Anypoint Platform Development: Production-Ready Integrations

Student Manual

Mule runtime 4.4
May 17, 2023

Table of Contents

Introducing the course

This course is for developers who have already internalized the basics of creating Mule apps with Studio and Anypoint Platform and now want to apply these skills in the context of professional software development projects, creating production-ready integrations.

Course goals and objectives

At the end of this course, you should be able to:

- Invoke REST APIs and SOAP web services using various client components taking into consideration the nonfunctional properties of API invocations.
- Pass messages asynchronously reliably between Mule flows and Mule apps using the VM and Anypoint MQ connectors.
- Assert contracts within and between Mule flows using the validation module and JSON/XML schema validation.
- Apply common Enterprise Integration Patterns to orchestrate and parallelize integration logic.
- Use the Object Store and related components to manage state for Mule apps and APIs to increase performance and resilience.
- Identify and extract reusable Mule app code into different Mule runtime extensions.
- Extract reusable integration functionality into XML SDK modules and custom API policies.

Product, component, and tool inventory

This course uses the following products, components, and tools.

Product, Component or Tool	Version
Anypoint Platform control plane	MuleSoft-hosted U.S. production version
Studio	7.15.0
Mule runtime	4.4.0-20230522 EE
Mule Maven plugin	3.8.2
Exchange Mule Maven plugin	0.0.18
Mule Extensions Maven plugin	1.5.0-20221117
MUnit and MUnit Maven plugin	2.3.16
APIkit	1.9.1
HTTP Connector	1.7.3

Product, Component or Tool	Version
Secure Configuration Properties module	1.2.5
Secure Properties tool	4.4
Validation module	2.0.4
JSON module	2.4.1
XML module	1.4.0
OAuth module	1.1.20
Object Store Connector	1.2.1
Web Service Consumer Connector	1.8.2
VM Connector	2.0.0
Anypoint MQ Connector	4.0.3
Tracing module	1.0.0

Table 1. MuleSoft products, components, and tools used in this course

Product, Component, or Tool	Version
OpenJDK 8	latest version
Apache Maven	3.6.3 to 3.8.6
Maven Resources plugin	3.3.1
HTTP client	recent

Course prerequisites

Third-party products, components, and tools used in this course

- Anypoint Platform Development: Fundamentals (Mule 4)
- Solid understanding of essential Maven concepts.
  - Apache Maven Tutorial
  - Maven in 5 Minutes
  - Maven Getting Started Guide

Understanding the approach of this course

This course uses a case study — AnyAirline — to illustrate the topics it addresses.

The course revolves around walkthroughs, which are broken down into sections and steps. You are expected to follow all walkthroughs in the given order, including all sections and all steps in each section.

The course's source code distribution contains two source directory trees, one strictly mirroring the walkthroughs, and another for the complete, final solution to the case study, irrespective of any walkthroughs.

Each walkthrough step typically specifies *what* to achieve rather than *how* to achieve it. This means, for example, that:

- API invocations are specified in terms of cURL commands, because this is a succinct, portable, text-only way of doing so. But this does not imply that you must use cURL. Use whatever tool you prefer to send HTTP requests and inspect HTTP responses, such as Advanced REST Client or Postman.
- File-level operations are expressed in Unix-style and for bash. Again, this is because it is a precise, text-only way of doing so. But you can use any other means of achieving the same result, such as a different Unix shell, the Windows command prompt, or any file explorer on any operating system. The only important thing is that the outcome is the same as when the given bash commands were executed.
- Editing of file contents that does not require Studio should be done in whatever text editor you prefer. This may well be Studio itself (which embeds capable XML and JSON editors) or any Eclipse plugin installed into Studio.
- Mule flows are specified by their XML code. This does not mean that code must or should be entered in XML form. Use whatever approach works best for you, including, but not limited to, coding Mule apps entirely in the visual Mule config flow editor, or the Mule config flow XML editor, or a combination thereof, or even outside of Studio in a Mule-agnostic editor (if you really prefer that). All that matters is that the XML code you produce in this fashion is equivalent to the code shown here. Mule flow XML code is also what matters at runtime and what you will typically see when doing code reviews in GitHub and the like. So it is essential to be confident in reading Mule flow XML code.
- For brevity, Mule flow XML code reproduced here omits doc:name and doc:id attributes and other code elements that have no effect at runtime. Your Mule app code does not have to — and probably should not — omit doc:name, doc:id and similar.
- For brevity, Mule flow XML code reproduced here omits XML namespace declarations.

Some walkthrough steps are declared as Homework. These are steps that have already been performed in previous steps or walkthroughs in a similar fashion, so they are repetitive and there is nothing new to learn. Also, these steps are not strictly necessary for the successful continuation of the current and later walkthroughs. On the other hand, in a real-world project, these steps would be performed and, in fact, the walkthrough solutions in the source code distribution do contain the

result of performing these steps. You are therefore advised to perform these steps if at all possible.

Walkthrough: Set up your development environment

In this walkthrough, you set up the local development environment you will use throughout class. It is essential that your development environment conforms in all aspects to the outcome of following this walkthrough.

1. Create a new **Anypoint Platform trial account** and remember its **username** and **password**.
2. Assign the **Anypoint Monitoring User** and **Visualizer Editor permissions** to that Anypoint Platform user.
3. Install a recent build of **OpenJDK 8**, add both the JVM and the compiler to your **path**, and verify the installation from a command-line interface:

   ```
   java -version
   ```

   ```
   javac -version
   ```

   Note: Both these commands must show the exact same build version — otherwise, you will inadvertently be using two different JDK installations.

4. Install a recent Maven version (3.6.3 to 3.8.6), add it to your path, and verify the installation from a command-line interface:

   ```
   mvn --version
   ```

   Note: This shows the Maven home (installation directory), which you will need shortly for configuring Studio.

   Note: This shows the Java version and installation directory used by Maven: The former must once more show the exact same JDK build version as in the previous step, while the latter you will need shortly for configuring Studio.

5. Back up and remove your local Maven configuration:

   ```
   cd ~/.m2/
   mv -f settings.xml settings.xml.before-apdevl2
   mv -f repository repository.before-apdevl2
   ```

6. Install Studio **7.15.0**, the Studio version required for this course, after downloading it from

https://www.mulesoft.com/lp/dl/studio.

7. Create a new empty Studio workspace and assign its absolute path to environment variable **APDL2WS**:

   Unix variants

   ```
   export APDL2WS=/path/to/studio/workspace
   ```

   Windows

   ```
   set APDL2WS=C:\path\to\studio\workspace
   ```

8. **Launch** and **configure** Studio:
   a. Use the **above workspace**.
   b. Select the **above JDK** in Installed JREs, using the JDK (not just JRE) installation directory shown as part of the Maven verification earlier.

      Note: Studio 7.15.0 has its own embedded JDK, but you use the one installed earlier for consistency with command-line interface usage.

   c. Use the **above Maven** installation by pointing Studio at the Maven home shown earlier, and test it; this must display the same output as the earlier Maven verification from the command-line interface.

      Note: Studio has its own embedded Maven, but you use the one installed earlier for consistency with command-line interface usage.

   d. Install **Mule runtime 4.4.0 EE** into Studio if it is not already installed; all Studio projects must be created with this version of the Mule runtime.
   e. Under XML > **XML Files** > Editor, set line width 140, clear all blank lines, don't format comments, don't insert white space before closing tag, and indent using two spaces.
   f. Authenticate to Anypoint Platform using the **trial account** credentials obtained above.
9. Create a new, trivial Mule app project in Studio and run it to confirm your setup is functional. Ensure that this and all future Studio projects use Mule runtime 4.4.0 EE.
10. Install an application for sending HTTP requests and inspecting HTTP responses, such as cURL, Advanced REST Client, or Postman.
11. Install/choose a text editor for editing XML files and the like.
12. Locate your course enrollment email and follow the instructions to download the student files ZIP.

13. Unpack the source code distribution package of this course revision and assign the absolute path of the filesystem directory to environment **variable APDL2DIST**:

```
cd     /path/to/course/directory
unzip  APDevPRInts*_studentFiles_*.zip
cd     APDevPRInts*_studentFiles_*
export APDL2DIST=$(pwd)
```

14. Familiarize yourself with the **source code distribution**.

    Note: The source code distribution contains two code directory trees, one for the complete, "final" solution to the AnyAirline case study, including all Mule apps, and other artifacts, the other for the solutions and starters for individual walkthroughs, each one relating directly to a walkthrough in this document, and addressing the aspects of the final solution under consideration in that walkthrough.

15. **Install API Catalog CLI:** In a command-line interface, run the npm install command for API Catalog CLI:

```
npm install -g api-catalog-cli@latest
```

    Note: If the command is not recognized, install Git and Node.js, then run the install command again.

How to install a walkthrough solution project

Under $APDL2DIST/walkthroughs, you find individual project directories for the solutions of all walkthroughs and the starters for some walkthroughs.

Typically, the solution to one walkthrough is the starter for the next walkthrough — unless a walkthrough has its own, explicit starter projects. When you follow the course from start to finish, you only have to install the explicit starter projects, and this is explained step-by-step as part of the walkthroughs that come with starter projects. However, when you cannot follow a walkthrough, you will typically have to install the solution projects for this walkthrough, so that you can continue with the subsequent walkthrough — unless that subsequent walkthrough has its own explicit starter projects.

Follow these steps to install the solution projects for a particular walkthrough — for example, walkthrough 3-1 — into your Studio workspace.

1. Discover the solution projects and files for the given walkthrough; for example, $APDL2DIST/walkthroughs/devprd/module03/wt3-1_solution has one subdirectory, for check-in-papi, and so walkthrough 3-1 only comprises that one solution project, which captures the state of check-in-papi at the end of walkthrough 3-1. Files such as parent-pom/pom.xml and pom.xml complete the Maven build setup for all solution projects of this walkthrough.
2. Copy all solution projects and files into your Studio workspace:

```
cd $APDL2WS
mkdir before-wt3-1
mv * before-wt3-1/
cp -r $APDL2DIST/walkthroughs/devprd/module02/wt2-5_solution/* ./
```

3. Edit **pom.xml** in the newly copied project directories, following any `students:`-instructions in those files.
4. Edit the newly copied **parent-pom/pom.xml**, following any `students:`-instructions in that file.
5. Install **bom/pom.xml and parent-pom/pom.xml** into your local Maven repository:

```
cd $APDL2WS
mvn install:install-file -Dfile=bom/pom.xml        -DpomFile
=bom/pom.xml
mvn install:install-file -Dfile=parent-pom/pom.xml -DpomFile=parent-
pom/pom.xml
```

6. Build the solution projects in the required order (if applicable):

```
cd $APDL2WS/apps-commons
mvn clean install
```

```
cd $APDL2WS/check-in-papi
mvn clean verify -U -Dencrypt.key=secure12345
```

7. Import the solution projects into Studio, without copying them into the workspace.
8. Create a Studio **run configuration** for the newly imported projects, setting **VM arguments**:

```
-M-Dencrypt.key=secure12345
-M-Danypoint.platform.gatekeeper=disabled
```

Introducing the AnyAirline case study

AnyAirline is a regional airline and an existing MuleSoft customer. It uses the MuleSoft-hosted Anypoint Platform control plane, and CloudHub to support a mobile app and a few other, ad-hoc integrations.

This case study focuses on a small selected number of use cases within this system landscape.

Requirements

The following systems are already deployed and functional, and form the immutable system landscape for this project, not to be changed but rather integrated with.

- **mobile app:** AnyAirline's native mobile app, the main strategic driver for this project.
- **PayPal:** For customer/passenger payments.
- **Flights Management system:**
  - Accessible via SOAP web services over mutually authenticated HTTPS.
  - Deployed on-premises in the AnyAirline data center.
- **Passenger Data system:**
  - In-house legacy PostgreSQL database to be accessed directly.
  - Deployed on-premises in the AnyAirline data center.
- **Salesforce CRM:** Recently introduced.

Functional requirements

Terminology

- **Record Locator:** A six- or seven-digit alphanumeric code that identifies a data record in an airline reservation system, for example, RW4TAB or KZVGX5. The term Record Locator is usually used to refer to a PNR. In this case study we do not make use of the term Record Locator itself.
- **PNR:** Passenger Name Record contains personal information about a passenger, as well as their itinerary (for example, flights). For simplicity, in this case study, itineraries consist of only one flight, so that each PNR identifies exactly one passenger and their flight. Furthermore, in this case study, we do not deal with the Passenger Name Records themselves, just with Record Locators identifying those Passenger Name Records. Therefore, in this case study, the term PNR is used to always mean Record Locator referring to a Passenger Name Record.

Actors

- **Passenger:** A customer of AnyAirline, in possession of an AnyAirline ticket.
- **Third party check-in partner company:** An external company that partners with AnyAirline to provide third party check-in for AnyAirline passengers.

User stories

The following user stories will be designed and implemented in this project.

US1: mobile Check-In

As a passenger, I want to be able to check in to an AnyAirline flight using the mobile app, by

providing my PNR and last name and paying for any baggage using PayPal.

Preconditions:

1. Passenger holds an AnyAirline ticket and knows its PNR.
2. Passenger has a PayPal account.
3. Passenger has mobile app installed.

US2: Flight Cancellation mobile Notifications

As a passenger, I want to be notified through the mobile app if a flight to which I've checked in is canceled.

Preconditions:

1. Passenger has checked in to the flight using the mobile app.
2. This flight is canceled.
3. AnyAirline is notified of the flight cancellation by an external party via the Flights Management system.

US3: Offline Check-In Submissions

As a Third party check-in partner company ("company" for short), I want to be able to submit all check-in data that was accumulated while being offline.

Context:

- Check-in must continue even if systems are offline due to an outage, so that normal online check-in using AnyAirline's online systems cannot be performed.
- After coming online again, the Third party check-in partner company must submit to AnyAirline data about all check-ins performed during the offline period.

Preconditions:

1. Company is a known partner of AnyAirline providing, check-in to AnyAirline passengers using AnyAirline online check-in services.
2. Company has suffered an outage so that it can't use AnyAirlines' online check-in services.
3. Company has continued checking in passengers while offline.
4. Company has since restored full connectivity.
5. Company now wants to send all check-in data accumulated during the offline period to AnyAirline.

Nonfunctional requirements

This is a small subset of nonfunctional requirements, namely those that are directly relevant to this project:

- **NFR1:** Application components interacting directly with on-premises systems (Flights Management system, Passenger Data system) must be deployed on-premises in the AnyAirline data center. This applies, for example, to System APIs.
- **NFR2:** All application components interacting only with cloud-hosted systems (PayPal, Salesforce CRM) or APIs must be deployed to CloudHub 2.0.
- **NFR3:** All data must be encrypted in flight (for example, using HTTPS).
- **NFR4:** API-led connectivity should be followed unless performance considerations suggest otherwise.

Solution architecture

Abbreviations used:

- SAPI for System API
- PAPI for Process API
- EAPI for Experience API

Please note that, for the purpose of simplicity, this section does not differentiate between an API and its API implementation. For example, Flights Management SAPI is used to denote both the REST API — defined via a RAML definition or OpenAPI definition — as well as the Mule app implementing and exposing that REST API. (However, later sections will differentiate; for example, the API implementation of Flights Management SAPI will be called flights-management-sapi.)

High-level architecture

The following diagram gives an overview of the relevant components and artifacts for this project, and where they interact to realize the use cases of this project. Deployment aspects of this diagram describe the production environment.

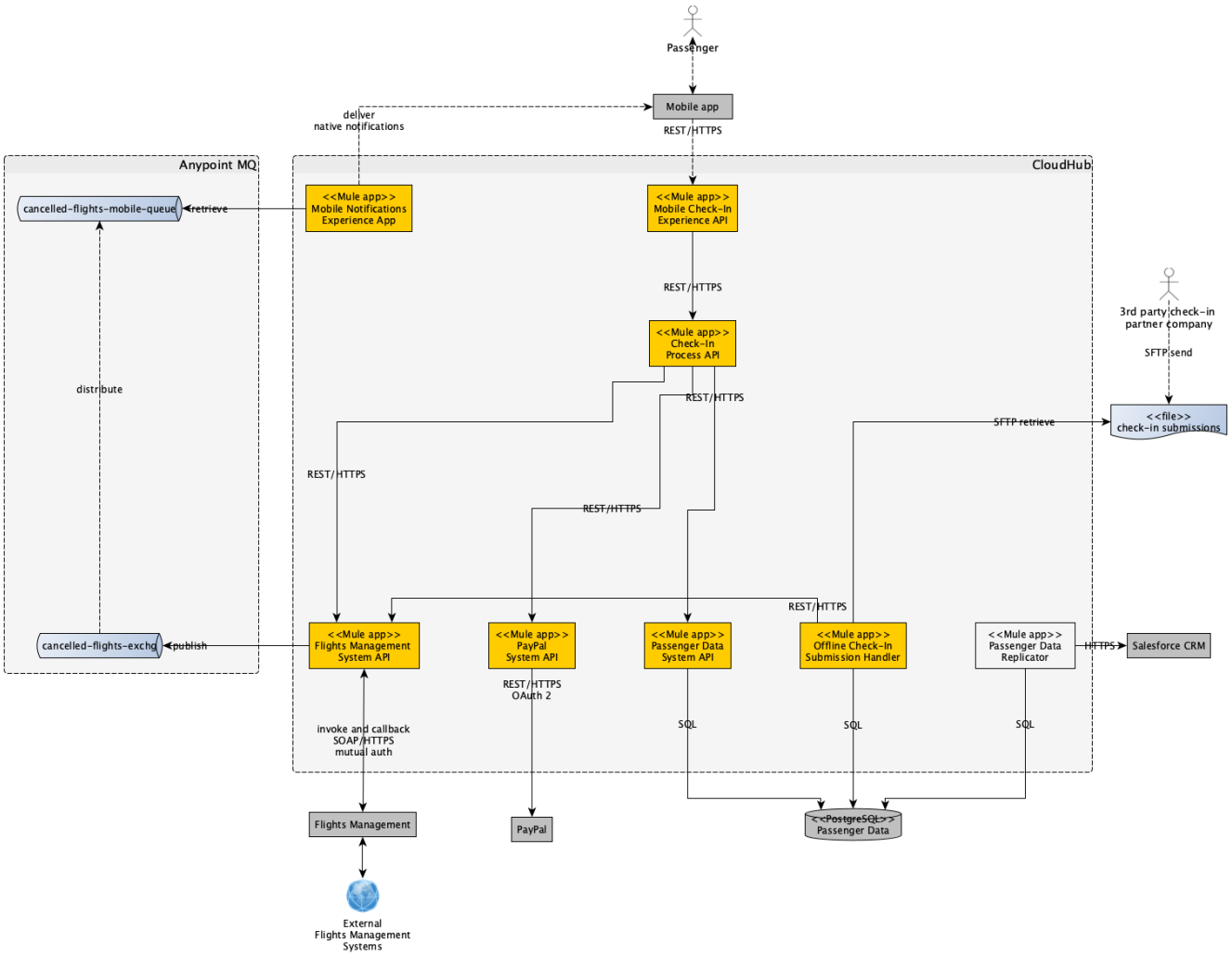

Figure 1. Components and artifacts under design and development in this project (yellow and blue), to be integrated with but not changed (grey), or out-of-scope for this project (transparent). A static view of all interactions between these components and artifacts is shown by arrows pointing from active to passive participant in the interaction. Dotted arrows signify interactions not directly addressed in this project. Deployment asepcts are for the production environment.

User story realizations

US1: mobile Check-In

Overview

The following diagram explains on a high level of detail the business process underlying US1: mobile Check-In.

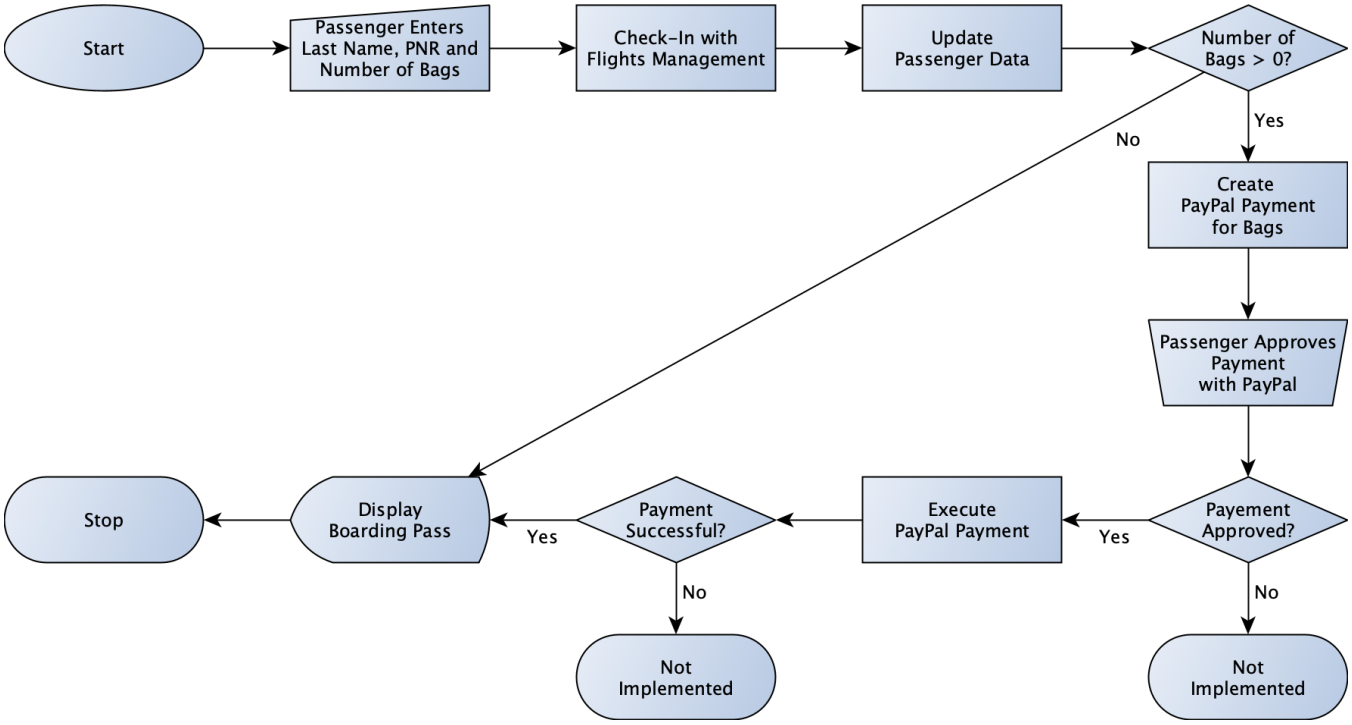

Figure 2. Flowchart visualizing the process by which US1: mobile Check-In is to be realized on a high level. Processing steps and decisions that do not require human interaction are shown as rectangles and rhombi, respectively.

PayPal payments

Payments are implemented via PayPal. This affects and is visible to the API implementations processing payments, the mobile app, and the passenger.

Integration with PayPal as designed here uses v1 of the PayPal REST API, which is deprecated as of early 2019.

The general approach for integrating with PayPal is:

- Use the PayPal v1 REST APIs from the PayPal SAPI to first create a PayPal payment and then, after approval by the passenger, to execute that PayPal payment.
- Use the web-based PayPal Check-Out Button from the mobile app to allow the passenger to approve a previously created PayPal payment.

Specifically, a PayPal payment must first be created by PayPal SAPI via a PayPal REST API invocation. The Payment ID created in the process by PayPal must then be used in the mobile app to present the passenger with a UI to approve that payment. In approving the payment, PayPal identifies the payer and hands its Payer ID to the mobile app. Finally, to execute a PayPal payment, PayPal SAPI must pass both the Payment ID and the Payer ID to PayPal in a PayPal REST API invocation.

Authentication of PayPal API clients follows the OAuth 2.0 client credentials grant type:

- At development time, a PayPal app representing the PayPal API client, which is PayPal SAPI, must be created or egistered with PayPal. In the process, PayPal generates a pair of client ID and secret for PayPal SAPI, for both the PayPal Sandbox and Production environments.
- At runtime, client ID and secret must be exchanged for a temporary bearer access token. All further PayPal REST API invocations must then present that bearer access token in the HTTP Authorization header. This process must be repeated once the bearer access token has expired.

Detailed component interactions

The following diagram shows how US1: mobile Check-In is realized through the interaction of all relevant components in the case where payment for bags needs to be made. In the case where there are no bags and no payment, the interaction simplifies accordingly.

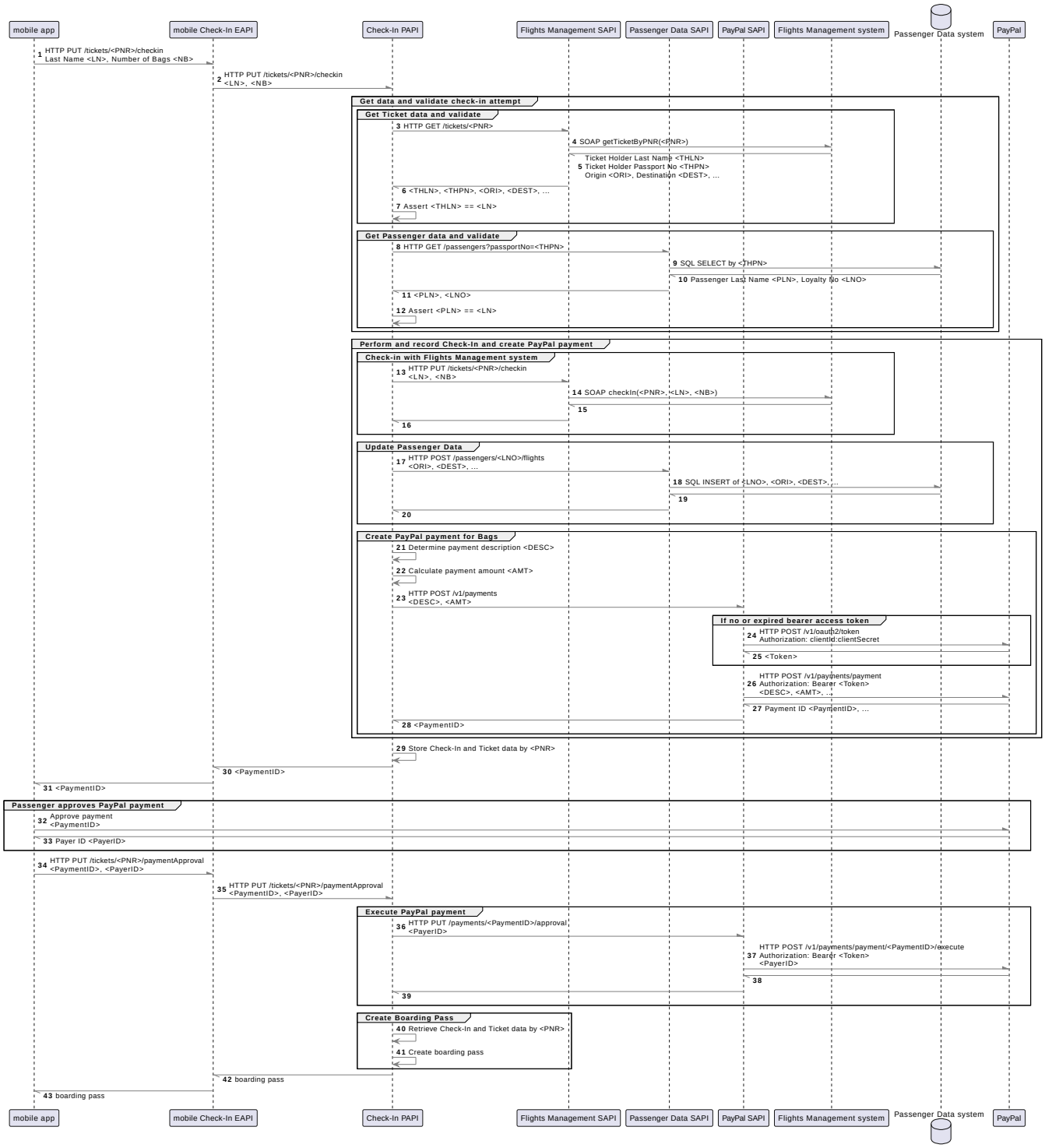

Figure 3. Sequence diagram visualizing the detailed interaction of components in the realization of US1: mobile Check-In. The depicted case is when the number of bags is positive and hence payment needs to be collected.

US2: Flight Cancellation mobile Notifications

From AnyAirline's perspective, flight cancellations materialize from outside of AnyAirline in the Flights Management system.

1. SOAP clients to the Flights Management system, such as Flights Management SAPI, can register an HTTP callback (webhook), which will then be invoked by the Flights Management system when a flight cancellation occurs. That registration should occur at startup of Flights Management SAPI (duplicate registrations are ignored).
2. The HTTP callback delivers cancellation notifications from the Flights Management system to Flights Management SAPI in the form of a HTTP POST requests from the former to the latter.
3. One cancellation notification for each previously successful check-in, now affected by the cancellation, is sent per HTTP POST request. Each cancellation notification contains the PNR and last name of the passenger in XML format:

```
<CancellationNotification>
  <PNR>PNR123</PNR>
  <PassengerLastName>Mule</PassengerLastName>
</CancellationNotification>
```

4. Flights Management SAPI uses the reliability pattern to accept a cancellation notification and immediately publish it onto a persistent VM queue. Then, asynchronously — in an XA transaction if supported by the message broker — the System API unqueues the notification, transforms it to a backend-neutral flight canceled event and publishes that event to an Anypoint MQ exchange (or, potentially, a JMS topic). A flight canceled event is JSON-formatted:

```
{
 "pnr": "RW4TAB",
 "lastNameOfPassenger": "Smith"
}
```

5. Because the HTTP callback processes cancellation notifications asynchronously rather than synchronously, it returns a HTTP 202 ACCEPTED rather than a 200 OK response code to the Flights Management system, to inform it of that fact.
6. flight canceled events are then delivered via publish-subscribe messaging to an Experience-layer mobile app, mobile Notifications EApp, which delivers them to the mobile app via native mobile notifications (such as Apple's APNs).

The component interactions to realize the first part of these steps, up to the publishing of flight canceled events to the Anypoint MQ exchange, are shown in the following sequence diagram.

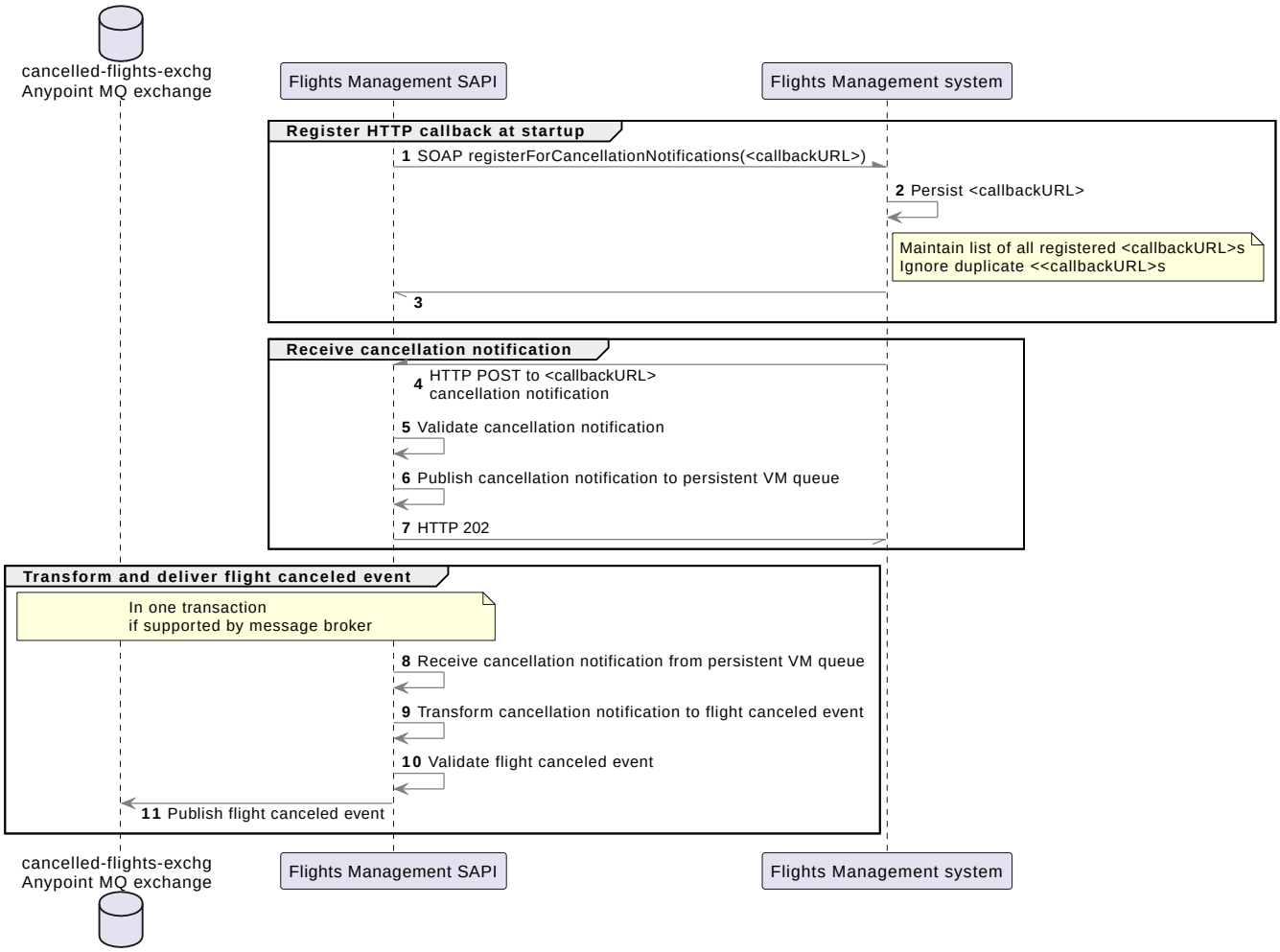

Figure 4. Sequence diagram visualizing the first part of the realization of US2: Flight Cancellation mobile Notifications: the registration of Flights Management SAPI with the Flights Management system to receive cancellation notifications, the delivery and transformation of cancellation notifications to flight canceled events, and the publishing of the latter to an Anypoint MQ exchange.

US3: Offline Check-In Submissions

Offline Check-In Submission Handler has a similar role to Check-In PAPI, with the difference that the former:

- Processes batches of check-ins submitted in a flat file,
- Does not deal with payments.

Files are submitted per SFTP and are in CSV format:

- Each submissions file contains zero to arbitrary many (typically tens of thousands) of records with one record on each line.

- Each record contains data about a passenger's check-in.
- Each record is independent of all other records. There is no need to deal with records that pertain to duplicate check-ins of the same passenger to the same flight.
- Each record must be individually passed to Flights Management SAPI via an API invocation, similar to what Check-In PAPI does.
- Batches of records must be inserted into the Passenger Data system, similar to what Passenger Data SAPI does (but in batches, for efficiency).

Deployment

All Mule apps are deployed to CloudHub 2.0 in the shared space in usa-e1 (N. Virginia) or usa-e2 (Ohio) under the control of the MuleSoft-hosted Anypoint Platform control plane in the U.S.

Module 1: Invoking web APIs and services

In this module, you invoke REST APIs and SOAP web services using various client components taking into consideration the nonfunctional properties of API invocations.

At the end of this module, you should be able to:

- Set up the starter code
- Invoke HTTP APIs using the HTTP Connector paying attention to the non-functional properties of API invocations.
- Map between HTTP requests and Mule events
- Enable an API client for OAuth 2.0.
- Invoke a SOAP web service using mutual TLS authentication.
- Implement an HTTP callback.

Walkthrough 1-1: Set up the starter code

This walkthrough starts with the latest state of check-in-papi and apps-commons as implemented in *Anypoint Platform Development: Production-Ready Development Practices*. The Mule app check-in-papi relies on the library-style Mule plugin apps-commons for common Mule app functionality in the areas of error handling and health checks. A library-style Mule plugin contains simple resources like Mule flow config files and is built from a Maven project with <packaging /> mule-application and <classifier /> mule-plugin.

In this walkthrough, you setup the starter code for check-in-papi, apps-commons, and their Maven build.

You will:

- [Setup starter code for check-in-papi]

Starting file

To follow this walkthrough, you need the starter project at $APDL2DIST/walkthroughs/devint/module01/wt1-1_starter.

Set up starter code for check-in-papi

In this section, you install a Maven settings.xml, which provides the credentials for authenticated access to several Maven repositories, and a parent POM and BOM, which form the basis of repeatable and convenient Maven builds for AnyAirline and this course. You then copy the starter code for apps-commons — a utility library-style Mule plugin for Mule apps — and check-in-papi into your Studio workspace, build them, and import them into Studio.

To run check-in-papi, you must disable autodiscovery, otherwise you would either have to create an API instance for Check-In PAPI in an Anypoint Platform organization you have access to or, alternatively, receive credentials to connect to AnyAirline's Anypoint Platform organization, which already has an API instance with the API ID configured in check-in-papi: you decide against both of these options and instead disable autodiscovery entirely.

1. **Study Solution Architecture:** Study the Solution architecture and in particular the High-level architecture and the design of US1: mobile Check-In; note the dependencies of Check-In PAPI on Flights Management SAPI, Passenger Data SAPI, and PayPal SAPI for the core check-in functionality being implemented in check-in-papi.
2. **Install global settings.xml:** Copy settings.xml from $APDL2DIST/etc to your ~/.m2/ directory and study its contents:

```
cp $APDL2DIST/etc/settings.xml ~/.m2/
```

Note: This settings.xml provides the credentials needed for accessing various Maven repositories, such as the MuleSoft EE releases repository and the Exchange of AnyAirline.

Note: There is also an entry for deploying to CloudHub 2.0 from the Mule Maven plugin, but this will not be used for the time being.

3. **Locate EE releases credentials:** In settings.xml, confirm the credentials for the MuleSoft EE releases repository to be **muletraining.nexus** and **eApHMgTm**.
4. **Create Exchange Viewer Connected App:** In Anypoint Access Management, create a new Connected App that acts on its **own behalf (client credentials)**, for reading assets from Exchange, adding the **Exchange Viewer** scope and retrieve its client ID and secret.
5. **Update settings.xml:** In a text editor, update settings.xml in ~/.m2/ with credentials for reading from your Anypoint Platform organization's Exchange as a Maven repository:

   settings.xml

   ```
   <settings>
     <servers>
       <server>
         <id>anypoint-exchange-v3</id>
         <username>~~~Client~~~</username>
         <password>your-AP-Exchange-Viewer-cid~?~your-AP-Exchange-Viewer-
   secret</password>
       </server>
     </servers>
   </settings>
   ```

   Note: You can also use Anypoint Platform credentials for an account of your Anypoint Platform organization with which you were able to search your Exchange — such as the trial account credentials created previously. However, it is best practice to use a Connected App for security.

   Note: To use Connected App authentication, provide basic authentication and define the username as ~~~Client~~~ and the password as clientID~?~clientSecret. Replace clientID with the client ID. Replace clientSecret with the client secret.

6. **Copy parent POMs:** Copy AnyAirline's BOM and parent POM into new **bom** and **parent-pom** directories, respectively, within your Studio workspace directory:

   ```
   cd $APDL2WS
   ```

```
mkdir bom
mkdir parent-pom
cp $APDL2DIST/walkthroughs/devint/module01/wt1-1_starter/bom/pom.xml
./bom/pom.xml
cp  $APDL2DIST/walkthroughs/devint/module01/wt1-1_starter/parent-
pom/pom.xml ./parent-pom/pom.xml
```

7. **Update parent POMs:** Update all parent POMs adapting the Maven coordinates to match the **groupId** of your Anypoint Platform organization ID for the project and the parent:

   bom/pom.xml

   ```
   <project>
     <groupId>your-AP-organization-id</groupId>
     ...
   </project>
   ```

   parent-pom/pom.xml

   ```
   <project>
     <parent>
       <groupId>your-AP-organization-id</groupId>
       ...
     </parent>
     <groupId>your-AP-organization-id</groupId>
     ...
   </project>
   ```

   Note: These POMs use the Anypoint Platform organization ID for the groupId, as this is a prerequisite for publishing to Exchange, which you will do later.

8. **Configure Anypoint Platform organization ID groupId:** Update the custom property with your Anypoint Platform organization ID used to configure the dependency groupId for apps-commons when referenced later:

   bom/pom.xml

   ```
   ...
   <properties>
     ...
     <student.deployment.ap.orgid>your-AP-organization-
   id</student.deployment.ap.orgid>
   ```

```
</properties>
```

9. **Install, and study parent POMs:** Install the parent POMs into your local Maven repository, and browse their contents:

```
cd $APDL2WS
mvn install:install-file -Dfile=bom/pom.xml        -DpomFile
=bom/pom.xml
mvn install:install-file -Dfile=parent-pom/pom.xml -DpomFile=parent-
pom/pom.xml
```

Note: AnyAirline's BOM manages all Maven dependencies and plugins supported by AnyAirline and this course, thereby fixing their versions and providing the repositories to download them from.

Note: AnyAirline's parent POM provides configuration of the Mule Maven plugin, MUnit Maven plugin, and other Maven aspects to facilitate build, test, and deploy of Mule apps.

Note: Installation into the local Maven repository is necessary because MUnit Maven tooling currently does not resolve parent POMs from the filesystem location given in <relativePath />.

10. **Copy apps-commons starter:** Copy the apps-commons starter project into your Studio workspace directory:

```
cd $APDL2WS
cp -r $APDL2DIST/walkthroughs/devint/module01/wt1-1_starter/apps-
commons ./
```

11. **Study Maven build:** Familiarize yourself briefly with the Maven build configuration of apps-commons.

Note: This Maven project packages a library-style Mule plugin containing Mule flow config files.

Note: Dependencies defined in provided scope must be provided by the context (Mule app, Mule runtime domain) in which apps-commons is imported and executed.

Note: Reusable Maven plugin configuration is inherited from the parent POM and Maven dependency management from the BOM. The parent-pom/pom.xml is read from the Studio workspace directory, and therefore in turn imports bom/pom.xml from that same directory.

12. **Update groupId:** Update pom.xml adapting the Maven coordinates to match the groupId of your Anypoint Platform organization ID for the project and the parent:

    pom.xml of apps-commons

    ```
    <project>
     <parent>
        <groupId>your-AP-organization-id</groupId>
        ...
      </parent>
      <groupId>your-AP-organization-id</groupId>
      ...
    </project>
    ```

    Note: This POM uses the Anypoint Platform organization ID for the groupId, as this is a prerequisite for publishing to Exchange, which you will do later.

13. **Maven-build:** Run a full Maven build of apps-commons, installing the JAR into your local Maven repository:

    ```
    cd $APDL2WS/apps-commons
    mvn clean install
    ```

14. **Import into Studio:** In Studio, import the apps-commons project as a Mule app, without copying it into the workspace, because it is already there; confirm that the import succeeded and update the classifier for the Mule Maven plugin configuration to output a mule-plugin:

    pom.xml of apps-commons

    ```
    <plugin>
      <groupId>org.mule.tools.maven</groupId>
      <artifactId>mule-maven-plugin</artifactId>
      <configuration>
        <classifier>mule-plugin</classifier>
      </configuration>
    </plugin>`
    ```

    Note: Importing the Mule app project into Studio changes the <classifier /> to mule-application and must be changed back.

    Note: The apps-commons project builds an artifact with Maven classifier mule-plugin but uses Maven packaging mule-application and so imports just like any other Mule app project into

Studio.

15. **Copy check-in-papi starter:** Copy the check-in-papi starter project into your Studio workspace directory, renaming it to check-in-papi:

```
cd $APDL2WS
cp -r $APDL2DIST/walkthroughs/devint/module01/wt1-1_starter/check-in-
papi ./
```

16. **Adapt POM:** In a text editor, adapt check-in-papi to your Maven build set-up by following the instructions in **pom.xml** including adapting the Maven coordinates of the parent POM to match the groupId of your Anypoint Platform organization ID for the project and the parent:

pom.xml of of check-in-papi

```
<project>
 <parent>
    <groupId>your-AP-organization-id</groupId>
    ...
  </parent>
  ...
  <groupId>your-AP-organization-id</groupId>
  ...
</project>
```

17. **Study Maven build:** Familiarize yourself briefly with the Maven build configuration of check-in-papi.

Note: The Check-In PAPI API specification is loaded as a Maven dependency with artifact ID check-in-papi in the form of an OpenAPI definition from the AnyAirline Exchange Maven repo, which requires authentication entries in settings.xml, created previously.

Note: The check-in-papi Mule app is assigned to the Process tier/layer in Visualizer.

Note: Again, reusable Maven plugin configuration is inherited from the parent POM and Maven dependency management from the BOM. Both parent-pom/pom.xml and bom/pom.xml are read from the Studio workspace directory.

18. **Maven-build:** Run a full Maven build of check-in-papi including all MUnit tests, providing the required secure properties encryption key; this should succeed and all unit tests should pass:

```
cd $APDL2WS/check-in-papi
```

```
mvn clean verify -U -Dencrypt.key=secure12345
```

19. **Import into Studio:** In Studio, import check-in-papi without copying it to the workspace, because it is already there; confirm that the import succeeded.
20. **Add run config disabling autodiscovery:** Add a Studio run configuration (via the **Run as > Mule application (configure)** dialog) for check-in-papi that sets **encrypt.key** and disables autodiscovery:

    ```
    -M-Dencrypt.key=secure12345
    -M-Danypoint.platform.gatekeeper=disabled
    ```

    Note: The API ID values configured in the properties files of check-in-papi for the various environments stem from the AnyAirline Anypoint Platform organization, to which you have not been given access; disabling autodiscovery works around this restriction by preventing the uplink to API Manager without requiring a code change (such as removing the autodiscovery configuration).

21. **Run and invoke:** Run check-in-papi and invoke the exposed Check-In PAPI and health check endpoints; this should succeed, returning meaningful HTTP responses:

    ```
    curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
    :\"Smith\",\"numBags\":2}"
    https://localhost:8081/api/v1/tickets/PNR123/checkin
    ```

    ```
    curl -ik -X PUT -H "Content-Type: application/json"  -d "{ \"payerID
    \": \"STJ8222K092ST\", \"paymentID\": \"PAY-1AKD7482FAB9STATKO\" }"
    https://localhost:8081/api/v1/tickets/N123/paymentApproval
    ```

    Note: Because this API implementation uses self-signed certificates, you must explicitly allow them when invoking these HTTPS endpoints.

22. **Study check-in-papi:** Inspect check-in-papi so that you can explain the HTTP requests just performed, the HTTP responses received, and the MUnit tests executed in the previous Maven build.

Walkthrough 1-2: Invoke HTTP APIs using the HTTP Connector

The readiness endpoint as implemented in check-in-papi is just a placeholder: it does not actually check the availability of all API dependencies of check-in-papi, as a proper readiness endpoint implementation should.

In this walkthrough, you implement the readiness endpoint of check-in-papi by invoking the liveness endpoints of the API implementations of all its downstream API dependencies: Flights Management SAPI, Passenger Data SAPI, and PayPal SAPI using the plain HTTP Connector. In doing so, you address the validation of HTTP responses and various important nonfunctional aspects of API invocations.

You will:

- Send HTTP GET requests using the HTTP Connector.
- Log HTTP traffic.
- Validate the HTTP status code returned in HTTP responses.
- Set meaningfully short HTTP response timeouts.

Solution file

You can see the end state of following this walkthrough in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module01/wt1-2_solution.

Send HTTP GET requests using the HTTP Connector

In this section, you implement check-all-dependencies-are-alive of check-in-papi, which is invoked by api-ready in apps-commons; that is, the actual readiness endpoint implementation. You implement this Mule flow using <http:request /> to send HTTP GET requests to the liveness endpoints of the implementations of all API dependencies of check-in-papi — the three System APIs Flights Management SAPI, Passenger Data SAPI, and PayPal SAPI — in the matching environment of AnyAirline. So check-in-papi in the dev environment of your Anypoint Platform organization invokes AnyAirline's dev endpoints of flights-management-sapi, passenger-data-sapi, and paypal-sapi, and similarly for test and prod.

It is a generic pattern that readiness endpoint implementations invoke liveness endpoints of all dependencies. If they were to invoke readiness endpoints then this would trigger a transitive invocation cascade with every readiness check.

1. **Implement readiness checks:** Implement **check-all-dependencies-are-alive** in health.xml to use a <http:request /> to invoke all three liveness endpoints using HTTP GET to the correct

URLs, without referring to an HTTP Request configuration, following HTTP redirects, and using defaults for everything else:

health.xml of check-in-papi

```
<sub-flow name="check-all-dependencies-are-alive">
  <http:request method="GET"
    url="https://tngaa-flights-management-sapi-dev-9yj2rh.rajrd4-
2.usa-e1.cloudhub.io/alive"
    followRedirects="true" />
  <http:request method="GET"
    url="https://tngaa-passenger-data-sapi-dev-9yj2rh.rajrd4-2.usa-
e1.cloudhub.io/alive"
    followRedirects="true" />
  <http:request method="GET"
    url="https://tngaa-paypal-sapi-dev-9yj2rh.rajrd4-2.usa-
e1.cloudhub.io/alive"
    followRedirects="true"/>
</sub-flow>
```

Note: It is a recommended defensive practice to always transparently follow HTTP redirects when possible, even if this is not needed here.

Note: This implementation of check-all-dependencies-are-alive can be improved along many dimensions, such as resilience and performance. You will do this shortly.

2. **Run and invoke:** Run check-in-papi in Studio and invoke its health check endpoints as before; both should succeed:

```
curl -ik https://localhost:8081/alive
```

```
curl -ik https://localhost:8081/ready
```

Note: The readiness endpoint should take noticeably longer to return than the liveness endpoint.

3. **Homework: Add props:** Add environment-dependent and specific for all environments, such as the hostnames of the System API dependencies, and the root paths of their API and health check endpoints.

Log HTTP traffic

In this section, you increase the log output produced by the HTTP Connector such that HTTP requests and responses to/from check-in-papi are logged. This affects both <http:listener /> and <http:request />.

4. **Set logger's log level:** In **log4j2.xml** decrease the log level of the relevant logger of the HTTP Connector to **DEBUG**:

   log4j2.xml of check-in-papi

   ```
   <Loggers>
     <AsyncLogger
       name="org.mule.service.http.impl.service.HttpMessageLogger"
       level="DEBUG" />
   </Loggers>
   ```

   Note: This entry is typically already present in Studio-generated log4j2.xml files and only needs to be commented-in.

   Note: It is not necessary to decrease the log level of the root logger.

   Note: This Log4J configuration is used in all environments; only unit tests are governed by a different Log4J configuration file (log4j2-test.xml). It is often helpful to use different logging configurations in prod versus test and dev, but this is beyond the scope of this course.

5. **Run and invoke:** Run check-in-papi and invoke the health check endpoints again; confirm that the HTTP interactions are logged and are as expected.
6. **Study threads:** Inspect the log output for information about the threads which perform HTTP communication, as this gives important insight into Mule runtime thread handling.

Validate the HTTP status code returned in HTTP responses

In this section, you enforce the agreed contract with health check endpoints in that the HTTP response must have a status code of 200. By default, <http:request /> considers all 2xx response codes to be successful, and raises errors for 4xx and 5xx response codes, while optionally following redirects denoted by 3xx response codes. So while default HTTP response code validation is sufficient for health checks, you explicitly enforce the contract by accepting only status code 200.

7. **Add status code validator:** Add explicit HTTP status code validation to all <http:request /> elements, requiring a value of 200:

health.xml of check-in-papi

```
<http:request method="GET" url="..." followRedirects="true">
  <http:response-validator>
    <http:success-status-code-validator values="200" />
  </http:response-validator>
</http:request>
```

Note: Status code ranges can also be expressed.

Note: An inverse status code validator that requires you to list error codes is also available.

8. **Run and invoke:** Run check-in-papi and confirm that the readiness endpoint implementation still works correctly.

Set meaningfully short HTTP response timeouts

In this section, you explore the impact of unavailable API dependencies and improve the responsiveness of the check-in-papi readiness endpoint implementation by setting a short timeout on the invocations of the downstream liveness endpoints.

9. **Misconfigure API dependency:** In **health.xml** change the paypal-sapi URL to use a timeout simulator service, thereby making all invocations to that API fail:

health.xml of check-in-papi

```
<sub-flow name="check-all-dependencies-are-alive">
  ...
  <http:request method="GET"
    url="https://tngaa-http-simulator-service-zkvtif.rajrd4-1.usa-
e1.cloudhub.io/wait"
    followRedirects="true">
    <http:response-validator>
                <http:success-status-code-validator
                    values="200" />
            </http:response-validator>
        </http:request>
</sub-flow>
```

Note: This is a naive way of simulating one particular kind out of a whole range of possible communication issues that may occur when invoking this or any other API.

10. **Run and invoke:** Invoke the readiness endpoint, ideally timing its execution; this should hang for at least 10 seconds before **failing**:

    ```
    time curl -ik https://localhost:8081/ready
    ```

    Note: The exact amount of time that an ultimately failing API invocation — or other HTTP request-response interaction — hangs until it returns an error is hard to predict in the absence of explicit timeout configuration, because it depends on many factors, including the nature of the failure, network components between API client and API implementation, and operating system, JVM and Mule runtime configuration, and defaults. The current Mule runtime default is 10 seconds.

11. **Reflect:** In Studio, follow the exact call chain that leads from the failed <http:request /> to the HTTP response just returned.
12. **Set timeout:** Add a short but achievable response timeout of **1000 milliseconds** to all <http:request /> elements:

    health.xml of check-in-papi

    ```
    <http:request method="GET" url="..." followRedirects="true"
      responseTimeout="1000">
      ...
    </http:request>
    ```

    Note: A meaningful value for the response timeout is the 95th percentile of the HTTP response time of the invoked API implementation when under typical load [Ref15], with the response time being measured from the point of view of the invoking API client. The response time thus depends on the performance characteristics and network location not only of the API implementation but also, to a certain extent, the API client.

    Note: This implementation of check-all-dependencies-are-alive can still be improved along many dimensions, such as resilience and performance. You will do this in later walkthroughs.

13. **Run and invoke:** Invoke the readiness endpoint again, ideally timing its execution; this should fail much faster.
14. **Revert misconfiguration:** Undo the misconfiguration of the paypal-sapi in **health.xml**.
15. **Homework:** Analyze the commonalities in the three <http:request /> instances in check-all-dependencies-are-alive of check-in-papi and factor-out that commonality into a new helper flow in apps-commons. This helper flow can then be used in all similar implementations of check-all-dependencies-are-alive in all Mule apps.

Walkthrough 1-3: Map between HTTP requests and Mule events

The check-in functionality currently implemented in check-in-papi is just a placeholder: it does not orchestrate any of the required downstream APIs to check-in a passenger.

In this walkthrough, you you begin the implementation of the required check-in functionality, invoking all its downstream API dependencies: Flights Management SAPI, Passenger Data SAPI, and PayPal SAPI. Including setting the required authentication and mapping HTTP request and responses to orchestrate multiple HTTP requests.

You will:

- Configure the HTTP Connector to send client ID and secret via HTTP Basic Authentication.
- Invoke downstream System APIs using the HTTP Connector with full control over the HTTP message.
- Handle all errors raised while processing an incoming HTTP request.

Solution file

You can see the end state of following this walkthrough in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module01/wt1-3_solution.

Configure the HTTP Connector to send client ID and secret via HTTP Basic Authentication

In this section, you configure the HTTP Connector for the three System APIs invoked by check-in-papi. This comprises supplying the endpoint URLs of the API instances which the HTTP Connector should invoke, as well as providing required security configuration.

In the case of the three System APIs, the API instances are protected by Client ID enforcement API policy instances, and hence require client ID and secret to be presented by API clients. client ID and secret could be sent in dedicated HTTP request headers or query parameters, but it is also possible to reuse the standard Authorization header defined by HTTP Basic Authentication. In the latter case, client ID and secret take the role of username and password in HTTP Basic Authentication, respectively. It is this approach that the Client ID enforcement API policy instances of all AnyAirline APIs take. Security configuration of the HTTP Connector therefore entails supplying username and password, which are really the client ID and secret supplied to you, for an API client already created.

1. **Add HTTP Connector configs:** In Studio, using the visual Mule config flow editor, add to **global.xml** a HTTP Connector configuration for Flights Management SAPI, Passenger Data SAPI,

and PayPal SAPI, configuring the correct host and authentication with the supplied client ID and secret for an API client already created in Anypoint Platform:

global.xml of check-in-papi

```
<http:request-config
  name="flightsManagementSapiConfig"
  basePath="/api/v1">
  <http:request-connection
    host="tngaa-flights-management-sapi-dev-9yj2rh.rajrd4-2.usa-
e1.cloudhub.io"
    protocol="HTTPS">
    <http:authentication>
      <http:basic-authentication
        username="e66105d66e8b4520b091127620221cce"
        password="A1c099D51C50420b84641D71b229EBac" />
    </http:authentication>
  </http:request-connection>
</http:request-config>
<http:request-config name="passengerDataSapiConfig"
  basePath="/api/v1">
  <http:request-connection
    host="tngaa-passenger-data-sapi-dev-9yj2rh.rajrd4-2.usa-
e1.cloudhub.io"
    protocol="HTTPS">
    <http:authentication>
      <http:basic-authentication
        username="e66105d66e8b4520b091127620221cce"
        password="A1c099D51C50420b84641D71b229EBac" />
    </http:authentication>
  </http:request-connection>
</http:request-config>
<http:request-config name="paypalSapiConfig"
  basePath="/api/v1">
  <http:request-connection
    host="tngaa-paypal-sapi-dev-9yj2rh.rajrd4-2.usa-e1.cloudhub.io"
    protocol="HTTPS">
    <http:authentication>
      <http:basic-authentication
        username="e66105d66e8b4520b091127620221cce"
        password="A1c099D51C50420b84641D71b229EBac" />
    </http:authentication>
  </http:request-connection>
</http:request-config>
```

Note: There are sample JSON files in the src/main/resources/examples directory for the request and response to use as metadata or as a guide.

Note: Username and password refer to client ID and secret because the Client ID enforcement API policy on all APIs is configured to use HTTP Basic Authentication to pass client ID and secret.

2. **Homework: Encrypt client ID and secret:** Encrypt and store client ID and secret in dev-secure-properties.yaml using the usual encryption key (secure12345).

Invoke downstream System APIs using the HTTP Connector with full control over the HTTP message

In this section, you implement the essential parts of the check-in functionality of check-in-papi using the HTTP Connector modifying the HTTP request as required.

3. **Invoke Flights Management SAPI check-in:** In Studio, add to the **check-in-flights-management** flow in **main.xml** a simple invocation of the check-in functionality provided by Flights Management SAPI, using the data already available in that flow to prepare the request payload, and extracting just the API invocation proper into a subflow:

   main.xml of check-in-papi

```
<flow name="check-in-flights-management">
  <ee:transform>
    <ee:message>
      <ee:set-payload><![CDATA[%dw 2.0
      output application/json
      ---
      {
        lastName: vars.checkIn.lastName,
        numBags:  vars.checkIn.numBags
      }]]></ee:set-payload>
    </ee:message>
  </ee:transform>
  <http:request config-ref="flightsManagementSapiConfig" method="PUT"
path="/tickets/{PNR}/checkin">
            <http:uri-params ><![CDATA[#[output application/java
---
{
    "PNR" : vars.PNR
}]]]></http:uri-params>
    </http:request>
</flow>
```

Note: By default the HTTP request body is taken from the payload, which is the approach used here, so that just the API invocation — and not the request payload creation — can be extracted into a subflow. This benefits testability.

4. **Homework:** Extract the API invocation into a subflow to improve testability.
5. **Run and invoke:** Run check-in-papi and invoke the check-in functionality it exposes; this should succeed, thereby confirming successful authentication with client ID and secret of check-in-papi against Flights Management SAPI in dev:

```
curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
:\"Smith\",\"numBags\":2}"
https://localhost:8081/api/v1/tickets/PNR123/checkin
```

6. **Study check-in validation:** In the solution design, review the validation of the check-in attempt performed as part of **US1: mobile Check-In**.
7. **Implement check-in validation:** In main.xml, implement the validation of the check-in attempt, for which you first need to retrieve data for ticket and passenger, storing them directly in variables using the **target** attribute:

main.xml of check-in-papi

```
<flow name="check-in-by-pnr">
  ...
  <set-variable variableName="checkIn" value="#[payload]" />
  <flow-ref name="validate-ticket-passport-matches" />
  ...
</flow>

<flow name="validate-ticket-passport-matches">
  <flow-ref name="get-ticket-by-pnr" />
  <choice>
    <when
      expression="#[not vars.ticket.ticketHolderLastName ==
vars.checkIn.lastName]">
      <raise-error type="APP:LASTNAME_MISMATCH" />
    </when>
  </choice>

  <flow-ref name="get-passenger-data-by-passport" />
  <choice>
    <when
      expression="#[not vars.passenger.lastName ==
```

```
vars.checkIn.lastName]">
      <raise-error type="APP:LASTNAME_MISMATCH" />
    </when>
  </choice>
</flow>

<flow name="get-ticket-by-pnr">
  <http:request config-ref="flightsManagementSapiConfig" method="GET"
path="/tickets/{PNR}" target="ticket" doc:name="FMS Get Ticket">
            <http:uri-params ><![CDATA[#[output application/java
---
{
    "PNR" : vars.PNR
}]]]></http:uri-params>
    </http:request>
</flow>

<flow name="get-passenger-data-by-passport">
  <http:request target="passenger" config-
ref="passengerDataSapiConfig" method="GET" path="/passengers">
            <http:query-params ><![CDATA[#[output application/java
---
{
    "passportNo" : vars.ticket.ticketHolderPassPortNo
}]]]></http:query-params>
    </http:request>
</flow>
```

Note: There are sample JSON files in the src/main/resources/examples directory for the request and response to use as metadata or as a guide.

Note: HTTP response bodies are assigned to variables using the target attribute of the HTTP Connector.

Note: The Passenger Data SAPI invocation requires the passport number returned by the Flights Management SAPI.

Note: Validation failures cause the custom application error APP:LASTNAME_MISMATCH to be raised.

Note: Using a Choice router to perform validation is arguably not idiomatic: in a later walkthrough you will use the Validation module for cases such as this one.

8. **Run and invoke:** Run check-in-papi and invoke the check-in functionality it exposes; this should execute and then return a HTTP 500 error with a particular payload:

```
curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
:\"Smith\",\"numBags\":2}"
https://localhost:8081/api/v1/tickets/PNR123/checkin
```

Note: A HTTP 500 error is not appropriate in this situation, where the API client sent a check-in request that fails validation; this should result in a HTTP 4xx error response.

Note: Error responses should never expose any Mule app internals.

9. **Confirm APP:LASTNAME_MISMATCH raised:** Inspect the log output to confirm that the APP:LASTNAME_MISMATCH error was raised and note the **ticket holder last name** actually returned from Flights Management SAPI.
10. **Explain HTTP response:** Inspect all error handling code to explain why in the case of APP:LASTNAME_MISMATCH being raise this particular HTTP response body is returned.

Note: The definition of api-error-handler in error-common.xml does not include a catch-all error handler, so the response payload is the payload at the time the error was raised. This is an important security leak you will correct later.

Note: The HTTP response status is determined by the HTTP Listener configuration in api-main in api.xml and is inappropriate for this situation, where a HTTP 4xx error code is expected.

11. **Handle APP:LASTNAME_MISMATCH:** Add error handling of APP:LASTNAME_MISMATCH to the appropriate Mule flow in **api.xml**, translating from the custom application error to an HTTP error response, taking this opportunity to simplify the setting of PNR:

api.xml of check-in-papi

```
<flow name="put:\tickets\(PNR)\checkin...">
  <set-variable variableName="PNR"
    value="#[attributes.uriParams.PNR]" />
  <flow-ref name="check-in-by-pnr" />
  <error-handler>
    <on-error-continue type="APP:LASTNAME_MISMATCH">
      <set-payload
        value='#[output application/json --- {message: "Invalid
passenger name record or bad data uploaded."}]'
      />
      <set-variable
```

```
        variableName="httpStatus" value="400" />
    </on-error-continue>
  </error-handler>
</flow>
```

Note: This is a client error (400) because the data sent by the API client does not pass validation.

12. **Run and invoke:** Run the API implementation and invoke the API again as before; this should now result in a 400 error response that does not leak app internals.
13. **Invoke:** Invoke the API with the correct ticket holder last name; this should succeed and return a payment ID:

```
curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
:\"Mule\",\"numBags\":2}"
https://localhost:8081/api/v1/tickets/PNR123/checkin
```

14. **Study check-in-by-pnr:** Confirm that check-in-by-pnr, after validation, invokes check-in-flights-management, register-passenger-data, and create-payment-for-bags, in that order.
15. **Raise custom error from check-in-flights-management:** Add a catch-all error handler raising a custom application error to the existing implementation of check-in-flights-management, as a first step to implementing the check-in functionality itself:

    main.xml of check-in-papi

```
<flow name="check-in-flights-management">
  ...
  <error-handler>
    <on-error-continue>
      <raise-error type="APP:CANT_UPDATE_CHECKINS" />
    </on-error-continue>
  </error-handler>
</flow>
```

16. **Implement register-passenger-data:** Fill in the missing implementation of register-passenger-data, transforming the check-in date to UTC as required by Passenger Data SAPI, invoking Passenger Data SAPI, and raising a custom application error from a catch-all error handler:

main.xml of check-in-papi

```
<flow name="register-passenger-data">
  <ee:transform>
    <ee:message>
      <ee:set-payload><![CDATA[%dw 2.0
      output application/json
      ---
      {
        "date_checkin": now() >> "UTC",
        "destination":  vars.ticket.destination,
        "flight_date":  vars.ticket.flightDate,
        "flight_no":    vars.ticket.flightNo,
        "origin":       vars.ticket.origin
      }]]></ee:set-payload>
    </ee:message>
  </ee:transform>

  <http:request config-ref="passengerDataSapiConfig" method="POST"
path="/passengers/{LNO}/flights">
            <http:uri-params ><![CDATA[#[output application/java
---
{
    "LNO" : vars.passenger.loyaltyNo
}]]]></http:uri-params>
    </http:request>

  <error-handler>
    <on-error-continue>
      <raise-error type="APP:CANT_CREATE_PASSENGER_FLIGHT"  />
    </on-error-continue>
  </error-handler>
</flow>
```

Note: There are sample JSON files in the src/main/resources/examples directory for the request and response to use as metadata or as a guide.

Note: DataWeave supports special syntax for time-zone conversions of dates.

Note: You use the same custom application error approach as in the other System API invocations.

17. **Homework:** Extract the API invocation into a subflow to improve testability.

18. **Implement create-payment-for-bags:** Fill in the missing implementation of create-payment-for-bags, assuming a constant cost per bag, invoking PayPal SAPI, and raising the already present custom application error from a catch-all error handler:

    main.xml of check-in-papi

```
<flow name="create-payment-for-bags">
  <ee:transform>
    <ee:message>
      <ee:set-payload><![CDATA[%dw 2.0
      output application/json
      var numBags = vars.checkIn.numBags
      var bagRate = 33.3
      ---
      {
        description: "Check-In of $(numBags) bags at USD $(bagRate)
each.",
        amount:      bagRate*numBags
      }]]></ee:set-payload>
    </ee:message>
  </ee:transform>

  <http:request config-ref="paypalSapiConfig" method="POST"
path="/payments"/>

  <error-handler>
    <on-error-continue>
      <raise-error type="APP:CANT_CREATE_PAYMENT"  />
    </on-error-continue>
  </error-handler>
</flow>
```

    Note: DataWeave supports the direct reading of configuration properties, the values of which are always strings and therefore often need to be type-converted.

    Note: DataWeave supports string interpolation, that is, the evaluation of expressions embedded in strings and subsequent substitution of these expressions with their values.

19. **Homework:** Add to **properties.yaml** a configuration property for the constant cost per bag.
20. **Homework:** Extract the API invocation into a subflow to improve testability.
21. **Return correct response from check-in-by-pnr:** Complete this first implementation of check-in-by-pnr by returning the expected response payload, replacing the current hard-coded payload:

main.xml of check-in-papi

```
<flow name="check-in-by-pnr">
  ...
  <flow-ref name="check-in-flights-management"  />
  <flow-ref name="register-passenger-data"  />
  <flow-ref name="create-payment-for-bags"  />

  <ee:transform>
    <ee:message>
      <ee:set-payload><![CDATA[%dw 2.0
      output application/json
      var paypalReturn = payload
      ---
      {
        paymentID: paypalReturn.paymentID
      }]]></ee:set-payload>
    </ee:message>
  </ee:transform>
  <logger level="INFO" message="..."/>
</flow>
```

Note: This code relies on create-payment-for-bags being the last invoked Mule flow before the response payload is constructed.

Note: The return payload from check-in-by-pnr becomes the HTTP response body.

22. **Run, invoke, and check log:** Run check-in-papi and invoke the check-in functionality as before; this should succeed and return a payment ID and the logs should confirm your understanding of the integration logic.

    Note: If the invocation of any downstream API fails, then the entire check-in process fails. This will be addressed in later walkthroughs.

23. **MUnit-test:** Run the MUnit test suite of check-in-papi; this should fail because some assumptions embedded in that test suite no longer hold:

    ```
    cd $APDL2WS/check-in-papi
    mvn clean verify -U -Dencrypt.key=secure12345
    ```

24. **Study failed test:** Inspect the log output of the test execution and compare it with the MUnit implementation of the happy-path test for check-in-by-pnr.

Note: This unit test performs API invocations, which is counter the nature of unit tests; these must be mocked.

Note: A test assertion fails because of a payload mismatch.

25. **Disable failing tests:** To **main-test-suite.xml**, ignore the happy-path test for check-in-by-pnr:

    main-test-suite.xml of check-in-papi

    ```
    <munit:test name="check-in-by-pnr-happy-path-test" ignore="true">
    ...
    </munit:test>
    ```

26. **MUnit-test:** Run the MUnit test suite of check-in-papi; this should succeed:

    ```
    cd $APDL2WS/check-in-papi
    mvn clean verify -U -Dencrypt.key=secure12345
    ```

27. **Homework:** Adapt the MUnit test suite to changes in invoked APIs.

Handle all errors raised while processing an incoming HTTP request

In this section, you ensure that no HTTP error raised during the processing of an HTTP request accepted by an <http:listener /> goes unhandled, thereby fixing a potential issue observed previously in that an uncontrolled HTTP response may be sent back to the HTTP client if an error is not handled. This is a potential security leak.

This activity is directly related to how <http:listener /> and APIkit deal with errors during the processing of incoming HTTP requests. <http:request /> are a fertile source of errors, and so the issue addressed here is so prevalent as to be unavoidable whenever HTTP requests are being made.

28. **Trace errors raised by <http:request />:** Inspecting all relevant source code, trace any error raised by the <http:request /> used to invoke the Flights Management SAPI back to the <http:listener /> that is responsible for sending the HTTP response back to the HTTP client.

    Note: When an error is propagated by the top-level Mule flow in api.xml that handles the HTTP request, it reaches APIkit, which in turn propagates the error. The error is then processed by api-error-handler, the error handler of api-main.

Note: The last opportunity for handling any such error is therefore in api-error-handler, which is defined in apps-commons.

29. **Add catch-all error handler:** In **apps-commons**, add a catch-all on-error-propagate error handler to **api-error-handler** defined in error-common.xml:

    error-common.xml of apps-commons

```
<error-handler name="api-error-handler">
...
  <on-error-propagate>
    <ee:transform>
      <ee:message>
        <ee:set-payload><![CDATA[%dw 2.0
        output application/json
        ---
        {message: "Internal server error"}]]></ee:set-payload>
      </ee:message>
      <ee:variables>
        <ee:set-variable variableName="httpStatus">500</ee:set-
variable>
      </ee:variables>
    </ee:transform>
  </on-error-propagate>
</error-handler>
```

    Note: This is an on-error-propagate error handler written in the chosen style to follow the example set by the other error handlers in api-error-handler.

30. **Maven-build:** Run a full Maven build of apps-commons, installing the JAR into your local Maven repository:

```
cd $APDL2WS/apps-commons
mvn clean install
```

31. **Refresh Studio project:** Close and reopen the Studio project for **check-in-papi**, to force Studio to reload the new build of apps-commons.

    Note: Inspect the contents of the apps-commons project library in check-in-papi to confirm that the new version was loaded.

32. **Run and invoke:** Prove that the implemented changes result in an HTTP error response being

returned if any otherwise unhandled error is raised, for example by forcing a timeout error.

Walkthrough 1-4: Enable an API client for OAuth 2.0

PayPal uses OAuth 2.0 to protect its API endpoints, and paypal-sapi must therefore be an OAuth 2.0 client.

In this walkthrough, you add OAuth 2.0 client capabilities to paypal-sapi on top of an otherwise complete implementation of API client functionality based on the HTTP Connector.

You will:

- Set up the starter code for paypal-sapi.
- Investigate OAuth 2.0 authentication against PayPal.
- Enable paypal-sapi for OAuth 2.0 against PayPal.
- Log and confirm OAuth 2.0 interactions.

Solution file

You can see the end state of following this walkthrough in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module01/wt1-4_solution.

Starting file

To follow this walkthrough, you need the starter project at $APDL2DIST/walkthroughs/devint/module01/wt1-4_starter.

Set up the starter code for paypal-sapi

In this section, you copy a basically functional starter implementation of paypal-sapi into your workspace and adapt it to build and run locally. To achieve this, you must once again disable autodiscovery.

The imported starter code for paypal-sapi lacks the required OAuth 2.0 authentication against PayPal, which is why it is not fully functional. You close this feature gap shortly.

1. **Study Solution Architecture:** Study the Solution architecture and in particular the design of US1: mobile Check-In and PayPal payments ; note the interaction of PayPal SAPI with PayPal and the usage of OAuth 2.0 in this interaction.
2. **Copy paypal-sapi starter:** Copy the paypal-sapi starter project at **$APDL2DIST/walkthroughs/devint/module01/wt1-4_starter/paypal-sapi** into your Studio workspace directory:

```
cd $APDL2WS
```

```
cp -r $APDL2DIST/walkthroughs/devint/module01/wt1-4_starter/paypal-
sapi ./
```

3. **Study Maven build:** Familiarize yourself briefly with the Maven build configuration of paypal-sapi.

   Note: The PayPal SAPI API specification is loaded as a Maven dependency with artifact ID paypal-sapi in the form of a RAML definition from the AnyAirline Exchange Maven repo, which requires authentication entries in settings.xml, created in a previous walkthrough.

   Note: The paypal-sapi Mule app is assigned to the System layer in Visualizer.

   Note: As usual, reusable Maven plugin configuration is inherited from the parent POM and Maven dependency management from the BOM. Both parent-pom/pom.xml and bom/pom.xml are read from the Studio workspace directory.

4. **Update groupId:** Update pom.xml adapting the Maven coordinates to match the groupId of your Anypoint Platform organization ID for the project and the parent:

   pom.xml of apps-commons

   ```
   <project>
    <parent>
       <groupId>your-AP-organization-id</groupId>
       ...
     </parent>
     <groupId>your-AP-organization-id</groupId>
     ...
   </project>
   ```

5. **Maven-build:** Run a full Maven build of paypal-sapi including all MUnit tests, providing the required secure properties encryption key; this should succeed and all unit tests should pass:

   ```
   cd $APDL2WS/paypal-sapi
   mvn clean verify -U -Dencrypt.key=secure12345
   ```

6. **Import into Studio:** In Studio, import paypal-sapi without copying it to the workspace, because it is already there; confirm that the import succeeded.
7. **Add temporary env property:** In Studio, near the top of **global.xml**, define a property **encrypt.key** to default the property required by the Studio tooling instance:

global.xml

```
<global-property name="encrypt.key" value="secure12345" />
```

Note: The encrypt.key property should not be defaulted and should be removed. This is a temporary solution to provide the runtime property to the Studio tooling instance, required to utilize DataSense metadata support.

8. **Add run config disabling autodiscovery:** Add a Studio run configuration (via the **Run as > Mule application (configure)** dialog) for paypal-sapi that sets **encrypt.key** and disables autodiscovery:

```
-M-Dencrypt.key=secure12345
-M-Danypoint.platform.gatekeeper=disabled
```

Note: The API ID values configured in the properties files of paypal-sapi for the various environments stem from the AnyAirline Anypoint Platform organization, to which you have not been given access; disabling autodiscovery works around this restriction by preventing the uplink to API Manager without requiring a code change (such as removing the autodiscovery configuration).

9. **Run and invoke:** Run paypal-sapi and invoke the exposed PayPal SAPI and health check endpoints; the health checks should succeed but invoking PayPal SAPI should fail externally with a 500 error and internally with a 4xx error when invoking PayPal, because PayPal requires API clients to authenticate with OAuth 2.0, which this version of paypal-sapi does not yet do:

```
curl -ik -X POST -H "Content-Type: application/json" -d "{\"amount\":
12.34, \"description\": \"something\"}"
https://localhost:8081/api/v1/payments
```

```
curl -ik -X PUT -H "Content-Type: application/json" -d "{\"payerID\":
\"STJ8222K092ST\"}" https://localhost:8081/api/v1/payments/PAY-
1B56960729604235TKQQIYVY/approval
```

```
curl -ik https://localhost:8081/alive
```

```
curl -ik https://localhost:8081/ready
```

Note: Because this API implementation uses self-signed certificates, you must explicitly allow them when invoking these HTTPS endpoints.

10. **Study paypal-sapi:** Inspect paypal-sapi so that you can explain the HTTP requests just performed, the HTTP responses received, and the MUnit tests executed in the previous Maven build.

Investigate successful and unsuccessful OAuth 2.0 authentication

In this section, you investigate how the API invocations of paypal-sapi to PayPal currently fail because the former does not honor the OAuth 2.0 authentication constraint enforced by the latter. You discover that PayPal uses the client credentials grant type for these invocations. You perform the required API invocations manually, such as from a command-line interface.

11. **Check log:** In the paypal-sapi logs from the previous API invocations, locate evidence of a PayPal invocation, and the failure thereof; you should see that the invocation of PayPal failed with an HTTP 401 Unauthorized response status.

    Note: If you see timeout errors instead, then either try again or increase the PayPal response timeout in dev-properties.yaml.

    Note: The PayPal endpoint invoked by paypal-sapi in the dev environment is that of a fake PayPal API implementation, at https://training-paypal-fake-api-sandbox-mjf1rw.5sc6y6-1.usa-e2.cloudhub.io.

12. **Get PayPal access token:** In the PayPal developer documentation for v1 of the PayPal REST API at https://developer.paypal.com/reference/get-an-access-token/ find the HTTP request details for getting an access token from PayPal via any of the documented methods (cURL or Postman), and perform such an API invocation against the PayPal endpoint used by paypal-sapi; this should return a token response similar to that shown in the PayPal developer documentation:

    ```
    curl -i -X POST  -H "Accept: application/json" -H "Accept-Language:
    en_US" -u "APP-80ANYAIRLINE8184JT3:1929FHDUAL8392K9ABKSNMM" -d
    "grant_type=client_credentials" https://training-paypal-fake-api-
    sandbox-mjf1rw.5sc6y6-1.usa-e2.cloudhub.io/v1/oauth2/token
    ```

    Note: Authentication of the token request is with client ID and secret known to be acceptable to

this fake PayPal implementation.

Note: This results in an HTTP POST request with content type application/x-www-form-urlencoded.

Note: This is the first step of OAuth 2.0 authentication with the client credentials grant type.

Note: The current implementation of paypal-sapi does not perform this step.

13. **Extract access token** From the PayPal HTTP response body, extract the value of the access token, either through copying and pasting or tool support:

    Unix variants

    ```
    atoken=$(curl -X POST -H "Accept: application/json" -H "Accept-Language: en_US" -u "APP-80ANYAIRLINE8184JT3:1929FHDUAL8392K9ABKSNMM" -d "grant_type=client_credentials" https://training-paypal-fake-api-sandbox-mjf1rw.5sc6y6-1.usa-e2.cloudhub.io/v1/oauth2/token | jq -r '.access_token')
    ```

    Note: This cURL command is configured to only return the HTTP response body, otherwise parsing the output as JSON fails.

    Windows

    ```
    curl -X POST -H "Accept: application/json" -H "Accept-Language: en_US" -u "APP-80ANYAIRLINE8184JT3:1929FHDUAL8392K9ABKSNMM" -d "grant_type=client_credentials" https://training-paypal-fake-api-sandbox-mjf1rw.5sc6y6-1.usa-e2.cloudhub.io/v1/oauth2/token
    ```

    Note: You must manually copy and paste the access token if not using tooling such as jq.

14. **Create PayPal payment:** In the PayPal developer documentation for v1 of the PayPal REST API at https://developer.paypal.com/docs/api/payments/v1/#payment_create, find the HTTP request details for creating a PayPal payment via any of the documented methods (cURL or Postman), and perform such an API invocation against the PayPal endpoint used by paypal-sapi, passing the previously extracted access token as a **bearer token**; this should successfully return a JSON payment creation response:

    Unix variants

    ```
    curl -i -X POST -H "Content-Type: application/json" -H "Authorization: Bearer $atoken" -d "{ \"intent\":\"sale\", \"payer\":{
    ```

```
\"payment_method\":\"paypal\" }, \"transactions\":[ { \"amount\":{
\"total\":\"80.00\", \"currency\":\"USD\" }, \"description\": \"Check-
In Baggage.\", \"custom\": \"ANYAIRLINE_90048630024435\",
\"invoice_number\": \"48787589673\", \"payment_options\":{
\"allowed_payment_method\":\"INSTANT_FUNDING_SOURCE\" },
\"soft_descriptor\":\"ANYAIRLINE BAGGAGE\" } ], \"note_to_payer\":\"Be
happy.\" }" https://training-paypal-fake-api-sandbox-mjf1rw.5sc6y6-
1.usa-e2.cloudhub.io/v1/payments/payment
```

Windows

```
SET atoken="VALUE RETURNED FROM PREVIOUS CURL REQUEST IF NOT USING JQ"
curl -i -X POST -H "Content-Type: application/json" -H "Authorization:
Bearer %atoken%" -d "{ \"intent\":\"sale\", \"payer\":{
\"payment_method\":\"paypal\" }, \"transactions\":[ { \"amount\":{
\"total\":\"80.00\", \"currency\":\"USD\" }, \"description\": \"Check-
In Baggage.\", \"custom\": \"ANYAIRLINE_90048630024435\",
\"invoice_number\": \"48787589673\", \"payment_options\":{
\"allowed_payment_method\":\"INSTANT_FUNDING_SOURCE\" },
\"soft_descriptor\":\"ANYAIRLINE BAGGAGE\" } ], \"note_to_payer\":\"Be
happy.\" }" https://training-paypal-fake-api-sandbox-mjf1rw.5sc6y6-
1.usa-e2.cloudhub.io/v1/payments/payment
```

Note: You can copy the sample request from https://developer.paypal.com/docs/api/payments/v1/#payment_create but first you need to paste it into a text editor so that you can edit the URL and token.

Note: This API invocation is to the now deprecated v1 of the PayPal API, which is implemented by the endpoints used here.

Note: This is the second step when invoking an API protected by OAuth 2.0 with the client credentials grant type.

Note: The current implementation of paypal-sapi performs this step but does not pass an access token and therefore receives a HTTP 4xx error response.

15. **Homework: Compare to OAuth 2.0 spec:** Compare the above interactions to the client credentials section in the OAuth 2.0 specification at https://tools.ietf.org/html/rfc6749.

Add OAuth 2.0 authentication to an API client using the HTTP Connector

In this section, you add OAuth 2.0 client support through the OAuth module to paypal-sapi.

Because this module must store tokens retrieved from the downstream OAuth 2.0-protected API, it is a stateful component. To store state, the OAuth module requires an Object Store, and you therefore also add the Object Store Connector.

16. **Confirm Maven dependency management:** In **bom/pom.xml**, locate the existing dependency management entries for the **OAuth module** and the **Object Store Connector**.
17. **Add Maven dependencies:** Add the corresponding Maven dependencies to the **POM** of paypal-sapi; confirm that Studio loads both modules:

    pom.xml of paypal-sapi

    ```
    <dependency>
      <groupId>org.mule.connectors</groupId>
      <artifactId>mule-objectstore-connector</artifactId>
      <classifier>mule-plugin</classifier>
    </dependency>
    <dependency>
      <groupId>org.mule.modules</groupId>
      <artifactId>mule-oauth-module</artifactId>
      <classifier>mule-plugin</classifier>
    </dependency>
    ```

    Note: Omit the <version /> elements so that the versions managed in the BOM take effect.

18. **Add OAuth 2.0 and Object Store config:** To the existing HTTP Request configuration in **global.xml** add authentication via the **client credentials** grant type, configuring an inline token manager referencing a new Object Store with suitable sizing and expiration settings:

    global.xml of paypal-sapi

    ```
    <http:request-config
      name="paypalServerHttpRequestConfig"
      followRedirects="true"
      responseTimeout="${paypal.responseTimeoutMillis}">
      <http:request-connection>
        <http:authentication>
          <oauth:client-credentials-grant-type
            clientId="APP-80ANYAIRLINE8184JT3"
            clientSecret="1929FHDUAL8392K9ABKSNMM"
            tokenUrl="https://training-paypal-fake-api-sandbox-
    mjf1rw.5sc6y6-1.usa-e2.cloudhub.io/v1/oauth2/token">
            <oauth:token-manager objectStore="tokenStore" />
          </oauth:client-credentials-grant-type>
    ```

```
    </http:authentication>
  </http:request-connection>
</http:request-config>

<os:object-store name="tokenStore"
  maxEntries="1000"
  entryTtl="60" entryTtlUnit="MINUTES"
  expirationInterval="30" expirationIntervalUnit="MINUTES" />
```

Note: The values for client ID and secret and the token URL are the same as in the previous manual token request.

Note: The default expressions for extracting the access token and expiration duration from the HTTP response to the retrieve token request match the response you have seen previously being returned from PayPal.

Note: The token Object Store is persistent by default, and this is also required here so that tokens are shared between multiple workers in a replicated CloudHub deployment.

Note: You limit the number of tokens that may be stored in the token Object Store to protect against attacks. This number can be small as all tokens authenticate paypal-sapi against PayPal (there is no user authentication in the client credentials grant type).

Note: You also limit the lifetime (TTL) of each token and explicitly specify at what interval this will be enforced. This client-side expiration is layered on top of the server-side expiration of tokens performed by PayPal.

19. **Run and invoke:** Run paypal-sapi and invoke the exposed PayPal SAPI as before; this should now succeed, proving that paypal-sapi now interacts via OAuth 2.0 with PayPal:

```
curl -ik -X POST -H "Content-Type: application/json" -d "{\"amount\":
12.34, \"description\": \"something\"}"
https://localhost:8081/api/v1/payments
```

```
curl -ik -X PUT -H "Content-Type: application/json" -d "{\"payerID\":
\"STJ8222K092ST\"}" https://localhost:8081/api/v1/payments/PAY-
1B56960729604235TKQQIYVY/approval
```

Note: The only change needed to make paypal-sapi perform the OAuth 2.0 dance was to add appropriate configuration to the HTTP Request configuration — the individual <http:request /> usages remained untouched.

20. **Homework: Extract config props:** Extract all hard-coded configuration values, for all supported environments, into appropriate configuration properties — environment-dependent and/or environment-independent, encrypted and/or cleartext, as appropriate.

    global.xml of paypal-sapi

```
<http:request-config
  name="paypalServerHttpRequestConfig"
  followRedirects="true"
  responseTimeout="${paypal.responseTimeoutMillis}">
  <http:request-connection>
    <http:authentication>
      <oauth:client-credentials-grant-type
        clientId="${secure::app.client_id}"
        clientSecret="${secure::app.client_secret}"
        tokenUrl="${paypal.tokenUrl}">
        <oauth:token-manager objectStore="tokenStore" />
      </oauth:client-credentials-grant-type>
    </http:authentication>
  </http:request-connection>
</http:request-config>

<os:object-store name="tokenStore"
  maxEntries="${paypal.tokenStore.maxEntries}"
  entryTtl="${paypal.tokenStore.entryTtlMins}"
  entryTtlUnit="MINUTES"
  expirationInterval="${paypal.tokenStore.expirationIntervalMins}"
  expirationIntervalUnit="MINUTES" />
```

    Note: You can use the same approach as in a previous walkthrough of associating just one pair of client ID and secret with the Mule app for invoking backend systems — PayPal, in this case. This works fine for a System API such as paypal-sapi, because it typically invokes only one backend system. It also works fine for a Process API such as check-in-papi, because when the System APIs it invokes are all managed by Anypoint Platform then the same client ID and secret pair is used for all these invocations (as was shown in a previous walkthrough).

Log and confirm OAuth 2.0 interactions

In this section, you turn on logging of all HTTP traffic and confirm that the HTTP requests performed by the OAuth module are equivalent to those you have previously performed manually, just as required by the client credentials OAuth 2.0 grant type.

21. **Check log:** In the log output shown in Studio for the previous invocation of PayPal SAPI, try to

trace OAuth 2.0-related interactions with PayPal; they are currently not logged directly.

22. **Log HTTP traffic:** In **log4j2.xml**, turn-on logging of HTTP traffic by setting the log level of the appropriate logger to **DEBUG**:

    log4j2.xml of paypal-sapi

    ```
    <Loggers>
      <AsyncLogger
        name="org.mule.service.http.impl.service.HttpMessageLogger"
        level="DEBUG" />
    </Loggers>
    ```

    Note: As always, this Log4J configuration applies to all environments, including prod, but not to unit tests.

23. **Run, invoke, and check log:** Restart paypal-sapi, invoke PayPal SAPI again, in the log find OAuth 2.0-related log entries, and confirm that this traffic fits the client credentials grant type and the manual HTTP requests you have performed previously.

    Note: The token retrieved by the OAuth module is the same that the module later sends in the Authorization HTTP request header.

24. **Confirm token re-use:** Perform a second invocation to confirm that the OAuth module reuses a previously retrieved token.
25. **Homework: Test token expiration:** Await the server-side and client-side token expiration timeouts and confirm that the OAuth module handles these cases correctly.

Walkthrough 1-5: Invoke a SOAP web service using mutual TLS authentication

The Flights Management system exposes a SOAP web service over an HTTPS endpoint that is protected by mutual TLS authentication (mTLS). The flights-management-sapi Mule app must implement Flights Management SAPI by invoking this SOAP web service.

In this walkthrough, you start with a skeleton of flights-management-sapi that does not interact with the Flights Management system at all, and add invocations of the Flights Management system SOAP web service that address both the functional and nonfunctional aspects of this interaction, including, but not limited to, the presentation of a client certificate to pass mutual TLS authentication.

You will:

- Setup the starter code for flights-management-sapi.
- Configure Web Service Consumer Connector to access the Flights Management system SOAP web service.
- Send a client certificate to pass mutual TLS authentication.

Solution file

You can see the end state of following this walkthrough in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module01/wt1-5_solution.

Starting file

To follow this walkthrough, you need the starter project at $APDL2DIST/walkthroughs/devint/module01/wt1-5_starter.

Setup the starter code for flights-management-sapi

In this section, you copy a minimal starter implementation of flights-management-sapi into your workspace and adapt it to build and run locally. To achieve this, you must disable autodiscovery, as done previously.

The imported starter code for flights-management-sapi exposes Flights Management SAPI over HTTPS but lacks all interaction with the Flights Management system. You implement this shortly.

1. **Study Solution Architecture:** Study the Solution architecture and in particular the design of US1: mobile Check-In; note the interaction of Flights Management SAPI with the Flights Management system and the usage of mTLS in this interaction.

2. **Copy flights-management-sapi starter:** Copy the flights-management-sapi starter project at **$APDL2DIST/walkthroughs/devint/module01/wt1-5_starter/flights-management-sapi** into your Studio workspace directory:

```
cd $APDL2WS
cp -r $APDL2DIST/walkthroughs/devint/module01/wt1-5_starter/flights-management-sapi ./
```

3. **Study Maven build:** Familiarize yourself briefly with the Maven build configuration of flights-management-sapi.

   Note: The Flights Management SAPI API specification is loaded in the form of a RAML definition from the AnyAirline Exchange Maven repository, which requires an authentication entry in settings.xml, which you have created in a previous walkthrough.

   Note: The Flights Management SAPI API specification imports and hence depends on RAML fragments.

   Note: The flights-management-sapi Mule app is assigned to the System layer in Visualizer.

   Note: As always, reusable Maven plugin configuration is done in a parent POM and Maven dependency management in a BOM. Both parent POM and BOM are read from the parent directory of flights-management-sapi.

4. **Update groupId:** Update pom.xml adapting the Maven coordinates to match the groupId of your Anypoint Platform organization ID for the project and the parent:

   pom.xml of apps-commons

```
<project>
 <parent>
    <groupId>your-AP-organization-id</groupId>
    ...
  </parent>
  <groupId>your-AP-organization-id</groupId>
  ...
</project>
```

5. **MUnit-test:** Run a full Maven build of flights-management-sapi including all unit tests using the same secure properties encryption key used in previous walkthroughs:

```
cd $APDL2WS/flights-management-sapi
```

```
mvn clean verify -U -Dencrypt.key=secure12345
```

6. **Import into Studio:** In Studio, import flights-management-sapi without copying it to the workspace, because it is already there; confirm that the import succeeded.
7. **Add temporary env property:** In Studio, near the top of **global.xml**, define a property **encrypt.key** to default the property required by the Studio tooling instance:

   global.xml of flights-management-sapi

   ```
   <global-property name="encrypt.key" value="secure12345" />
   ```

   Note: The encrypt.key property should not be defaulted and should be removed. This is a temporary solution to provide the runtime property to the Studio tooling instance, required to utilize DataSense metadata support.

8. **Add run config disabling autodiscovery:** Add a Studio run configuration (via the **Run as > Mule application (configure)** dialog) for flights-management-sapi that sets **encrypt.key** and disables autodiscovery:

   ```
   -M-Dencrypt.key=secure12345
   -M-Danypoint.platform.gatekeeper=disabled
   ```

   Note: The API ID configured in the environment-dependent properties files of flights-management-sapi stems from the AnyAirline Anypoint Platform organization, to which you have not been given access; disabling autodiscovery works around this restriction by preventing the uplink to API Manager without requiring a code change (such as removing the autodiscovery configuration).

9. **Run and invoke:** Run flights-management-sapi and invoke the exposed Flights Management SAPI; this should succeed, returning hard-coded HTTP responses:

   ```
   curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
   :\"Mule\",\"numBags\":2}"
   https://localhost:8081/api/v1/tickets/PNR123/checkin
   ```

   ```
   curl -ik https://localhost:8081/api/v1/tickets/PNR123
   ```

```
curl -ik https://localhost:8081/alive
```

```
curl -ik https://localhost:8081/ready
```

10. **Study flights-management-sapi:** Inspect flights-management-sapi so that you can explain the HTTP requests just performed and the HTTP responses received, including the health check endpoints in your investigation.

Configure Web Service Consumer Connector to access a SOAP web service

In this section, you enable flights-management-sapi to invoke the SOAP web service exposed by the Flights Management system. This SOAP web service is exposed over HTTPS and enforces mTLS. To ease development of API clients, however, the deployments of Flights Management system in the dev and test environments expose additional HTTPS endpoints that do not require mTLS. You first invoke the non-mTLS dev endpoint of Flights Management system, before adding mTLS support later.

11. **Get WSDL:** Retrieve the WSDL of the **Flights Management system SOAP web service** in the **dev** environment:

    ```
    curl -ik https://tngaa-flights-management-devx-9yj2rh.d5n5q8.usa-e1.cloudhub.io/api/v1/FlightsManagementService?wsdl
    ```

    Note: This sends a HTTP GET request to a public endpoint of the Mule app that uses the default TLS context of a CloudHub 2.0 Private Space without mTLS configured.

12. **Confirm Maven dependency management:** In **bom/pom.xml**, locate the existing dependency management entries for the **Web Service Consumer Connector**, which has artifact ID **mule-wsc-connector**.
13. **Add Maven dependency:** Add the corresponding Maven dependency for the Web Service Consumer Connector to the POM of flights-management-sapi; confirm that Studio loads the module:

    pom.xml of flights-management-sapi

    ```
    <dependency>
      <groupId>org.mule.connectors</groupId>
      <artifactId>mule-wsc-connector</artifactId>
    ```

```
  <classifier>mule-plugin</classifier>
</dependency>
```

Note: Omit the <version /> element so that the version managed in the BOM takes effect.

14. **Config Web Service Consumer Connector:** In **global.xml**, add configuration for the Web Service Consumer Connector, using information from the WSDL retrieved previously: configure a bespoke HTTP transport configuration pointing to a new HTTP Request configuration that requires HTTPS and in turn points to a new global TLS context configuration:

    global.xml of flights-management-sapi

    ```
    <wsc:config name="flightsWSCConfig">
      <wsc:connection
        wsdlLocation="https://tngaa-flights-management-devx-
    9yj2rh.d5n5q8.usa-e1.cloudhub.io/api/v1/FlightsManagementService?wsdl"
        service="FlightsManagementService"
        port="FlightsManagementPort"
        address="https://tngaa-flights-management-devx-9yj2rh.d5n5q8.usa-
    e1.cloudhub.io/api/v1/FlightsManagementService">
      </wsc:connection>
    </wsc:config>
    ```

    Note: As per typical SOAP conventions, the WSDL location (URL) is simply derived from the endpoint address (URL).

15. **Call SOAP web service:** In **main.xml**, in the **check-in-by-pnr** flow, replace the placeholder with the actual SOAP call, that is, consume the Flights Management system SOAP web service, transforming the data to the SOAP request body inline:

    main.xml of flights-management-sapi

    ```
    <flow name="check-in-by-pnr">
      ...
      <try>
        <flow-ref name="flights-wsc-check-in" />
        ...
      </try>
      ...
    </flow>
    <sub-flow name="flights-wsc-check-in">
      <wsc:consume operation="checkIn" config-ref="flightsWSCConfig">
        <wsc:message>
    ```

```
        <wsc:body><![CDATA[#[%dw 2.0
      output application/xml
      ns ns0 http://flightsMgmt.sword.com/
     ---
      {
        ns0#checkIn: {
          pnr:               vars.PNR,
          passengerLastName: payload.lastName,
          numOfBags:         payload.numBags
       }
      }]]]></wsc:body>
      </wsc:message>
  </wsc:consume>
</sub-flow>
```

Note: This syntax assumes that the message payload contains the SOAP request body.

Note: You must specify the correct XML namespace in DataWeave to construct an XML document that obeys the SOAP web service contract expressed in the WSDL.

Note: This is only the SOAP request body, which will be wrapped into a SOAP envelope by the Web Service Consumer Connector.

Note: If Studio fails to load the WSDL at design time, then it can't assist you in designing this transformation.

16. **Log HTTP traffic:** In **log4j2.xml**, turn on logging of HTTP traffic.
17. **Run and invoke:** Run flights-management-sapi and invoke the check-in functionality of the exposed Flights Management SAPI; this should succeed:

```
curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
:\"Mule\",\"numBags\":2}"
https://localhost:8081/api/v1/tickets/PNR123/checkin
```

18. **Check log:** Scan the log output for all SOAP-related traffic.

Note: The Web Service Consumer Connector retrieves the WSDL before invoking the SOAP operation for the first time.

Note: SOAP invocations are by definition HTTP POST requests.

Note: The SOAP request body is a faithful XML rendering of the output of the above DataWeave

transform expression.

Note: The SOAP response is essentially a boolean wrapped into a SOAP body wrapped into a SOAP envelope.

19. **Unwrap SOAP response and return:** In **main.xml**, in the **check-in-by-pnr**, set the payload to the boolean contained in the SOAP response body, replacing the existing setting of the payload to a hard-coded JSON response, so that this flow returns the boolean returned by the SOAP call:

    main.xml of flights-management-sapi

    ```
    <flow name="check-in-by-pnr">
      ...
      <set-payload
        value="#[output application/java ---
    payload.body.checkInResponse.return as Boolean]" />
    </flow>
    ```

    Note: This encapsulates the handling of SOAP messages in main.xml.

    Note: If desired, XML namespaces can be ignored when reading an XML document in DataWeave.

20. **Return API responses:** In **api.xml**, handle the boolean returned from the check-in-by-pnr flow by creating a suitable HTTP response, either a success response as previously created in main.xml, or a new error response:

    api.xml of flights-management-sapi

    ```
    <flow name="put:\tickets\(PNR)\checkin:application\json:apiConfig">
      <set-variable variableName="PNR"
        value="#[attributes.uriParams.PNR]" />
      <flow-ref name="check-in-by-pnr" />
      <choice>
        <when expression="#[payload]">
          <set-payload
            value="#[output application/json --- {message: 'Passenger
    check-in successful.'}]" />
        </when>
        <otherwise>
          <set-payload
            value="#[output application/json --- {message: 'Invalid
    passenger name record given.'}]" />
    ```

```
      <set-variable variableName="httpStatus"
        value="400" />
    </otherwise>
  </choice>
</flow>
```

Note: This encapsulates the construction of API responses in api.xml.

21. **Run and invoke:** Run flights-management-sapi and invoke the check-in functionality again; this should succeed:

```
curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
:\"Mule\",\"numBags\":2}"
https://localhost:8081/api/v1/tickets/PNR123/checkin
```

22. **MUnit-test:** Run the MUnit test suite of Flights Management SAPI.
23. **Homework:** Implement the get-ticket-by-pnr flow including the invocation of the Flights Management system SOAP web service and the transformations to/from the SOAP payloads.
24. **Homework:** Improve the nonfunctional properties of the SOAP web service invocations by setting a meaningfully short response timeout.

Send a client certificate to pass mutual TLS authentication

In this section, you configure a keystore to be used by the Web Service Consumer Connector through which flights-management-sapi invokes the Flights Management system SOAP web service. That keystore must contain a public/private keypair, of which the public key is sent to the HTTPS server (the Flights Management system). The HTTPS server, in turn, must be configured with that public key so that it can authenticate clients.

For the purposes of this walkthrough, the client's public/private keypair has already been generated and packaged into a keystore, and the Flights Management system (or, more precisely, the CloudHub 2.0 Private Space hosting the Flights Management system) has already been configured to allow access from clients that send the public key from that keypair. What remains to be done in this section is the corresponding configuration of flights-management-sapi.

First you change flights-management-sapi to access a differnet public endpoint of the Flights Management system SOAP web service that requires mutual TLS authentication.

25. **Change SOAP web service endpoint:** In **global.xml**, change the SOAP web service endpoint being invoked, by changing both the address and the derived WSDL location in the Web Service Consumer Connector configuration:

global.xml of flights-management-sapi

```
<wsc:config name="flightsWSCConfig">
  <wsc:connection
    wsdlLocation="https://tngaa-flights-management-devx.nonprod-
internalps.anyair.net/api/v1/FlightsManagementService?wsdl"
    service="FlightsManagementService"
    port="FlightsManagementPort"
    address="https://tngaa-flights-management-devx.nonprod-
internalps.anyair.net/api/v1/FlightsManagementService">
  </wsc:connection>
</wsc:config>
```

26. **Run, invoke, and check log:** Run the Mule app and invoke the check-in functionality of Flights Management SAPI; this should fail with a HTTP 500 error that was triggered internally by a HTTP 4xx error when calling the Flights Management system SOAP web service, because the HTTP client did not send the client-side certificate needed for mTLS.
27. **Locate client's keystores:** Inspect the PKCS12 keystores containing the public/private RSA keypairs to be used for clients to the Flights Management system, provided to you in **src/main/resources/certs**.
28. **Use keystore:** Add and configure a bespoke HTTP transport configuration pointing to a new HTTP Request configuration that requires HTTPS and in turn points to a new global TLS context configuration a keystore configuration referencing the client's **nonprod** keystore, which will be used when the Web Service Consumer Connector connects over HTTPS to a SOAP web service:

global.xml of flights-management-sapi

```
<wsc:config name="flightsWSCConfig">
  <wsc:connection
    wsdlLocation="https://tngaa-flights-management-devx.nonprod-
internalps.anyair.net/api/v1/FlightsManagementService?wsdl"
    service="FlightsManagementService"
    port="FlightsManagementPort"
    address="https://tngaa-flights-management-devx.nonprod-
internalps.anyair.net/api/v1/FlightsManagementService">
    <wsc:custom-transport-configuration>
      <wsc:http-transport-configuration
        requesterConfig="flightsWSHTTPConfig" />
    </wsc:custom-transport-configuration>
  </wsc:connection>
</wsc:config>
```

```
<http:request-config name="flightsWSHTTPConfig">
  <http:request-connection protocol="HTTPS"
    tlsContext="flightsWSTLSContext"/>
</http:request-config>

<tls:context name="flightsWSTLSContext">
  <tls:key-store type="pkcs12"
    path="certs/clients-to-nonprod-internalps.p12"
    password="muleclient2021"
    keyPassword="muleclient2021"
    alias="client" />
</tls:context>
```

Note: Following PKCS12, both key and keystore share the same password. The alias was specified when creating the keystore.

Note: This keystore and the keypair it contains were created using the Java keytool.

29. **Run, invoke, and check log:** Run the Mule app and invoke the check-in API; this should succeed, proving that mutual TLS authentication of HTTPS client and HTTPS server succeeded.
30. **Homework: Extract config props:** Extract all hard-coded configuration values, for all supported environments, into appropriate configuration properties — environment-dependent and/or environment-independent, encrypted and/or cleartext, as appropriate.

Walkthrough 1-6: Implement an HTTP callback

The Flights Management system can notify its clients about flight cancellations by performing a HTTP POST request to a callback endpoint exposed by the API client — a so-called webhook.

In this walkthrough, you extend flights-management-sapi to expose such a webhook and register it with the Flights Management system. This requires you to think about network topology and different types of API clients.

You will:

- Expose an HTTP callback from flights-management-sapi.
- Register the HTTP callback with the Flights Management system.

Solution file

You can see the end state of following this walkthrough in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module01/wt1-6_solution.

Starting file

If you did not complete the previous walkthrough, you can get a starting file located in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module01/wt1-6_solution.

Expose an HTTP callback

In this section, you expose an HTTP callback from flights-management-sapi that will be invoked by the Flights Management system when a flight has been cancelled.

Rather than defining a new HTTP Listener configuration for the callback endpoint, you reuse the HTTP Listener configuration through which the main API exposed by flights-management-sapi, namely Flights Management SAPI, is exposed.

This implies that flights-management-sapi listens on the same network interface and port for invocations from (upstream) clients to Flights Management SAPI as well as for callback invocations from the (downstream) Flights Management system.

These two types of API clients to flights-management-sapi are of a different nature, however, and so you decide that the callback shall not be defined in the API specification of Flights Management SAPI. This API specification defines the interface, which has business purpose, for upstream API clients, which are typically Process APIs such as check-in-papi. On the other hand, the HTTP callback is a technical detail of the interaction with the Flights Management system, and is

mandated by the Flights Management system interface.

This distinction between API clients to flights-management-sapi also applies to versioning: The Check-In PAPI API specification is under the control of AnyAirline. When the version of that API specification changes, then flights-management-sapi has to implement that updated version when it is ready to do so. The HTTP callback, on the other hand, must be implemented by flights-management-sapi as required by the Flights Management system. When the Flights Management system changes its requirements for this callback, then flights-management-sapi has to change its implementation thereof immediately. Thus flights-management-sapi exposes two types of APIs, for two different audiences, with two different lifecycles, and with two different versioning approaches.

Because the HTTP callback is not defined by an API specification, flights-management-sapi does not use APIkit for its implementation.

Also, the HTTP callback endpoint does not use autodiscovery, is not under API Manager control and hence no API policies can be applied to the HTTP callback.

1. **Study Solution Architecture:** Study the Solution architecture and in particular the design of US2: Flight Cancellation mobile Notifications; note how flights-management-sapi registers a HTTP callback with the Flights Management system, and how the Flights Management system POSTs cancellation notifications to that callback.
2. **Expose and implement HTTP callback:** In **main.xml** of **flights-management-sapi** add a flow exposing the HTTP callback, using the same HTTP Listener configuration used for exposing Flights Management SAPI, and assuming an appropriate configuration property for the path:

   main.xml of flights-management-sapi

   ```
   <flow name="receive-cancellation-notification">
     <http:listener
       path="/api/cancelFlight"
       allowedMethods="POST"
       config-ref="apiHttpListenerConfig">
       <http:response statusCode="#[vars.httpStatus default 202]">
         <http:body>#[output text/plain --- vars.response default
   'OK']</http:body>
       </http:response>
     </http:listener>

     <logger level="INFO"
       message="Received Cancellation Notification"/>
   </flow>
   ```

   Note: You return HTTP 202 ACCEPTED rather than HTTP 200 OK to inform the Flights

Management system that the cancellation notification has been accepted for later asynchronous processing.

Note: This implementation just logs the cancellation notification; you will validate and use it in a later walkthrough.

Note: In a later walkthrough, you will add defensive error handling because this endpoint is not protected by API Manager policies.

3. **Homework:** Turn the hard-coded path into a property.
4. **Run and invoke:** Run the Mule app and POST a cancellation notification to the HTTP callback just like the Flights Management system would; this should succeed:

```
curl -ik -H "Content-Type:text/xml" -d
"<CancellationNotification><PNR>PNR123</PNR><PassengerLastName>Mule</P
assengerLastName></CancellationNotification>"
https://localhost:8081/api/cancelFlight
```

Register the HTTP callback with an HTTP service

In this section, you extend flights-management-sapi to invoke a SOAP web service operation on the Flights Management system to register the HTTP callback exposed by flights-management-sapi. Registration is done just once, shortly after the startup of flights-management-sapi, and from that moment on, the Flights Management system sends HTTP POST requests to that callback whenever a flight is cancelled.

The callback URL to be registered with the Flights Management system must be that of a load balancer in front of a (typically) replicated deployment of flights-management-sapi. In environments where flights-management-sapi is deployed to a CloudHub 2.0 Shared Space, this must be based on the DNS entry for flights-management-sapi that maps to the CloudHub Shared Space Load Balancer (such as https://tngaa-flights-management-sapi-dev-9yj2rh.rajrd4-2.usa-e1.cloudhub.io). Other deployment topologies might require a different address, such as the DNS entry of customer-provisioned load balancer. In any case, the Flights Management system must eventually send HTTP POST requests to that load-balanced callback URL. And for flights-management-sapi to register that URL with the Flights Management system, flights-management-sapi must be configured with its own load balancer hostname in each environment.

The network topology of AnyAirline does not allow the Flights Management system installations in any environment to reach a HTTP callback endpoint exposed on your development machine (localhost). The invocation by the Flights Management system of the callback exposed by flights-management-sapi can therefore only be tested by deploying flights-management-sapi to CloudHub.

5. **Register callback URL:** To **main.xml**, add a flow that is scheduled to run after startup and registers the HTTP callback URL, stored in a property, by invoking the Flights Management system SOAP web service:

   main.xml of flights-management-sapi

```
<flow name="register-callback">
  <scheduler>
    <scheduling-strategy>
      <fixed-frequency
        frequency="365000" timeUnit="DAYS" />
    </scheduling-strategy>
  </scheduler>
  <ee:transform>
    <ee:message>
      <ee:set-payload><![CDATA[%dw 2.0
      output application/xml
      ns ns0 http://flightsMgmt.sword.com/
      ---
      {
        ns0#registerForCancellationNotifications: {
          callbackURL: p("api.callback.url")
        }
      }]]></ee:set-payload>
    </ee:message>
  </ee:transform>
  <try>
    <flow-ref name="register-for-cancellation-notifications" />
    <error-handler>
      <on-error-continue>
        <raise-error type="APP:CANT_REGISTER_CALLBACK" />
      </on-error-continue>
    </error-handler>
  </try>
</flow>
<sub-flow name="register-for-cancellation-notifications">
  <wsc:consume
    operation="registerForCancellationNotifications"
    config-ref="flightsWSCConfig" />
</sub-flow>
```

   Note: This scheduler configuration runs this flow effectively only once after startup of the Mule app.

Note: DataWeave expressions can directly refer to configuration properties.

Note: The callback URL to be invoked must be configured for each environment, because it cannot be determined automatically.

6. **Add callback URL prop:** To **properties.yaml**, add the callback URL as a function of an environment-dependent base URL:

   properties.yaml of flights-management-sapi

```
api:
  callback:
    path:   "/api/cancelFlight"
    url:    "${api.callback.base}${api.callback.path}"
```

7. **Add base URL prop for dev:** To **dev-properties.yaml**, add the callback base URL:

   dev-properties.yaml of flights-management-sapi

```
api:
  callback:
    base: "https://flights-management-sapi-dev-uniqid.shard.usa-
e2.cloudhub.io"
```

   Note: The uniqid and shard in the URL is just a placeholder for now. The complete URL with the unique id for your application will be generated once you deploy to CloudHub 2.0 next. You will update this value later.

8. **Update Mule app and dependencies to release versions:** Update the **BOM**, **parent POM**, **apps-commons** and **flights-management-sapi** artifact version and references to a release version and not a SNAPSHOT:

   bom/pom.xml

```
...
<version>1.0.0</version>
...
<properties>
  ...
  <apps-commons.version>1.0.0</apps-commons.version>
  ...
</properties>
```

```
...
```

parent-pom/pom.xml

```
...
<parent>
  <!-- students: replace with your AP org ID -->
  <groupId>your-AP-organization-id</groupId>
  <artifactId>solutions-bom</artifactId>
  <version>1.0.0</version>
  <relativePath>../bom/pom.xml</relativePath>
</parent>
...
<version>1.0.0</version>
...
```

pom.xml of flights-management-sapi

```
...
<parent>
  <!-- students: replace with your AP org ID -->
  <groupId>your-AP-organization-id</groupId>
  <artifactId>solutions-parent-pom</artifactId>
  <version>1.0.0</version>
  <relativePath>../parent-pom/pom.xml</relativePath>
</parent>
...
<version>1.0.0</version>
...
```

pom.xml of apps-commons

```
...
<parent>
  <!-- students: replace with your AP org ID -->
  <groupId>your-AP-organization-id</groupId>
  <artifactId>solutions-parent-pom</artifactId>
  <version>1.0.0</version>
  <relativePath>../parent-pom/pom.xml</relativePath>
</parent>
...
<version>1.0.0</version>
```

```
...
```

Note: Although Exchange supports SNAPSHOT versions, not all Anypoint Platform components support them such as RTF. Therefore, for consistency you will use release versions for all dependencies and release and increase version numbers before every CloudHub 2.0 deployment.

9. **Install, and study parent POMs:** Install the parent POMs into your local Maven repository:

```
cd $APDL2WS
mvn install:install-file -Dfile=bom/pom.xml        -DpomFile
=bom/pom.xml
mvn install:install-file -Dfile=parent-pom/pom.xml -DpomFile=parent-
pom/pom.xml
```

10. **Maven-build:** Run a full Maven build of apps-commons, installing the JAR into your local Maven repository:

```
cd $APDL2WS/apps-commons
mvn clean install
```

11. **Maven-build:** Run a full Maven build of flights-management-sapi including all unit tests using the same secure properties encryption key used in previous walkthroughs:

```
cd $APDL2WS/flights-management-sapi
mvn clean verify -U -Dencrypt.key=secure12345
```

12. **Create Exchange Contributor Connected App:** In **Anypoint Access Management**, create a new **Connected App** that acts on its **own behalf (client credentials)**, for writing assets to Exchange, adding the **Exchange Contributor** scope and retrieve its client ID and secret.
13. **Update settings.xml:** Change in settings.xml the credentials of the matching **server** entry for the repository defined in the distributionManagement section of the bom/pom.xml to use the Connected App credentials created previously for writing to that repository:

settings.xml

```
<server>
  <id>anypoint-exchange-v3-student-deployment</id>
  <username>~~~Client~~~</username>
```

```
  <password>your-capp-contributor-cid~?~your-capp-contributor-
secret</password>
</server>
```

Note: Only the Exchange Connected App is configured the Maven settings.xml, not the CloudHub Connected App as the Mule Maven plugin does not support using Connected App client ID and secret as Maven credentials and must be configured directly in the plugin configuration.

Note: You can also use Anypoint Platform credentials for an account of your Anypoint Platform organization with which you were able to search your Exchange — such as the trial account credentials created previously. However, it is best practice to use a Connected App for security.

Note: To use Connected App authentication, provide basic authentication and define the username as ~~~Client~~~ and the password as clientID~?~clientSecret. Replace clientID with the client ID. Replace clientSecret with the client secret.

14. **Deploy parent POMs to Exchange:** Deploy the **BOM** and **parent POM** used by apps-commons to your remote Exchange repository:

```
cd $APDL2WS/bom
mvn deploy -f pom.xml -Pdeploy-to-exchange-v3

cd $APDL2WS/parent-pom
mvn deploy -f pom.xml -Pdeploy-to-exchange-v3
```

Note: When a SNAPSHOT version is deployed to Exchange, it is created in the development lifecycle phase allowing the asset version to be overwritten and permanently deleted, whenever. When a stable asset version is deployed, Exchange will treat the asset as a production asset and enforce the standard Exchange rules upon it. Such as locking the version number so it cannot be overwritten and only allowing permanent deletion within the first 7 days.

15. **Deploy custom library to Exchange:** Deploy apps-commons to your remote Exchange repository:

```
cd $APDL2WS/apps-commons
mvn deploy
```

Note: The Mule Maven plugin is automatically integrated with Exchange Mule Maven plugin, therefore there is no need to define the Exchange Mule Maven plugin configuration as well. It is only needed to directly reference the Exchange Mule Maven plugin when deploying non-Mule

artifacts such as custom libraries and POMs. If both are configured, deployment will happen twice, and the second deployment will fail with a conflict.

Note: The Exchange Maven Facade API enables you to both create your asset and set the mutable data describing it in the same request. The mutable data of an asset includes tags, custom fields, categories, and documentation pages. The final solution includes a complete example of creating a API policy with documentation and tags.

16. **Deploy:** In **Runtime Manager**, deploy the packaged flights-management-sapi artifact from the **target** directory using the application name: **flights-management-sapi-dev**, selecting the **CloudHub 2.0 US East Ohio Shared Space** as a deployment target, enabling **Last-Mile Security** on the Ingress and adding **properties** that sets encrypt.key and disables autodiscovery:

```
encrypt.key=secure12345
anypoint.platform.gatekeeper=disabled
```

Note: As the application exposes a HTTPs endpoint itself, this option must be checked to forward the traffic from the CloudHub Shared Space Load Balancer to the application itself. This ensures the "Last-Mile" traffic between the CloudHub Shared Space Load Balancer and the application are encrypted.

17. **Locate fully qualified domain name:** In the Runtime Manager dashboard, check the fully qualified domain name of the Mule app and copy the API's endpoint URL with the additional six-character uniq-id and shard.
18. **Update base URL prop for dev:** To **dev-properties.yaml**, update the callback base URL with your fully qualified domain name:

dev-properties.yaml of flights-management-sapi

```
api:
  callback:
    base: "https://flights-management-sapi-dev-uniqid.shard.usa-
e2.cloudhub.io"
```

Note: Ensure to replace the placeholders for the six-character unique id and shard for your individual application.

19. **Maven-build: and Redeploy:** In **Runtime Manager** Run a full Maven build of flights-management-sapi again and redeploy the packaged flights-management-sapi artifact from the **target** directory:

```
cd $APDL2WS/flights-management-sapi
mvn clean verify -U -Dencrypt.key=secure12345
```

20. **Check log:** Study the log output of flights-management-sapi; the logs should show the callback being registered with the Flights Management system in the dev environment and the return invocations of the callback.

    Note: If the callback is not invoked, check in a passenger; the Flights Management system will then cancel these flights:

    ```
    curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
    :\"Mule\",\"numBags\":2}" https://flights-management-sapi-dev-
    uniqid.shard.usa-e2.cloudhub.io/api/v1/tickets/PNR123/checkin
    ```

    Note: Ensure to replace the placeholders for the six-character unique id and shard for your individual application.

Module 2: Passing messages asynchronously

In this module, you set up queues using the VM Connector to pass messages asynchronously and reliably between Mule flows and use Anypoint MQ exchanges and Anypoint MQ queues to pass messages asynchronously and reliably between Mule apps.

At the end of this module, you should be able to:

- Publish messages to a VM queue.
- Listen for messages in a VM queue.
- Publish messages to an Anypoint MQ exchange.
- Subscribe to messages in an Anypoint MQ queue.

Walkthrough 2-1: Publish messages to a VM queue

When the Flights Management system sends a cancellation notification to flights-management-sapi, it does so in the body of a HTTP POST request to the callback exposed by flights-management-sapi.

In this walkthrough, you extend the implementation of that HTTP callback to perform the first phase of the reliability pattern, the reliable acquisition flow, by sending the HTTP POST body unchanged to a VM queue. In doing so you must guard against DoS attacks by always limiting the size of VM queues.

You will:

- Configure a VM Connector and persistent VM queues.
- Publish cancellation notification XML messages to a VM queue.

Solution file

You can see the end state of following this walkthrough in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module02/wt2-1_solution.

Starting file

If you did not complete the previous walkthrough, you can get a starting file located in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module01/wt1-5_solution.

Configure a VM Connector and persistent VM queues

In this section, you add the VM Connector as a Maven dependency to flights-management-sapi and configure it with two size-limited VM queues, one for cancellation notifications received from the Flights Management system and the other a Dead Letter Queue.

The first queue is an essential part of the reliability pattern, because it is a reliable buffer of cancellation notifications sent by the Flights Management system. You want to accept cancellation notifications and send them to this queue as quickly and reliably as possible, so that none are lost. Later, asynchronously, these cancellation notifications will be processed.

You also configure a Dead Letter Queue to send those cancellation notifications to that cannot be processed. This is done in a later walkthrough.

1. **Study Solution Architecture:** Study the solution design for US2: Flight Cancellation mobile Notifications in the Solution architecture.

2. **Add managed VM Connector Maven dependency:** In Studio, add to flights-management-sapi a managed dependency on the VM Connector in **pom.xml**:

   pom.xml of flights-management-sapi

   ```
   <dependency>
     <groupId>org.mule.connectors</groupId>
     <artifactId>mule-vm-connector</artifactId>
     <classifier>mule-plugin</classifier>
   </dependency>
   ```

3. **Maven-build:** In a command-line interface, navigate to the **base directory** of **flights-management-sapi** and run a Maven build of this Mule app, providing the required secure properties encryption key; this should succeed and all unit tests should pass:

   ```
   cd $APDL2WS/flights-management-sapi
   mvn clean verify -Dencrypt.key=secure12345
   ```

4. **Add VM Connector config:** To **global.xml**, add a VM Connector configuration declaring these VM queues as persistent and limiting their size:

   global.xml of flights-management-sapi

   ```
   <vm:config name="vmConfig">
     <vm:queues>
       <vm:queue
         queueName="flight-cancel-notifs-q"
         queueType="PERSISTENT"
         maxOutstandingMessages="100" />
       <vm:queue
         queueName="flight-cancel-notifs-dlq"
         queueType="PERSISTENT"
         maxOutstandingMessages="1000" />
     </vm:queues>
   </vm:config>
   ```

   Note: VM queues must be declared to be used.

   Note: The reliability pattern also strongly suggests that the queues should be persistent.

   Note: Explicitly setting the queue type to persistent in the queue configuration is only honored by standalone customer-hosted deployments. Configuring persistent VM queues differs per

deployment topology. Some topologies have many more options, such as cluster-wide, in-memory replication, or persistence on disk or in a database, but may require management of persistent storage. It is also possible to use an external broker for reliable messaging such as Anypoint MQ which you will look at in a later walkthrough.

Note: If messages are dequeued slower than they are enqueued — for instance in the case of a DoS attack on the HTTP callback — then the Mule app will ultimately fail with an out-of-memory error: to guard against this it is good practice to always limit the size of VM queues.

5. **Homework:** Extract the hard-coded queue size into an environment-dependent configuration property, sizing it more generously in the prod environment.
6. **Homework:** Extract the hard-coded queue names into environment-independent configuration properties.

Publish XML messages to a VM queue

In this section, you publish cancellation notifications received from the Flights Management system to the configured VM queue.

7. **Publish message:** In **main.xml**, send the HTTP POST body received by the HTTP callback to the appropriate VM queue, to be processed later, asynchronously:

   main.xml of flights-management-sapi

   ```
   <flow name="receive-cancellation-notification">
     ...
     <vm:publish
       queueName="flight-cancel-notifs-q"
       sendCorrelationId="ALWAYS"
       timeout="5000" timeoutUnit="MILLISECONDS"
       config-ref="vmConfig" />
   </flow>
   ```

   Note: This sends the current payload to the VM queue. No other parts of the current Mule event, such as attributes or variables, are sent — with the exception of the correlation ID.

   Note: The Mule event's correlation ID is typically created by the <http:listener />, unless it was sent by the HTTP client.

   Note: An explicit timeout ensures that sending to the queue does not block indefinitely in case of an error.

   Note: If the queue size has been reached, this enqueue operation blocks until either a message

has been dequeued or the timeout has been reached.

Note: Errors in sending the message to the VM queue are currently handled by logging a misleading error log entry: you will fix this in a later walkthrough.

8. **Run, invoke, and check log:** Run flights-management-sapi and send a few cancellation notifications to the callback; observe the corresponding log entries:

```
curl -ik -H "Content-Type:text/xml" -d
"<CancellationNotification><PNR>PNR123</PNR><PassengerLastName>Mule</P
assengerLastName></CancellationNotification>"
https://localhost:8081/api/cancelFlight
```

```
curl -ik -H "Content-Type:text/xml" -d
"<CancellationNotification><PNR>PNR456</PNR><PassengerLastName>Max</Pa
ssengerLastName></CancellationNotification>"
https://localhost:8081/api/cancelFlight
```

```
curl -ik -H "Content-Type:text/plain" -d "invalid content type and
content" https://localhost:8081/api/cancelFlight
```

Note: All HTTP POST bodies are sent to the VM queue, even obviously invalid ones; In a later walkthrough you will add validation of the received message, before it is sent to the queue.

9. **Homework:** Turn the hard-coded timeout into an environment-dependent property.

Walkthrough 2-2: Listen for messages in a VM queue

cancellation notifications POSTed from the Flights Management system to flights-management-sapi are sent to a persistent VM queue, to ensure they are not lost. But they are not currently processed: What is missing is the second phase of the reliability pattern, the application logic flow.

In this walkthrough, you extend flights-management-sapi to asynchronously retrieve messages from that VM queue and start processing them as cancellation notifications. Despite processing logic being simple for now, this nonetheless requires you to think about processing errors, a redelivery policy for dealing with those errors, and the transactionality of message processing. VM queues are transactional resources that can participate in local and XA transactions.

You will:

- Create a flow that listens on a VM queue in a local transaction.
- Set a redelivery policy when consuming cancellation notifications.
- Avoid reprocessing of invalid cancellation notifications.

Solution file

You can see the end state of following this walkthrough in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module02/wt2-2_solution.

Starting file

If you did not complete the previous walkthrough, you can get a starting file located in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module02/wt2-1_solution.

Create a flow that listens on a VM queue in a local transaction

In this section, you create a Mule flow with a <vm:listener /> as the message source to receive putative cancellation notifications from a VM queue. You transform the cancellation notifications to flight canceled events and begin to understand the effect of errors that may be raised in that transformation.

cancellation notifications are sent in a format not under AnyAirline's control. It is essential to insulate AnyAirline from changes in that message format, such as by transforming to the internally controlled format of flight canceled events.

1. **Study Solution Architecture:** Study the solution design for US2: Flight Cancellation mobile Notifications in the Solution architecture.

2. **Listen for notification messages:** In Studio, add to **main.xml** a flow with a message source that listens on the cancellation notifications VM queue in a new local transaction and logs the received message:

   main.xml of flights-management-sapi

   ```
   <flow name="deliver-flight-cancelled-event">
     <vm:listener
       queueName="flight-cancel-notifs-q"
       transactionalAction="ALWAYS_BEGIN"
       config-ref="vmConfig"/>
     <logger level="INFO" message="Received"/>
   </flow>
   ```

   Note: By default, messages are retrieved from the VM queue concurrently; this is OK because cancellation notifications are not inherently ordered.

   Note: The transaction starts with the de-queuing of the message, ends at the end of the Mule flow, and involves the VM Connector as the only transactional resource (is a resource-local transaction).

3. **Run, invoke, and check log:** Run flights-management-sapi and send several HTTP POST requests — valid cancellation notifications and invalid messages — to the HTTP callback as before; observe the log entries:

   ```
   curl -ik -H "Content-Type:text/xml" -d
   "<CancellationNotification><PNR>PNR123</PNR><PassengerLastName>Mule</P
   assengerLastName></CancellationNotification>"
   https://localhost:8081/api/cancelFlight
   ```

   ```
   curl -ik -H "Content-Type:text/xml" -d
   "<CancellationNotification><PNR>PNR456</PNR><PassengerLastName>Max</Pa
   ssengerLastName></CancellationNotification>"
   https://localhost:8081/api/cancelFlight
   ```

   ```
   curl -ik -H "Content-Type:text/plain" -d "invalid content type and
   content" https://localhost:8081/api/cancelFlight
   ```

   Note: Messages are retrieved from the VM queue immediately after they have been added to it.

4. **Transform XML messages to JSON:** In **deliver-flight-cancelled-event**, transform each XML-formatted cancellation notification into a JSON-formatted flight canceled event, immediately after logging the received message:

   main.xml of flights-management-sapi

   ```
   <flow name="deliver-flight-cancelled-event">
     ...
     <ee:transform>
       <ee:message>
         <ee:set-payload><![CDATA[%dw 2.0
         output application/json
         ---
         {
           pnr:                payload.CancellationNotification.PNR,
           lastNameOfPassenger:
   payload.CancellationNotification.PassengerLastName
         }]]></ee:set-payload>
       </ee:message>
     </ee:transform>
   </flow>
   ```

   Note: In a later walkthrough, you will send this flight canceled event to an Anypoint MQ exchange, for now you just log it.

5. **Run, invoke, and check log:** Run flights-management-sapi again and post one valid and one invalid cancellation notification to the HTTP callback; observe the log and note the error being raised.

   Note: The HTTP client — which, in real use, is the Flights Management system — sees no difference between valid and invalid cancellation notifications.

   Note: When asynchronously processing an invalid cancellation notification, an error such as MULE:EXPRESSION is raised when transforming it to a flight canceled event, which causes the transaction and hence the dequeuing of the message to roll back. This leads to an infinite loop as the same invalid message is received and processed again and again, always failing with the same error.

Set a redelivery policy when consuming messages from a VM queue

In this section, you limit the number of times the same message is dequeued from a VM queue after it had previously caused an error during message processing. This is done by setting a

redelivery policy on the <vm:listener />. After redeliveries have been exhausted, you send the message to the Dead Letter Queue.

A redelivery policy must decide when a message being dequeued is the same message as a previously dequeued message. By default the identity criterion is a hash of the message payload. But this approach is inappropriate for XML message like the cancellation notification, which can be formatted differently while still representing the same XML content. Instead, you use the correlation ID that was explicitly sent with the message to the VM queue.

A redelivery policy is a stateful component because it needs to store the IDs and counts of previously processed messages: it requires an Object Store to maintain this state between invocations, and uses the default Object Store if none is explicitly configured.

6. **Configure redelivery policy:** Add a redelivery policy to the <vm:listener /> in main.xml, add, using the correlation ID rather than the payload as the identity criterion:

   main.xml of flights-management-sapi

   ```
   <flow name="deliver-flight-cancelled-event">
     <vm:listener
       queueName="flight-cancel-notifs-q"
       transactionalAction="ALWAYS_BEGIN"
       config-ref="vmConfig">
       <redelivery-policy
         maxRedeliveryCount="3"
         idExpression="#[correlationId]" />
     </vm:listener>
     ...
   </flow>
   ```

   Note: The correlation ID of the current Mule event is available as a predefined variable in DataWeave.

   Note: The default Object Store, which is persistent, is used by the redelivery policy.

7. **Run, invoke, and check log:** Run flights-management-sapi again and note the errors being raised; the Mule app should automatically start processing messages from the persistent VM queue.

   Note: After the last permissible redelivery of a message has also failed, a MULE:REDELIVERY_EXHAUSTED error is raised.

8. **Add Dead Letter error handling:** Add an error handler for MULE:REDELIVERY_EXHAUSTED

that sends the message to the Dead Letter Queue:

main.xml of flights-management-sapi

```
<flow name="deliver-flight-cancelled-event">
  ...
  <error-handler>
    <on-error-continue
      type="MULE:REDELIVERY_EXHAUSTED"
      enableNotifications="false"
      logException="false">
      <logger level="ERROR"
        message="Giving up"/>
      <flow-ref name="send-to-vm-dlq" />
    </on-error-continue>
  </error-handler>
</flow>

<sub-flow name="send-to-vm-dlq">
  <vm:publish
    queueName="flight-cancel-notifs-dlq"
    timeout="5000" timeoutUnit="MILLISECONDS"
    config-ref="vmConfig" />
</sub-flow>
```

Note: Set timeout values as usual when sending to a VM queue, as the send operation may fail, for example when the Dead Letter Queue is full.

Note: The default is for the correlation ID to be sent to the VM queue if one is present, which there is in this case.

Note: The trigger of the REDELIVERY_EXHAUSTED error is caused by the arrival of a new event, and no context is kept between the original failing events and the new one. Therefore the event is in its original state with the original payload and no variables.

9. **Consume Dead Letter messages:** In main.xml, add a simple Mule flow with a <vm:listener /> consuming messages sent to the **Dead Letter Queue**:

   main.xml of flights-management-sapi

```
<flow name="handle-dlq">
  <vm:listener
    queueName="flight-cancel-notifs-dlq"
    config-ref="vmConfig">
```

```
    <redelivery-policy
      maxRedeliveryCount="3"
      idExpression="#[correlationId]"/>
  </vm:listener>
  <logger level="INFO"
    message="Processing message from DLQ"/>
</flow>
```

10. **Run, invoke, and check log:** Run flights-management-sapi again; it should drain all messages from the VM queue, sending invalid ones to the Dead Letter Queue after the third retry.
11. **Invoke:** Send an individual valid and invalid cancellation notification to flights-management-sapi and follow the processing of each.
12. **Homework:** Extract all hard-coded values into environment-dependent configuration properties.

Avoid redelivery of invalid messages

In this section, you address a fundamental inefficiency in the current redelivery configuration for cancellation notifications in flights-management-sapi. If a message retrieved from the VM queue is inherently invalid, then it will never be possible to treat this message as a cancellation notification and transform it to a valid flight canceled event. But in the current redelivery configuration, these messages are redelivered and reprocessed several time, before finally being sent to the Dead Letter Queue. This kind of permanent errors should not waste resources and should be dealt with by sending the corresponding messages straight to the Dead Letter Queue.

13. **Add localized Dead Letter error handling:** Add a Try scope around the transformation of an (assumed) cancellation notification to a flight canceled event, treating all raised errors as permanent and sending the message to the Dead Letter Queue. Set a boolean **msgValid** variable for processors outside of the Try scope to determine whether validation was successful:

    main.xml of flights-management-sapi

```
<flow name="deliver-flight-cancelled-event">
  <vm:listener
    transactionalAction="ALWAYS_BEGIN">
    ...
  </vm:listener>
  <try
    transactionalAction="BEGIN_OR_JOIN">
    <ee:transform>...</ee:transform>
    <set-variable variableName="msgValid" value="#[true]"/>
    <error-handler>
      <on-error-continue
```

```
        enableNotifications="false"
        logException="false">
        <flow-ref name="send-to-vm-dlq" />
        <set-variable variableName="msgValid" value="#[false]"/>
      </on-error-continue>
    </error-handler>
  </try>
  <error-handler>...</error-handler>
</flow>
```

Note: The Try scope serves just to isolate any errors raised during preparing the flight canceled event and must therefore participate in the ongoing transaction started by the <vm:listener />.

Note: Errors during transforming to the flight canceled event must be handled with an on-error-continue so as to consume the message and not trigger redeliveries.

Note: Because errors within the Try scope are consumed by an on-error-continue to not trigger redeliveries, future processors outside of the Try scope are unaware an error occurred and need another way of filtering invalid messages.

Note: The trigger of the REDELIVERY_EXHAUSTED error is caused by the arrival of a new event, and no context is kept between the original failing. Therefore the event is in its original state with the original payload and no variables.

14. **Store original payload:** Add a variable that stores the original payload so that this can be sent to the Dead Letter Queue:

    main.xml of flights-management-sapi

```
<flow name="deliver-flight-cancelled-event">
  ...
  <try>
    <set-variable
      variableName="originalPayload"
      value="#[output text/plain --- payload.^raw]"
    />
    ...
  </try>
  ...
</flow>

<sub-flow name="send-to-vm-dlq">
  <vm:publish>
    <vm:content>#[vars.originalPayload]</vm:content>
```

```
    </vm:publish>
</sub-flow>
```

Note: At this point in the Mule flow, the error is not triggered by a REDELIVERY_EXHAUSTED event, therefore the payload must be manually reverted to its original state before publishing to the Dead Letter Queue.

15. **Run, invoke, and check log:** Run flights-management-sapi and post invalid cancellation notifications to it; confirm that they are sent to the Dead Letter Queue straight away.

Walkthrough 2-3: Publish messages to an Anypoint MQ exchange

In this walkthrough, you extend the processing of VM messages in flights-management-sapi to publish flight canceled events to an Anypoint MQ exchange.

You will:

- Configure an Anypoint MQ Connector for message publishing.
- Publish flight canceled events to an Anypoint MQ exchange.

Solution file

You can see the end state of following this walkthrough in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module02/wt2-3_solution.

Starting file

If you did not complete the previous walkthrough, you can get a starting file located in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module02/wt2-2_solution.

Configure an Anypoint MQ Connector for message publishing

In this section, you add the Anypoint MQ Connector to flights-management-sapi as a Maven dependency and configure it to connect to the us-e1 (N. Virginia) Anypoint MQ broker using the client ID and secret of a pre-created Anypoint MQ client app.

6. **Study Solution Architecture:** Study the solution design for US2: Flight Cancellation mobile Notifications in the Solution architecture paying particular attention to the role of Anypoint MQ. .
7. **Add managed Anypoint MQ Connector Maven dependency:** In Studio, add to flights-management-sapi a managed dependency on the Anypoint MQ Connector in **pom.xml**:

   pom.xml of flights-management-sapi

   ```
   <dependency>
     <groupId>com.mulesoft.connectors</groupId>
     <artifactId>anypoint-mq-connector</artifactId>
     <classifier>mule-plugin</classifier>
   </dependency>
   ```

8. **Add Anypoint MQ Connector global config:** To **global.xml**, add <anypoint-mq:config />

with the supplied client ID and secret for an Anypoint MQ client app already created in Anypoint Platform:

global.xml of flights-management-sapi

```
<anypoint-mq:config name="amqConfig">
  <anypoint-mq:connection
    clientId="b58581e1242d4e10bdca4103cee00181"
    clientSecret="b063BAe915a449A8A00AAe4f1e2aA3D4"
    url="https://mq-us-east-
1.anypoint.mulesoft.com/api/v1/organizations/a63e6d25-8aaf-4512-b36d-
d91b90a55c4a/environments/129441e8-ae69-4cf9-a70b-cdabca4823ff"/>
</anypoint-mq:config>
```

Note: The Anypoint MQ Connector configuration is called <anypoint-mq:config /> both for sending and receiving messages to/from an Anypoint MQ broker.

Note: Use the client ID and secret of an Anypoint MQ client app created previously.

Note: Since v4.x of the Anypoint MQ Connector, the Anypoint Platform organization ID and the Anypoint Platform environment ID must be supplied in the Anypoint MQ URL. This URL can be copied from the Anypoint MQ UI.

Publish JSON messages to an Anypoint MQ exchange

In this section, you use the previously configured Anypoint MQ Connector to publish messages containing JSON flight canceled events to an Anypoint MQ destination.

9. **Publish flight canceled event to an Anypoint MQ destination:** In Studio, add to **main.xml** the operation to publish a flight canceled event with a custom message ID to its Anypoint MQ destination within a Choice router filtering any invalid messages:

   main.xml of flights-management-sapi

```
<flow name="deliver-flight-cancelled-event">
  <vm:listener />
  <try />
  <choice>
    <when expression="#[vars.msgValid]">
      <anypoint-mq:publish
        destination="cancelled-flights-exchg-dev"
        messageId="#[correlationId]"
        config-ref="amqConfig" />
    </when>
```

```
  </choice>
  <error-handler />
</flow>
```

Note: By default, the Anypoint MQ message body is taken from the current payload.

Note: You use the correlation ID that has been passed along this message from the start as the ID of the Anypoint MQ message. If you do not specify a message ID, then Anypoint MQ creates a new one.

Note: Transformation to the flight canceled event is performed in the Try scope, so that publishing to Anypoint MQ is under the control of the flow-level error handler, resulting in the correct retry behavior.

Note: Because errors within the Try scope are consumed by an on-error-continue to not trigger redeliveries, future processors outside of the Try scope are unaware an error occurred and need another way of filtering invalid messages.

10. **Run, invoke, and check log:** Run flights-management-sapi and POST a cancellation notification to the HTTP callback; observe the logs:

```
curl -ik -H "Content-Type:text/xml" -d
"<CancellationNotification><PNR>PNR123</PNR><PassengerLastName>Mule</P
assengerLastName></CancellationNotification>"
https://localhost:8081/api/cancelFlight
```

Note: Because HTTP traffic is logged, you can observe the API invocation of the Anypoint MQ REST API.

Note: Confirm that when the Anypoint MQ Connector authenticates towards the Anypoint MQ broker, it sends the correct client ID and secret and receives the Anypoint Platform organization ID of AnyAirline.

Note: Confirm that the message ID sent to the Anypoint MQ broker is the correlation ID.

Note: The content type of the Anypoint MQ message is correctly set to application/json because that's the output set in the DataWeave transformation to the flight canceled event.

11. **Homework:** Extract all configuration values hard-coded in this walkthrough into appropriate configuration properties, environment-dependent or independent and encrypted, as fitting. Note that you have not been given an Anypoint MQ client app for the test and prod environments.

Walkthrough 2-4: Subscribe to messages in an Anypoint MQ queue

Flight cancellation events sent to the cancelled-flights-exchg Anypoint MQ exchange are distributed to the cancelled-flights-mobile-queue Anypoint MQ queue. This walkthrough creates mobile-notifications-eapp as a Mule app located in the Experience API tier that consumes messages from Anypoint MQ queue and forwards them to the mobile app (simulated).

In this walkthrough, you first set-up the starter code for mobile-notifications-eapp. You then implement a Mule flow to subscribe to an Anypoint MQ queue, manual acknowledging received messages, and forwarding them onto a mobile app simulated via the VM Connector. You then implement a circuit breaker to control subscription to the Anypoint MQ queue if the mobile app network is unavailable.

You will:

- Create mobile-notifications-eapp.
- Configure the Anypoint MQ Connector for message subscription.
- Subscribe to messages in an Anypoint MQ queue.
- Control Anypoint MQ message subscription with a Circuit Breaker.

Solution file

You can see the end state of following this walkthrough in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module02/wt2-4_solution.

Starting file

To follow this walkthrough, you need the starter project at $APDL2DIST/walkthroughs/devint/module02/wt2-4_starter.

Create mobile-notifications-eapp

In this section, you copy the starter code for mobile-notifications-eapp — a Mule app for receiving Flight cancellation events in the Experience API layer into your Studio workspace, build and import it into Studio.

1. **Copy mobile-notifications-eapp starter:** Copy the **mobile-notifications-eapp** starter project into your Studio workspace directory, renaming it to mobile-notifications-eapp:

```
cd $APDL2WS
cp -r $APDL2DIST/walkthroughs/devint/module02/wt2-4_starter/mobile-
```

```
notifications-eapp ./
```

2. **Study Maven build:** Familiarize yourself briefly with the Maven build configuration of mobile-notifications-eapp.

   Note: Again, a reusable Maven plugin configuration is inherited from the parent POM and Maven dependency management from the BOM. Both parent-pom/pom.xml and bom/pom.xml are read from the Studio workspace directory.

3. **Maven-build:** Run a full Maven build of mobile-notifications-eapp including all MUnit tests, providing the required secure properties encryption key; this should succeed and all unit tests should pass:

   ```
   cd $APDL2WS/mobile-notifications-eapp
   mvn clean verify -U -Dencrypt.key=secure12345
   ```

4. **Import into Studio:** In Studio, import mobile-notifications-eapp without copying it to your workspace, as it is already there; confirm that the import succeeded.
5. **Add temporary env property:** In Studio, near the top of global.xml, define a property **encrypt.key** to default the property required by the Studio tooling instance:

   global.xml of mobile-notifications-eapp

   ```
   <global-property name="encrypt.key" value="secure12345" />
   ```

   Note: The encrypt.key property should not be defaulted and should be removed. This is a temporary solution to provide the runtime property to the Studio tooling instance, required to utilize DataSense metadata support.

6. **Add run config :** Add a Studio run configuration (via the **Run as > Mule application (configure)** dialog) for mobile-notifications-eapp that sets **encrypt.key**:

   ```
   -M-Dencrypt.key=secure12345
   ```

Configure the Anypoint MQ Connector for message subscription

In this section, you configure the Anypoint MQ Connector with the client ID and secret for a new Anypoint MQ client app.

7. **Add Anypoint MQ Connector global config:** To **global.xml**, add <anypoint-mq:config /> with the client ID and secret (**b58581e1242d4e10bdca4103cee00181** and **b063BAe915a449A8A00AAe4f1e2aA3D4**):

   global.xml of mobile-notifications-eapp

   ```
   <anypoint-mq:config name="amqConfig">
     <anypoint-mq:connection
       clientId="b58581e1242d4e10bdca4103cee00181"
       clientSecret="b063BAe915a449A8A00AAe4f1e2aA3D4"
       url="https://mq-us-east-
   1.anypoint.mulesoft.com/api/v1/organizations/a63e6d25-8aaf-4512-b36d-
   d91b90a55c4a/environments/129441e8-ae69-4cf9-a70b-cdabca4823ff"/>
   </anypoint-mq:config>
   ```

   Note: It is best practice to create a new application for each Mule app and environment combination.

Subscribe to messages in an Anypoint MQ queue

In this section, you create a Mule flow with a <anypoint-mq:subscriber /> as the message source to receive flight canceled events from an Anypoint MQ queue. You transform the flight canceled events and publish to a simple VM queue simulating a downstream mobile app.

8. **Select a queue:** Choose an individual **cancelled-flights-mobile-queue-student-XX** Anypoint MQ queue, where XX represents a student number, from 01 to 15, for example, 01, 02, etc.
9. **Listen for notification messages:** In Studio, add to **main.xml** a flow with a message source that subscribes on the cancelled-flights-mobile-queue-student-XX Anypoint MQ queue with **MANUAL** acknowledgment and a reconnection strategy to reconnect forever and logs the received message:

   main.xml of mobile-notifications-eapp

   ```
   <flow name="retrieve-cancellation-event">
     <anypoint-mq:subscriber config-ref="amqConfig"
   destination="cancelled-flights-mobile-queue-dev-student05"
   acknowledgementMode="MANUAL">
       <reconnect-forever frequency="1000"/>
     </anypoint-mq:subscriber>
     <logger level="INFO" message="Received"/>
   </flow>
   ```

Note: Default property values are predefined in properties.yaml.

Note: As Anypoint MQ is a remote service and used to trigger a Mule flow, it is best practice to set a reconnection strategy to reconnect if there are any connectivity issues.

Note: You reference an Anypoint MQ queue to subscribe, whereas you previously published to an Anypoint MQ exchange.

10. **Run, invoke, and check log:** Run **mobile-notifications-eapp**, invoke the **AnyAirline** hosted **flights-management-sapi** to populate the queue sand observe the mobile-notifications-eapp log entries:

```
curl -ik -H "Content-Type:text/xml" -d
"<CancellationNotification><PNR>PNR123</PNR><PassengerLastName>Mule</P
assengerLastName></CancellationNotification>" https://tngaa-flights-
management-sapi-dev-9yj2rh.rajrd4-2.usa-
e1.cloudhub.io/api/cancelFlight
```

Note: If more than one client subscribes to the same queue, you must send multiple requests as they will be competing consumers, consuming the message before another client can.

11. **Transform to mobile app JSON format:** Transform each JSON-formatted message to the new format required by the mobile app immediately after logging the received message:

main.xml of mobile-notifications-eapp

```
<flow name="retrieve-cancellation-event">
  ...
  <ee:transform>
    <ee:message>
      <ee:set-payload><![CDATA[%dw 2.0
        output application/json
        ---
        {
          pnr: payload.pnr,
          lastName: payload.lastNameOfPassenger
        }]]></ee:set-payload>
    </ee:message>
    </ee:transform>
</flow>
```

12. **Add VM Connector config:** To **global.xml**, add a VM Connector configuration declaring a VM

queue simulating the mobile app network:

global.xml of mobile-notifications-eapp

```
<vm:config name="mobileAppNetworkVMConfig">
  <vm:queues>
    <vm:queue queueName="mobile-app-native-notifs-q"/>
  </vm:queues>
</vm:config>
```

Note: Default property values are predefined in properties.yaml.

Note: VM queues must be declared to be used.

13. **Publish message:** In **main.xml**, send the transformed JSON payload received by Anypoint MQ subscriber to the VM queue:

    main.xml of mobile-notifications-eapp

    ```
    <flow name="retrieve-cancellation-event">
      ...
      <vm:publish config-ref="mobileAppNetworkVMConfig" queueName="mobile-
    app-native-notifs-q"/>
    </flow>
    ```

    Note: This sends the current payload to the VM queue. Except for the correlation ID, no other parts of the Mule event, such as attributes or variables.

14. **Wait and observe:** Run mobile-notifications-eapp again; observe the log and note the same messages are being received.

    Note: Messages will continue to be received until the flow acknowledges the message as the acknowledgement mode is manual.

    Note: This is difficult to demonstrate if there are multiple clients consuming from the same queue.

15. **Store acknowledgment token:** In **main.xml**, set a variable to store the acknowledgment token received from the attributes of the Anypoint MQ message before the <vm:publish />:

    main.xml of mobile-notifications-eapp

    ```
    <flow name="retrieve-cancellation-event">
    ```

```
  ...
  <set-variable variableName="mqAckToken"
value="#[attributes.ackToken]"/>
  <vm:publish config-ref="mobileAppNetworkVMConfig" queueName="mobile-
app-native-notifs-q"/>
</flow>
```

Note: The token must be stored prior to crossing a transport barrier to be used later as attributes are overwritten.

16. **Manually acknowledge the message:** In **main.xml**, manually acknowledge the message has been successfully processed after publishing to the VM queue using the stored **mqAckToken**:

    main.xml of mobile-notifications-eapp

```
<flow name="retrieve-cancellation-event">
  ...
  <anypoint-mq:ack ackToken="#[vars.mqAckToken]" config-
ref="amqConfig"/>
</flow>
```

17. **Explicitly Not Acknowledge a message:** Add an error handler and send a **NACK** to put the message back on the queue using the same stored **mqAckToken**:

    main.xml of mobile-notifications-eapp

```
<flow name="retrieve-cancellation-event">
  ...
  <error-handler>
    <on-error-propagate>
      <anypoint-mq:nack ackToken="#[vars.mqAckToken]" config-
ref="amqConfig"/>
    </on-error-propagate>
  </error-handler>
</flow>
```

Note: Manual acknowledgement gives you flexibility to choose which error types result in an ACK or NACK.

Control Anypoint MQ message subscription with a Circuit Breaker

When an external service is not available (the VM Connector simulating the mobile app network in this case), every attempt to process messages results in a failure, forcing the app to retrieve messages that cannot succeed and are eventually moved to a Dead Letter Queue. You can avoid this behavior by notifying the subscriber of the error in a way that prevents it from consuming more messages for some time.

If a Mule flow finishes its execution with an error the <anypoint-mq:subscriber />, will check if the error is one of a preconfigured error type that indicates an external service error, and counts consecutive occurrences until a configurable threshold is reached. When the threshold is reached, the circuit trips into an Open state and stops polling for new messages for the duration of a specified configurable trip timeout.

Once the timeout has elapsed, the circuit goes into a Half-Open, state consuming a single message from the Anypoint MQ queue to test if the external error has been resolved. If successful, the circuit goes back into a Closed state and the <anypoint-mq:subscriber /> continues normal message processing. If unsuccessful, the circuit goes back into an Open state until the timeout has elapsed once again.

By default, the circuit breaking feature is disabled. In this section, you augment the <anypoint-mq:subscriber /> with circuit breaking capability, to control how the Anypoint MQ Connector handles errors that occur while processing a consumed message.

18. **Add Anypoint MQ Connector global Circuit Breaker:** To **global.xml**, add <anypoint-mq:circuit-breaker /> to trip using the error types from the VM connector:

    global.xml of mobile-notifications-eapp

    ```
    <anypoint-mq:circuit-breaker name="vmConnectCircuitBreaker"
      onErrorTypes="VM:CONNECTIVITY, VM:QUEUE_TIMEOUT"
      errorsThreshold="1"
      tripTimeout="120"
      tripTimeoutUnit="SECONDS"/>
    ```

    Note: The errorThreshold argument configures the max number of the configured error type that must occur for the circuit breaker to trip into an open state.

    Note: The tripTimeout argument configures how long the circuit remains in an open state once the errors threshold is reached.

    Note: You configure error types on which to trip the circuit. In this case, errors from the VM

namespace; a VM queue is simulating the downstream system.

Note: An error occurrence counts only when the flow finishes with an error. By default, all error types count as a circuit failure.

Note: These values are artificially low for testing purposes.

19. **Configure subscriber with Circuit Breaker:** In **main.xml**, add to the <anypoint-mq:subscriber /> a reference to the global Circuit Breaker configuration:

    main.xml of mobile-notifications-eapp

    ```
    <anypoint-mq:subscriber config-ref="amqConfig" destination="cancelled-flights-mobile-queue-dev" acknowledgementMode="MANUAL" circuitBreaker="vmConnectCircuitBreaker">
      ...
    </anypoint-mq:subscriber>
    ```

20. **Simulate VM Connector error:** To **global.xml**, modify the VM Connector configuration to only allow one outstanding message on the queue:

    global.xml of mobile-notifications-eapp

    ```
    <vm:config name="mobileAppNetworkVMConfig">
      <vm:queues>
        <vm:queue queueName="mobile-app-native-notifs-q" maxOutstandingMessages="1"/>
      </vm:queues>
    </vm:config>
    ```

21. **Tune log config:** Change **log4j2-test.xml** to show DEBUG log entries created by the **com.mulesoft.extension.mq** logger:

    log4j2.xml of mobile-notifications-eapp

    ```
    <AsyncLogger name="com.mulesoft.extension.mq" level="DEBUG"/>
    ```

22. **Run, invoke, and check log:** Run **mobile-notifications-eapp**, invoke the **AnyAirline** hosted **flights-management-sapi** to populate the queue and observe the mobile-notifications-eapp log entries, paying close attention to the Notify Circuit messages and the delay between attempts at picking up messages which should correspond with the configured trip timeout property:

```
curl -ik -H "Content-Type:text/xml" -d
"<CancellationNotification><PNR>PNR123</PNR><PassengerLastName>Mule</P
assengerLastName></CancellationNotification>" https://tngaa-flights-
management-sapi-dev-9yj2rh.rajrd4-2.usa-
e1.cloudhub.io/api/cancelFlight
```

Note: If more than one client subscribes to the same queue, you must send multiple requests as they will be competing consumers, consuming the message before another client can.

23. **Homework:** Extract all hard-coded configuration values into environment-dependent and independent configuration properties as necessary.

Module 3: Validating messages

In this module, you use various modules and techniques to implement message validation across Mule flows and Mule apps.

At the end of this module, you should be able to:

- Validate Mule events.
- Validate XML messages.
- Validate JSON messages.

Walkthrough 3-1: Validate Mule events

The flights-management-sapi Mule app implemented earlier receives flight from the Flights Management system in XML format via an HTTP callback. Because this HTTP callback endpoint does not use APIkit for contract enforcement and is not protected by API Manager policies, invalid messages can be published to the VM queue.

In this walkthrough, you enhance flights-management-sapi to validate that the minimum pre-conditions are met before storing the cancellation notifications.

You will:

- Filter HTTP POST requests with empty bodies or the wrong content-type using the Validation module.

Solution file

You can see the end state of following this walkthrough in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module03/wt3-1_solution.

Starting file

If you did not complete the previous walkthrough, you can get a starting file located in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module02/wt2-4_solution.

Filter HTTP POST requests with empty bodies or the wrong content-type using the Validation module

In this section, you use the Validation module to validate that the received HTTP request meets the minimum requirements before publishing to the VM queue.

1. **Confirm Maven dependency management:** In **bom/pom.xml**, locate the existing dependency management entries for the Validation module, which has artifact ID mule-validation-module.
2. **Add Maven dependency:** Add the corresponding Maven dependency for the Validation module to the **POM** of **flights-management-sapi**; confirm that Studio loads the module:

   pom.xml of flights-management-sapi

   ```
   <dependency>
     <groupId>org.mule.modules</groupId>
     <artifactId>mule-validation-module</artifactId>
   ```

```
  <classifier>mule-plugin</classifier>
</dependency>
```

Note: Omit the <version /> element so that the version managed in the BOM takes effect.

3. **Validate Content-type:** Immediately after the <http:listener /> that accepts HTTP POST requests from the Flights Management system, add a Validation module component configured with a DataWeave expression that validates the Content-type HTTP header is an XML variant: text/xml or application/xml:

   main.xml of flights-management-sapi

   ```
   <flow name="receive-cancellation-notification">
     <http:listener />

     <validation:is-true
       expression="#[var ct = attributes.headers.'content-type' --- ((not
   isBlank(ct)) and (lower(ct) contains '/xml'))]"/>
     ...
   </flow>
   ```

   Note: You store the content-type header in a variable first as it is accessed multiple times.

   Note: You first verify the header is not empty before performing String operations upon it.

   Note: You lowercase the header value to perform consistent String comparisons.

4. **Validate payload:** Add another Validation module component, this time validating the payload is not empty:

   main.xml of flights-management-sapi

   ```
   <flow name="receive-cancellation-notification">
     ...
     <validation:is-not-blank-string value="#[payload.^raw]"
   message="Payload is missing"/>
     ...
   </flow>
   ```

   Note: You use the specific is-not-blank-string operation from the Validation module; You can use other Validation module operations in conjunction with DataWeave expressions to configure the same functionality.

Note: You use the raw DataWeave selector to validate the raw underlying String, instead of the DataWeave object automatically created.

5. **Store original payload:** Immediately after the <http:listener /> store the original payload in a variable as plain text, avoiding issues with invalid payloads:

   main.xml of flights-management-sapi

   ```
   <flow name="receive-cancellation-notification">
     <http:listener />

     <set-variable variableName="originalPayload" value="#[output
   text/plain --- payload.^raw]"/>
     ...
   </flow>
   ```

   Note: This variable will be used to return to clients in case of errors.

   Note: You use the raw payload value output in a simplistic text/plain format as the payload has not been validated at this point and could contain anything.

6. **Catch validation errors:** Add an error handler containing an on-error-continue scope that sets the appropriate HTTP status code and response body to the original payload stored previously:

   main.xml of flights-management-sapi

   ```
   <flow name="receive-cancellation-notification">
     <http:listener />
     ...
     <error-handler>
       <on-error-continue when="#[['VALIDATION'] contains
   error.errorType.namespace]">
         <set-variable variableName="httpStatus" value="400"/>
         <set-variable variableName="response" value="#[output text/plain
   --- vars.originalPayload]"/>
       </on-error-continue>
     </error-handler>
   </flow>
   ```

   Note: No further validation is done before sending the XML doc to the VM queue in order not to slow down accepting these HTTP POST requests.

   Note: You use an array of module namespaces as others will be added in a later walkthrough.

7. **Run, invoke, and check log:** Run flights-management-sapi and send a few cancellation notifications to the callback; observe the corresponding log entries:

```
curl -ik -H "Content-Type:text/xml" -d
"<CancellationNotification><PNR>PNR123</PNR><PassengerLastName>Mule</P
assengerLastName></CancellationNotification>"
https://localhost:8081/api/cancelFlight
```

```
curl -ik -H "Content-Type:text/plain" -d "invalid content type"
https://localhost:8081/api/cancelFlight
```

```
curl -ik -H "Content-Type:text/xml" -d ""
https://localhost:8081/api/cancelFlight
```

```
curl -ik -H "Content-Type:text/xml" -d "invalid content"
https://localhost:8081/api/cancelFlight
```

Note: Most invalid requests are now correctly validated before being sent to the VM queue. Some invalid HTTP POST bodies are still sent to the VM queue as they are not are syntactically checked. In the next walkthrough, you will add syntactical validation before sending it to the queue.

Walkthrough 3-2: Validate XML messages

The flights-management-sapi Mule app implemented earlier receives flight cancellation notifications from the Flights Management system in XML format via an HTTP POST request. Each notification is transformed into JSON-formatted flight canceled events, which are sent to an Anypoint MQ exchange. Because message-based integration of this kind is not covered by an automatically enforced contract like an API specification, neither the incoming XML messages nor the outgoing JSON messages are syntactically checked.

In this walkthrough, you enhance flights-management-sapi to validate incoming XML messages against an XML-Schema.

You will:

- Import XML module.
- Create an XML-Schema.
- Validate an XML document against an XML-Schema.
- Validate an XML document leniently.

Solution file

You can see the end state of following this walkthrough in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module03/wt3-2_solution.

Starting file

If you did not complete the previous walkthrough, you can get a starting file located in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module03/wt3-1_solution.

Import XML module

In this section, you import the XML module in flights-management-sapi to use utility operations for validating XML Messages against a XML-Schema.

1. **Confirm Maven dependency management:** In **bom/pom.xml**, locate the existing dependency management entries for the XML module, which has artifact ID mule-xml-module.
2. **Add Maven dependency:** Add the corresponding Maven dependency for the XML module to the **POM** of flights-management-sapi; confirm that Studio loads the module:

pom.xml of flights-management-sapi

```
<dependency>
  <groupId>org.mule.modules</groupId>
  <artifactId>mule-xml-module</artifactId>
  <classifier>mule-plugin</classifier>
</dependency>
```

Note: Omit the <version /> element so that the version managed in the BOM takes effect.

Create an XML-Schema

In this section, you generate an XML-Schema for validating XML Messages against using a sample XML document.

3. **Generate cancellation notifications XML-Schema:** In the XML-Schema at **https://www.freeformatter.com/xsd-generator.html** generate an XML-Schema from the sample cancellation notifications XML:

```
<CancellationNotification>
  <PNR>PNR123</PNR>
  <PassengerLastName>Mule</PassengerLastName>
</CancellationNotification>
```

4. **Create XML-Schema file:** Create XML-Schema file **CancellationNotification.xsd** in a new directory **src/main/resources/schemas** copying the contents of the generated XML-Schema:

CancellationNotification.xsd of flights-management-sapi

```
<xs:schema attributeFormDefault="unqualified"
elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="CancellationNotification">
    <xs:complexType>
      <xs:sequence>
          <xs:element type="xs:string" name="PNR" minOccurs="1"/>
          <xs:element type="xs:string" name="PassengerLastName"
minOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Validate an XML document against an XML-Schema

In this section, you use the <xml-module:validate-schema /> component to validate incoming XML Messages against the generated XML-Schema.

6. **Validate XML messages against an XML-Schema:** Replace the Validation module component validating the payload with an <xml-module:validate-schema /> component defining the location of CancellationNotification.xsd created previously:

   main.xml of flights-management-sapi

   ```
   <flow name="receive-cancellation-notification">
     <http:listener />
     ...
     <xml-module:validate-schema
   schemas="schemas/CancellationNotification.xsd"/>
     ...
   </flow>
   ```

   Note: XML-Schema validation overrides the need to validate empty payloads.

7. **Catch validation errors:** Add to the error handler, the XML module namespace to catch any validation errors:

   main.xml of flights-management-sapi

   ```
   <flow name="receive-cancellation-notification">
     <http:listener />
     ...
     <error-handler>
       <on-error-continue when="#[['VALIDATION', 'XML-MODULE'] contains
   error.errorType.namespace]">
       ...
       </on-error-continue>
     </error-handler>
   </flow>
   ```

8. **Run, invoke, and check log:** Run flights-management-sapi and send a few cancellation notifications to the callback; observe the corresponding log entries:

   ```
   curl -ik -H "Content-Type:text/xml" -d
   "<CancellationNotification><PNR>PNR123</PNR><PassengerLastName>Mule</P
   assengerLastName></CancellationNotification>"
   ```

```
https://localhost:8081/api/cancelFlight
```

```
curl -ik -H "Content-Type:text/plain" -d "invalid content type"
https://localhost:8081/api/cancelFlight
```

```
curl -ik -H "Content-Type:text/xml" -d ""
https://localhost:8081/api/cancelFlight
```

```
curl -ik -H "Content-Type:text/xml" -d "invalid content"
https://localhost:8081/api/cancelFlight
```

Note: All invalid requests are now correctly validated before being sent to the VM queue.

Validate an XML document leniently

The Robustness principle, also known as Postel's Law, states that you should be conservative in what you send and liberal in what you accept from others. In this section, you update the XML-Schema, to implement the Robustness principle and leniently validate the received cancellation notifications, only validating required fields to make integrations more resilient to changes from other systems.

9. **Run, invoke, and check log:** Run flights-management-sapi and send cancellation notifications to the callback with an additional field for the Passenger's first name; this should fail:

```
curl -ik -H "Content-Type:text/xml" -d
"<CancellationNotification><PNR>PNR123</PNR><PassengerLastName>Mule</P
assengerLastName><PassengerFirstName>Max</PassengerFirstName></Cancell
ationNotification>" https://localhost:8081/api/cancelFlight
```

10. **Update XML-Schema file:** Update **CancellationNotification.xsd** adding an xsd:any element with lax validation to the CancellationNotification complex type:

CancellationNotification.xsd of flights-management-sapi

```
<xs:schema attributeFormDefault="unqualified"
elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="CancellationNotification">
```

```
    <xs:complexType>
    <xs:sequence>
     <xs:element type="xs:string" name="PNR" minOccurs="1"/>
     <xs:element type="xs:string" name="PassengerLastName"
minOccurs="1"/>
     <xs:any processContents="lax" minOccurs="0" />
    </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Note: The <xs:any /> element enables the author to extend the XML document with elements not specified by the schema.

Note: A minOccurs value of 0 allows zero or more occurrences of that element.

Note: Setting the processContents flag to lax instructs the XML processor to not validate any elements where the schema cannot be obtained.

Note: Similar functionality exists for JSON-Schema using the additionalProperties keyword.

Note: Other approaches to lenient validation include extracting individual fields using DataWeave or XPath and validating them using the Validation module.

11. **Run, invoke, and check log:** Run flights-management-sapi and send cancellation notifications to the callback with an additional field for the passenger's first name; this should succeed:

```
curl -ik -H "Content-Type:text/xml" -d
"<CancellationNotification><PNR>PNR123</PNR><PassengerLastName>Mule</P
assengerLastName><PassengerFirstName>Max</PassengerFirstName></Cancell
ationNotification>" https://localhost:8081/api/cancelFlight
```

Walkthrough 3-3: Validate JSON messages

In flights-management-sapi Mule app, cancellation notifications are transformed into internally controlled JSON-formatted flight canceled events. Although internally controlled, it can be advantageous to use defensive programming methods such as Design By Contract(DBC) to improve code quality and understanding.

In this walkthrough, you enhance flights-management-sapi to validate outgoing JSON messages against a JSON-Schema.

You will:

- Import JSON module.
- Create a JSON-Schema.
- Validate a JSON document against a JSON-Schema.

Solution file

You can see the end state of following this walkthrough in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module03/wt3-3_solution.

Starting file

If you did not complete the previous walkthrough, you can get a starting file located in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module03/wt3-2_solution.

Import JSON module

In this section, you import the JSON module in flights-management-sapi to use utility operations for validating JSON Messages against a JSON-Schema.

1. **Confirm Maven dependency management:** In **bom/pom.xml**, locate the existing dependency management entries for the JSON module, which has artifact ID mule-json-module.
2. **Add Maven dependency:** Add the corresponding Maven dependency for the JSON module to the **POM** of flights-management-sapi; confirm that Studio loads the module:

   pom.xml of flights-management-sapi

   ```
   <dependency>
     <groupId>org.mule.modules</groupId>
     <artifactId>mule-json-module</artifactId>
     <classifier>mule-plugin</classifier>
   ```

```
</dependency>
```

Note: Omit the <version /> element so that the version managed in the BOM takes effect.

Create a JSON-Schema

In this section, you generate an JSON-Schema for validating JSON Messages against using a sample JSON document.

3. **Generate flight canceled events JSON-Schema:** At **https://jsonschema.net/**, generate a JSON-Schema from the sample flight canceled events JSON:

```
{
  "pnr":"PNR123",
  "lastNameOfPassenger":"Mule"
}
```

4. **Create JSON-Schema file:** Create JSON-Schema file **FlightCancelledEvent.schema.json** in **src/main/resources/schemas** copying the contents of the generated JSON-Schema:

   FlightCancelledEvent.schema.json of flights-management-sapi

```
{
   "definitions": {},
   "$schema": "http://json-schema.org/draft-07/schema#",
   "$id":
"http://training.mulesoft.com/FlightCancelledEvent.schema.json",
   "type": "object",
   "title": "FlightCancelledEvent",
   "properties": {
     "pnr": {
       "$id": "#/properties/pnr",
       "type": "string",
       "examples": [
         "PNR123"
       ]
     },
     "lastNameOfPassenger": {
       "$id": "#/properties/lastNameOfPassenger",
       "type": "string",
       "examples": [
         "Mule"
       ]
```

```
      }
    },
    "required": ["pnr", "lastNameOfPassenger"]
}
```

Note: As the data structure is internally controlled, you can make use of the first statement of Postel's Law, to be conservative in what you send.

Validate a JSON document against a JSON-Schema

In this section, you use the <json-module:validate-schema /> component to validate outgoing JSON Messages against the generated JSON-Schema.

5. **Validate JSON messages against a JSON-Schema:** Add a <json-module:validate-schema /> component defining the location of FlightCancelledEvent.schema.json created previously:

   main.xml of flights-management-sapi

```
<flow name="deliver-flight-cancelled-event">
  <try>
    <ee:transform doc:name="To Flight Cancelled Event">
      ...
    </ee:transform>

    <json:validate-schema
schema="schemas/FlightCancelledEvent.schema.json" />
    ...
  </try>
  ...
</flow>
```

   Note: Although the JSON format is internally controlled, you use Design by Contract (DBC) postconditions to assert the transformation correctly changed the message state. You can disable this in production environments for performance.

   Note: The existing on-error-continue error handler already handles all validation errors and commits the ongoing transaction, thereby preventing redelivery of the message.

6. **Modify JSON-Schema:** Temporarily modify FlightCancelledEvent.schema.json configuring pnr to be a number type:

FlightCancelledEvent.schema.json of flights-management-sapi

```
{
   ...
   "properties": {
     "pnr": {
       "$id": "#/properties/pnr",
       "type": "number",
       "examples": [
         "PNR123"
       ]
     },
     ...
  }
}
```

7. **Run, invoke, and check log:** Run flights-management-sapi and send valid cancellation notifications to the callback; observe the corresponding log entries for the validation errors:

```
curl -ik -H "Content-Type:text/xml" -d
"<CancellationNotification><PNR>PNR123</PNR><PassengerLastName>Mule</P
assengerLastName></CancellationNotification>"
https://localhost:8081/api/cancelFlight
```

8. **Revert JSON-Schema:** Revert FlightCancelledEvent.schema.json configuring pnr back to a string type:

FlightCancelledEvent.schema.json of flights-management-sapi

```
{
   ...
   "properties": {
     "pnr": {
       "$id": "#/properties/pnr",
       "type": "string",
       "examples": [
         "PNR123"
       ]
     },
     ...
  }
}
```

Module 4: Orchestrating integration functionality

In this module, you apply essential Enterprise Integration Patterns to orchestrate multiple Mule apps and API invocations.

At the end of this module, you should be able to:

- Parallelize integration logic.
- Trace transactions across an application network.
- Retry failed API invocations.

Walkthrough 4-1: Parallelize integration logic

The invocations of the three System APIs in the check-in functionality of check-in-papi are independent of each other and should therefore be performed in parallel to reduce the overall latency. This walkthrough enhances check-in-papi with parallel API invocations using the Scatter-Gather router.

In this walkthrough, you parallelize the three System API invocations using the Scatter-Gather router. Importantly, you implement error handling to compensate for successful API invocations in the case that at least one of the other API invocations have failed.

You will:

- Invoke independent APIs concurrently using the Scatter-Gather router.
- Handle errors in one or more routes of a Scatter-Gather router.

Solution file

You can see the end state of following this walkthrough in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module04/wt4-1_solution.

Starting file

If you did not complete the previous walkthrough, you can get a starting file located in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module01/wt1-4_solution.

Invoke independent APIs concurrently using the Scatter-Gather router

In this section, you parallelize all independent System API invocations using the Scatter-Gather router and modify any post-processing to use the newly created event structure.

1. **Study current check-in logic:** In **check-in-by-pnr** of check-in-papi **main.xml**, review the flow references for each System API dependency and note that all are independent from one another but still sequentially called.
2. **Parallelize flow references:** Encapsulate the flow references for each System API in a Scatter-Gather router:

   main.xml of check-in-papi

   ```
   <flow name="check-in-by-pnr">
     ...
   ```

```
  <scatter-gather>
    <route>
      <flow-ref name="check-in-flights-management"/>
    </route>
    <route>
      <flow-ref name="register-passenger-data"/>
    </route>
    <route>
      <flow-ref name="create-payment-for-bags"/>
    </route>
  </scatter-gather>
  ...
</flow>
```

Note: The Scatter-Gather router combines the Mule events returned by each processing route into a new Mule event that is passed to the next event processor only after every route completes successfully.

3. **Access route result:** Modify the transformation to access the result of the PayPal SAPI create-payment-for-bags flow reference:

   main.xml of check-in-papi

```
<flow name="check-in-by-pnr">
  ...
  <scatter-gather>
    ...
  </scatter-gather>
  <ee:transform>
    <ee:message>
      <ee:set-payload><![CDATA[%dw 2.0
        output application/json
        var paypalReturn = payload['2'].payload
        ---
        {
          paymentID: paypalReturn.paymentID
        }]]></ee:set-payload>
    </ee:message>
  </ee:transform>
</flow>
```

Note: The combined Mule event is a map structure where each key is an integer that identifies the index of each route. The create-payment-for-bags is the third route called and the Scatter-

Gather router uses a zero-based index, so the result is accessed using index two.

4. **MUnit-test:** Run the MUnit test suite of check-in-papi; this should fail because the exception path tests assume an error type that is subsumed by the Scatter-Gather router:

```
cd $APDL2WS/check-in-papi
mvn clean verify -U -Dencrypt.key=secure12345
```

5. **Disable failing tests:** To **main-test-suite.xml**, ignore the test for **check-in-by-pnr-exception-path-test**:

main-test-suite.xml of check-in-papi

```
<munit:test name="check-in-by-pnr-exception-path-test" ignore="true">
...
</munit:test>
```

6. **MUnit-test:** Run the MUnit test suite; this should now succeed:

```
cd $APDL2WS/check-in-papi
mvn clean verify -U -Dencrypt.key=secure12345
```

7. **Homework: Update tests.** Adapt the MUnit test suite to changes in invoked APIs.

Handle errors in one or more routes of a Scatter-Gather router

Sending messages to multiple APIs, whether sequentially or in parallel, does not provide transactional guarantees: One API invocation may succeed while the next API invocation may fail, leaving the overall system in an inconsistent state. Sequential API invocations may stop any subsequent API invocations if a failure occurs, but any previous successful API invocations may still leave the system in an inconsistent state. Parallelizing API invocations with the Scatter-Gather router increases the number of successful API invocations if a failure occurs in any particular route thus increasing the amount of system inconsistency. However, some actions can be reversed, for example, a passenger successfully checked-in to a flight can be removed if the payment for bags fails.

In this section, you handle errors raised from the Scatter-Gather router. Using the Scatter-Gather router results, calculate any successful routes, invoking specific Mule flows to compensate for or reverse any actions created by the route.

8. **Study test failures:** Observe the log and note the error being raised of the expected error type no longer matching the actual error type: **MULE:COMPOSITE_ROUTING**.
9. **Catch routing errors:** Encapsulate the entire processing in a Try scope configured with an error handler containing an on-error-propagate scope catching the MULE:COMPOSITE_ROUTING. The scope should set two variables identifying all successful and failed routes:

   main.xml of check-in-papi

   ```
   <flow name="check-in-by-pnr">
     ...
     <try>
       <scatter-gather>
       ...
       </scatter-gather>
       <error-handler>
         <on-error-propagate type="MULE:COMPOSITE_ROUTING">
           <set-variable variableName="successfulRouteIndexes"
   value="#[(error.errorMessage.payload.results pluck $$ as Number)]"/>
           <set-variable variableName="failedRouteIndexes"
   value="#[(error.errorMessage.payload.failures pluck $$ as Number)]"/>
         </on-error-propagate>
       </error-handler>
     </try>
   </flow>
   ```

   Note: Errors occurring in any route, causing the route to fail, will in turn cause the Scatter-Gather router to throw a MULE:COMPOSITE_ROUTING error.

   Note: When a failure occurs, the results of both the successful and failed routes are contained in the error itself.

   Note: If a failure occurs in one route, it does not affect other routes as they are executed in parallel. This can lead to inconsistent data across each of the System APIs. To remedy this, you can use compensating transactions that can undo any event caused by the successful routes, which will lead to eventual consistency across each of the System APIs.

   Note: The DataWeave pluck function is used to iterate over the returned object and create an array of all successful and failed route indexes using the $$ operator to retrieve the key of each route. These two arrays of route indexes can then be used to determine any compensation logic required for successful routes.

10. **Create compensation Mule flows:** To **main.xml**, add the three Mule flows, one for each of the routes of the Scatter-Gather router, and store each flow name with its corresponding route

index in a variable before invoking the Scatter-Gather router:

main.xml of check-in-papi

```
<flow name="check-in-by-pnr">
  ...
  <set-variable variableName="allCompensationFlows" value="#[ ['check-
in-flights-management-compensate', 'register-passenger-data-
compensate', 'create-payment-for-bags-compensate'] ]"/>

  <try>
    <scatter-gather>
    ...
    </scatter-gather>
  </try>
  ...
</flow>
<flow name="check-in-flights-management-compensate">
  <logger level="INFO" message="Must compensate for successful check-
in-flights-management"/>
</flow>
<flow name="register-passenger-data-compensate">
  <logger level="INFO" message="Must compensate for successful
register-passenger-data"/>
</flow>
<flow name="create-payment-for-bags-compensate">
  <logger level="INFO" message="Must compensate for successful create-
payment-for-bags"/>
</flow>
```

Note: This implementation just logs when the compensation route is invoked. Realistically, each flow would contain logic to undo any action performed by the corresponding route, for example, executing an API invocation to remove a passenger from a flight.

11. **Reflect:** Discuss what each of these compensation API resources would have to do in practice.
12. **Invoke compensation Mule flows:** To the error handler in check-in-by-pnr add a variable to calculate the successful route indexes and create a list of all compensation Mule flows to be invoked. Create and reference a new Mule flow to iterate the list of successful routes and dynamically invoke the corresponding compensation flow:

main.xml of check-in-papi

```
<flow name="check-in-by-pnr">
```

```
  ...
  <try>
    <scatter-gather>
    ...
    </scatter-gather>
    <error-handler>
      <on-error-propagate type="MULE:COMPOSITE_ROUTING">
        ...
         <set-variable variableName="successfulRouteIndexes"
value="#[(error.errorMessage.payload.results pluck $$ as Number)]"/>
         <set-variable variableName="failedRouteIndexes"
value="#[(error.errorMessage.payload.failures pluck $$ as Number)]"/>
         <set-variable variableName="compensationFlows"
value="#[vars.successfulRouteIndexes map
vars.allCompensationFlows[$]]"/>
            <flow-ref name="compensate-successful-routes-of-check-in-
by-pnr"/>
      </on-error-propagate>
    </error-handler>
  </try>
</flow>
<flow name="compensate-successful-routes-of-check-in-by-pnr">
  <foreach collection="#[vars.compensationFlows]">
    <flow-ref name="#[payload]"/>
  </foreach>
</flow>
```

13. **Misconfigure API dependency:** In global.xml, prepend **X** to the host property for PayPal SAPI, thereby making all invocations to that API fail:

    global.xml of check-in-papi

```
<http:request-config name="paypalSapiConfig"
  basePath="...">
  <http:request-connection
    host="Xtngaa-paypal-sapi-dev-9yj2rh.rajrd4-2.usa-e1.cloudhub.io">
      ...
    </http:authentication>
  </http:request-connection>
</http:request-config>
```

14. **Run, invoke, and check log:** Run the Mule app, and invoke the check-in API, and observe the check-in-papi log entries, paying close attention the log statements from the corresponding

compensation flows:

```
curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
:\"Mule\",\"numBags\":2}"
https://localhost:8081/api/v1/tickets/PNR123/checkin
```

15. **Revert misconfiguration:** Undo the misconfiguration of the host in global.xml.

Walkthrough 4-2: Trace transactions across an application network using correlation IDs

flight canceled events in the mobile-notifications-eapp originate from multiple distributed Mule apps in the application network. cancellation notifications are POSTed from the Flights Management system to the flights-management-sapi, sent to a persistent VM queue, and subsequently transformed to flight canceled events and sent to an Anypoint MQ queue. Only then is the mobile-notifications-eapp aware of the message.

In this walkthrough, you trace cancellation notifications from flights-management-sapi through to flight canceled events in mobile-notifications-eapp. You review how correlation IDs propagate across various transports. Then you modify the publishing and consuming of flight canceled events to use an Anypoint MQ custom property to manually propagate correlation IDs.

You will:

- Follow transactions across API invocations.
- Follow transactions propagated via VM messages.
- Follow transactions propagated via Anypoint MQ messages.

Solution file

You can see the end state of following this walkthrough in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module04/wt4-2_solution.

Starting file

If you did not complete the previous walkthrough, you can get a starting file located in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module03/wt3-3_solution.

Follow transactions across API invocations

In this section, you — using an account with sufficient entitlements, or an instructor, perform a distributed log search in Anypoint Monitoring to track a distributed transaction across multiple Mule apps. This is a licensed feature that is not available with trial accounts, which is why you must have an account with sufficient entitlements or demonstrated by an instructor.

1. **Study Solution Architecture:** Study the solution design for US2: Flight Cancellation mobile Notifications in the Solution architecture.
2. **Run, invoke, and check log:** Invoke **check-in-papi**, inspect the log entries, and copy logged correlation ID:

```
curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
:\"Mule\",\"numBags\":2}"
https://localhost:8081/api/v1/tickets/PNR123/checkin
```

3. **Navigate to Anypoint Monitoring:** Using credentials with sufficient permissions, log in to the AnyAirline Anypoint Platform organization and navigate to Anypoint Monitoring.
4. **Search logs:** Using the correlation ID, perform a distributed log search and inspect the search results spanning multiple Mule apps, including the downstream System APIs.

   Note: Surround the search string in double quotes for an exact match:

   ```
   "65f8af40-0d30-11eb-93c7-12742c9eae8f"
   ```

   Note: Expand your time range if there is no matching data.

   Note: Mule automatically generates a correlation ID for every Mule event unless one is already set or propagated from another Mule app.

   Note: The HTTP Connector makes use of the X-CORRELATION-ID HTTP request header to propagate the correlation ID across HTTP requests.

   Note: Transports like HTTP automatically send the correlation ID by default, this can be configured on each transports global configuration.

   Note: By default, correlation IDs are a Java Universally Unique Identifier (UUID) string. You can override the default format by using the global configuration component and setting the DataWeave expression for generating correlation IDs. Although it is best to avoid making changes to the correlation ID generator, you might need to format the correlation ID for the events if your company has its own standard or format for correlation IDs, for example.

Follow transactions propagated via VM messages

In this section, you publish cancellation notifications received from the Flights Management system to the configured VM queue and review how the correlation ID is propagated across the VM transport barrier.

5. **Review cancellation notifications VM queues:** In **main.xml** of flights-management-sapi, review the Mule flows responsible for receiving an HTTP request, publishing to a VM queue, and reading from the VM queue:

main.xml of flights-management-sapi

```
<flow name="receive-cancellation-notification">
  <http:listener />
  ...
  <vm:publish
    queueName="${vm.cancelNotif.q.name}"
    sendCorrelationId="ALWAYS"
    config-ref="vmConfig"/>
</flow>

<flow name="deliver-flight-cancelled-event">
  <vm:listener
    queueName="${vm.cancelNotif.q.name}"
    transactionalAction="ALWAYS_BEGIN"
    config-ref="vmConfig">
    <redelivery-policy
      maxRedeliveryCount="${vm.maxRedeliveryCount}"
      idExpression="#[correlationId]"/>
  </vm:listener>
</flow>
```

Note: This code does not set or specify the correlation ID but rather just that redelivery of an event is identified by that event's correlation ID.

Note: The correlation ID is sent and propagated automatically.

Note: The correlation ID of the current Mule event is available as a predefined variable in DataWeave.

6. **Run, invoke, and check log:** Run flights-management-sapi and send a cancellation notification to the callback; observe the corresponding log entries:

```
curl -ik -H "Content-Type:text/xml" -d
"<CancellationNotification><PNR>PNR123</PNR><PassengerLastName>Mule</P
assengerLastName></CancellationNotification>"
https://localhost:8081/api/cancelFlight
```

Note: The Mule event's correlation ID is typically created by the <http:listener />, unless the correlation ID was sent by the HTTP client.

Note: The default is for the correlation ID to be sent to the VM queue if one is present, which there is in this case.

7. **Manually invoke with custom correlation ID:** Modify the HTTP request to specify an **X-CORRELATION-ID** header and send a cancellation notification to the callback; observe the corresponding log entries:

```
curl -ik -H "Content-Type:text/xml" -H "X-CORRELATION-ID: PNR123" -d
"<CancellationNotification><PNR>PNR123</PNR><PassengerLastName>Mule</P
assengerLastName></CancellationNotification>"
https://localhost:8081/api/cancelFlight
```

Note: As a custom correlation ID was set by the HTTP client, the <http:listener /> does not generate one. This is how a correlation ID can be passed from one API to another.

Follow transactions propagated via Anypoint MQ messages

In this section, you observe mobile-notifications-eapp consuming flight canceled events from an Anypoint MQ exchange and review how the correlation ID is propagated across the Anypoint MQ transport barrier. You modify the publishing and consuming of flight canceled events, to use a custom property to propagate correlation IDs, and use the Tracing module to update the correlation ID for a given Mule flow.

8. **Check log:** Run **mobile-notifications-eapp** and observe the log entries from consuming the flight canceled events resulting from the previous cancellation notification requests.

   Note: The Anypoint MQ connector does not automatically propagate correlation IDs (in contrast to the JMS, VM, and HTTP connectors, for example).

9. **Publish:** Update the Anypoint MQ publish operation in **flights-management-sapi** to send the correlationId as a custom property:

   main.xml of flights-management-sapi

```
<flow name="deliver-flight-cancelled-event">
  <vm:listener />
  ...
  <choice>
    <when expression="#[vars.msgValid]">
      <anypoint-mq:publish
        destination="cancelled-flights-exchg-dev"
        messageId="#[correlationId]"
        config-ref="amqConfig">
        <anypoint-mq:properties ><![CDATA[#[output application/java
        ---
        {
```

```
          "correlationId" : correlationId
        }]]]>
        </anypoint-mq:properties>
      </anypoint-mq:publish>
    </when>
  </choice>
</flow>
```

Note: The ID of an Anypoint MQ message must be unique. Make sure to choose a unique custom ID to avoid unwanted side effects of duplicated IDs. In FIFO queues, messages with duplicate IDs are overwritten.

Note: If you do not specify a message ID, then Anypoint MQ creates a new one.

Note: You use the correlation ID that has been passed along this message from the start as the ID of the Anypoint MQ message.

Note: The Anypoint MQ messageId has no effect on the Mule message correlation ID.

Note: It may not be enough to rely on messageId for propagating the correlationId due to the unique constraints of the field. Therefore, it is recommended to use a custom property for propagating the correlation ID. You can define properties for outgoing messages, for example, to provide compatibility with other messaging systems or to communicate the content type of a message.

10. **Log custom correlationId property:** In **mobile-notifications-eapp**, after receiving the flight canceled event, update the log statement to log the correlationId user property field:

    main.xml of mobile-notifications-eapp

```
<flow name="retrieve-cancellation-event">
  <anypoint-mq:subscriber />
  <logger level="INFO" message="#['Retrieve cancellation event from
queue: ' ++ (attributes.properties.correlationId default '')]"/>
</flow>
```

    Note: You use DataWeave to extract the custom user property from the received Anypoint MQ message.

    Note: Without additional modules, the Mule message correlationId field is immutable and can only be set by an event source. Therefore, when using Anypoint MQ, the value is always regenerated and you have to instead rely on a custom user property for tracing the transaction.

11. **Run, invoke, and check log:** Run **mobile-notifications-eapp**, invoke the AnyAirline-hosted **flights-management-sapi** in the **dev** environment to populate the queue, and observe the mobile-notifications-eapp log entries, paying close attention to the message correlation ID versus the logged user property correlation ID:

```
curl -ik -H "Content-Type:text/xml" -d
"<CancellationNotification><PNR>PNR123</PNR><PassengerLastName>Mule</P
assengerLastName></CancellationNotification>" https://tngaa-flights-
management-sapi-dev-9yj2rh.rajrd4-2.usa-
e1.cloudhub.io/api/cancelFlight
```

Note: You must send multiple requests as there are multiple clients subscribing to the same queue and consuming the message in a round-robin fashion.

12. **Add managed library dependency:** Locate the existing dependency management of mule-tracing-module in the **BOM** and add a matching entry to the mobile-notifications-eapp **POM**:

pom.xml of mobile-notifications-eapp

```
<dependencies>
  ...
  <dependency>
    <groupId>org.mule.modules</groupId>
    <artifactId>mule-tracing-module</artifactId>
    <classifier>mule-plugin</classifier>
  </dependency>
</dependencies>
```

13. **Process Mule flow with modified correlation ID:** Wrap the entire Mule flow processing, including error handlers, inside a Try scope within a With Correlation ID scope setting the Mule event correlation ID to the correlationId user property field:

main.xml of mobile-notifications-eapp

```
<flow name="retrieve-cancellation-event">
  <anypoint-mq:subscriber />
  <tracing:with-correlation-id
correlationId="#[attributes.properties.correlationId]">
    <try>
    ...
    </try>
  </tracing:with-correlation-id>
```

```
</flow>
```

Note: The Tracing module enables you to use the With Correlation ID scope to modify the correlation ID during the execution of said scope.

Note: You use a Try scope so that any errors raised within the With Correlation ID scope use the overriden correlation ID. Any error handlers outside of the With Correlation ID scope will use the original correlation ID if the With Correlation ID scope failed and prevented the scope from setting the new correlation ID.

14. **Run, invoke, and check log:** Run **mobile-notifications-eapp** and invoke the AnyAirline-hosted **flights-management-sapi** in the **dev** environment as before; log entries should now show the modified correlation ID.

Walkthrough 4-3: Retry failed API invocations

The check-in-papi Mule app implemented in previous walkthroughs invokes three System APIs and is only successful if all System API invocations are successful. Because each individual instance of an API invocation may fail for various reasons (many of them transient in nature), it is essential to retry failed API invocations to increase the chance of ultimate success.

When retrying API invocations, it is essential that the corresponding API implementation is idempotent. According to the HTTP spec, the implementations of GET, HEAD, OPTIONS, PUT, and DELETE (but not POST) to RESTful resources must be idempotent. This is the case for the three System APIs invoked by check-in-papi, so it's legitimate to retry here. This walkthrough adds retry logic to check-in-papi.

In this walkthrough, you use the Until Successful scope to add retry logic to the flights-management-sapi HTTP Request configuration determining transient and permanent errors.

You will:

- Use the Until Successful scope to retry failed API invocations.
- Differentiate between transient and permanent errors.

Solution file

You can see the end state of following this walkthrough in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module04/wt4-3_solution.

Starting file

If you did not complete the previous walkthrough, you can get a starting file located in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module04/wt4-2_solution.

Use the Until Successful scope to retry failed API invocations

In this section, you use the Until Successful scope with one of the System API HTTP Request configuration to retry failed API invocations.

1. **Run and invoke:** Run check-in-papi and invoke the check-in functionality it exposes; this should succeed, thereby confirming successful authentication with client ID and secret of check-in-papi against Flights Management SAPI in dev:

```
curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
:\"Mule\",\"numBags\":2}"
```

```
https://localhost:8081/api/v1/tickets/PNR123/checkin
```

2. **Misconfigure API dependency:** In global.xml, prepend **X** to the username property for flights-management-sapi, thereby making all invocations to the flights-management-sapi fail:

   global.xml of check-in-papi

```
<http:request-config
  name="flightsManagementSapiConfig"
  basePath="/api/v1">
  <http:request-connection
    host="tngaa-flights-management-sapi-dev-9yj2rh.rajrd4-2.usa-
e1.cloudhub.io"
    protocol="HTTPS">
    <http:authentication>
      <http:basic-authentication
        username="X<insert-your-client-id>"
        password="<insert-your-client-secret>" />
    </http:authentication>
  </http:request-connection>
</http:request-config>
```

3. **Run, invoke, and check log:** Run the Mule app and invoke the check-in API; this should fail, triggered internally by a **HTTP:UNAUTHORIZED**:

```
curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
:\"Mule\",\"numBags\":2}"
https://localhost:8081/api/v1/tickets/PNR123/checkin
```

4. **Retry flights-management-sapi HTTP Request configuration:** In **get-ticket-by-pnr** of **check-in-papi** main.xml, encapsulate the Flights Management SAPI GET ticket HTTP Request configuration in an Until Successful scope configured to retry any failures a maximum of **3 times** with a **1000 millisecond delay**:

   main.xml of check-in-papi

```
<flow name="get-ticket-by-pnr">
  <until-successful maxRetries="#[3]" millisBetweenRetries="#[1000]">
    <http:request config-ref="flightsManagementSapiConfig"
method="GET" path="/tickets/{PNR}" target="ticket" doc:name="FMS Get
Ticket" doc:id="2aaf810a-33e3-476d-9080-28ce3465dc2a">
              <http:uri-params ><![CDATA[#[output application/java
```

```
---
{
    "PNR" : vars.PNR
}]]]></http:uri-params>
      </http:request>
  </until-successful>
</flow>
```

Note: Both configuration arguments support expressions.

Note: This operation performs an HTTP GET requests and is expected to be idempotent. Therefore it can be retried upon failure.

Note: The Until Successful scope is synchronous and blocks further flow execution until complete.

Note: If the Until Successful scope is ultimately unsuccessful, a MULE:RETRY_EXHAUSTED error is raised.

5. **Run, invoke, and check log:** Run the Mule app and invoke the check-in API; this should still ultimately fail, but the log should indicate that the request was retried three times:

```
curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
:\"Mule\",\"numBags\":2}"
https://localhost:8081/api/v1/tickets/PNR123/checkin
```

Note: Any error types propagated from the Until Successful scope are subsumed by the MULE:RETRY_EXHAUSTED error type.

6. **Review error types:** Review the possible error types that can be raised by the HTTP Connector.

Note: All error types, regardless whether they are transient in nature, are retried. Retrying permanent errors,such as HTTP:UNAUTHORIZED, is wasteful and increases overall latency as retries are unlikely to solve the issue.

Differentiate between transient and permanent errors

In this section, you modify the Until Successful scope to only retry API invocations that are transient in nature.

7. **Catch permanent errors:** Encapsulate the HTTP Request configuration operation in a Try scope configured with an error handler. Configure the error handler with an on-error-propagate scope

to rethrow transient errors and an on-error-continue scope to consume known permanent errors and non-transient errors that originate from the same namespace so they are not retried. Configure a final on-error-propagate scope to rethrow non-transient errors that are unknown and originate outside of the module namespace. All error handlers must set a Boolean successful variable to false that is defaulted to true after invoking the Until Successful scope:

main.xml of check-in-papi

```
<flow name="get-ticket-by-pnr">
  <until-successful maxRetries="#[3]" millisBetweenRetries="#[1000]">
    <try>
      <http:request config-ref="flightsManagementSapiConfig"
method="GET" path="/tickets/{PNR}" target="ticket" doc:name="FMS Get
Ticket" doc:id="2aaf810a-33e3-476d-9080-28ce3465dc2a">
             <http:uri-params ><![CDATA[#[output application/java
---
{
    "PNR" : vars.PNR
}]]]></http:uri-params>
      </http:request>
    <set-variable variableName="successful" value="#[true]"/>
      <error-handler>
        <on-error-propagate
when="#[(['TOO_MANY_REQUESTS','INTERNAL_SERVER_ERROR','SERVICE_UNAVAIL
ABLE','TIMEOUT','CONNECTIVITY'] contains
error.errorType.identifier)]">
          <set-variable variableName="successful" value="#[false]"/>
        </on-error-propagate>
        <on-error-continue when="#[(error.errorType.namespace ==
'HTTP') or (['EXPRESSION','STREAM_MAXIMUM_SIZE_EXCEEDED'] contains
error.errorType.identifier))]">
          <set-variable variableName="successful" value="#[false]"/>
        </on-error-continue>
        <on-error-propagate>
          <set-variable variableName="successful" value="#[false]"/>
        </on-error-propagate>
      </error-handler>
    </try>
  </until-successful>
</flow>
```

Note: Although the HTTP Connector uses standard HTTP error types, it is not as simple as treating all 4xx HTTP statuses as permanent and all 5xx statuses as transient. For example, some 5xx error codes can be retried (such as 504 - Gateway Timeout, which may be transient).

But others such as 501 — Not Implemented are likely to be permanent in nature. Some HTTP APIs go as far as providing a HTTP response header indicating whether or not a request can be retried.

Note: The error handler makes use of DataWeave expressions to validate only the error identifier for known transient errors, regardless of namespace. This will be useful later in the walkthrough.

8. **Handle MULE:RETRY_EXHAUSTED:** Encapsulate the Until Successful scope operation in a Try scope configured with an error handler to consume the MULE:RETRY_EXHAUSTED error type if any errors ultimately fail:

   main.xml of check-in-papi

   ```
   <flow name="get-ticket-by-pnr">
     <try>
       <until-successful>
         ...
       </until-successful>
       <error-handler>
         <on-error-continue>
           <set-variable variableName="successful" value="#[false]"/>
         </on-error-continue>
       </error-handler>
     </try>
   </flow>
   ```

   Note: The multiple error handler levels are used to provide a consistent interface, otherwise transient errors would ultimately propagate and return an error after execution, whereas permanent errors would not.

9. **Run, invoke, and check log:** Run the Mule app and invoke the check-in API; this should still ultimately fail with APP:LASTNAME_MISMATCH indicating that only transient error types are in fact retried:

   ```
   curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
   :\"Mule\",\"numBags\":2}"
   https://localhost:8081/api/v1/tickets/PNR123/checkin
   ```

10. **Add Validation module Maven dependency:** In check-in-papi, add the corresponding Maven dependency for the Validation module to the **POM** of flights-management-sapi and then confirm that Studio loads the module:

pom.xml of check-in-papi

```
<dependency>
  <groupId>org.mule.modules</groupId>
  <artifactId>mule-validation-module</artifactId>
  <classifier>mule-plugin</classifier>
</dependency>
```

Note: Omit the <version /> element so that the version managed in the BOM takes effect.

11. **Validate success:** Validate whether the API invocation was successful and map to a custom error type **EXT:CANT_RETRIEVE_TICKET_DATA** if it was not:

    main.xml of check-in-papi

    ```
    <flow name="get-ticket-by-pnr">
      ...
      <validation:is-true expression="#[vars.successful]" message="Error
    getting ticket data">
        <error-mapping targetType="EXT:CANT_RETRIEVE_TICKET_DATA"/>
      </validation:is-true>
    </flow>
    ```

    Note: You must explicitly validate the result as the flow no longer returns errors to stop flow processing.

12. **Revert misconfiguration:** Undo the misconfiguration of the client_id in global.xml.
13. **Homework:** Analyze the retry commonalities in the three System API Mule flows of check-in-papi and factor out those commonalities into a new helper flow in apps-commons. This helper flow can then be used in all similar API invocations across all Mule apps. Repeat this for the PayPal SAPI and Passenger Data SAPI HTTP Request configurations in main.xml of check-in-papi and the HTTP Request configurations in health-common.xml of apps-commons.

Module 5: Storing objects for persistence, performance, and resilience

In this module, you use various techniques and Anypoint Platform components to store and manage state in an application network to increase performance and resilience.

At the end of this module, you should be able to:

- Persist data in an Object Store.
- Avoid expensive operations with the Cache scope.
- Apply a caching API policy.

Walkthrough 5-1: Persist data in an Object Store

This walkthrough extends check-in-papi by handling approved payments in payment-approval-by-pnr in addition to modifying the previously implemented check-in functionality. In doing so, it becomes apparent that ticket and check-in data available during check-in must be preserved for later handling of approved payments. Temporary persistent storage of this kind, for example, caching — is a good use case for an Object Store.

In this walkthrough, you configure a persistent Object Store using the Object Store Connector to temporarily persist data across Mule flow invocations before deploying to CloudHub 2.0 and inspecting the CloudHub Object Store v2 implementation.

You will:

- Configure a persistent Object Store.
- Access entries in an Object Store.
- Store entries in an Object Store.
- Deploy to CloudHub 2.0 and use Object Store v2.

Solution file

You can see the end state of following this walkthrough in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module05/wt5-1_solution.

Starting file

If you did not complete the previous walkthrough, you can get a starting file located in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module04/wt4-1_solution.

Configure a persistent Object Store

In this section, you study the current architecture and the state requirements across the check-in and payment approval Mule flows before beginning the approved payment handling implementation. You add the Object Store Connector and configure a persistent Object Store to store state across Mule flow invocations.

1. **Study System APIs invoked by check-in-papi:** Review the Solution architecture, particularly the High-level architecture and the design of US1: mobile Check-In. Note the dependency of check-in-papi on paypal-sapi for the core handling of approved payments being implemented in payment-approval-by-pnr. Also notice the logic required to display a boarding pass is contained within check-in-papi without dependencies.

2. **Review stubbed payment-approval-by-pnr logic:** Review the current implementation of payment-approval-by-pnr of check-in-papi main.xml. Note the two skeleton Mule flows invoked: **update-approvals** and **get-boarding-pass**.

   Note: This is the simplest, nonfunctional implemenation of payment-approval-by-pnr. The update-approvals Mule flow uses a Transform Message component to return a stub status response from PayPal SAPI. You will implement the logic to invoke PayPal SAPI shortly.

3. **Run, invoke, and check log:** Run the Mule app in Studio and invoke the API via cURL, supplying an arbitrary PNR and example JSON request body; this should return an HTTP 200 OK response:

```
curl -ik -X PUT -H "Content-Type: application/json"  -d "{ \"payerID
\": \"STJ8222K092ST\", \"paymentID\": \"PAY-1AKD7482FAB9STATKO\" }"
https://localhost:8081/api/v1/tickets/N123/paymentApproval
```

4. **Implement payment-approval-by-pnr logic:** In **update-approvals**, invoke PayPal SAPI, building the required POST body:

   main.xml of check-in-papi

```
<set-variable variableName="payerID" value="#[output application/json
--- {payerID: payload.payerID}]"/>

<flow name="update-approvals">
  <http:request config-ref="paypalSapiConfig" method="PUT"
path="/payments/{PaymentID}/approval" doc:id="24f7105a-3034-4603-a6a6-
d5610080374f"> <error-mapping sourceType="HTTP:BAD_REQUEST"
targetType="APP:INVALID_PAYMENT"/>
        <http:body ><![CDATA[#[vars.payerID]]]></http:body>
            <http:uri-params ><![CDATA[#[output application/java
---
{
    "PaymentID" : payload.paymentID
}]]]></http:uri-params>
    </http:request>
</flow>
```

   Note: There are sample JSON files in the src/main/resources/examples directory for the request and response to use as metadata or as a guide.

   Note: You build a payerID JSON object and store it in a variable for later use by the HTTP

Connector.

Note: If the HTTP Request operation operation fails with a HTTP:BAD_REQUEST, the error is mapped to a custom APP:INVALID_PAYMENT error type.

5. **Homework:** Use the common retry flow from the previous homework.
6. **Build boarding pass:** In **get-boarding-pass**, update the Transform Message component to start building the boarding pass assuming the payload will eventually contain a **ticket** and **checkIn** object with the missing required data:

   main.xml of check-in-papi

```
<flow name="get-boarding-pass">
  <ee:transform>
    <ee:message>
      <ee:set-payload><![CDATA[%dw 2.0
        output application/json
        var numBags = if (payload.checkIn.numBags > 0)
payload.checkIn.numBags else 0
        ---
        {
          PNR:           vars.PNR,
          lastName:      payload.ticket.ticketHolderLastName,
          airportArrive: payload.ticket.destination,
          airportDepart: payload.ticket.origin,
          bagsCount:     numBags,
          flight:        payload.ticket.flightNo,
          flightDate:    payload.ticket.flightDate,
          depart:        payload.ticket.depart,
          boarding:      payload.ticket.boarding,
          class:         payload.ticket.class,
          gate:          payload.ticket.gate,
          seat:          payload.ticket.seat
        }]]></ee:set-payload>
    </ee:message>
  </ee:transform>
</flow>
```

   Note: Only the PNR is available at this point. For now, assume the existence of a payload object that contains a checkIn object and a ticket object with the required data.

7. **Reflect:** The majority of the information needed is not available at this point in time of the Mule flow. Note that this information was available at the end of the check-in-by-pnr. If this data

could be persisted between the two isolated and separate flow invocations, then payment-approval-by-pnr could make use of it. One way to store state in Mule is to use an Object Store.

Note: Mule event state is lost between subsequent executions of Mule flows.

Note: A MuleSoft Object Store is a key-value store available as a service to all Mule runtimes, where the implementation of that service differs by deployment model (standalone versus cluster versus CloudHub).

Note: The Object Store component was designed to store state information between Mule flow invocations.

Note: From Mule application code, an Object Store is typically accessed via the Object Store Connector.

8. **Confirm Maven dependency management:** In **bom/pom.xml**, locate the existing dependency management entries for the Object Store Connector, which has artifact ID mule-objectstore-connector.
9. **Add Maven dependency:** Add the corresponding Maven dependency for the Object Store Connector to the **POM** of check-in-papi; confirm that Studio loads the module:

   pom.xml of check-in-papi

   ```
   <dependency>
     <groupId>org.mule.connectors</groupId>
     <artifactId>mule-objectstore-connector</artifactId>
     <classifier>mule-plugin</classifier>
   </dependency>
   ```

   Note: Omit the <version /> element so that the version managed in the BOM takes effect.

10. **Create persistent Object Store:** In global.xml, create a persistent Object Store named **pnrObjectStore**:

    global.xml of check-in-papi

    ```
    <os:object-store name="pnrObjectStore" persistent="true"/>
    ```

    Note: By default, each Mule application has an Object Store that is persistent and is always available to the Mule app. Here, however, we are creating a named persistent Object Store instead.

    Note: Persistence usually refers to storage that is copied to disk or some external storage, or

replicated across several nodes, depending on the deployment model.

Note: In-memory usually refers to data that is only stored in the JVM memory of one Mule runtime. When the persistent argument is set to false, the Object Store is transient, storing data only in-memory.

Access entries in an Object Store

In this section, you configure the Object Store Connector to preemptively retrieve the check-in data to be stored by check-in-by-pnr, validate the key exists in the Object Store, and remove entries from the Object Store when no longer required.

11. **Retrieve missing data from Object Store:** In **get-boarding-pass**, retrieve from the Object Store using the PNR as the key. If they key is found, it will eventually return an object containing the missing **ticket** and **checkIn** data. Add a default value to return an empty object if the PNR is not found:

    main.xml of check-in-papi

    ```
    <flow name="get-boarding-pass">
      <os:retrieve key="#[vars.PNR]" objectStore="pnrObjectStore">
        <os:default-value>#[{}]</os:default-value>
      </os:retrieve>
      <ee:transform>
      ...
      </ee:transform>
    </flow>
    ```

    Note: There is no query mechanism; objects are only retrievable by key.

    Note: If a key is not found and no default value is configured, an OS:KEY_NOT_FOUND error will be raised.

    Note: The maximum number of characters in a key is currently 256.

    Note: The values can be any serializable Java object.

12. **Validate Object Store contains missing data:** Before invoking get-boarding-pass, use the Object Store Connector to verify that the Object Store contains an entry keyed under the specified PNR. Store the result in a variable before using the Validation module to validate that the Object Store does in fact contain data under the specified key:

main.xml of check-in-papi

```
<flow name="payment-approval-by-pnr">
  <os:contains key="#[vars.PNR]" target="existsPNR"
objectStore="pnrObjectStore"/>
  <validation:is-true expression="#[vars.existsPNR]" message="PNR
check-in expired for this passenger. Passenger needs to check in
again.">
    <error-mapping targetType="EXT:BAD_REQUEST"/>
  </validation:is-true>
  ...
  <flow-ref name="update-approvals"/>
  <flow-ref name="get-boarding-pass"/>
</flow>
```

Note: Although isolated processes, the check-in-by-pnr Mule flow must be invoked before the payment-approval-by-pnr Mule flow. Validating that the check-in-by-pnr was in fact invoked and stored the required data in the Object Store provides a degree of safety.

Note: If the Object Store does not contain any data keyed by the PNR, a validation error is thrown and mapped to the EXT:BAD_REQUEST error type.

13. **Remove used Object Store data:** In **get-boarding-pass**, after building the boarding pass object, remove the Object Store entry keyed under the specified PNR:

    main.xml of check-in-papi

    ```
    <flow name="get-boarding-pass">
      <ee:transform>
      ...
      </ee:transform>
      <os:remove key="#[vars.PNR]" objectStore="pnrObjectStore"/>
    </flow>
    ```

    Note: The payment approval handling process can only be completed once per check-in. Therefore it safe to remove the Object Store entry to prevent duplicate payment approvals.

Store entries in an Object Store

In this section, you configure the Object Store Connector to store the check-in data required by payment-approval-by-pnr.

14. **Store missing data from Object Store:** In **check-in-by-pnr**, before building the check-in response, store the missing checkIn and ticket data required by payment-approval-by-pnr in the expected object structure keyed by the PNR:

    main.xml of check-in-papi

    ```
    <flow name="check-in-by-pnr">
      ...
      <os:store key="#[vars.PNR]" objectStore="pnrObjectStore">
        <os:value><![CDATA[#[{
          checkIn: vars.checkIn,
          ticket: vars.ticket
        }]]]></os:value>
      </os:store>

      <ee:transform>
      ...
      </ee:transform>
      ...
    </flow>
    ```

    Note: You build the object inline, directly within the operation itself.

    Note: The value can be any serializable Java object. There is currently a 10MB max value size restriction.

15. **Run, invoke, and check log:** Run the Mule app and invoke the paymentApproval API with no prior check-in; this should fail, triggered internally by an EXT:BAD_REQUEST, which indicates no value was found in the Object Store for the given key:

    ```
    curl -ik -X PUT -H "Content-Type: application/json"  -d "{ \"payerID
    \": \"STJ8222K092ST\", \"paymentID\": \"PAY-1AKD7482FAB9STATKO\" }"
    https://localhost:8081/api/v1/tickets/N123/paymentApproval
    ```

16. **Invoke:** Invoke the checkIn API via cURL, supplying a PNR and example JSON request body; this should return an HTTP 200 OK :

    ```
    curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
    :\"Mule\",\"numBags\":2}"
    https://localhost:8081/api/v1/tickets/PNR123/checkin
    ```

17. **Extract paymentID:** From the checkIn HTTP response body, extract the value of the paymentID, either by copy and pasting or by using tool support:

```
paymentID=$(curl -k -X PUT -H "Content-Type: application/json" -d "{
\"lastName\":\"Mule\",\"numBags\":2}"
https://localhost:8081/api/v1/tickets/PNR123/checkin | jq -r
".paymentID")
```

Note: This cURL command is configured to only return the HTTP response body, otherwise parsing the output as JSON fails.

Windows

```
curl -k -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
:\"Mule\",\"numBags\":2}"
https://localhost:8081/api/v1/tickets/PNR123/checkin
```

Note: You must manually copy and paste the payment ID if not using tooling such as jq.

18. **Invoke paymentApproval:** Invoke the paymentApproval via cURL, supplying matching PNRs and the extracted paymentID; this should return a HTTP 200 OK response:

Unix variants

```
curl -ik -X PUT -H 'Content-Type: application/json' -d "{ \"payerID\":
\"STJ8222K092ST\", \"paymentID\": \"$paymentID\" }"
https://localhost:8081/api/v1/tickets/PNR123/paymentApproval
```

Windows

```
SET paymentID="VALUE RETURNED FROM PREVIOUS CURL REQUEST IF NOT USING
JQ"
curl -ik -X PUT -H 'Content-Type: application/json' -d "{ \"payerID\":
\"STJ8222K092ST\", \"paymentID\": \"%paymentID%\" }"
https://localhost:8081/api/v1/tickets/PNR123/paymentApproval
```

Note: The two endpoints(checkIn and paymentApproval) act as part of a session-based cache when one Mule flow relies on the temporal storing of data from the other. Therefore it is important that each endpoint is invoked in order.

19. **Review Object Store configuration:** In **global.xml**, review the global configuration of the Object Store Connector and configure it with meaningful values to evict data entries after **one hour** and check for expired entries every **30 minutes**:

    global.xml of check-in-papi

    ```
    <os:object-store name="pnrObjectStore" entryTtl="1"
    entryTtlUnit="HOURS" expirationInterval="30" persistent="true"/>
    ```

    Note: You can configure how long data gets stored in an Object Store using entryTtl specified in the global configuration parameters for the Object Store Connector. This argument determines when to evict key-value pairs from the store. The values should be configured on a use-case basis depending on how long the stored resource is required. An Object Store, although being persistent, should not be used for permanent storage.

    Note: For Object Store v2, since Mule 4.2.1, the entryTTL is rolling by default. If no entryTTL value is configured, the default is used and as long as an entry is accessed at least once a week, the TTL is extended for a further 30 days automatically. If an entryTTL is configured, the TTL is static and tne entry will be evicted once the entryTTL expires, regardless of how often it is accessed.

    Note: The standard Mule Object Store, the original on premise based Object Store that is part of Mule Runtime has no limit on the TTL value and is always static.

    Note: The maximum time to live (TTL) is 2592000 seconds (30 days).

    Note: Object Stores in Mule versions earlier than 4.2.1 have a static TTL of 30 days by default.

    Note: You will be working with these settings in more detail in the next walkthrough.

Deploy to CloudHub 2.0 and use Object Store v2

In this section, you deploy Check-In PAPI to CloudHub 2.0 and view the Object Store v2 implementation.

20. **Update Mule app and dependencies to release versions:** Update the artifact version and references to a release version and not a SNAPSHOT:

    pom.xml of check-in-papi

    ```
    ...
    <parent>
      <!-- students: replace with your AP org ID -->
    ```

```
  <groupId>your-AP-organization-id</groupId>
  <artifactId>solutions-parent-pom</artifactId>
  <version>1.0.0</version>
  <relativePath>../parent-pom/pom.xml</relativePath>
</parent>
...
<version>1.0.0</version>
...
```

Note: You already updated the parent POMs versions in a previous walkthrough.

21. **Maven-build:** Run a full Maven build of check-in-papi skipping all unit tests using the same secure properties encryption key used in previous walkthroughs:

```
cd $APDL2WS/check-in-papi
mvn clean verify -U -Dencrypt.key=secure12345 -DskipTests=true
```

Note: You skip unit tests as you now need to update the payment-approval-by-pnr-happy-path-test to pre-seed the Object Store with a check-in, otherwise it will fail validation as seen previously in this walkthrough.

22. **Deploy:** In **Runtime Manager**, deploy the packaged check-in-papi artifact from the **target** directory using an application named: **check-in-papi-dev**, selecting the **CloudHub 2.0 US East Ohio Shared Space** as a deployment target, enabling **Object Store v2** on the deployment settings, enabling **Last-Mile Security** on the Ingress and adding **properties** that sets encrypt.key and disables autodiscovery:

```
encrypt.key=secure12345
anypoint.platform.gatekeeper=disabled
```

23. **Invoke:** Invoke the API using its endpoint URL; this should return an HTTP 200 OK response:

```
curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
:\"Mule\",\"numBags\":2}" https://check-in-papi-uniqid.shard.usa-
e2.cloudhub.io/api/v1/tickets/PNR123/checkin
```

Note: Ensure to replace the placeholders for the six-character unique id and shard for your individual application.

24. **Inspect Object Store:** In Runtime Manager, select the Object Store navigate through the partition, and view the stored keys.

    Note: Object Store v2 is the latest version of the CloudHub Object Store. It is a cloud-native implementation of the Object Store interface that is external to the Mule app and used by CloudHub-deployed applications.

    Note: The only way non-CloudHub Mule apps can access Object Store v2 is through the Object Store REST API, not through the Object Store Connector.

    Note: A non-persistent Object Store does not use the Object Store v2 service. The Object Store is implemented locally in each CloudHub replica and data is isolated within each CloudHub replica.

    Note: The Object Store v2 instance used by each Mule app is located in the same region as the worker where the Mule app is initially deployed. For example, if you deploy to the Singapore region, the Object Store persists in the Singapore region. If, after the initial deployment, you move the app to a different region, the Object Store v2 instance remains in the original region to avoid data loss. Your use of Object Store v2 never moves from one region to another.

    Note: The key for each entry is visible and human-readable. However, the value is stored as binary regardless whether the value was stored as JSON or as an object. Behind the scenes, Mule 4 executes binary serialization with the Mule internal serializer. The user interface cannot deserialize the object; hence, it can only tell you that the value is now binary in the Anypoint Platform user interface.

    Note: The TTL can be either rolling or static depending whether it should restart from every update to a key or just continue from the initial creation.

Walkthrough 5-2: Cache expensive operations

The previous walkthrough demonstrates storing state between separate Mule flows temporarily as part of a session-based cache when one Mule flow relies on the temporary storing of data from the other. Caching can also improve performance of client-side applications by avoiding transferring the same data over the network repeatedly. Rather than sending each and every request to a downstream API, you can use a cache to store previous responses and return a cached response instead where appropriate. One technique to caching in a Mule app is to use the Object Store Connector to manually store and retrieve data based on a key. However, for this particular use case, Mule also provides a Cache scope, which still uses the Object Store behind the scenes but provides a simplified interface.

Similar to retrying API invocations, it is essential that the API implementation being cached is idempotent and also safe to cache. Caching works well with HTTP GET requests because, for the same HTTP request data (including the URL) repeated invocation does not change the response. Operations invoked by check-in-papi to retrieve Passengers by their passport number from passenger-data-sapi is matching use case. This walkthrough adds caching logic to check-in-papi.

In this walkthrough, you use the Cache scope to add caching logic to a Passenger Data SAPI request to limit calls to the Passenger Data SAPI API.

You will:

- Create a persistent Object Store and cache strategy.
- Store data using the Cache scope and a cache strategy.

Solution file

You can see the end state of following this walkthrough in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module05/wt5-2_solution.

Starting file

If you did not complete the previous walkthrough, you can get a starting file located in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module05/wt5-1_solution.

Create a persistent Object Store and cache strategy

In this section, you configure a global cache strategy with a persistent Object Store for storing cached passenger data.

1. **Reflect:** In Studio, inspect the current check-in logic paying close attention to get-passenger-data-by-passport for retrieving passenger data via a passport number.
2. **Create persistent Object Store cache strategy:** In **global.xml**, create a cache strategy with an inline, private named **persistent** Object Store named **passengerObjectStore** that caches values for **30 days** and then set the cache key to a DataWeave expression to extract the passenger's passport number:

   global.xml of check-in-papi

   ```
   <ee:object-store-caching-strategy name="passengerCachingStrategy"
   keyGenerationExpression="#[vars.ticket.ticketHolderPassPortNo]">
     <os:private-object-store alias="passengerObjectStore"
   persistent="true" maxEntries="10000" entryTtl="30" entryTtlUnit="DAYS"
   expirationInterval="1" expirationIntervalUnit="DAYS"/>
   </ee:object-store-caching-strategy>
   ```

   Note: By default, the Cache scope uses a cache strategy that stores data in an in-memory Object Store. You can instead use a custom Object Store by overriding the cache strategy.

   Note: You can reference a standard existing Object Store instead if necessary. The benefit of using an inline private Object Store is that other components cannot interfere with the cache.

   Note: You set the entryTtl and expirationInterval to evict cached entries. The values should be configured on a use-case basis depending on how likely the cached resource will change and on the performance requirements of the client application. In this case, the maximum of 30 days has been used because the data being cached is unlikely to change.

   *Note: The cache is set with a maxEntries value of **10000**. This has been roughly calculated based on the size of the payload being cached and the entryTTL to avoid the cache consuming too much JVM heap (in-memory) or too much disk space (persistent).*

   Note: The keyGenerationExpression is a DataWeave expression to determine if two requests are considered equal. If a cache entry has a matching key, it is considered a cache hit and the cached value is returned. Otherwise, it is considered a cache miss and the downstream API will be invoked and the resulting response cached under the generated key (if the response is repeatable).

   Note: By default, the Cache scope uses an SHA256KeyGenerator and an SHA256 digest of the message payload to generate a unique key. However, you can set up your own key through a custom Caching Strategy as shown here.

   Note: Only the Mule message is cached — variables are not.

Store data using the Cache scope and a cache strategy

In this section, you encapsulate a cache-safe passenger-data-sapi request operation in a Cache scope and modify the Mule event to correctly set required variables.

3. **Cache safe request operation:** In **main.xml**, locate **get-passenger-data-by-passport** and encapsulate the HTTP request operation in a Cache scope, referencing the cache strategy created previously:

   main.xml of check-in-papi

   ```
   <flow name="get-passenger-data-by-passport">
    <ee:cache cachingStrategy-ref="passengerCachingStrategy">
       <http:request config-ref="passengerDataSapiConfig" method="GET"
   path="/passengers" doc:id="c08b805e-bf4c-49ec-a56d-d5edd5c90f5d">
           <http:query-params><![CDATA[#[output application/java
   ---
   {
       "passportNo" : vars.ticket.ticketHolderPassPortNo
   }]]]></http:query-params>
       </http:request>
     </ee:cache>
   </flow>
   ```

   Note: When a message enters the Cache scope, it uses the referenced cache strategy to determine if the Object Store already contains the key. If not, the nested operation is invoked, the Cache scope determines whether the message payload is nonrepeatable, and it stores the result for future invocations.

   Note: You can configure a cache strategy to be synchronized if concurrent cache access should not be allowed.

4. **Run and invoke:** Run check-in-papi and invoke the exposed check-in endpoint twice; the first request should succeed with an HTTP 200 OK response and the second should fail with an HTTP 400 Bad Request:

   ```
   curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
   :\"Mule\",\"numBags\":2}"
   https://localhost:8081/api/v1/tickets/PNR123/checkin
   ```

   ```
   curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
   :\"Mule\",\"numBags\":2}"
   ```

```
https://localhost:8081/api/v1/tickets/PNR123/checkin
```

Note: The ticket validation is performed against the cached operation, which stores the result in a target variable passenger. The Cache scope does not cache variables. Therefore it is important that no variables or targets are set within a Cache scope that are relied upon outside of the scope itself.

5. **Move target variable:** Move the setting of the target **passenger** variable outside of the Cache scope, from **get-passenger-data-by-passport** to the flow reference in **validate-ticket-passport-matches**:

   main.xml of check-in-papi

```
<flow name="validate-ticket-passport-matches">
  ...
  <flow-ref name="get-passenger-data-by-passport" target="passenger"/>
  ...
</flow>
<flow name="get-passenger-data-by-passport">
 <ee:cache cachingStrategy-ref="passengerCachingStrategy" >
  <http:request config-ref="passengerDataSapiConfig" method="GET"
path="/passengers" doc:id="c08b805e-bf4c-49ec-a56d-d5edd5c90f5d">
        <http:query-params ><![CDATA[#[output application/java
---
{
    "passportNo" : vars.ticket.ticketHolderPassPortNo
}]]]></http:query-params>
      </http:request>
  </ee:cache>
</flow>
```

6. **Tune log config:** Change **log4j2-test.xml** to show DEBUG log entries created by the **com.mulesoft.mule.runtime.cache** logger:

   log4j2.xml of check-in-papi

```
<AsyncLogger name="com.mulesoft.mule.runtime.cache" level="DEBUG" />
```

7. **Run, invoke, and check log:** Run check-in-papi again, making sure to **clear application data** and invoke the exposed Check-In PAPI check-in endpoint **twice** and paying close attention to the cache hit messages on subsequent requests:

```
curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
:\"Mule\",\"numBags\":2}"
https://localhost:8081/api/v1/tickets/PNR123/checkin
```


```
curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
:\"Mule\",\"numBags\":2}"
https://localhost:8081/api/v1/tickets/PNR123/checkin
```


Note: The first request should result in a cache miss as it is the first time the API is invoked.

Note: Subsequent requests should result in a cache hit with the timer showing subsequent requests being more performant.

Note: Application data refers to any state stored by the Mule runtime, such as Object Store data whether manually stored via the Object Store Connector, or stored by other components such as watermarks and OAuth tokens.

Note: You can configure and clear application data from the Studio Run Configuration.

Note: You can manually clear application data in ARM via the Application Data page of a deployed Mule app.

Walkthrough 5-3: Apply a caching API policy

The previous walkthrough demonstrates caching data on the client side to avoid transferring the same data over the network repeatedly. This avoids unnecessary network calls and improves overall application performance. This is all governed by each individual API client. However, there is another scenario where the API implementation may want to enlist its own caching functionality on the server side to optimize against computationally expensive processing such as executing expensive database operations. In this scenario, API clients still make network calls, but it is generally dictated by the server whether data is returned from the cache or not. When implementing this style of caching, the same constraints on cache-safe data apply.

In this walkthrough, you apply and configure HTTP caching as an automated policy to implement a server-side cache.

You will:

- Apply an API policy for HTTP caching

Solution file

You can see the end state of following this walkthrough in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module05/wt5-1_solution.

Apply an API policy for HTTP caching

In this section, you apply HTTP caching as an API policy to passenger-data-sapi in the dev environment.

1. **Navigate to API Manager:** Navigate to API Manager and using credentials with sufficient permissions, log in to your Anypoint Platform organization and navigate to API Manager.
2. **Manage API:** Manage passenger-data-sapi using the **Create new API** option as an **HTTP API**. Create a new **Endpoint with Proxy** with the following implementation endpoint URL: **https://tngaa-passenger-data-sapi-dev-9yj2rh.rajrd4-2.usa-e1.cloudhub.io** deployed to **CloudHub 2.0** and the proxy application name: **passenger-data-sapi-dev**.

   Note: You use the HTTP API option over managing an API from Exchange to save time.

3. **Apply API policy:** In API Manager apply the HTTP caching API policy to **passenger-data-sapi**, with **attributes.requestUri** as the cachingKey, **2592000** as the Entry Time To Live (in Seconds), **10000** as the Maximum Cache Entries, enabling **Persistent Cache** and set the Invalidation Header to **X-CACHE-INVALIDATE**.

   *Note: The DataWeave expression: attributes.requestUri uses the **cachingKey**, which contains*

the query parameter passportNo.

Note: The cache is set as distributed because passenger-data-sapi is deployed to multiple CloudHub workers and you consider it more important to avoid accessing the Passenger Data system than to avoid invoking the CloudHub Object Store v2 implementation (which is also a remote service).

Note: In the cache, the Entry Time To Live (in Seconds) value is set to 2592000, which is the maximum Object Store TTL of 30 days, because the data being cached is unlikely to change.

Note: In the cache, the Maximum Cache Entries value is set to 10000. This has been roughly calculated on the size of the payload being cached and the entry TTL to avoid the cache consuming too much JVM heap (in-memory) or too much disk space (persistent).

Note: You leave the default expression to cache only GET and HEAD requests as these are the only cache-safe methods exposed by passenger-data-sapi.

Note: You can set up a custom expression to invalidate cache entries if a certain condition in the request is met.

4. **Enable Object Store v2:** In **Runtime Manager**, locate the proxy application created for Passenger Data SAPI in the **dev** and enable **Object Store v2**.

   Note: Object Store v2 must be enabled on the application to use persistent caching for the HTTP caching API policy.

5. **Locate fully qualified domain name:** In the Runtime Manager dashboard, check the fully qualified domain name of the Mule app and copy the API's endpoint URL with the additional six-character uniq-id and shard.
6. **Invoke:** Invoke the **passenger-data-sapi** proxy endpoint multiple times, timing its execution with an existing **passportNo** and the client ID and secret for the **dev** instances of Passenger Data SAPI; each should succeed with an HTTP 200 OK response:

   ```
   time curl -ik -X GET -u
   e66105d66e8b4520b091127620221cce:A1c099D51C50420b84641D71b229EBac
   "http://gl-passenger-data-sapi-dev-uniqid.shard.usa-
   e2.cloudhub.io/api/v1/passengers?passportNo=P3JR0BZ2OY"
   ```

   Note: Ensure to replace the placeholders for the six-character unique id and shard for your individual application.

   Note: The scheme is HTTP not HTTPS.

*Note: You use the same client ID and secret that you configured as part of check-in-papi for **passenger-data-sapi** in a previous walkthrough.*

Note: The first request should result in a cache miss as it is the first time the API is invoked.

Note: Subsequent requests should result in a cache hit, returning an 'age' HTTP header indicating how long in seconds the resource has been cached for and the timer showing subsequent requests being more performant.

7. **Verify cache:** In **Runtime Manager**, verify the Object Store for the Passenger Data SAPI proxy application in the **dev** contains a matching key for the request path containing the passportNo.
8. **Invalidate cache:** Invoke Passenger Data SAPI, specifying the cache invalidation header; this should succeed with an HTTP 200 OK response:

```
time curl -ik -X GET -u <insert-your-client-id>:<insert-your-client-
secret> -H "X-CACHE-INVALIDATE:invalidate" "http://gl-passenger-data-
sapi-dev-uniqid.shard.usa-
e2.cloudhub.io/api/v1/passengers?passportNo=P3JR0BZ2OY"
```

Note: The hostname will be different for you.

Note: The scheme is HTTP not HTTPS.

Note: This request will not return an 'age' HTTP header as it will invalidate the matching cache entry and bypass the cache.

Note: The timer should indicate the request taking longer as it is bypassing the cache.

Note: The header value invalidate will invalidate a cache entry with a matching cache key. You can also specify the value invalidate-all, which will invalidate all the key-value pairs from the cache.

Module 6: Componentizing reusable integration functionality

In this module, you identify and extract reusable Mule app code from multiple Mule apps and the apps-commons library-style Mule plugin. Then you use it in two separate first-class Mule runtime extensions: an XML SDK module and a custom API policy.

At the end of this module, you should be able to:

- Create an XML SDK component.
- Create a custom API policy.

Walkthrough 6-1: Create an XML SDK component

The current check-all-dependencies-are-alive implementation of check-in-papi shares many commonalities across the three <http:request /> instances. If you completed the optional homework, this logic is reused via the apps-commons shared library. This implementation is currently verbose and awkward to reuse and can further be improved along many dimensions, such as resilience and performance.

In this walkthrough, you create a custom XML SDK module to encapsulate and reuse the logic to check whether an endpoint is alive and to retry HTTP requests.

You will:

- Generate an XML SDK module from a Maven archetype.
- Create XML SDK module operations.
- Reuse XML SDK module operations.
- Deploy an XML SDK module to Exchange.
- Import an XML SDK module into another Mule app.

Solution file

You can see the end state of following this walkthrough in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module06/wt6-1_solution.

Starting file

If you did not complete the previous walkthrough, you can get a starting file located in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module05/wt5-2_solution.

Generate an XML SDK module from a Maven archetype

In this section, you run the mule-extensions-xml-archetype to generate the skeleton for the new custom XML SDK module and modify the generated XML SDK module to inherit from a parent POM to gain access to common dependencies.

1. **Execute archetype:** In a command-line interface, navigate to the **base directory** of your Studio workspace and run the Maven archetype, replacing the groupId with your Anypoint Platform organization ID; this should succeed:

Unix variants

```
cd $APDL2WS

mvn -B -f bom/pom.xml archetype:generate -DarchetypeGroupId
=org.mule.extensions -DarchetypeArtifactId=mule-extensions-xml-
archetype -DarchetypeVersion=1.2.0 -DgroupId=your-AP-organization-id
-DartifactId=resilience-mule-extension -DmuleConnectorName=resilience-
mule-extension -DextensionName=resilience -Dpackage=.
-DoutputDirectory=../
```

Windows

```
cd $APDL2WS

mvn -B -f bom/pom.xml archetype:generate -DarchetypeGroupId
=org.mule.extensions -DarchetypeArtifactId=mule-extensions-xml-
archetype -DarchetypeVersion=1.2.0 -DgroupId=your-AP-organization-id
-DartifactId=resilience-mule-extension -DmuleConnectorName=resilience-
mule-extension -DextensionName=resilience -Dpackage=.
-DoutputDirectory=C:\windows\full\path\to\your\workspace
```

Note: The archetype artifact is retrieved from the mulesoft-releases repository already defined in the BOM.

Note: The muleConnectorName argument is the fully qualified name of the module project.

Note: The extensionName argument is used to create the namespace prefix for module operations.

Note: The package argument is set to create the module-resilience.xml file in the root directory of src/main/resources.

Note: This POM uses the Anypoint Platform organization ID for the groupId, as this is a prerequisite for publishing to Exchange, which you will do later.

Note: The Exchange Maven Facade API does not support dynamic parameters within groupIds such as ${aa.ap.org.id}, and it requires the value to be the hardcoded Anypoint Platform organization ID.

2. **Import into Studio:** Import the module into Studio by creating a **New General Project**, setting the project name as **resilience-mule-extension**, and changing the default location to the location of your generated module.

3. **Browse the generated artifact:** Inspect the generated directory structure of the newly created XML SDK module named resilience-mule-extension.

   Note: The module-resilience.xml file in the src/main/resources directory defines the module including all its operations.

   Note: The assertion-munit-test.xml file in the src/test/munit directory contains a generated test-suite for the generated operations defined in module-resilience.xml.

4. **Turn POM into child POM:** Turn the XML SDK module into a Maven child project of your existing **parent POM** in its parent directory:

   pom.xml of resilience-mule-extension

   ```
   <parent>
     <!-- students: replace with your AP org ID -->
     <groupId>a63e6d25-8aaf-4512-b36d-d91b90a55c4a</groupId>
     <artifactId>solutions-parent-pom</artifactId>
     <version>1.0.0-SNAPSHOT</version>
     <relativePath>../parent-pom/pom.xml</relativePath>
   </parent>
   ```

   Note: The mule.version property must be defined despite not being directly referenced from within the POM, as it is indirectly used by mule-extensions-maven-plugin.

5. **Homework:** Remove the now unnecessary properties, dependency, and plugin versions managed by the inherited BOM.
6. **Add managed HTTP Connector dependency:** Add a dependency entry for the HTTP Connector without a version to <dependencies /> in the resilience-mule-extension **POM**:

   pom.xml of resilience-mule-extension

   ```
   <dependencies>
     ...
     <dependency>
       <groupId>org.mule.connectors</groupId>
       <artifactId>mule-http-connector</artifactId>
       <classifier>mule-plugin</classifier>
     </dependency>
   </dependencies>
   ```

7. **Disable tests:** Back up and disable the MUnit test by renaming **assertion-munit-test.xml**:

```
cd $APDL2WS/resilience-mule-extension

mv src/test/munit/assertion-munit-test.xml src/test/munit/assertion-
munit-test.xml.bkp
```

Note: The generated MUnit test suite contains the original target namespace and schemaLocations that must be manually kept in sync with the module. The operation tests are soon to be obsolete as you implement your own custom operations.

Note: The Module Testing Framework leverages MUnit to test custom modules. The Maven Resources plugin must copy the MUnit resources to the target directory for the MUnit Maven plugin to find and run the tests.

8. **Maven-build:** In a command-line interface, navigate to the **base directory** of resilience-mule-extension and run a Maven build of this XML SDK module; this should succeed:

```
cd $APDL2WS/resilience-mule-extension

mvn clean verify
```

9. **Homework:** Adapt the MUnit test suite to the correct module's namespace and operations.

Create XML SDK module operations

In this section, you modify the module template to create a new operation: is-endpoint-alive to encapsulate the reusable logic for checking whether an endpoint is alive.

10. **Refactor generated module:** In Studio, modify the namespace **prefix**, **tns**, and **schemaLocation**, removing **module-** as well as removing Smart Connector from the module name and removing all the autogenerated operations:

module-resilience.xml of resilience-mule-extension

```
<module name="Resilience Smart Connector"
  prefix="resilience"
  xmlns:tns="http://www.mulesoft.org/schema/mule/resilience"
  xsi:schemaLocation="
    http://www.mulesoft.org/schema/mule/resilience
http://www.mulesoft.org/schema/mule/resilience/current/mule-
resilience.xsd">
</module>
```

Note: At the time of this writing, Studio does not support the project type of XML SDK modules.

Note: The term Smart Connector is a legacy term no longer in use.

11. **Study existing Mule flows:** Study the <http:request /> instances, the variables and properties required by **check-all-dependencies-are-alive** of **check-in-papi** and any additional parameters that will be required to retry HTTP requests.
12. **Add operations:** Add a new operation to **module-resilience.xml** called **is-endpoint-alive**:

    module-resilience.xml of resilience-mule-extension

    ```
    <module name="Resilience"
      prefix="resilience">
      <operation name="is-endpoint-alive">
        ...
      </operation>
    </module>
    ```

    Note: The combination of the prefix resilience and the operation name is-endpoint-alive will result in clients calling the operation using resilience:is-endpoint-alive.

13. **Add parameters:** Add parameters for **url**, **responseTimeoutInMillis**, **maxRetries**, and **millisBetweenRetries** that will be required by the <http:request />:

    module-resilience.xml of resilience-mule-extension

    ```
    <operation name="is-endpoint-alive">
      <parameters>
        <parameter name="url" displayName="URL" type="string"
    use="REQUIRED"/>
        <parameter name="responseTimeoutMillis" displayName="Response
    Timeout in Milliseconds" type="number" use="OPTIONAL"
    defaultValue="#[2000]"/>
        <parameter name="maxRetries" displayName="Max Retries"
    type="number" use="OPTIONAL" defaultValue="#[3]"/>
        <parameter name="millisBetweenRetries" displayName="Milliseconds
    Between Retries" type="number" use="OPTIONAL" defaultValue="#[2000]"/>
      </parameters>
    </operation>
    ```

    Note: Each parameter will become an XML attribute on the operation element.

    Note: Parameters are locally scoped to each operation and must be passed on each invocation.

If you want to pass global configuration that affects all operations of an XML SDK component instance in a project, you can use a <property /> instead. Properties are defined at the root of the module, outside of any operation element.

Note: To define parameter data types with complex structures, you can create a catalog of data types that you can use within the module. This catalog of data types can make use of both XML-Schema and JSON-Schema.

Note: The XML SDK is strongly typed, so the data type of defined parameters for every operation is statically set for both the input and output.

14. **Retry a generic HTTP Request configuration:** Add an Until Successful scope and a generic <http:request /> to the body of the operation with a matching **global configuration**, **namespace**, and **schemaLocation**. Refactor the attributes to use the new operation parameters as variables. Handle any errors with an error handler containing an on-error-continue scope consuming all errors and sets a Boolean successful variable to false, which defaults to true after invoking the Until Successful scope:

    module-resilience.xml of resilience-mule-extension

```
<module name="Resilience Smart Connector"
  prefix="resilience"
  xmlns:http="http://www.mulesoft.org/schema/mule/http"
  xsi:schemaLocation="
   http://www.mulesoft.org/schema/mule/http
   http://www.mulesoft.org/schema/mule/http/current/mule-http.xsd">

  <http:request-config name="httpRequestConfig"/>

  <operation name="is-endpoint-alive">
    <parameters>
      <parameter name="url" displayName="URL" type="string"
use="REQUIRED"/>
      <parameter name="responseTimeoutMillis" displayName="Response
Timeout in Milliseconds" type="number" use="OPTIONAL"
defaultValue="#[2000]"/>
      <parameter name="maxRetries" displayName="Max Retries"
type="number" use="OPTIONAL" defaultValue="#[3]"/>
      <parameter name="millisBetweenRetries" displayName="Milliseconds
Between Retries" type="number" use="OPTIONAL" defaultValue="#[2000]"/>
    </parameters>
    <body>
      <mule:until-successful maxRetries="#[vars.maxRetries]"
millisBetweenRetries="#[vars.millisBetweenRetries]">
```

```
        <mule:try>
          <http:request method="GET" url="#[vars.url]"
followRedirects="true" responseTimeout=
"#[vars.responseTimeoutMillis]">
            <http:response-validator>
              <http:success-status-code-validator values="200..299"/>
            </http:response-validator>
          </http:request>
          <mule:set-payload value="#[true]"/>
          <mule:error-handler>
            <mule:on-error-continue>
              <mule:set-payload value="#[false]"/>
            </mule:on-error-continue>
          </mule:error-handler>
        </mule:try>
      </mule:until-successful>
    </body>
  </operation>
</module>
```

Note: The target namespace is that of the connector itself, not the standard core Mule namespace as seen in Studio applications. Thereforem it is required to prefix any core Mule element with the namespace prefix defined in the module (root) element: mule:

Note: Each operation parameter becomes a variable scoped to the enclosing operation.

15. **Define output type:** Add the expected output type to the operation, i.e., a Boolean indicating whether or not the endpoint is alive:

    module-resilience.xml of resilience-mule-extension

    ```
    <operation name="is-endpoint-alive">
      ...
      <output type="boolean"/>
    </operation>
    ```

    Note: The output type must be the last element of an operation.

    Note: You can set the output to void by removing the element. This prevents the operation from setting the payload of the calling Mule flow's Mule event, even if the operation's internal behavior involves modifying the payload.

    Note: To define output types with complex structures, you can create a catalog of data types

that you can use within the module. This catalog of data types can make use of both XML-Schema and JSON-Schema.

16. **Maven-build:** In a command-line interface, navigate to the **base directory** of resilience-mule-extension and run a Maven build of this module; this should succeed:

```
mvn clean install
```

17. **Homework:** Parameterize other configuration values that can make the module more flexible.

Reuse XML SDK module operations

In some cases, operations have repeated message processors, on which you can rely if they are encapsulated in a new operation and called from other places.

Each operation element defined in a module element can be reused in the same module if the operation does not have cyclic dependencies.

In this section, you create retry functionality to retry HTTP requests in a new operation that can be used internally by the module or externally by a Mule app. Then you refactor is-endpoint-alive to reuse this functionality internally instead of making its own HTTP requests.

18. **Create new retry-http operation:** Move the existing Until Successful scope and <http:request /> logic to a new operation named **retry-http**, supplying the same operation parameters as previously used:

module-resilience.xml of resilience-mule-extension

```
<module>
  ...
  <operation name="retry-http">
    <parameters>
      <parameter name="url" displayName="URL" type="string"
use="REQUIRED"/>
      <parameter name="responseTimeoutMillis" displayName="Response
Timeout in Milliseconds" type="number" use="OPTIONAL"
defaultValue="#[2000]"/>
      <parameter name="maxRetries" displayName="Max Retries"
type="number" use="OPTIONAL" defaultValue="#[3]"/>
      <parameter name="millisBetweenRetries" displayName="Milliseconds
Between Retries" type="number" use="OPTIONAL" defaultValue="#[2000]"/>
    </parameters>
    <body>
```

```
      <mule:until-successful maxRetries="#[vars.maxRetries]"
millisBetweenRetries="#[vars.millisBetweenRetries]">
        <http:request method="GET" url="#[vars.url]"
          followRedirects="true"
          responseTimeout="#[vars.responseTimeoutMillis]">
          <http:response-validator>
            <http:success-status-code-validator values="200..299"/>
          </http:response-validator>
        </http:request>
     </mule:until-successful>
    </body>
  </operation>
</module>
```

19. **Handle errors and define output and error types:** Catch any errors raised from the Until Successful scope component with an on-error-continue error handler and map them to a new module-specific error via the raise-error component, setting the output type to **any** and declaring the new **error** type:

    module-resilience.xml of resilience-mule-extension

```
<module>
  ...
  <operation name="retry-http">
    ...
    <body>
      <mule:try>
        <mule:until-successful maxRetries="#[vars.maxRetries]"
millisBetweenRetries="#[vars.millisBetweenRetries]">
          <http:request method="GET" url="#[vars.url]"
followRedirects="true" responseTimeout=
"#[vars.responseTimeoutMillis]">
            <http:response-validator>
              <http:success-status-code-validator values="200..299"/>
            </http:response-validator>
          </http:request>
        </mule:until-successful>
        <mule:error-handler>
          <mule:on-error-continue>
            <mule:raise-error type="RESILIENCE:RETRIES_EXHAUSTED"
description="Exhausted re-tries executing http request"/>
          </mule:on-error-continue>
        </mule:error-handler>
      </mule:try>
```

```
    </body>
    <output type="any"/>
    <errors>
      <error type="RETRIES_EXHAUSTED"/>
    </errors>
  </operation>
</module>
```

Note: The errors element declares the error types the XML SDK can raise (or map) within the operation body.

Note: Defined errors can be discovered when using the Studio UI.

20. **Invoke internal operation:** In the **is-endpoint-alive** operation, replace the previous <http:request /> logic with an internal reference to the new **retry-http** operation, adding the same operation parameters and forwarding them on to the new operation:

module-resilience.xml of resilience-mule-extension

```
<module xmlns:tns="http://www.mulesoft.org/schema/mule/resilience"
 xsi:schemaLocation="...
http://www.mulesoft.org/schema/mule/resilience
http://www.mulesoft.org/schema/mule/resilience/current/mule-
resilience.xsd">

<module>
 ...
 <operation name="is-endpoint-alive">
    <parameters>
      <parameter name="url" displayName="URL" type="string"
use="REQUIRED"/>
      <parameter name="responseTimeoutMillis" displayName="Response
Timeout in Milliseconds" type="number" use="OPTIONAL"
defaultValue="#[2000]"/>
      <parameter name="maxRetries" displayName="Max Retries"
type="number" use="OPTIONAL" defaultValue="#[3]"/>
      <parameter name="millisBetweenRetries" displayName="Milliseconds
Between Retries" type="number" use="OPTIONAL" defaultValue="#[2000]"/>
    </parameters>
    <body>
      <mule:try>
        <tns:retry-http url="#[vars.url]"
responseTimeoutMillis="#[vars.responseTimeoutMillis]"
          maxRetries="#[vars.maxRetries]"
```

```
millisBetweenRetries="#[vars.millisBetweenRetries]"/>
        <mule:set-payload value="#[true]"/>
        <mule:error-handler>
          <mule:on-error-continue>
            <mule:set-payload value="#[false]"/>
          </mule:on-error-continue>
        </mule:error-handler>
      </mule:try>
    </body>
    <output type="boolean"/>
  </operation>
</module>
```

Note: Both operations are visible to any Mule app using the module. If you want an operation to be truly internal, you can mark the visibility attribute as PRIVATE on the operation element.

Note: As this is an internal reference, you use the tns namespace identifier defined in the module element.

21. **Maven-build:** In a command-line interface, navigate to the **base directory** of resilience-mule-extension and run a Maven build of this module; this should succeed:

```
mvn clean install
```

Deploy an XML SDK module to Exchange

With the Anypoint Exchange Maven Facade API, Apache Maven clients can publish and consume Exchange assets, including Mule 4 extensions.

Mule 4 Extensions that declare dependencies that don’t exist in Maven Central or the MuleSoft Maven repositories are not currently supported. Therefore, any artifact deployed to Exchange that relies on a custom asset, such as the parent POM and BOM introduced in the previous sections, is required to deploy to Exchange prior to the extension.

The steps to publish an asset with Maven are slightly different for each asset type, and different assets use different Maven plugins. In this case, you will use the Exchange Mule Maven plugin to deploy each POM as a custom asset and finally deploy resilience-mule-extension to Exchange.

22. **Study BOM:** Browse the content of bom/pom.xml, identifying the **Exchange Mule Maven plugin** configuration located in a Maven profile with the ID **deploy-to-exchange-v3**:

bom/pom.xml

```
<profile>
  <id>deploy-to-exchange-v3</id>
  <build>
    <plugins>
      <plugin>
        <groupId>org.mule.tools.maven</groupId>
        <artifactId>exchange-mule-maven-plugin</artifactId>
        <version>${exchange.mule.maven.plugin.version}</version>
        <executions>
          <execution>
            <id>validate</id>
            <phase>validate</phase>
            <goals>
              <goal>exchange-pre-deploy</goal>
            </goals>
          </execution>
          <execution>
            <id>deploy</id>
            <phase>deploy</phase>
            <goals>
              <goal>exchange-deploy</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</profile>
```

Note: Mule extensions can reuse the Exchange Mule Maven plugin configuration because extensions are built using the Mule Extensions Maven plugin and not the Mule Maven plugin, therefore not causing multiple deployments.

Note: The exchange-pre-deploy goal must be run to prevalidate the asset with Exchange, where the plugin checks various pre-conditions such as unique asset versions. Without explicitly running this goal, the Exchange API can return a status code of 412 (Precondition Failed). The plugin goal can be bound to other phases such as the validate phase, but it is set to the deploy phase in this course to not interfere with local builds.

Note: The Exchange Maven Facade API does not support dynamic parameters within groupIds such as ${aa.ap.org.id}, and it requires the value to be the hardcoded Anypoint Platform

organization ID.

Note: The Exchange Maven Facade API requires that every asset deployed must have an artifact name element defined in the POM.

Note: The Exchange Maven Facade API requires a Maven property named type with the value set custom for custom assets.

23. **Study BOM distributionManagement:** Notice the custom property with your Anypoint Platform organization used to configure **<distributionManagement />**:

bom/pom.xml

```
...
<properties>
  ...
  <student.deployment.ap.orgid>your-AP-organization-
id</student.deployment.ap.orgid>
</properties>
...
<repositories>
  <repository>
    <id>anypoint-exchange-v3-student-deployment</id>
    <name>Anypoint Exchange</name>
    <url>https://maven.anypoint.mulesoft.com/api/v3/maven</url>
  </repository>
  ...
</repositories>
<distributionManagement>
  <repository>
    <id>anypoint-exchange-v3-student-deployment</id>
    <name>AnyAirline Anypoint Exchange</name>

<url>https://maven.anypoint.mulesoft.com/api/v3/organizations/${studen
t.deployment.ap.orgid}/maven</url>
    <layout>default</layout>
  </repository>
</distributionManagement>
...
```

Note: The Exchange Mule Maven plugin also requires the Exchange v3 API to publish custom assets, which is reflected in the repository URL.

24. **Create Exchange Contributor Connected App:** In Anypoint Access Management, create a new Connected App that acts on its **own behalf (client credentials)**, for writing assets to Exchange, adding the **Exchange Contributor** scope and retrieve its client ID and secret.
25. **Update settings.xml:** In **settings.xml**, change the credentials of the matching server entry for the repository (defined in the distributionManagement section of bom/pom.xml to use the Connected App credentials created previously for writing to that repository):

    settings.xml

    ```
    <server>
      <id>anypoint-exchange-v3-student-deployment</id>
      <username>~~~Client~~~</username>
      <password>your-capp-contributor-cid~?~your-capp-contributor-
    secret</password>
    </server>
    ```

    Note: You can also use Anypoint Platform credentials for an account of your Anypoint Platform organization with which you were able to search your Exchange, such as the trial account credentials created previously. However, it is best practice to use a Connected App for security.

    Note: To use Connected App authentication, provide basic authentication and define the username as ~~~Client~~~ and the password as clientID~?~clientSecret. Replace clientID with the client ID. Replace clientSecret with the client secret.

26. **Deploy parent POMs to Exchange:** Deploy the **BOM** and **parent POM** used by resilience-mule-extension to your remote Exchange repository:

    ```
    cd $APDL2WS/bom
    mvn deploy -f pom.xml -Pdeploy-to-exchange-v3

    cd $APDL2WS/parent-pom
    mvn deploy -f pom.xml -Pdeploy-to-exchange-v3
    ```

    Note: When a SNAPSHOT version is deployed to Exchange, it is created in the development lifecycle phase allowing the asset version to be overwritten and permanently deleted, whenever. When a stable asset version is deployed, Exchange will treat the asset as a production asset and enforce the standard Exchange rules upon it. Such as locking the version number so it cannot be overwritten and only allowing permanent deletion within the first 7 days.

27. **Deploy XML SDK module to Exchange:** Deploy resilience-mule-extension to your remote Exchange repository:

```
cd $APDL2WS/resilience-mule-extension
mvn deploy -Pdeploy-to-exchange-v3
```

Import an XML SDK module into another Mule app

In this section, you import resilience-mule-extension as a Maven dependency into check-in-papi to reuse the operations of the XML SDK module.

28. **Add managed XML SDK module dependency:** Locate the existing dependency management of resilience-mule-extension in the **BOM** and add a matching entry to the check-in-papi **POM**:

    pom.xml of check-in-papi

    ```
    <dependencies>
      ...
      <dependency>
        <groupId>${student.deployment.ap.orgid}</groupId>
        <artifactId>resilience-mule-extension</artifactId>
        <classifier>mule-plugin</classifier>
      </dependency>
    </dependencies>
    ```

29. **Refresh Studio project:** In Studio, close and reopen the **check-in-papi** project; Studio should resolve the resilience-mule-extension dependency and show it as a project dependency.

    Note: If Studio does not find resilience-mule-extension, then you may need to force reloading Maven dependencies from a command-line interface with the `-U` option, and then close and reopen the check-in-papi Studio project:

    ```
    mvn clean verify -U
    ```

30. **Replace imported Flow references:** In **check-all-dependencies-are-alive** of health.xml replace each <flow-ref /> with the new **resilience:is-endpoint-alive** operation with the relevant URL property:

    health.xml of check-in-papi

    ```
    <sub-flow name="check-all-dependencies-are-alive">
      <resilience:is-endpoint-alive
    url="${external.flightsManagementSapi.aliveUrl}"/>
      <resilience:is-endpoint-alive
    ```

```
url="${external.passengerDataSapi.aliveUrl}"/>
  <resilience:is-endpoint-alive url=
"${external.paypalSapi.aliveUrl}"/>
</sub-flow>
```

31. **Run and invoke:** Run check-in-papi in Studio and invoke its readiness health check endpoint as usual; this should succeed:

```
curl -ik https://localhost:8081/ready
```

32. **Homework:** Update the operation to only retry failed API invocations if the error is transient in nature.

Walkthrough 6-2: Create a custom API policy

In this walkthrough, you create a custom API policy for operational logging of HTTP requests and responses. You then apply this API policy both offline in Studio and online via API Manager using an automated policy.

You will:

- Generate a custom API policy skeleton from a Maven archetype.
- Develop a custom API policy to log HTTP requests and responses.
- Apply an offline custom API policy.
- Apply an online custom API policy.
- Develop complex policies using Handlebars.

Solution file

You can see the end state of following this walkthrough in the solutions folder of the student files ZIP located in the Course Resources: $APDL2DIST/walkthroughs/devint/module06/wt6-2_solution.

Generate a custom API policy skeleton from a Maven archetype

In this section, you generate a custom API policy skeleton project from a Maven archetype.

1. **Execute archetype:** In a command-line interface, navigate to the **base directory** of your Studio workspace and run the Maven archetype with your Anypoint Platform organization ID; this should succeed:

   Unix variants

   ```
   cd $APDL2WS

   mvn -B -f bom/pom.xml archetype:generate -DarchetypeGroupId
   =org.mule.tools -DarchetypeArtifactId=api-gateway-custom-policy-
   archetype -DarchetypeVersion=1.2.0 -DgroupId=your-AP-organization-id
   -DartifactId=custom-message-logging-policy -Dversion=1.0.0-SNAPSHOT
   -Dpackage=mule-policy -DpolicyDescription="Policy for logging
   messages" -DpolicyName="Custom Message Logging" -DoutputDirectory=../
   ```

   Windows

   ```
   cd $APDL2WS
   ```

```
mvn -B -f bom/pom.xml archetype:generate -DarchetypeGroupId
=org.mule.tools -DarchetypeArtifactId=api-gateway-custom-policy-
archetype -DarchetypeVersion=1.2.0 -DgroupId=your-AP-organization-id
-DartifactId=custom-message-logging-policy -Dversion=1.0.0-SNAPSHOT
-Dpackage=mule-policy -DpolicyDescription="Policy for logging
messages" -DpolicyName="Custom Message Logging" -DoutputDirectory=C:
\windows\full\path\to\your\workspace
```

Note: The Maven groupId is required to be an Anypoint Platform organization ID in order to successfully deploy to Exchange. Refer to the final solution to see an advanced Maven configuration that supports maintaining two POM files: One for deploying to Exchange and one for standard development processes.

2. **Browse the generated artifact:** Inspect the generated directory structure of the newly created API policy named custom-message-logging-policy.

Develop a custom API policy to log HTTP requests and responses

In this section, you modify the generated API policy to inherit from a parent POM to gain access to the repository settings. Then you modify the policy template to log messages before and after receiving HTTP requests.

3. **Import into Studio:** Import the API policy into Studio by creating a **New General Project**, setting the project name as **custom-message-logging-policy**, and changing the default location to the location of your generated API policy.
4. **Turn POM into child POM:** Turn the API policy into a Maven child project of your existing **parent POM** in its parent directory and remove the <properties />, <distributionManagement />, and <plugins /> elements except for the Mule Maven plugin:

   pom.xml of custom-message-logging-policy

```
<parent>
  <!-- students: replace with your AP org ID -->
  <groupId>a63e6d25-8aaf-4512-b36d-d91b90a55c4a</groupId>
  <artifactId>solutions-parent-pom</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <relativePath>../parent-pom/pom.xml</relativePath>
</parent>
<groupId>your-AP-organization-id</groupId>
<artifactId>custom-message-logging-policy</artifactId>
<version>1.0.0-SNAPSHOT</version>
<name>Custom Message Logging</name>
```

```
<description>Policy for logging messages</description>
<packaging>mule-policy</packaging>
<build>
  <plugins>
    <plugin>
      <groupId>org.mule.tools.maven</groupId>
      <artifactId>mule-maven-plugin</artifactId>
      <extensions>true</extensions>
    </plugin>
  </plugins>
</build>
```

Note: The mule.maven.plugin.version overrides the Mule Maven plugin version defined in the BOM, which uses a newer, incompatible configuration that will cause the overall build to fail.

Note: The default Exchange URL added is the v2 API whereas this course uses a newer v3 API.

Note: The archetype generates a Maven Deploy plugin configuration that is rendered obsolete by the v3 API.

5. **Maven-build:** In a command-line interface, navigate to the **base directory** of custom-message-logging-policy and run a Maven build of this API policy; this should succeed:

```
cd $APDL2WS/custom-message-logging-policy
mvn clean verify
```

6. **Log:** Add log statements to the **http-policy:proxy** in **template.xml** before and after the **http-policy:execute-next** element parameterizing the message using handlebar syntax and logging input and output for each HTTP request received:

template.xml of custom-message-logging-policy

```
<http-policy:proxy
    name="{{{policyId}}}-custom-policy">
  <http-policy:source>
    <logger level="INFO"
      message="{{{message}}}"
      category="org.mule.runtime.logging.policy-{{{policyId}}}"/>

    <http-policy:execute-next />

    <logger level="INFO"
      message="{{{message}}}"
```

```
      category="org.mule.runtime.logging.policy-{{{policyId}}}"/>
  </http-policy:source>
</http-policy:proxy>
```

Note: Any Mule event operation that is defined before the http-policy:execute-next element will be executed before the start of a flow with an HTTP Listener.

Note: The http-policy:execute-next element is required to continue the Mule event processing. The http-policy:execute-next element can trigger other policies or the application flow. Any policy can stop the Mule event processing chain by not executing http-policy:execute-next.

Note: Mule event operations defined after the http-policy:execute-next element will be executed after the flow completes. These Mule Event processors are able to process the Mule Event returned from the flow.

Note: You remove the http-transform:add-headers element because it is not needed in this case. This element is available from mule-http-policy-transform-extension plugin that can be used to modify HTTP requests within an API policy.

Note: Whereas the http-policy:source element intercepts incoming HTTP requests to <http:listener /> elements, the http-policy:operation element can intercept outgoing HTTP requests made be the <http:request />.

7. **Configure user parameters:** In **custom-message-logging-policy.yaml**, add the configuration for the parameterized values used in our template:

   custom-message-logging-policy.yaml of custom-message-logging-policy

```
id: Custom Message Logging
name: Custom Message Logging
description: Policy for logging messages
category: Custom
type: custom
resourceLevelSupported: true
encryptionSupported: false
standalone: true
requiredCharacteristics: []
providedCharacteristics: []
configuration:
  - propertyName: message
    name: Message
    description: |
      Mule Expression for extracting information from the message to
log.
```

```
      e.g. #[attributes.headers['id']]
    type: expression
    optional: false
```

Note: Every parameter listed here is rendered as an expected user input in API Manager's UI.

Note: You can use propertyName to specify the internal name of the parameter. This value must be unique within the policy.

Note: Parameters can have various types, including expression, string etc. Parameters also be masked, mandated and defaulted via configuration. Each option influences how the policy configuration page is displayed in API Manager's UI.

8. **Maven-build:** In a command-line interface, navigate to the **base directory** of custom-message-logging-policy and run a Maven build of this API policy; this should succeed:

```
cd $APDL2WS/custom-message-logging-policy

mvn clean package
```

Apply an offline custom API policy

In this section, you test the API policy locally by applying it as an offline API policy directly to the runtime and not through API Manager.

9. **Go to MULE_HOME policies directory:** In a command-line interface, navigate to the **MULE_HOME** directory of the Studio-embedded Mule runtime:

```
cd <MULE_HOME>/policies
```

Note: You can find the MULE_HOME location by searching for MULE_HOME in a running Studio console.

10. **Copy policy:** In a command-line interface, copy the packaged policy from the **target** directory to the local **policy-templates** directory:

```
cp $APDL2WS/custom-message-logging-policy/target/custom-message-
logging-policy-1.0.0-SNAPSHOT-mule-policy.jar
<MULE_HOME>/policies/policy-templates
```

11. **Create offline policy definition:** In a text editor, create a new file named **custom-message-logging-policy-definition.json**, supplying the API ID for check-in-papi from dev-properties.yaml in the **<MULE_HOME>/policies/offline-policies** directory:

    custom-message-logging-policy-definition.json

    ```
    {
     "template" : {
       "groupId" : "<your-ap-organization-id>",
       "assetId" : "custom-message-logging-policy",
       "version" : "1.0.0-SNAPSHOT"
     },
     "api": [
       {
         "id": "<insert-your-api-id>"
       }
     ],
     "order": 1,
     "configuration" : {
       "message" : "#['Hello']"
     }
    }
    ```

    Note: These values are configured by API Manager's UI when using an online policy.

    Note: You must manually specify all the API IDs for the APIs to which you want this policy to apply.

    Note: You must specify the Exchange groupId, assetId, and version of the template used to deploy the offline policy definition.

12. **Run, invoke, and check log:** Run check-in-papi from Studio, invoke the API using cURL as before, and study the log entries originating from the API policy:

    ```
    curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
    :\"Smith\",\"numBags\":2}"
    https://localhost:8081/api/v1/tickets/PNR123/checkin
    ```

Apply an online custom API policy

In this section, you deploy custom-message-logging-policy to Exchange and apply it as an online API policy through API Manager.

13. **Update to release version:** Update the artifact version from a SNAPSHOT to a release version:

    pom.xml of custom-message-logging-policy

    ```
    ...
    <version>1.0.0</version>
    ..
    ```

    Note: API Manager does not allow you to choose development assets versions of policies such as Maven snapshots deployed via Exchange Mule Maven plugin.

    Note: All stable policy versions are immutable. After a stable policy is deployed to Exchange, that version number can no longer be changed or reused. Therefore, you must increase the artifact version for every deployment to Exchange.

14. **Deploy to Exchange:** In a command-line interface, navigate to the **base directory** of custom-message-logging-policy and run a Maven deploy of this API policy; this should succeed:

    ```
    cd $APDL2WS/custom-message-logging-policy
    mvn deploy
    ```

    Note: The Mule Maven plugin is automatically integrated with the Exchange Mule Maven plugin; therefore, there is no need to define the Exchange Mule Maven plugin configuration as well. It is only needed to directly reference the Exchange Mule Maven plugin when deploying non-Mule artifacts such as custom libraries and POMs. If both are configured, deployment will happen twice, and the second deployment will fail with a conflict.

    Note: The parent POMs this policy relies on were deployed in the previous walkthrough. If they had not been, deployment would be required prior to deploying anAPI policy.

15. **Apply automated policy:** In the API Manager **dev** environment, apply a new automated policy.
16. **Configure custom-message-logging-policy:** Select the latest custom-message-logging-policy API policy, logging a custom message:

    ```
    #['Hello']
    ```

Develop complex policies using Handlebars

In this section, you use the Handlebars templating framework to decide which sections of the policy are applied depending on the user configuration. Specifically, you use the if helper to conditionally

render which logger element is added to the policy dependent on whether the user has chosen to log requests before or after an HTTP request.

17. **Configure conditional parameters:** In **custom-message-logging-policy.yaml**, add the configuration options to log before or after an HTTP request is received:

    custom-message-logging-policy.yaml of custom-message-logging-policy

```
id: Custom Message Logging
name: Custom Message Logging
description: Policy for logging messages
category: Custom
type: custom
resourceLevelSupported: true
encryptionSupported: false
standalone: true
requiredCharacteristics: []
providedCharacteristics: []
configuration:
  - propertyName: message
    name: Message
    description: |
      Mule Expression for extracting information from the message to
log.
      e.g. #[attributes.headers['id']]
    type: expression
    optional: false

  - propertyName: beforeRequest
    name: Before Calling API
    type: boolean
    optional: false
    defaultValue: true

  - propertyName: afterRequest
    name: After Calling API
    type: boolean
    optional: false
    defaultValue: false
```

18. **Conditionally log:** In **template.xml**, wrap the log statements before and after **http-policy:execute-next** with if blocks that validate the beforeRequest and afterRequest conditions, respectively:

template.xml of custom-message-logging-policy

```
<http-policy:proxy name="{{{policyId}}}-custom-policy">
  <http-policy:source>
    {{#if beforeRequest}}
    <logger:logger level="INFO" message="{{{message}}}"
category="org.mule.runtime.logging.policy-{{{policyId}}}"/>
    {{/if}}
    <http-policy:execute-next />
    {{#if afterRequest}}
    <logger level="INFO" message="{{{message}}}"
category="org.mule.runtime.logging.policy-{{{policyId}}}"/>
    {{/if}}
  </http-policy:source>
</http-policy:proxy>
```

Note: Handlebars also supports other built-in helper blocks such as else, each and unless.

19. **Bump Maven artifact version:** In **pom.xml**, bump the patch version of custom-message-logging-policy:

    pom.xml of custom-message-logging-policy

    ```
    ...
    <version>1.0.1</version>
    ...
    ```

    Note: API Manager does not allow you to choose SNAPSHOT versions of policies.

    Note: All stable policy versions are immutable. After a stable policy is deployed to Exchange, that version number can no longer be changed or reused. Therefore, you must increase the artifact version for every deployment to Exchange.

20. **Deploy to Exchange:** In a command-line interface, navigate to the **base directory** of custom-message-logging-policy and run a Maven deploy of this API policy; this should succeed:

    ```
    mvn clean deploy
    ```

    Note: The Exchange Maven Facade API enables you to both create your asset and set the mutable data describing it in the same request. The mutable data of an asset includes tags, custom fields, categories, and documentation pages. The final solution includes a complete example of creating an API policy with documentation and tags.

21. **Apply new automated policy version:** In the API Manager **dev** environment, apply the new automated policy, removing the previous policy.
22. **Configure custom-message-logging-policy:** Select the latest custom-message-logging-policy API policy, logging a custom message as before, but this time choose to log only before an API invocation.

Appendix A: Coding conventions

A.1. General

- **Importing API specifications into a Studio project:** Don't import from API Designer but import a well-defined version of the API specification (RAML definition or OpenAPI definition) from Exchange using the API Sync feature available in Studio 7.4.0 or later.
- **Abbreviations:** Use wherever space is limited, such as for API Designer project names, Mule app names, and the like:
  - SAPI for System API
  - PAPI for Process API
  - EAPI for Experience API
- **Definition of Done:**
  - Implementation meets all functional and nonfunctional requirements.
  - Readiness endpoint checks liveness of all the Mule app's dependencies.
  - MUnit tests assert all functional requirements and all tests pass.
  - Functional Monitors source code checked in to src/test/funmon.
  - Mule app deployed to all CloudHub environments using Maven and scripts to invoke Maven.
  - Functional Monitors deployed for all Anypoint Platform environments and all statuses displayed in green.

A.2. RAML definitions and API Designer projects

- **API Designer project**:
  - Use a business-friendly project name, choose name of dominant RAML object and keep all other naming defaults:
    - Project name for RAML definition: Passenger Data SAPI
      - title: Passenger Data SAPI
      - Defaults:
        - RAML file: passenger-data-sapi.raml (visible in API implementation)
        - Exchange asset ID: passenger-data-sapi
        - Exchange name: Passenger Data SAPI (visible to all Exchange users)
    - Project name for RAML type or RAML library containing that type: PaymentApproval
      - Type name: PaymentApproval
      - Defaults:

        - RAML file: paymentapproval.raml
        - Exchange asset ID: paymentapproval
        - Exchange name: PaymentApproval (visible to all Exchange users)
    - Combine RAML type with examples into a **RAML library**, don't just use a RAML type fragment
    - Use subdirectories within project:
        - For RAML type: types
        - For examples: examples
- **API specification attributes:**
    - title: Check-In PAPI, see above
    - version: v1
    - baseUri: use the one from the prod environment, for example https://check-in-papi-uniqid.shard.usa-e2.cloudhub.io/api/v1

A.3. Mule apps

- **Project-level:**
    - Maven <groupId />: com.mulesoft.training.anyairline
    - Studio project name = Maven <artifactId /> = Maven project name:
        - For API implementation: passenger-data-sapi = Exchange asset ID of API exposed by that API implementation
- **Mule flows:** check-all-dependencies-are-alive, register-passenger-data
    - Also correct names autogenerated by Studio
    - **Prefer subflows** over flows where possible, for example, unless error handling or a message source is needed in the flow
- **Global config elements:**
    - Use camelCase: apiHttpListenerConfig
    - With the exception of global error handlers (which are conceptually more like flows): api-error-handler
- **doc:name:**
    - For event processors: sentence case without punctuation: "Set PNR from query param"
        - Such that the entire graphical representation of each flow reads as much as possible like a paragraph in a story
    - For <set-variable /> include the name of the variable and further detail only if really helpful) "origPayload"

    - For <raise-error /> use the type of error being raised, omitting the APP namespace but including all other namespaces
    - For <flow-ref /> use the name of the flow being invoked: check-all-dependencies-are-alive
        - Flow names should be descriptive already
- **File-level:**
    - Use double quotes (") for XML attributes and single quotes for DataWeave strings (so that an XML formatter can enforce XML file layout):

```
<set-payload
  value="#[output application/json --- {message: 'Invalid passenger
name record given.'}]"
  doc:name="error" />
```

    - Mule flow config files:
        - error.xml for global error handlers
        - global.xml for global definitions that are not error handlers
        - api.xml for top-level, API-related Mule flows called from APIkit
        - main.xml for main integration logic (and additional Mule flow config files if helpful)
        - health.xml for liveness and readiness endpoints (see below)
    - Configuration properties files:
        - properties.yaml for environment-independent config
        - dev-properties.yaml, dev-secure-properties.yaml for env-dependent config for the dev environment
- **Endpoints:**
    - Use **http.port** or **https.port** for the name of the configuration property that holds the port at which the API is exposed
    - API endpoint for the one API exposed by an API implementation: /api/v1 for the API itself and /console/v1 for API Console for that API (assuming major version v1)
    - Health check endpoints: Mule apps expose endpoints for Kubernetes-style "probes":
        - For a liveness probe at /alive, returning 200 if alive or 500 if not
        - For a readiness probe at /ready, returning 200 if ready or 500 if not
            - Readiness requires the Mule app to verify that all its dependencies are alive. For API dependencies, this means invoking /alive
        - These should use the same HTTP Listener configuration as the main API endpoints
- **Logging:**

- Merge two or more subsequent loggers into one
- Log-levels:
  - INFO for start and end of externally visible flow, but not those delegated to by APIkit because they use Message Logging API policy
  - INFO before and after every invocation of an external system
  - INFO before raising error
  - DEBUG for internal flows and other log entries

- **Error handling:**
  - Use descriptive error types: APP:INVALID_CHECKIN_RESPONSE
  - By default, define all custom application error types in the APP namespace
    - Omit that namespace in doc:name and the like because it is assumed to be the most common namespace
  - Define all custom application errors that should be communicated to a client (if possible) in the EXT namespace and always supply description for client: EXT:CANT_CHECKIN, EXT:BAD_REQUEST etc.
  - Raise errors only for error conditions, not to control happy-path message processing
  - Reuse common error response creation encapsulated in error-common.xml in apps-commons
    - Specially handles EXT:BAD_REQUEST and all errors in the EXT namespace
- **MUnit tests:**
  - Do not test the APIkit-generated main flow (containing the APIkit Router) and console flow
  - Do not test for validations already performed by APIkit: required parameters, data types of parameters, payload message format, etc.
  - Do not "enable flow sources" for HTTP/S and do not test APIs by invoking them over HTTP/S. Instead, refactor Mule flows that do actual work so that they are easy to test and independent of APIkit, and then test those flows directly
  - In general, do not accept any incoming network communication and do not perform any outgoing network communication from unit tests: MUnit tests must run in isolation in a sandboxed Mule runtime without connectivity or dependency on remote components
  - Use a synchronous logger configuration for tests, with all relevant log levels set to DEBUG
  - Testable Mule flows:
    - Avoid nested XML elements in event processors: they can't be mocked or spied on in MUnit tests. Instead, make them trivial so that they can't fail
- **Functional Monitors:**
  - To be implemented in code and checked in underneath src/test/funmon for each environment the Mule app is deployed to

- **Reconnection strategies:**
  - All global configurations for connectors that are likely to pool or cache connections (Database Connector, JMS Connector, etc.) should define a modest, finite reconnection strategy, such as reconnect 3 times with 1 second intervals. This then applies by default to all operations using this connector config and blocks the operation until completed
    - No other global connector configurations should define a reconnection strategy
    - Most connector operations require retry logic using Until Successful scope and Try scope, irrespective of any reconnection strategy
  - All listeners should define <reconnect-forever /> with a realistic interval locally to the listener itself, thereby overriding any globally defined reconnection strategy
    - The only exception is listeners that cannot realistically re-establish connectivity once lost, such as <http:listener />
- **General:**
  - Avoid absolute file system paths in Mule apps if at all possible; most often, a relative path suffices
    - Relative paths do not work for the sslrootcert in the PostgreSQL JDBC URL
  - Reconnection strategies: Do not fail deployment when backend systems are down, and don't retry forever as this blocks requests when the backend system is down
  - Use Choice router for content-based routing, not to validate dynamic data
  - Use the Validation module validators to validate dynamic data and specifically to enforce invariants, preconditions, postconditions, assertions, expected service responses, and the like
    - Raise descriptive custom application errors from within the validators (by error mapping all errors to such a custom error)

A.4. CloudHub deployments

- **CloudHub deployment names:**
  - tngaa-flights-management-sapi-dev for a Mule app called flights-management-sapi deployed to the dev environment
  - tngaa-flights-management-sapi for the same Mule app deployed to the prod environment

A.5. API Manager

- Use **automated policies** as much as possible, instead of repetitively defining the same policies on every API instance
- Set the **consumer endpoint** of every API instance to the correct endpoint URL of the CloudHub deployment of the API implementation in that environment: https://tngaa-flights-management-sapi-dev-9yj2rh.rajrd4-2.usa-e1.cloudhub.io/api/v1 for Flights Management SAPI in dev

environment

A.6. Functional Monitors

- **Functional Monitors names** and **asset IDs**:
  - flights-management-sapi-dev-funmon for flights-management-sapi in dev
  - flights-management-sapi-prod-funmon for the same Mule app in prod (which exposes its API on https://tngaa-flights-management-sapi-zkvtif.rajrd4-1.usa-e1.cloudhub.io/api/v1)
- Functional Monitors must execute in a **different CloudHub/AWS region** than the Mule app they are monitoring is deployed to: usa-e2 (Ohio) if Mule app runs in usa-e1 (N. Virginia)
- Report violations by posting to appropriate **Slack channel** via webhook
- For any Mule app that supports it, Functional Monitors should invoke the **liveness and readiness endpoints** exposed by that Mule app
- For Experience APIs, Functional Monitors should also invoke a real, read-only business transaction that exercises as many nodes in the application network as possible

A.7. Anypoint MQ

- Consider **encrypting** messages in prod but not in dev and test
- Every "normal" Anypoint MQ queue should be associated with a **Dead Letter Queue**, the TTL of which should be as long as possible (currently 14 days)
- **Anypoint MQ exchange names:**
  - cancelled-flights-exchg-dev for an Anypoint MQ exchange in the dev environment
  - cancelled-flights-exchg for the equivalent Anypoint MQ exchange in the prod environment
    - Note that technically the same name could be used in all environments
- **Anypoint MQ queue and corresponding Dead Letter Queue names:**
  - cancelled-flights-mobile-queue-dev for an Anypoint MQ queue in dev
  - cancelled-flights-mobile-queue for the equivalent Anypoint MQ queue in prod
    - Note that technically the same name could be used in all environments
  - cancelled-flights-mobile-dlq for the Dead Letter Queue belonging to cancelled-flights-mobile-queue (and therefore in prod)

Bibliography

- [Ref1] MuleSoft, "Mule Runtime", https://docs.mulesoft.com/mule-runtime. 2019.
- [Ref2] MuleSoft, "Design Center", https://docs.mulesoft.com/design-center. 2019.
- [Ref3] MuleSoft, "API Manager", https://docs.mulesoft.com/api-manager. 2019.
- [Ref4] MuleSoft, "Anypoint Exchange", https://docs.mulesoft.com/exchange. 2019.
- [Ref5] MuleSoft, "Runtime Manager", https://docs.mulesoft.com/runtime-manager. 2019.
- [Ref6] MuleSoft, "Access Management", https://docs.mulesoft.com/access-management. 2019.
- [Ref7] MuleSoft, "Anypoint Monitoring", https://docs.mulesoft.com/monitoring. 2019.
- [Ref8] MuleSoft, "Anypoint MQ", https://docs.mulesoft.com/mq. 2019.
- [Ref9] MuleSoft, "Anypoint Studio", https://docs.mulesoft.com/studio, 2019.
- [Ref10] MuleSoft, "MUnit", https://docs.mulesoft.com/munit, 2019.
- [Ref11] MuleSoft, "API Functional Monitoring", https://docs.mulesoft.com/api-functional-monitoring, 2019.
- [Ref12] MuleSoft, "APIkit", https://docs.mulesoft.com/apikit, 2019.
- [Ref13] MuleSoft, "Object Store v2", https://docs.mulesoft.com/object-store, 2019.
- [Ref14] Wikipedia, "Monorepo", https://en.wikipedia.org/wiki/Monorepo, 2020.
- [Ref15] M.T. Nygard, *Release It! Second Edition*. Raleigh, NC: Pragmatic Bookshelf, 2018.

Version history

2022-05-17	4.4.0	replace REST connectors with HTTP connector;API Manager CH2 support;upgraded dependencies;minor bug fixes;
2022-11-23	4.3.0	upgraded dependencies;Minor bug fixes;DIYs;
2022-10-31	4.2.0	upgraded dependencies including Studio.7.14.0;Minor bug fixes;
2022-09-07	4.0.1	Minor bug fixes and studentFiles use flat RAMLs;
2022-08-31	4.0.0	CloudHub 2.0 Overhaul;
2022-08-04	3.3.0	upgraded dependencies including Studio.7.13.0;
2022-06-27	3.2.0	upgraded dependencies;bump MMP and update for version 3.6.3+ as no longer needs build parameter -DskipASTValidations;added back all wt and module numbering;
2022-05-20	3.1.0	upgraded dependencies including Studio.7.12.1;Bump MMP and update for version 3.6.0+ to include an additional build parameter -DskipASTValidations;Removed steps where student logs in to AA org, this is due to upcoming MFA. Any client creds required are now directly provided in the manual;
2022-05-04	3.0.0	removed all numbering from walkthroughs and modules except for student file solutions;upgraded dependencies;
2022-03-31	2.4.0	upgraded dependencies including Studio.7.12;
2022-03-09	2.3.0	upgraded dependencies;Exchange Lifecycle support for SNAPSHOT dependencies;Update doc since fixed issue: MULE-19915: Redelivery Policy: infinite loop when the redelivery is exhausted in a source configured with transactions in walkthrough 2-1; Update doc as Windows does not support relative path in -DoutputDirectory arg for arechetype in walkthrough 6-1;
2022-01-10	2.2.0	upgraded dependencies incl. Studio 7.11.1, app.runtime and munit post Log4j resolutions;parent-pom.xml refactored to correct pom.xml naming convention contained in its own directory parent-pom/pom.xml;bom.xml refactored to correct pom.xml naming convention contained in its own directory bom/pom.xml;editorial updates so the courses work for self-paced without instructor demos;
2021-10-05	2.0.0	upgraded dependencies, incl. Studio 7.11.0; Added Mule 4.4 tracing module and With Correlation ID scope section to walkthrough 4-2; Removed json-logger from all walkthroughs including json logger policy is a standard logger policy in walkthrough 6-2; Added Exchange V3 and deployment of apps-commons and parent POMs to walkthrough 1-1 and deployment of module-resilience.xml in walkthrough 6-1; Switch to use Connected Apps instead of credentials in walkthrough 1-1, walkthrough 6-1 and walkthrough 6-2;
2021-08-03	1.9.1	fixed student files

2021-07-30	1.9	explained downgrade of maven-deploy-plugin in walkthrough 6-2; added project/app name to all code snippets; upgraded dependencies, incl. Studio 7.10.0
2021-04-29	1.8	upgraded dependencies, incl. Studio 7.9.0, improved DW expression to extract SGR routes in walkthrough 4-1, introduced choice router in walkthrough 2-3 and boolean var for FMSSAPI validation failures in in walkthrough 2-2 to filter invalid messages and stop AMQ publishing
2021-03-29	1.7	initial public release