are implemented using the message mapping step, SAP Cloud Platform Integration comes with the Mapping Editor, which can be used to graphically visualize structures of source and target messages,

mappings, and rules for target elements. Although integration developers don't need to develop the entire Groovy script, they can still develop Groovy functions, add them to the message mapping, and use them together with standard mapping functions in mapping rules.

In the message mapping context, developed Groovy functions are called *custom functions* in contrast to *standard functions* that are offered out-of-the-box by SAP Cloud Platform Integration. Custom functions can be grouped into scripts and can be created and developed using the built-in code editor of the Mapping Editor, or they can be assigned from script files that have to be added to the iFlow.

Although implementations of custom functions are diverse, there are two principal types:

- **Custom functions that use a single value of the context**
  Parameters of the function can include several arguments. Each argument represents a single value of one of the supported types. The return type of the function is a `String`, that is, a string representation of an output value of the function. The simplest form of such a function is provided in the following. The function implements no conversion logic and returns the same value as was passed to it, but it illustrates the described concept:

```
import com.sap.it.api.mapping.*
def String func(String arg1) {
  return arg1
}
```

- **Custom functions that use all values of the context**
  Similar to the previous type, the function can have several arguments. The fundamental difference is that each argument doesn't represent a single value, but the entire array of values contained in the context for the input field. Another difference is that such a function doesn't return a value; instead, output of the function has to be passed to an additional function parameter, an instance of `com.sap.it.api.mapping.Output`, which represents an array of output values or the entire output queue.

Along with adding values to the output queue, context changes can be added if needed using corresponding methods of the `Output` object. In the example that will be introduced later in this section and is depicted in Figure 4.1, you'll use a custom function of this type and get a closer look at its implementation later in this section.

Custom functions of both described types can also access some information that is exposed by the mapping runtime—for example, message headers and exchange properties—using the optional parameter, an instance of `com.sap.it.api.mapping.MappingContext`.

Let's look into this feature further and see how you can make use of Groovy in custom functions. We'll reuse an example that was introduced when discussing regular expressions in Section 2.4 and will build an iFlow around it. The iFlow expects an input XML message containing phone numbers and will produce an output XML message containing results of submitted phone numbers' validation using the earlier introduced regular expression. As with earlier examples, the iFlow can be called from the outside by sending the HTTP request to its end point, but the demonstration and testing can be expedited by using the iFlow simulation tool.

The iFlow used in this example is illustrated in Figure 4.1.

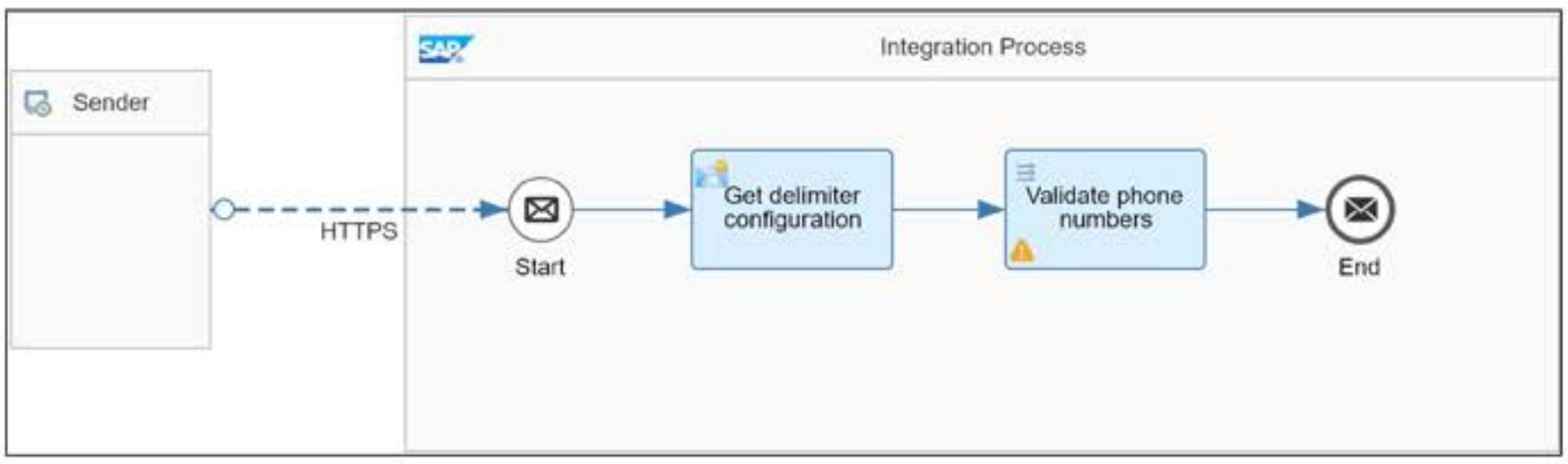


Figure 4.1 iFlow Used in the Custom Function in Message Mapping Example

The iFlow will consist of two message flow steps—content modifier and message mapping.

The **Content Modifier** step shown in Figure 4.2 is used to externalize a parameter for a delimiter and assign its value to an exchange property that

is further used in the custom function contained in the message mapping step. The delimiter is introduced to enable support of multiple phone numbers in a single XML node value. By default, the expected result is CSV.

Figure 4.2 Content Modifier Step to Set the Exchange Property from the Externalized Parameter

The **Message Mapping** step contains a simplified mapping logic that is required to produce the output message. A mapping rule for the target field `IsValid`, shown in Figure 4.3, consists of a single newly introduced custom function `isPhoneValid` that validates the array of values of the source field `PhoneNumbers`.

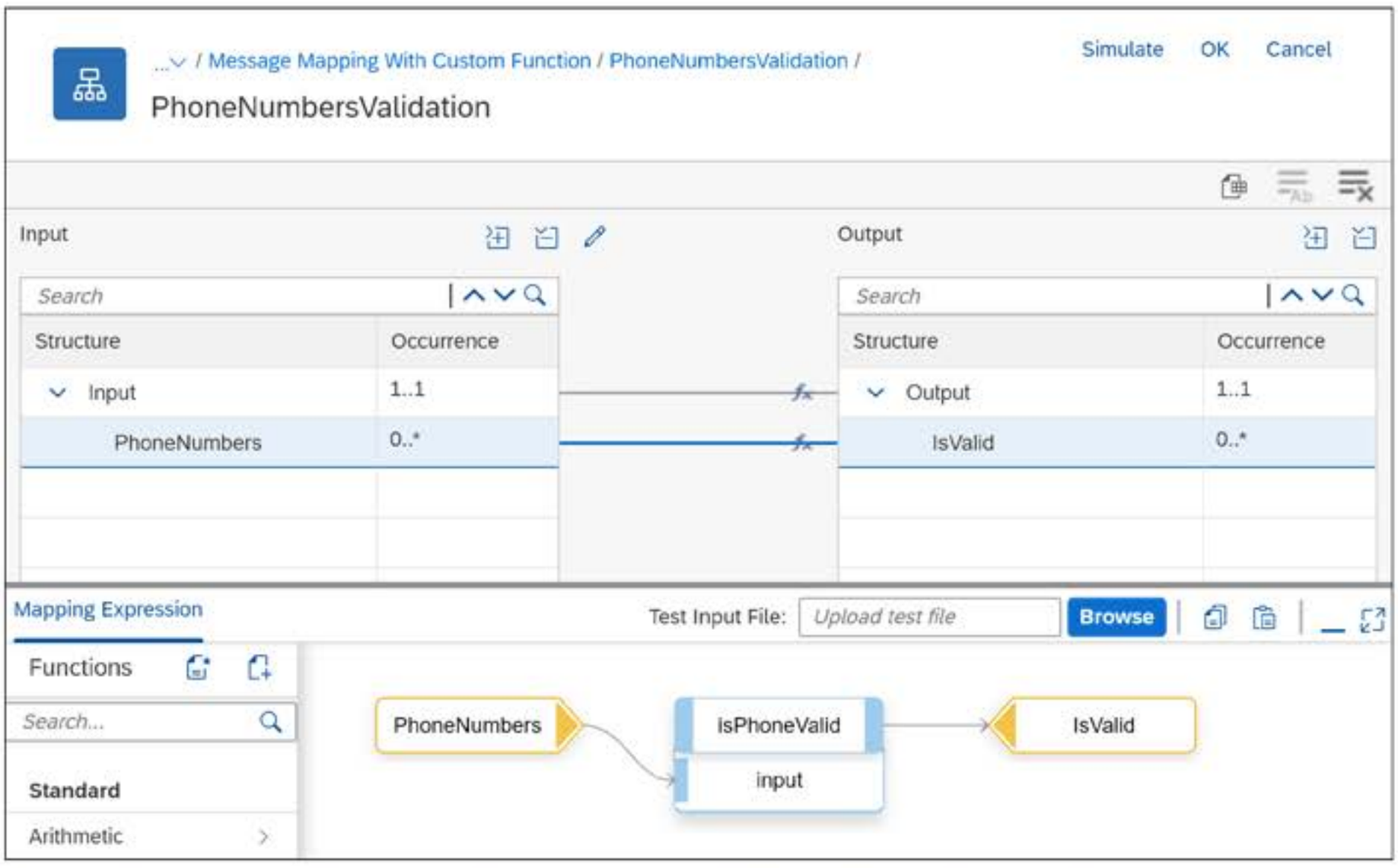


Figure 4.3 Message Mapping Step: A Mapping Rule with the Custom Function

To introduce the custom function, in the Mapping Editor, a new script named `Validators` has been created. In our example, the script contains only one custom function, `isPhoneValid`, which is provided in Listing 4.11.

```
import com.sap.it.api.mapping.*

def void isPhoneValid(String[] input, Output output, MappingContext
context) {

  def delimiter = context.getProperty('Delimiter') as String
  def pattern = /^\+(?:\d\u0020?){6,14}\d$/
  input.each { entry ->
    entry.tokenize(delimiter).each { phoneNumber ->
      output.addValue(phoneNumber ==~ pattern)
    }
  }

}
```

Listing 4.11 Custom Function Used in the Message Mapping Step

Note that the function makes use of the `MappingContext` object to access the earlier set exchange property `Delimiter`. Using an already familiar regular expression, the function iterates over the input argument, which is an array of string values that represents the queue of the source field `PhoneNumbers` and validates them. Where necessary and in case several phone numbers were submitted in a single field value, those are split using the determined delimiter. Results of validations of individual phone numbers are added to the queue of the target field `IsValid` using the `Output` object.

Let's put this mapping under test via the iFlow simulation tool. A simulation start point is added right before the message mapping step (after the content modifier step), and the end point is added right after the message mapping step. This setup allows you to unit test the message mapping alone and mock other required dependencies such as the exchange property `Delimiter`.

Input for the start point is as follows:

- The property `Delimiter` is added and assigned the comma value (,).
- Content of the XML document provided in Listing 4.12 is used as the body.

```
<Input>
  <PhoneNumbers>+6012 345 6789,+7 812 1234567</PhoneNumbers>
  <PhoneNumbers>+48 12 345 67 89 12345,44 1223 123456
  </PhoneNumbers>
</Input>
```

Listing 4.12 Simulation Input: Source Message Payload

After the simulation is run, its output will be similar to what you see in Figure 4.4. Given an input containing a combination of valid and invalid phone numbers, you can see occurrences of **true** (for valid phone numbers) and **false** values in the output.

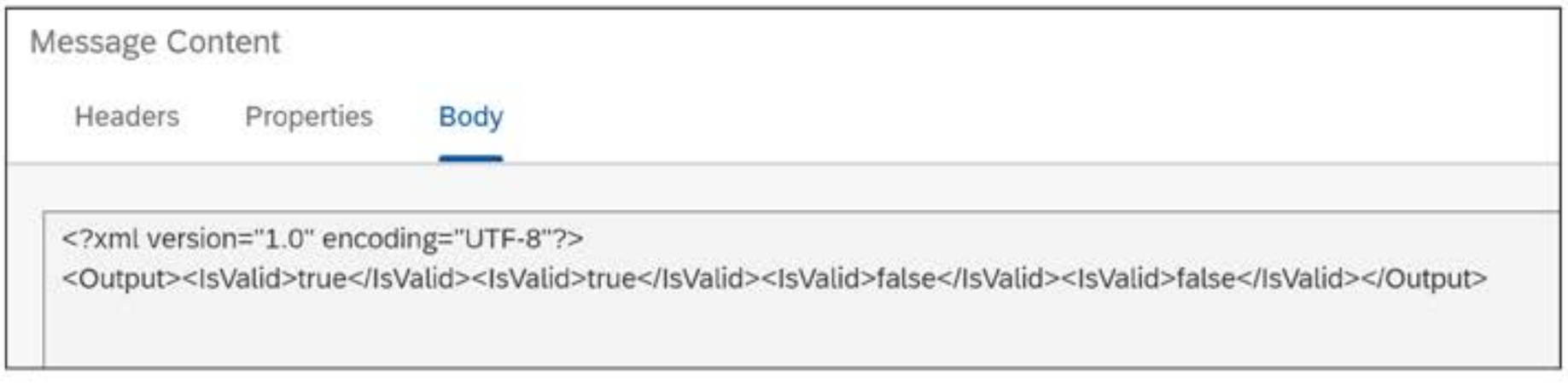


Figure 4.4 Simulation Output: Target Message Payload Created by the Mapping

5 Testing Groovy Scripts

In Section 3, you learned about using IntelliJ IDEA as a local IDE for developing Groovy scripts. This allows Groovy scripts to be developed faster with the aid of code completion and syntax check that IntelliJ IDEA natively provides. Although Section 3 also introduced you to the iFlow simulation tool, it still requires access to the SAP Cloud Platform Integration tenant to execute testing of Groovy scripts.

In this section, we'll dive into advanced techniques that allow you to test Groovy scripts without deployment into an SAP Cloud Platform Integration tenant. Using IntelliJ IDEA as a local environment for both development *and* testing of Groovy scripts significantly boosts productivity.

First, you'll create an additional component required for testing Groovy scripts locally. Then you'll learn how to create a testing program that will inject an input message to a Groovy script and display the contents of the output message produced by the script. Finally, you'll learn how *test-driven development* (TDD) can be achieved by using the open-source Spock framework when testing Groovy scripts. This advanced technique allows you to easily execute mass testing of multiple scenarios.

5.1 Mocking the Message Class

In Section 3, you saw that Groovy scripts in SAP Cloud Platform Integration use a `Message` interface as the container of a message being processed. The Script API library that contains the `Message` interface is publicly available, but that *definition* is only sufficient for development. For runtime execution, an *implementation* of the `Message` interface is required.

Typically, the approach would be to create a class that *implements* the `Message` interface. However, we'll deviate and create a mock `Message` class that replaces the `Message` interface in order to keep the class simple by just having methods that are frequently used, instead of implementing all the methods defined by the `Message` interface. You'll begin by creating a Groovy project, followed by the addition of Camel dependencies. Next, you'll create the mock `Message` class containing the most frequently used methods. Finally, you'll execute the build process to compile the class and generate a JAR artifact file.

After you have the JAR file containing the mock `Message` class, you can execute local testing of Groovy scripts.

Create New Project

In IntelliJ IDEA, start by creating a new Groovy project via the following steps:

1. In the menu bar, choose **File • New • Project**.
2. In the **New Project** window, select **Groovy** as the project type, and click **Next**.
3. Enter a suitable project name, for example, "sap-press-cpi-groovy-mock-msg", and click **Finish**.

Add a Dependency for Camel

After the project and its default module are created, you need to add Camel to the module's dependencies. Follow the steps described in Section 3.3 to add the following library from Maven as a dependency (as a global library): *org.apache.camel:camel-core:2.17.4*.

Warning

When copying the name of this and other libraries, be sure to not include the final period!

Note on Camel Version

In Section 6, you'll learn how to determine the Camel version used in the runtime of SAP Cloud Platform Integration, which is 2.17.4 at the time of writing.

Create Message Class

You can now create the mock version of the `Message` class, as follows:

1. First, create a new package. In the project directory structure, right-click on the **src** directory, and select **New • Package**. Enter "com.sap.gateway.ip.core.customdev.util" as the package name.

2. Next, create the mock `Message` class. Right-click on the newly created package, and then select **New • Groovy Class**. Enter "Message" as the class name.
3. Enter the code from Listing 5.1 in the newly created `Message` class, overwriting any existing generated code.

The code for the `Message` class is broken down into the following parts:

- Import declarations for Camel classes
- Declarations of private attributes
- Class constructor that takes in a Camel `Exchange` object as an input
- Methods to access the message body (`getBody(Class)` provides the type conversion capability, which will be further described in Section 6)
- Methods to access the headers of the message, as a whole `Map` object or as individual items
- Methods to access the properties of the message, as a whole `Map` object or as individual items

```
package com.sap.gateway.ip.core.customdev.util

import org.apache.camel.Exchange
import org.apache.camel.TypeConversionException

class Message {

  Exchange exchange
Object msgBody
Map<String, Object> msgHeaders
Map<String, Object> msgProps

Message(Exchange exchange) {
  this.exchange = exchange
}

public <Klass> Klass getBody(Class<Klass> klass)
        throws TypeConversionException {
```

```
    def body = this.exchange.getIn().getBody(klass)
    return body ?: null
}

Object getBody() {
    return this.msgBody
}

void setBody(Object msgBody) {
    this.msgBody = msgBody
}

Map<String, Object> getHeaders() {
    return this.msgHeaders
}

public <Klass> Klass getHeader(String name, Class<Klass> klass)
        throws TypeConversionException {
    if (!this.exchange.getIn().getHeader(name)) {
        return null
    } else {
        def header = this.exchange.getIn().getHeader(name, klass)
        return header ?: null
    }
}

void setHeaders(Map<String, Object> msgHeaders) {
    this.msgHeaders = msgHeaders
}

void setHeader(String name, Object value) {
    if (!this.msgHeaders)
        this.msgHeaders = [:]
    this.msgHeaders.put(name, value)
}

Map<String, Object> getProperties() {
    return this.msgProps
}
```

```
void setProperties(Map<String, Object> msgProps) {
  this.msgProps = msgProps
}

void setProperty(String name, Object value) {
  if (!this.msgProps)
    this.msgProps = [:]
  this.msgProps.put(name, value)
}

Object getProperty(String name) {
    return (this.msgProps) ? this.msgProps.get(name) : null
  }
}
```

Listing 5.1 Mock Implementation of Message Class

Build JAR Artifact

With the code in place, now you can build the JAR artifact. First, configure the project to build a JAR artifact, as follows:

1. In the menu, choose **File • Project Structure**.
2. In the **Project Structure** window, select **Project Settings • Artifacts**.
3. Click **+** in the middle pane, and then select **JAR • Empty**.
4. Enter a suitable name for the JAR, for example, "cpi-mock-msg".
5. Under **Available Elements**, expand the top-level element, right-click **'sap-press-cpi-groovy-mock-msg' compile output**, and select **Put into Output Root** (see Figure 5.1).
6. Click **OK** to finish the configuration.

Next, execute the build process from the main window, as follows:

1. In the menu, choose **Build • Build Artifacts**.
2. In the **Build Artifact** popup window, select the action **Build**.

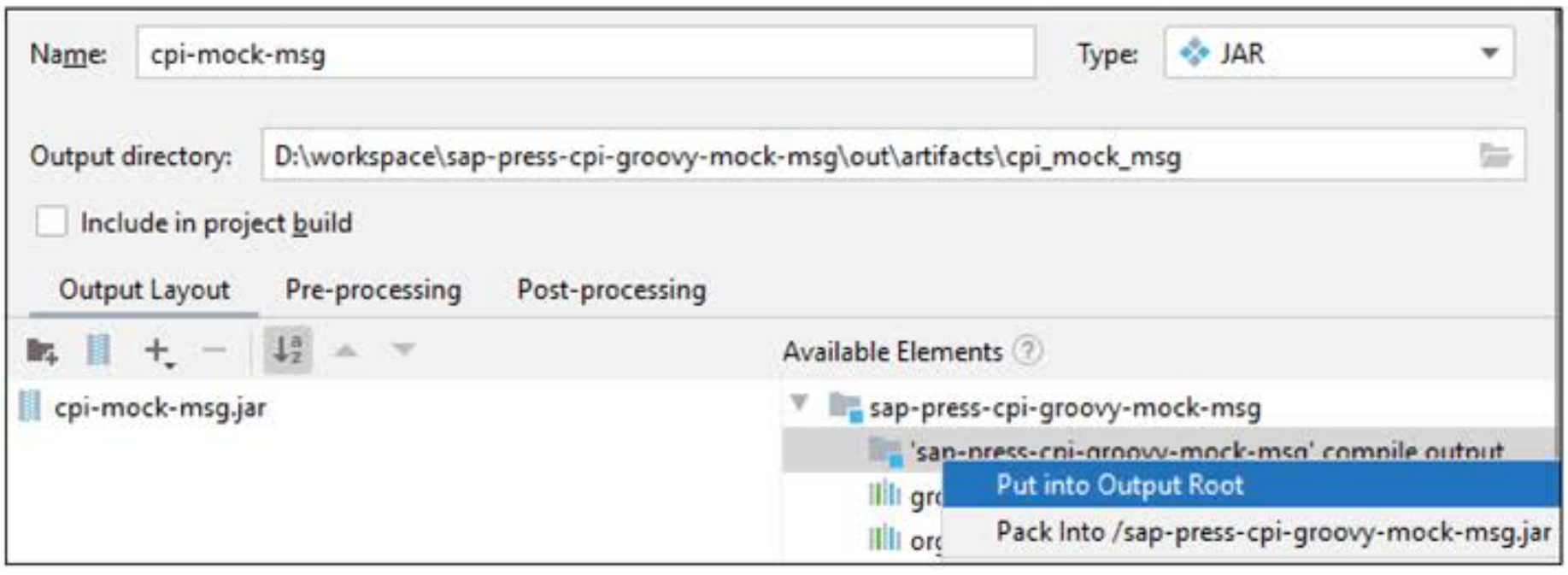


Figure 5.1 JAR Artifact Build Configuration

After the build has completed successfully, the JAR file will be generated in the output folder, as shown in Figure 5.2. Copy this JAR file to the local directory that was used in Section 3 to store the Script API library (e.g., *D:\lib\sap\cpi*). It will be used next in Section 5.2.

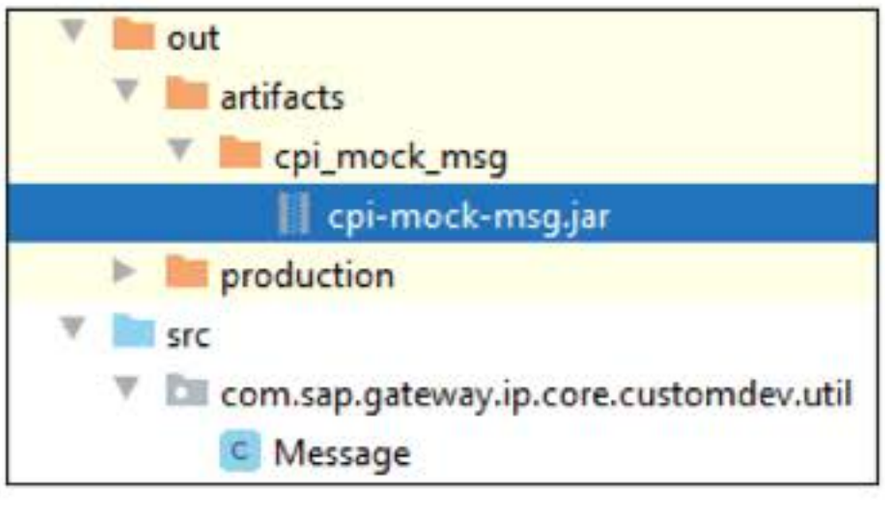


Figure 5.2 Generated JAR file with Mock Message Class

5.2 Testing Groovy Scripts Locally

After you have a library containing the mock `Message` class, it can then be added as a dependency into a project containing the Groovy script that you want to test.

You begin by creating a new Groovy project and setting up its directory structure and dependencies. Next, you add the input and output XML files to the project, and then create the Groovy script that will be tested. Lastly,

you create a testing program and execute it to perform the testing of the Groovy script.

For this illustration, we'll reuse the example of XML-to-XML transformation from Section 4.1. Therefore, the listings from that example (input/output XMLs and transformation script) won't be reproduced here.

Create a New Project

Create a new Groovy project in IntelliJ IDEA, and provide a suitable name, for example, "sap-press-cpi-groovy-testing".

Set Up the Directory Structure

The default project provides only a single *src* directory. You'll update the directory structure in the project structure, as follows:

1. In the menu bar, choose **File • Project Structure**.
2. In the **Project Structure** window, select **Project Settings • Modules**, and then select the **Sources** tab on the right pane.
3. First, in the directory structure in the lower pane, unmark the **src** directory as a **Source Folder**.
4. Next, using the context menu, build the directory structure as listed in Table 5.1.

An example of the complete directory structure is shown in Figure 5.3.

Directory	Additional Note
data/in	
data/out	
src/main/groovy	Mark as **Source Folder**.
src/test/groovy	Mark as **Test Source Folder**.

Table 5.1 Directory Structure for Groovy Testing Project

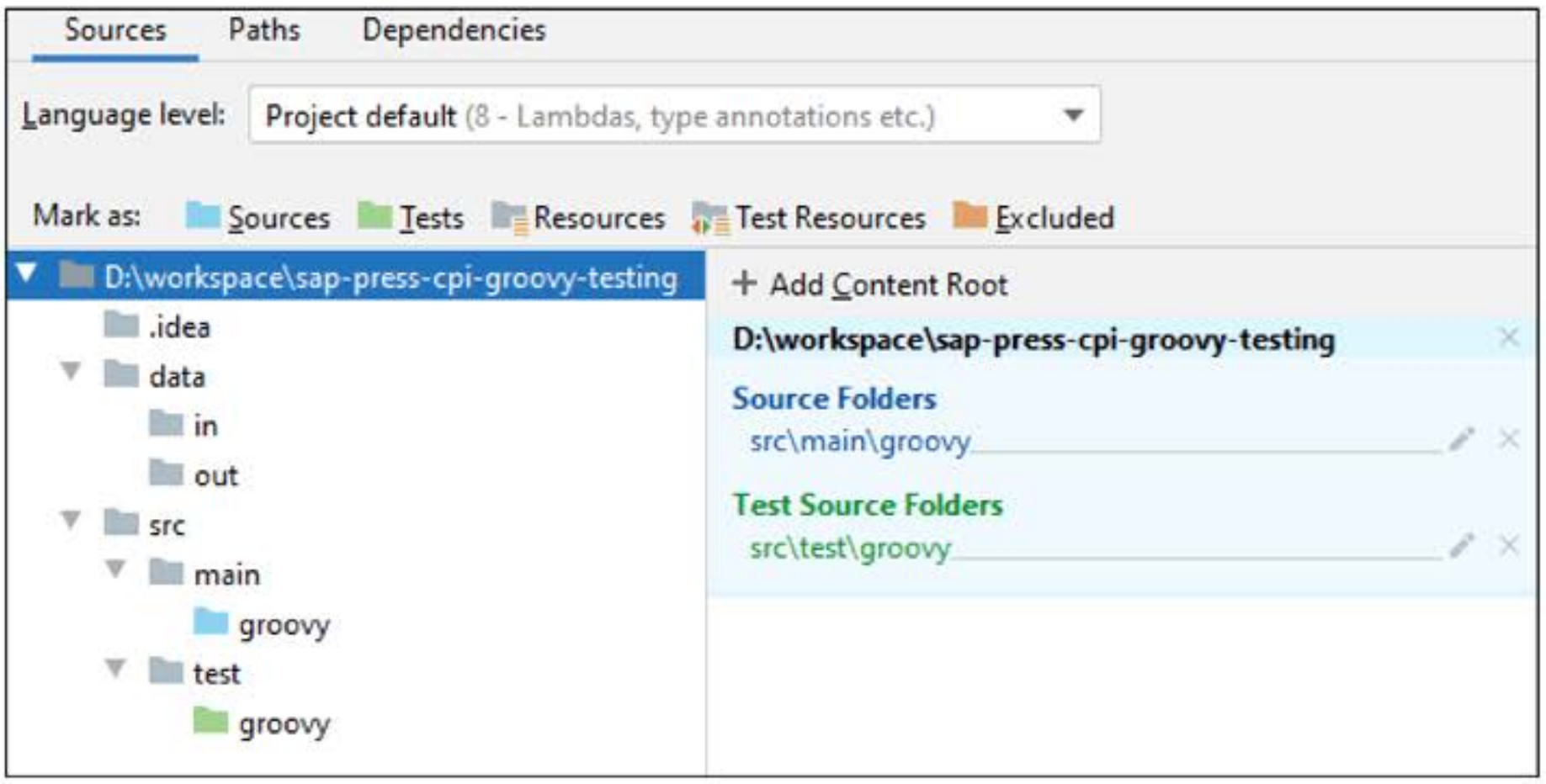


Figure 5.3 Directory Structure for Groovy Testing Project

Add Dependencies

Switch to the **Dependencies** tab, and add the following library (previously created as a global library) from Maven as a dependency: *org.apache.camel:camel-core:2.17.4*.

Next, go to **Platform Settings • Global Libraries**, create a new global library (of type Java) using the JAR file created previously (*cpi-mock-msg.jar*), and add it as a dependency to the module.

Click **OK** to apply the changes and return to the main window.

Warning on Message Version Conflict

If a project template (as described in Section 3) is used to create this Groovy project and the template that contains the Script API library as a dependency, the Script API library needs to be removed. This will prevent the conflict of having multiple versions of the `Message` interface/class in the project's dependency. If the Script API library isn't removed, the execution of the testing program can fail with the error **You cannot create an instance from the abstract interface 'com.sap.gateway.ip.core.customdev.util.Message'.**

Set Up Input and Output Files

Create new files in the following directories. Right-click on the directory, select **New • File**, and enter the file name as shown in Table 5.2.

Directory	File Name
data/in	*input1.xml*
data/out	*output1.xml*

Table 5.2 Directory of Input and Output Files

Enter the content of Listing 4.1 and Listing 4.5 in the corresponding newly created XML files, ensuring there are no additional lines added at the end of the files.

Create a Groovy Script

Next, you'll create the Groovy script that will be tested. Right-click on the **src/main/groovy** directory, and select **New • Groovy Script**. Enter a suitable name, for example, "XMLTransformation". Enter the code from Listing 4.4 in the newly created `XMLTransformation` script.

Create a Testing Program

Next, you'll create the testing program. Right-click on the **src/test/groovy** directory, and select **New • Groovy Script**. Enter a suitable name, for example, "TestingProgram". Enter the code from Listing 5.2 in the newly created `TestingProgram` script.

The code for the testing program is broken down into the following parts:

- Import declarations for `Message` class and `Camel` classes.
- Load the Groovy script under test using `GroovyShell`.
- Initialize the `CamelContext` and exchange for the message.
- Inject the input XML content into the input message using the `setBody(Object)` method of the `Message` class.

- Execute the `processData(Message)` method of the Groovy script under test.
- Display the contents of the message after it was processed by the Groovy script.

```
import com.sap.gateway.ip.core.customdev.util.Message
import org.apache.camel.CamelContext
import org.apache.camel.Exchange
import org.apache.camel.impl.DefaultCamelContext
import org.apache.camel.impl.DefaultExchange

// Load Groovy Script
GroovyShell shell = new GroovyShell()
Script script = shell.parse(new File(
  '../../../src/main/groovy/XMLTransformation.groovy'))

// Initialize CamelContext and exchange for the message
CamelContext context = new DefaultCamelContext()
Exchange exchange = new DefaultExchange(context)
Message msg = new Message(exchange)

// Initialize the message body with the input file
def body = new File('../../../data/in/input1.xml')

// Set exchange body in case Type Conversion is required
exchange.getIn().setBody(body)
msg.setBody(exchange.getIn().getBody())

// Execute script
script.processData(msg)
exchange.getIn().setBody(msg.getBody())

// Display results of script in console
println("Body:\r\n${msg.getBody(String)}")
println('Headers:')
msg.getHeaders().each { k, v -> println("$k = $v") }
println('Properties:')
msg.getProperties().each { k, v -> println("$k = $v") }
```

Listing 5.2 Testing Program to Test Groovy Script

Execute the Testing Program

In IntelliJ IDEA, Groovy script testing is executed using a *run configuration*. Run configurations can be created manually from the menu using **Run • Edit Configurations**, or automatically using the context menu by right-clicking the Groovy script and selecting **Run {ScriptName}**. The latter is the preferred choice as it automatically creates the run configuration and executes the testing at the same time.

Right-click on the **TestingProgram** script, and select **Run 'TestingProgram'**. This will run the testing program, which in turn will execute the Groovy script. The result of the test run appears in the console window on the lower pane. The contents of the body, headers (optional), and properties (optional) are displayed as shown in Figure 5.4. As expected, the output is the XML document with the same content as the *output1.xml* file.

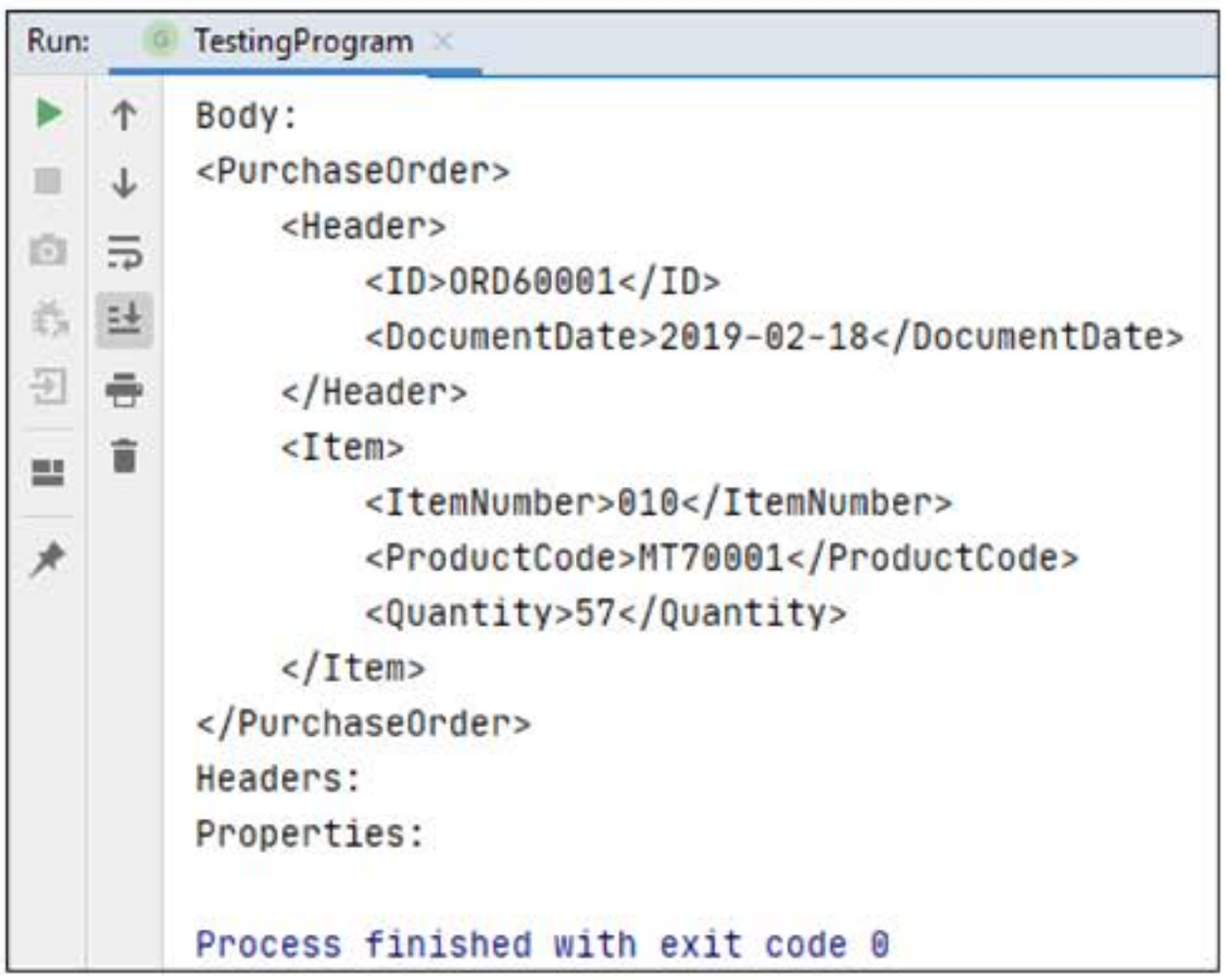


Figure 5.4 Console Output with Output XML Content

5.3 Test-Driven Development with Spock

Next, we'll look into using Spock (http://spockframework.org) to implement TDD and unit testing. This is an approach that will be beneficial even

to experienced developers, allowing them to further enhance their productivity. Unit tests are written as methods in a Spock *specification*, which is a Groovy class that extends `spock.lang.Specification`. A method is structured into blocks for the different phases of the test execution. Following are some of the commonly used phases in Spock.

- **`given`**
  Sets up the test.
- **`when`**
  Provides input for the test.
- **`then`**
  Sets the expected output of the test.

To showcase the benefit of Spock, we'll continue with our XML transformation example but with an additional new requirement. This allows us to demonstrate the TDD approach to handling requirements, as well as show how multiple scenarios can be easily unit tested in just a single execution.

Returning to our XML transformation, we add a new requirement that if there are no valid line items, set the new field `DocumentType` in `Header` segment of XML using the value from exchange property `DocType`.

Let's continue with the project from Section 5.2 by first adding dependency for Spock into the project. Next, you'll add additional input and output files related to the new requirement. With this in place, you'll create the Spock specification that contains tests for both scenarios. You'll initially execute the test without implementing any changes to the Groovy script for XML transformation. Finally, you'll introduce the required changes to the Groovy script and execute the Spock specification again to verify that both scenarios finally pass the tests.

Add a Dependency for Spock

Continue with the *sap-press-cpi-groovy-testing* project, and add the following library from Maven as a dependency (as a global library): *org.spockframework:spock-core:1.3-groovy-2.4*. Make sure to uncheck **Transitive**

dependencies to prevent downloading an additional Groovy version that can cause conflicts with the existing Groovy version used.

Spock has a dependency on JUnit for runtime execution. JUnit is included in the Groovy library distribution from Apache's archive repository. However, if the project's Groovy library is using the *org.codehaus.groovy:groovy-all:{version}* library from Maven (instead of the Groovy library distribution), then JUnit needs to be added separately. This is done by either including it as a transitive dependency of the Spock library or by adding the following library from Maven: *junit:junit:4.12*.

Note on Spock Version

The Spock version used needs to be compatible with the Groovy version. The version naming convention for Spock versions in Maven is *{spock_version}-groovy-{groovy_major_version}*. The version of Spock that supports Groovy major version 2.4 is Spock 1.3. Therefore, Maven version *1.3-groovy-2.4* is used. When the Groovy major version in SAP Cloud Platform Integration changes in the future, refer to the available versions in Maven (https://search.maven.org/artifact/org.spockframework/spock-core) to select a compatible Spock version.

Set Up Additional Input and Output Files

In addition to the existing XML files from the previous scenario, create new input/output XML files for the additional requirement in the following directories. Right-click on the directory, select **New** • **File**, and enter the file name as shown in Table 5.3.

Directory	File Name
data/in	*input2.xml*
data/out	*output2.xml*

Table 5.3 Directory of Additional Input and Output Files

Enter the content of Listing 5.3 and Listing 5.4 in the corresponding newly created XML files, ensuring there are no additional lines added at the end of the files.

```
<Order>
    <Header>
        <OrderNumber>ORD80002</OrderNumber>
        <Date>20200218</Date>
    </Header>
    <Item>
        <ItemNumber>10</ItemNumber>
        <MaterialNumber>MT90001</MaterialNumber>
        <Quantity>55</Quantity>
        <Valid>false</Valid>
    </Item>
</Order>
```

Listing 5.3 Content of Input XML File with No Valid Items

```
<PurchaseOrder>
    <Header>
        <ID>ORD80002</ID>
        <DocumentDate>2020-02-18</DocumentDate>
        <DocumentType>HDR</DocumentType>
    </Header>
</PurchaseOrder>
```

Listing 5.4 Content of Expected Output XML File

Create Spock Specification

Next, you'll create a Spock specification. Right-click on the **src/test/groovy** directory, and select **New • Groovy Class**. Enter a suitable name, for example, "XMLTransformationSpec". Enter the code from Listing 5.5 in the newly created `XMLTransformationSpec` class, overwriting any existing generated code.

The code for the Spock specification is broken down into the following parts:

- Import declarations for `Message`, `Camel`, and `Spock` classes
- Class definition (extending `Specification`) and declarations of attributes
- `setupSpec()` method (executed once before first unit test) to parse the script to be tested and instantiate the `CamelContext`

- `setup()` method (executed before each unit test) to instantiate the exchange and message
- Two unit test methods to test the different scenarios, which both set up the input (body and property) in the `given` block, execute the Groovy script in the `when` block, and verify the output in the `then` block

```
import com.sap.gateway.ip.core.customdev.util.Message
import org.apache.camel.CamelContext
import org.apache.camel.Exchange
import org.apache.camel.impl.DefaultCamelContext
import org.apache.camel.impl.DefaultExchange
import spock.lang.Shared
import spock.lang.Specification

class XMLTransformationSpec extends Specification {
  @Shared Script script
  @Shared CamelContext context
  Message msg
  Exchange exchange

  def setupSpec() {
    GroovyShell shell = new GroovyShell()
    script = shell.parse(new File(
      'src/main/groovy/XMLTransformation.groovy'))
    context = new DefaultCamelContext()
  }

  def setup() {
    exchange = new DefaultExchange(context)
    msg = new Message(exchange)
  }

  def 'Scenario 1 - Order has items'() {
    given: 'the message body is initialized'
    def msgBody = new File('data/in/input1.xml')
    exchange.getIn().setBody(msgBody)
    msg.setBody(exchange.getIn().getBody())

    when: 'we execute the Groovy script'
```

```
    script.processData(msg)
    exchange.getIn().setBody(msg.getBody())

    then: 'the output message body is as expected'
    msg.getBody(String) ==
      new File('data/out/output1.xml').text.normalize()
}

def 'Scenario 2 - Order does not have items'() {
    given: 'the message body and property are initialized'
    def msgBody = new File('data/in/input2.xml')
    exchange.getIn().setBody(msgBody)
    msg.setBody(exchange.getIn().getBody())
    msg.setProperty('DocType', 'HDR')

    when: 'we execute the Groovy script'
    script.processData(msg)
    exchange.getIn().setBody(msg.getBody())

    then: 'the output message body is as expected'
    msg.getBody(String) ==
      new File('data/out/output2.xml').text.normalize()
  }
}
```

Listing 5.5 Spock Specification to Test the XML Transformation

Execute Initial Unit Tests

Following the TDD approach, you'll next perform an initial execution of the unit tests prior to any further changes of the Groovy script under test. This will test the existing `XMLTransformation` script for both scenarios although it only has logic for the first scenario.

Similar to the earlier testing program, you execute the Spock specification using the context menu to automatically create the run configuration and execute the testing at the same time. Right-click on the Spock specification **XMLTransformationSpec**, and select **Run 'XMLTransformationSpec'**.

The lower pane will display the results of the Spock specification test run. As expected, the first scenario passes while the second scenario fails. Scroll to the end of the result (in the bottom-right pane), and click **<Click to see difference>**. This provides a comparison of the actual output (right side) against the expected output (left side), as shown in Figure 5.5. This is a useful view when checking for unit test failures. You can see that the failure is because the additional field `DocumentType` isn't in the actual output.

Figure 5.5 Spock Comparison View for Unit Test Failure

Complete Groovy Script and Execute Final Tests

Continuing with the TDD approach, you'll implement the minimum amount of changes required to cater for the additional requirement. Enter the code from Listing 5.6 in the existing `XMLTransformation` script, overwriting the existing code. The changes implemented are as follows:

- Move the position of the logic to find the valid items.
- Within the `Header` section, add the new field `DocumentType` if there are no items found. The field is populated with the value from exchange property `DocType`.

```
import com.sap.gateway.ip.core.customdev.util.Message
import groovy.xml.MarkupBuilder

import java.time.LocalDate
import java.time.format.DateTimeFormatter

def Message processData(Message message) {
  Reader reader = message.getBody(Reader)
```

```
  def Order = new XmlSlurper().parse(reader)
  Writer writer = new StringWriter()
  def indentPrinter = new IndentPrinter(writer, '    ')
  def builder = new MarkupBuilder(indentPrinter)

  def items = Order.Item.findAll { it.Valid.text() == 'true' }
  builder.PurchaseOrder {
    'Header' {
      'ID' Order.Header.OrderNumber
      'DocumentDate' LocalDate.parse(Order.Header.Date.text(), Date
TimeFormatter.ofPattern('yyyyMMdd')).format(DateTimeFormatter.ofPat
tern('yyyy-MM-dd'))
      if (!items.size())
        'DocumentType' message.getProperty('DocType')
    }

    items.each { item ->
      'Item' {
        'ItemNumber' item.ItemNumber.text().padLeft(3, '0')
        'ProductCode' item.MaterialNumber
        'Quantity' item.Quantity
      }
    }
  }

  message.setBody(writer.toString())
  return message
}
```

Listing 5.6 Final XML Transformation Groovy Script

Rerun the test for the Spock specification, and the final result will be displayed in the lower pane. Figure 5.6 shows an example of a successful execution; tests that passed show a checkmark.

Figure 5.6 Successful Execution of Spock Specification

As shown, we were able to implement changes to the Groovy script while ensuring that it didn't break any existing functionality. Additionally, you can see that using Spock, we were able to test multiple scenarios easily in just a single execution.

Tip on Changing the Indent Setting

The indent setting can also affect the result of the Spock specification test run. The indent in the XML output from the Groovy script is dependent on the indent value passed during instantiation of `IndentPrinter` in Listing 5.6. This should match the indent setting used in the expected output XML file. The examples in this section use an indent setting of four spaces.

The default indent setting for XML files in IntelliJ IDEA is four spaces. The following steps describe how this can be changed:

1. In the menu, choose **File • Settings**.
2. In the **Settings** window, expand the **Editor • Code Style** section. Settings for various file types are listed in the **Code Style** section.
3. Select **XML**, choose the **Tabs and Indents** tab, and change the value in the **Indent** field.
4. Click **OK** to apply the changes.

Next, for each of the XML files, apply the following steps to reformat the file with the new indent setting:

1. Double-click the XML file to open it.
2. In the menu, choose **Code • Reformat Code**.

6 Accessing SAP Cloud Platform Integration's Internal Frameworks

SAP Cloud Platform Integration is built on technologies from the open-source ecosystem. In this section, we'll explore two of these open-source technologies that form its underlying layers, namely, the Camel integration framework (https://camel.apache.org) and the OSGi framework (https://www.osgi.org) specifically.

In general, SAP Cloud Platform Integration provides an abstraction from these underlying layers, and you don't necessarily need to interact with them directly. Due to the abstraction, access to these frameworks isn't provided in SAP's documentation. Nevertheless, advanced knowledge of these components allows you to take advantage of their capabilities to craft creative and innovative solutions. Because these components are open-source Java-based frameworks, you can use Groovy to seamlessly access their publicly documented APIs.

This section begins with a brief overview of these frameworks. Subsequently, you'll learn how to use Groovy to access the Camel exchange and CamelContext within a message processed on SAP Cloud Platform Integration. Next, you'll learn about the Camel type conversion system and how it can be used for automatic conversion from one payload type to another. Finally, we'll discuss how to incorporate Camel's Simple Expression Language in Groovy scripts.

6.1 Runtime Architecture

After an iFlow is modeled in the SAP Cloud Platform Integration web UI, it's deployed to the runtime for message processing. The runtime node consists of multiple layers: at the top are SAP-specific components, in the middle are the integration and OSGi frameworks, and at the bottom are the JVM and operating system. Figure 6.1 shows the underlying layers of the runtime node in SAP Cloud Platform Integration.

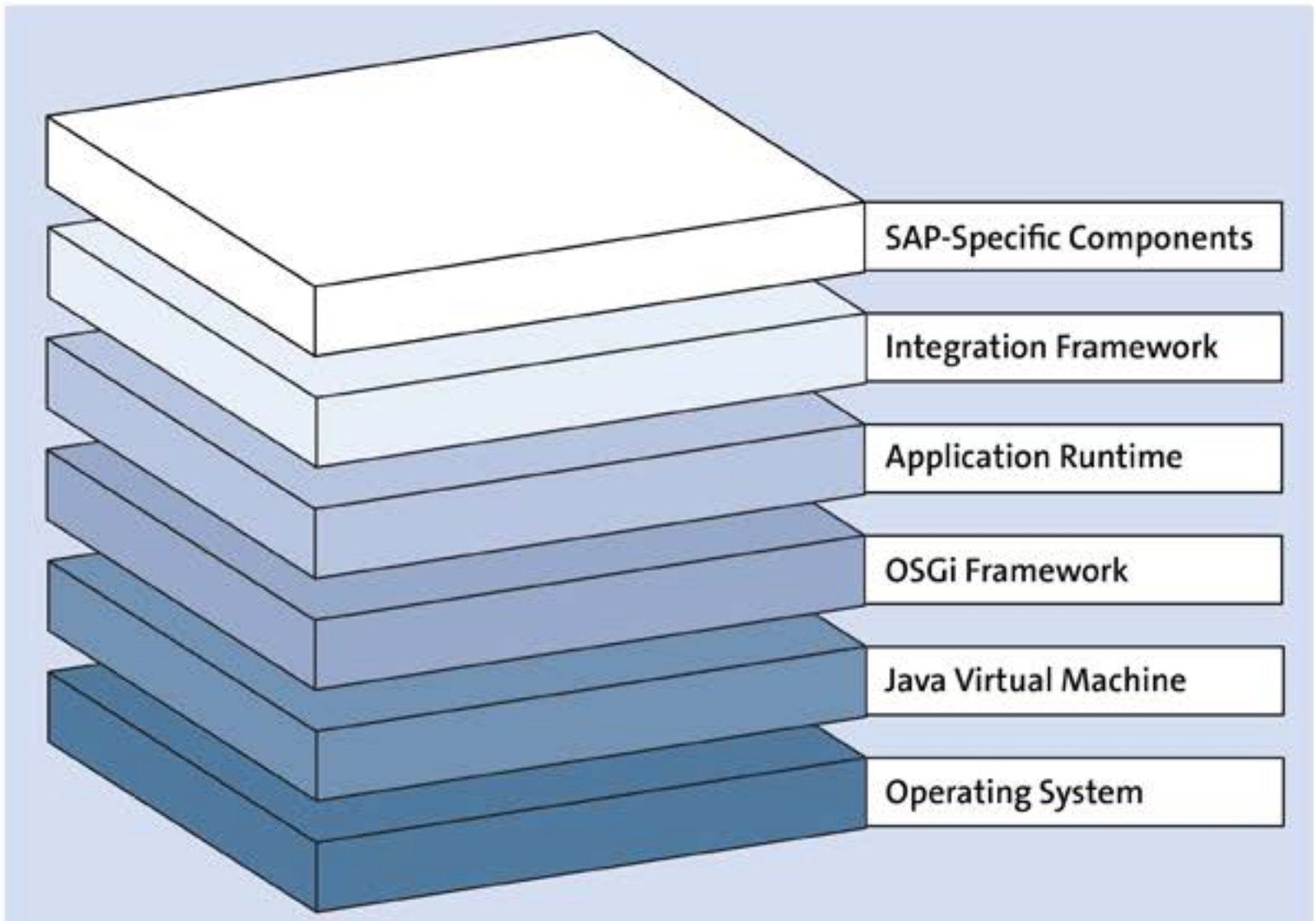


Figure 6.1 Underlying Layers of the Runtime Node

SAP Cloud Platform Integration uses Camel as its integration framework. When an iFlow is saved during design time in SAP Cloud Platform Integration, the graphical model is automatically converted to an XML file in the Business Process Model and Notation (BPMN) format, which is Camel agnostic.

Subsequently during the deployment process, the BPMN model is converted to a Camel *route*, which is a chain of processors applied on a message passing through the iFlow. Camel supports various domain-specific languages (DSL) for creating routes, for example, Java DSL, Spring XML, and Blueprint XML. SAP Cloud Platform Integration uses Blueprint XML, which allows the route to be declared using XML notation and to be executed in an OSGi environment. The contents of the iFlow together with the generated Blueprint XML route are combined into an OSGi bundle and deployed onto the underlying OSGi environment. At the time of writing, SAP Cloud Platform Integration uses Apache Karaf (https://karaf.apache.org) as the application runtime that runs on top of an OSGi framework using Apache Felix (https://felix.apache.org).

6.2 Accessing CamelContext with Groovy

In Section 5, you learned about the implementation of the `Message` class used during runtime execution of an interface message. This `Message` class wraps over a Camel *exchange*, which is Camel's message container. An exchange consists of in message and out message (optional), exchange ID, properties, and exception details. It's represented as interface `org.apache.camel.Exchange` in Java.

An exchange is processed within a *CamelContext*, the runtime container that keeps all the pieces together, such as routes, end points, components, and converters. In SAP Cloud Platform Integration, a CamelContext has a 1-to-1 relationship with a deployed iFlow.

To access the underlying Camel components, you can rely on a unique undocumented behavior of Groovy—the ability to access private attributes.

Listing 6.1 defines a Groovy script to access the Camel exchange and CamelContext from an SAP Cloud Platform Integration message. The Camel exchange is a private instance attribute of `Message` and can be accessed using dot notation `message.exchange`. From the exchange, you use the `getContext()` method to retrieve the CamelContext.

Both Camel exchange and CamelContext provide various methods to access different parts of the exchange or CamelContext. Listing 6.1 uses the following methods of the CamelContext to retrieve some information from the underlying Camel framework:

- `getName()`
  Provides the name/ID of the CamelContext. This is equivalent to the ID of the deployed iFlow.
- `getVersion()`
  Provides the version of Camel used by the CamelContext. This is also the version of the underlying Camel framework.

```
import com.sap.gateway.ip.core.customdev.util.Message
import org.apache.camel.CamelContext
```

```
import org.apache.camel.Exchange

def Message processData(Message message) {
  Exchange ex = message.exchange
  CamelContext ctx = ex.getContext()

  StringBuilder sb = new StringBuilder()
  sb << "CamelContext Name/ID: ${ctx.getName()}\r\n"
  sb << "CamelContext Version: ${ctx.getVersion()}\r\n"

  message.setBody(sb.toString())
  return message
}
```

Listing 6.1 Groovy Script to Access CamelContext

Use the iFlow simulation tool to execute Listing 6.1, as well as subsequent examples. Following are the results of the Groovy script execution:

```
CamelContext Name/ID: {iFlowName/GeneratedID}
CamelContext Version: 2.17.4-sap-29
```

Note on Camel Version

From the result, we find that the version of Camel used in SAP Cloud Platform Integration at the time of writing is 2.17.4-sap-29. This indicates the public version 2.17.4 of Camel is used with some additional SAP-specific modifications. For the purpose of local testing in IntelliJ IDEA, the public version of Camel should be used.

6.3 Camel Type Conversion System

Built into the core of Camel is a type conversion system that enables automatic conversion between well-known types. This feature supports Camel's payload-agnostic model, providing a noninvasive, under-the-hood capability to convert payload types.

Camel comes with more than 100 preloaded type converters, as well as support for the addition of custom type converters. In the following sections, you'll learn how you can explore the various type converters available in SAP Cloud Platform Integration.

Finding Available Type Converters

Listing 6.2 defines a Groovy script that lists all the type converters in SAP Cloud Platform Integration. You can achieve this using publicly documented APIs of the Camel framework. After you have access to the CamelContext, you use method `getTypeConverterRegistry()` to access the type converter registry. From this registry, you can get all the available type converters using method `listAllTypeConvertersFromTo()`, listing the class it converts from and the class it converts to. This is then stored in a `StringBuilder` in comma-delimited rows and saved into the message body.

```
import com.sap.gateway.ip.core.customdev.util.Message
import org.apache.camel.CamelContext
import org.apache.camel.Exchange
import org.apache.camel.spi.TypeConverterRegistry

def Message processData(Message message) {
  Exchange ex = message.exchange
  CamelContext ctx = ex.getContext()

  StringBuilder sb = new StringBuilder()
  sb << "FromClass,ToClass\r\n"

  TypeConverterRegistry registry = ctx.getTypeConverterRegistry()
  List<Class<?>[]> list = registry.listAllTypeConvertersFromTo()
  if (list) {
    list.each { converter ->
      sb << "${converter[0].getCanonicalName()},"
      sb << "${converter[1].getCanonicalName()}\r\n"
    }
  }
```

```
  message.setBody(sb.toString())
  return message
}
```

Listing 6.2 Groovy Script to List Available Camel Type Converters

When you execute Listing 6.2, the result is a comma-delimited list of all the type converters. At the time of writing, this list has more than 200 type converters, including some SAP custom type converters.

For the sake of brevity, Table 6.1 lists some of the available type converters without going through the entire list.

FromClass	ToClass
java.io.InputStream	byte[]
byte[]	java.io.Reader
javax.xml.soap.SOAPMessage	java.lang.String
java.io.InputStream	org.w3c.dom.Document
org.w3c.dom.NodeList	java.util.List
java.lang.String	java.util.TimeZone

Table 6.1 Sample List of Camel Type Converters

With this knowledge, you can take advantage of the built-in type converters in a Groovy script without writing excessive or complex logic. The `Message` interface contains the following two methods that provide native capability to use type conversion for the message body and headers.

- `getBody(Class)`
- `getHeader(String, Class)`

Type Conversion in Message Body

In Section 2, Section 4, and Section 5, you saw the use of the method `getBody(Class)` in various Groovy script examples. Conversion of the message

body using this method is supported by the underlying Camel type conversion system that we just discussed.

Because you've come across many Groovy scripts using type conversion for the message body in previous sections, there won't be a full Groovy script example in this part. Instead, we'll revisit an example on XML parsing to understand how type conversion works under the hood:

```
Reader reader = message.getBody(Reader)
def root = new XmlSlurper().parse(reader)
```

In these code snippets, the `getBody(Reader)` call enables you to retrieve the message body (typically an `InputStream` object) as a `Reader` object. This uses the underlying Camel type converter to convert from `InputStream` to `Reader`. With a `Reader` object available, you can then proceed to parse the XML in an efficient manner by passing it to the `parse(Reader)` method of `XmlSlurper`.

Type Conversion in Message Header

Besides the message body, SAP Cloud Platform Integration also supports type conversion for message headers via the `getHeader(String, Class)` method.

Listing 6.3 defines a Groovy script that retrieves the message header `X-TimeZone` with a type conversion to a `TimeZone` object. Then the `getDisplayName()` method is called to retrieve the long standard time name of the time zone.

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

def Message processData(Message message) {
  TimeZone tz = message.getHeader('X-TimeZone', TimeZone)
  message.setBody(tz.getDisplayName())
  return message
}
```

Listing 6.3 Groovy Script with Type Conversion on Message Header

Use the iFlow simulation tool to execute Listing 6.3, additionally providing the header `X-TimeZone` as input to the simulation. Figure 6.2 displays the simulation input with header **X-TimeZone** set to **UTC**.

Figure 6.2 Add X-TimeZone Header in Simulation Input

Following is the result of the Groovy script execution:

```
Coordinated Universal Time
```

6.4 Using Camel's Simple Expression Language

As mentioned earlier, Camel comes with the *Simple Expression Language* (Simple, for short). Simple enables evaluation of expressions on the current exchange being processed. It's widely used in the building of Camel routes and, therefore, appears in many components of SAP Cloud Platform Integration modeling steps, such as content modifiers, routers, and adapters.

Simple uses `${ }` placeholders for dynamic expressions. It was originally created for simple expressions used directly in Camel routes. For more complex needs in Camel, it's recommended to use a scripting language such as Groovy. However, the Camel framework provides `SimpleBuilder`, another publicly documented API that allows you to have the best of both worlds by using Simple expressions in Groovy scripts.

Listing 6.4 defines a Groovy script that evaluates the following Simple expressions:

- **`${camelId}`**
  Name of Camel ID (and also the iFlow).
- **`${exchangeId}`**
  ID of the exchange.
- **`${date:now:yyyy-MM-dd}`**
  Current date in yyyy-MM-dd format.
- **`${messageHistory(false)}`**
  Processing history of the exchange.

```
import com.sap.gateway.ip.core.customdev.util.Message
import org.apache.camel.Exchange
import org.apache.camel.builder.SimpleBuilder

def Message processData(Message message) {
  Exchange ex = message.exchange
  StringBuilder sb = new StringBuilder()

  def evalSimple = { simpleExpression ->
    SimpleBuilder.simple(simpleExpression).evaluate(ex, String)
  }

  sb << 'Camel ID: ' + evalSimple('${camelId}') + '\r\n'
  sb << 'Exchange ID: ' + evalSimple('${exchangeId}') + '\r\n'
  sb << 'Date: ' + evalSimple('${date:now:yyyy-MM-dd}') + '\r\n'
  sb << evalSimple('${messageHistory(false)}') + '\r\n'

  message.setBody(sb.toString())
  return message
}
```

Listing 6.4 Groovy Script to Execute Camel Simple Expressions

Tip

To avoid conflict and interpolation with Groovy strings that also use the `${ }` notation, Listing 6.4 doesn't use strings encapsulated in double quotation marks.

Figure 6.3 shows the results of the Groovy script execution using the iFlow simulation tool.

Figure 6.3 Results of Camel Simple Expressions in Groovy Script

7 Additional Resources

At last, we come to the final part of our journey through this E-Bite. However, it isn't the end of your journey in Groovy and SAP Cloud Platform Integration. Instead, it marks the beginning as you learn to master and unleash the power of Groovy for your SAP Cloud Platform Integration developments.

As we mentioned early on, it isn't possible to cover every minute aspect of developing Groovy scripts for SAP Cloud Platform Integration. We've covered most of the important aspects sufficient for you to start developing Groovy scripts for SAP Cloud Platform Integration. At this point, you should have a solid foundation to fuel further experimentation and learning.

Throughout this E-Bite, we've covered a wide range of topics. To continue your journey, we provide some links to online resources for your further learning. Some of these are resources that you'll revisit time and again as you continue to hone your skills in Groovy and SAP Cloud Platform Integration development.

- **Groovy** (https://groovy-lang.org)
  To do justice to Groovy, we must admit that we've only scratched the surface of this powerful yet easy-to-use language in this E-Bite. The best publicly accessible source of information for Groovy is its official site. A few of the topics that are useful are closures, differences between Groovy and Java, and the GDK enhancements. The GDK enhancements provide additional methods to the classes of JDK, thereby making them "Groovier" and easier to use. It's important to refer to the documentations of both JDK and GDK to get a holistic picture of these classes and their methods.
- **Camel** (https://camel.apache.org)
  It's impossible to avoid Camel when you're working on SAP Cloud Platform Integration. Therefore, a strong understanding of Camel is beneficial in improving your ability to develop robust, enterprise-grade integrations. Camel's official site provides a comprehensive guide to help you understand the key aspects of Camel, such as EIPs and the Simple Expression Language.
- **OSGi** (https://www.osgi.org) and **Karaf** (https://karaf.apache.org)
  Both OSGi and Karaf's official sites provide further details for you to understand the modular system of the underlying application runtime used in SAP Cloud Platform Integration. This allows you to explore more about the OSGi bundles that are generated and deployed during the iFlow deployment process in SAP Cloud Platform Integration.
- **IntelliJ IDEA** (https://www.jetbrains.com/idea/documentation/)
  IntelliJ IDEA is recommended as the IDE of choice for developing Groovy scripts for SAP Cloud Platform Integration. While we illustrated some productivity tips and tools when using IntelliJ IDEA, again these only scratched the surface of this feature-rich IDE. It contains other useful features such as code inspections, refactoring assistance, and version control management, which all can be explored further in its documentation. Additionally, IntelliJ IDEA can be further enhanced with a multitude of plug-ins that are available through its marketplace.

- **Maven** (https://maven.apache.org)
  In the world of software development, it isn't uncommon that a particular functionality is already available that you can reuse without "reinventing the wheel." A dependency management system allows you to reuse such libraries with ease. In the Java ecosystem, Maven is one such popular system alongside others such as Gradle (https://gradle.org). As shown in our examples, a good grasp of dependency management allows you to easily incorporate external libraries into your Groovy scripts to extend their functionalities.
- **Spock** (http://spockframework.org)
  When using Spock together with a TDD approach, it's possible to develop lean Groovy scripts and iron-out bugs early in the software development life cycle. However, Spock isn't just limited to comparison of the message body. Its other features, such as mocking/stubbing and data-driven testing, provide a solid framework for writing comprehensive testing specifications for Groovy script developments.
- **CPITracker** (https://twitter.com/cpitracker)
  Developed by Morten Wittrock, this is a tracker for various parameters and component versions within SAP Cloud Platform Integration. As SAP Cloud Platform Integration undergoes its regular rolling updates, this tracker provides updates in the form of Twitter feeds. It's especially useful to remain aware of changes to versions of Groovy or Camel (among other things) in SAP Cloud Platform Integration.
- **RealCore CPI Dashboard** (https://github.com/codebude/cpi-dashboard)
  Developed by Raffael Herrmann, this open-source dashboard allows you to monitor many aspects of an SAP Cloud Platform Integration tenant. The revolutionary aspect is that it's wholly developed as an iFlow covering both the frontend and backend. With many of its functionalities realized using Groovy scripts, it's an example of a development that pushes the boundary of what could be developed in an iFlow when it's backed with the power of Groovy.
- **Int4 IFTT** (https://int4.com/iftt)
  Your journey on SAP Cloud Platform Integration would not be complete if

we didn't cover the aspect of testing. Without comprehensive testing, it isn't possible to guarantee that developments are bug free and robust. Int4 IFTT is a feature-rich toolset covering many critical aspects of testing, such as regression testing, continuous testing, and TDD. One of its prominent features is allowing you to easily create test cases and execute them in a matter of minutes.

We hope that the content and resources provided throughout this E-Bite have enlightened you, spurred your imagination and creativity, and fueled your enthusiasm for your ongoing learning journey in Groovy and SAP Cloud Platform Integration.

8 What's Next?

Now that you know all that Groovy can do in SAP Cloud Platform Integration, it's time to learn even more! Whether you're looking to get the full picture on SAP Cloud Platform Integration or expand your programming skills into other areas, we've got the resources you need.

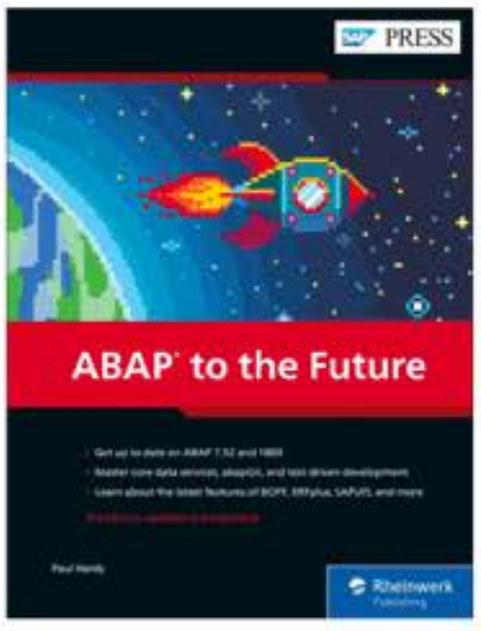
Recommendation from Our Editors

Set up and use SAP Cloud Platform Integration with *SAP Cloud Platform Integration: The Comprehensive Guide* by John Mutumba Bilay, Peter Gutsche, Mandy Krimmel, and Volker Stiehl! This book will teach you how to integrate processes and data in your system by developing and configuring integration flows.

Visit www.sap-press.com/4650 to check out *SAP Cloud Platform Integration: The Comprehensive Guide*!

In addition to this book, our editors picked a few other SAP PRESS publications that you might also be interested in. Check out the next page to learn more!

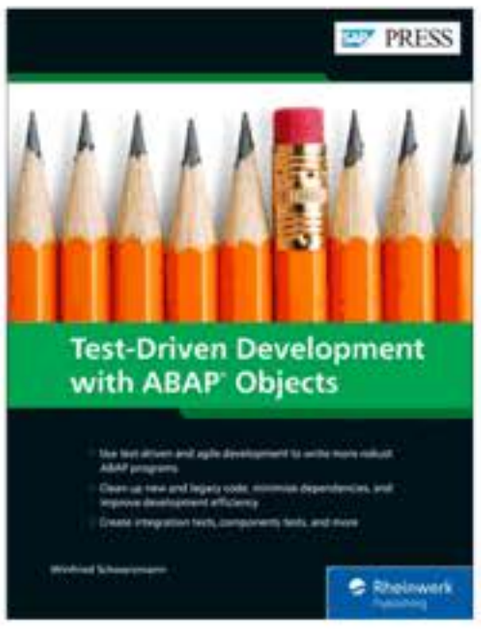
More from SAP PRESS



ABAP to the Future

Stay on the cutting edge of ABAP technology! Learn what's new with the latest ABAP releases—7.52 and 1809—and see what other SAP technologies are now bringing to the table. New to this edition: abapGit, ABAP SQL, the RESTful ABAP programming model, and test tools.

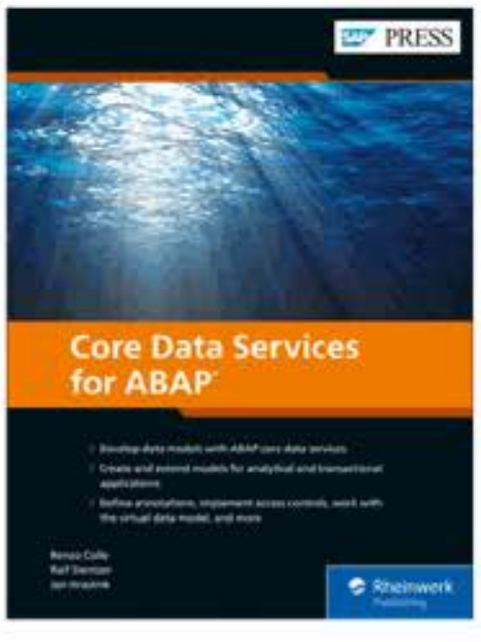
864 pages, 3rd edition, pub. 02/2019
E-book: $69.99 | **Print:** $79.95 | **Bundle:** $89.99
www.sap-press.com/4751



Test-Driven Development with ABAP Objects

From creating a robust test infrastructure to designing methods, classes, and packages that will stand the test of time, revitalize your programming. Whether you're writing new code or fixing legacy code, you'll learn to select test cases, use test doubles, and generate test data.

594 pages, pub. 07/2019
E-book: $79.99 | **Print:** $89.95 | **Bundle:** $99.99
www.sap-press.com/4882



Core Data Services for ABAP

Get the skills you need to create data models with in-depth information on CDS syntax, its key components, and its capabilities. Walk step-by-step through modeling application data in SAP S/4HANA and developing analytical and transactional application models

490 pages, pub. 05/2019
E-book: $69.99 | **Print:** $79.95 | **Bundle:** $89.99
www.sap-press.com/4822

SAP PRESS E-Bites

SAP PRESS E-Bites provide you with a high-quality response to your specific project need. If you're looking for detailed instructions on a specific task; or if you need to become familiar with a small, but crucial sub-component of an SAP product; or if you want to understand all the hype around product xyz: SAP PRESS E-Bites have you covered. Authored by the top professionals in the SAP universe, E-Bites provide the excellence you know from SAP PRESS, in a digestible electronic format, delivered (and consumed) in a fraction of the time!

Massimo Tuscano
SAP Cloud Platform Mobile Services: Application Development and Operations
www.sap-press.com/4781 | $24.99 | 117 pages

Rohit Khan, Rajiv Shivdev Pandey
Introducing SAP Cloud Platform Workflow
www.sap-press.com/4541 | $19.99 | 92 pages

Abani Pattanayak, Imran Rashid
Introducing SAP S/4HANA Extensions in SAP Cloud Platform
www.sap-press.com/4542 | $24.99 | 125 pages

The Authors of this E-Bite

Dr. Vadim Klimov is an SAP integration architect, responsible for the design and optimization of integration solutions, and the evolution of integration capabilities in customer SAP landscapes, using hybrid technologies stacks that include products like SAP Process Orchestration and SAP Cloud Platform Integration. He has worked with SAP software since 2005.

Eng Swee Yeoh is an integration architect at int4, specializing in SAP integration, particularly using SAP Process Orchestration and SAP Cloud Platform Integration. He studied electrical engineering at University of Malaya, Malaysia. His knowledge-sharing activities include publishing online articles and speaking at SAP TechEd.

Learn more about Vadim and Eng Swee at www.sap-press.com/5121.