Anypoint Platform Development: Production-Ready Development Practices

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 Mule apps.

In this course, students focus on configuration, build, test, and deployment of Mule apps following these themes:

- Automation with Maven-based tooling of build, unit-test and deployment of Mule apps.
- Full lifecycle, multi-environment configuration and deployment.
- Unit-testing with MUnit.
- Security: data protection at rest (in Mule app configuration files and Runtime Manager) and in transit (using HTTPS).
- Monitoring and observability of Mule apps through health checks and operational logging.
- Coding standards and naming conventions.

In covering these topics, this course teaches the foundations for applying a DevOps mindset to Mule app development projects — but it does not cover DevOps or CI/CD in itself, it only lays the necessary foundation.

Course goals and objectives

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

- Understand and execute the fundamental API-related workflows in Anypoint Platform.
- Expose APIs from Mule apps over HTTPS.
- Configure Mule apps succinctly and securely for different deployment environments.
- Automate, configure, and secure Mule app deployments to CloudHub.
- Manage complex Maven dependency relationships of Mule apps.
- Improve operational logging and monitorability of Mule apps.
- Implement simple reusable libraries of Mule app code.
- Unit-test successful and erroneous execution of Mule flows with MUnit.
- Structure MUnit test code and data for reuse and maintainability.
- Run MUnit tests in Studio and as part of Maven builds.

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
MUnit and MUnit Maven plugin	2.3.16
APIkit	1.9.1
HTTP Connector	1.7.3
Secure Configuration Properties module	1.2.5
Secure Properties tool	4.4
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  APDevPRDev*_studentFiles_*.zip
cd     APDevPRDev*_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.

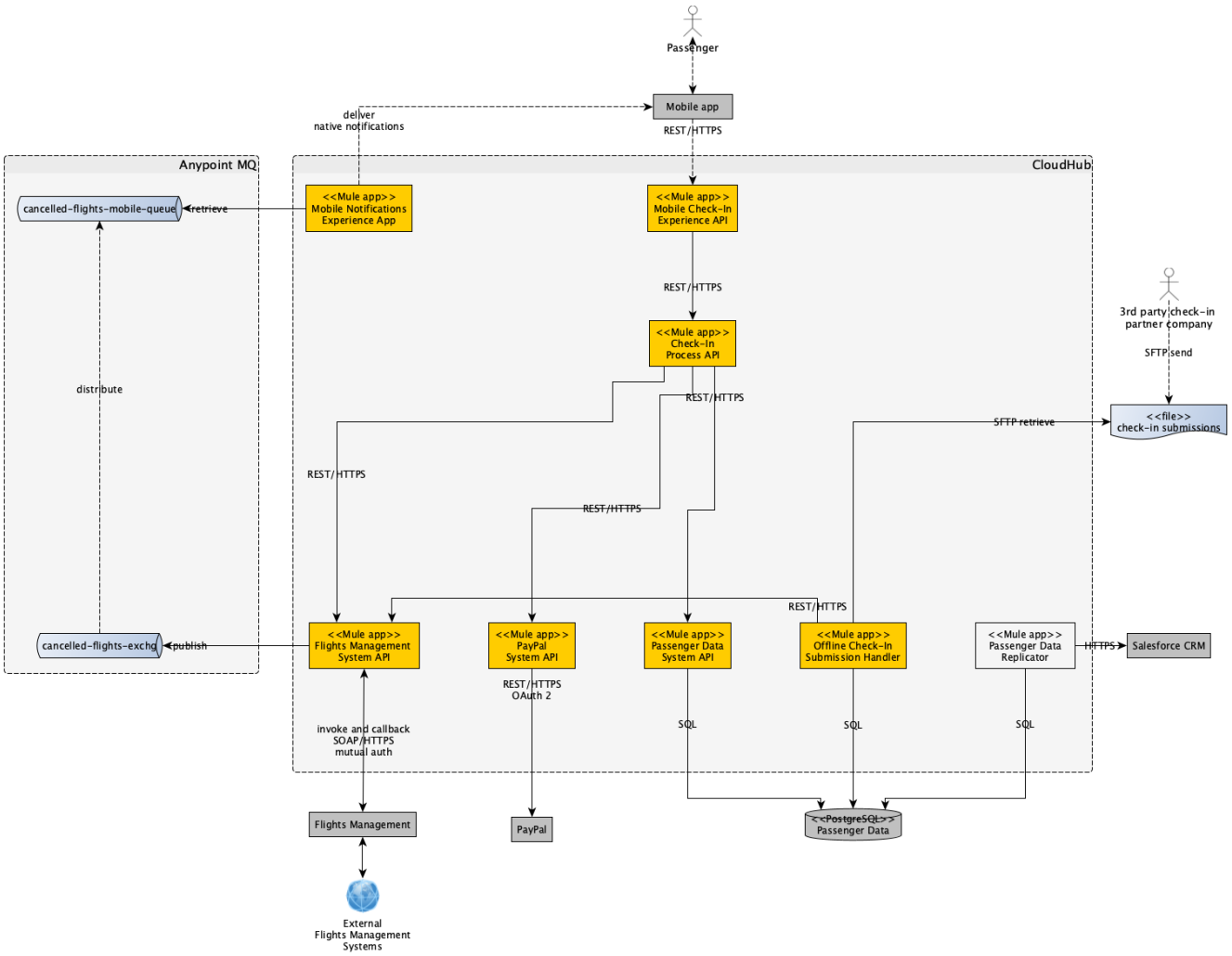

Figure 1. Components and artifacts under design and development in this project (yellow and blue), to be integrated with but not changed (grey), or out-of-scope for this project (transparent). A static view of all interactions between these components and artifacts is shown by arrows pointing from active 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.

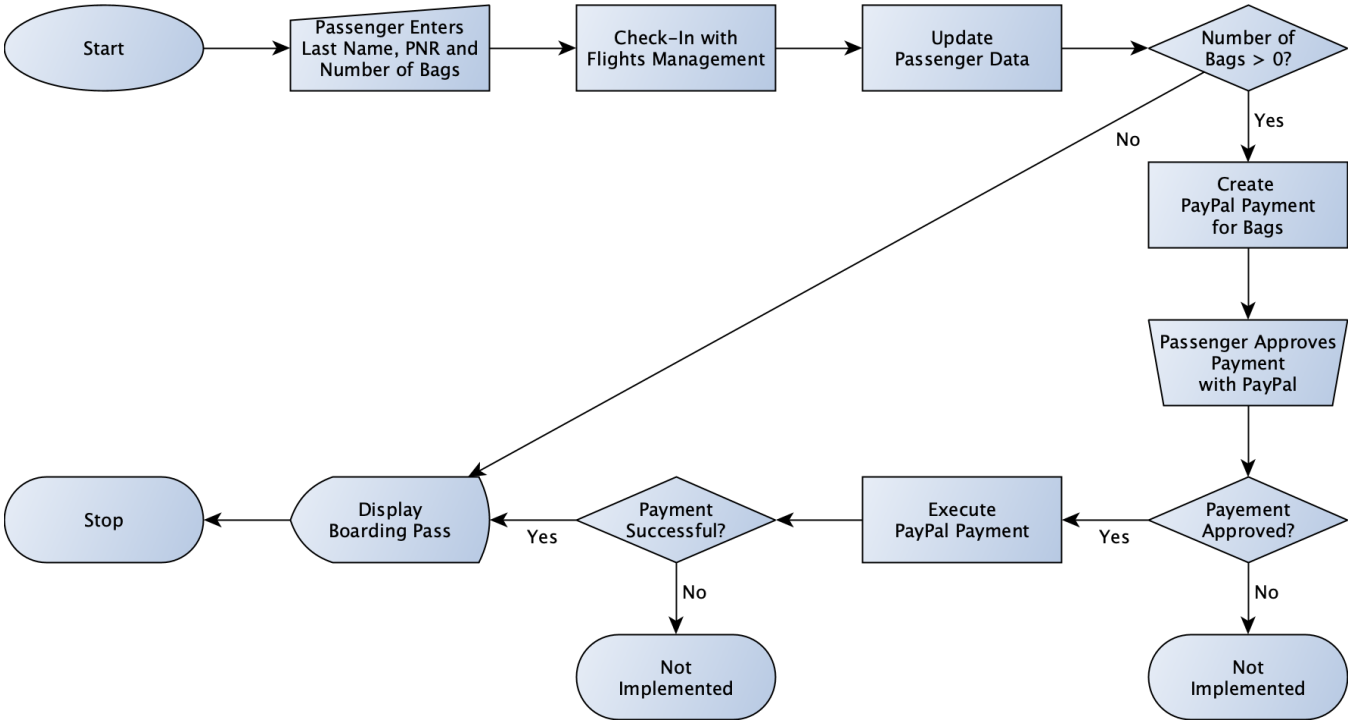

Figure 2. Flowchart visualizing the process by which US1: mobile Check-In is to be realized on a high level. Processing steps and decisions that do not require human interaction are shown as rectangles and rhombi, respectively.

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.

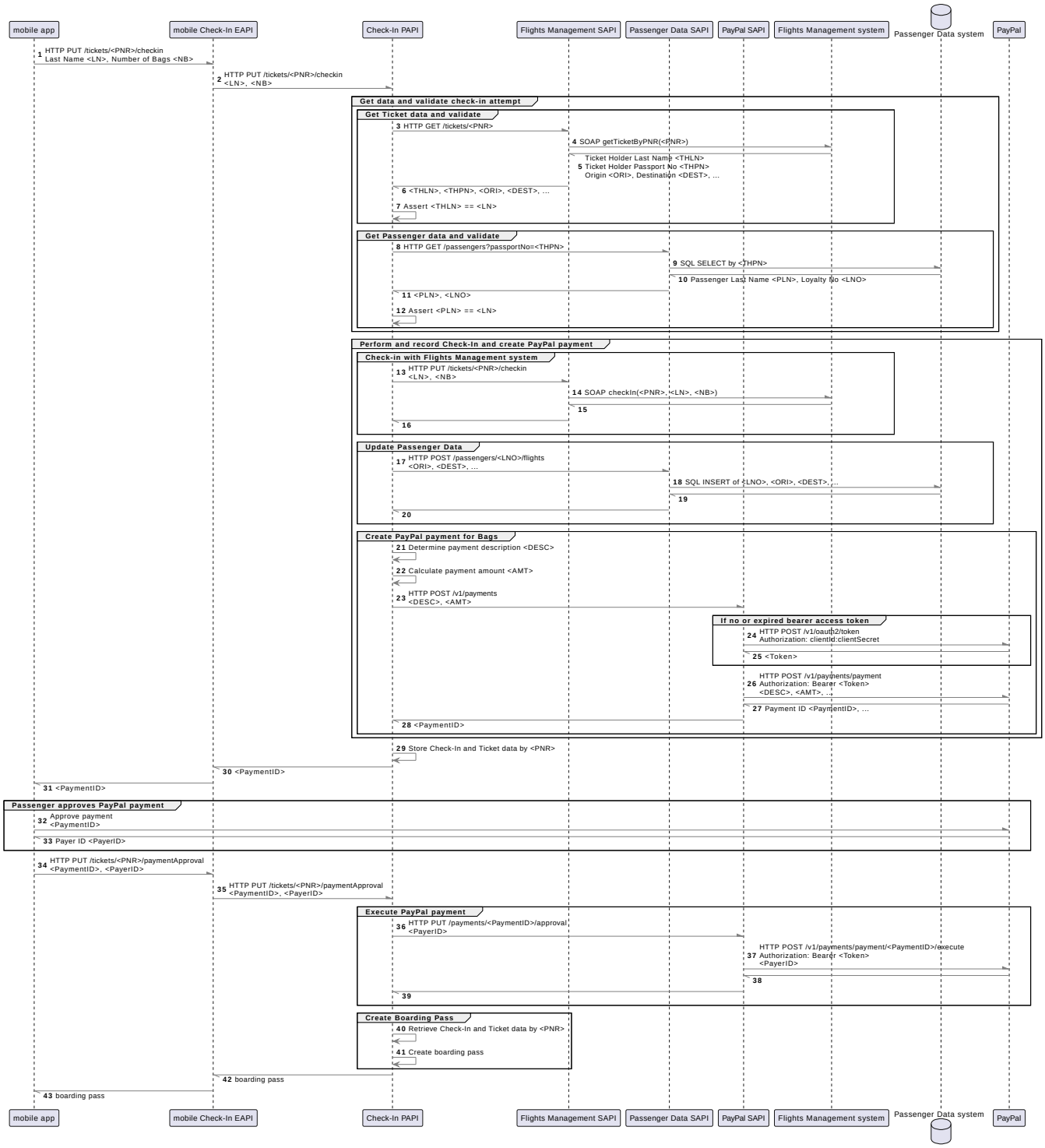

Figure 3. Sequence diagram visualizing the detailed interaction of components in the realization of US1: mobile Check-In. The depicted case is when the number of bags is positive and hence payment needs to be collected.

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.

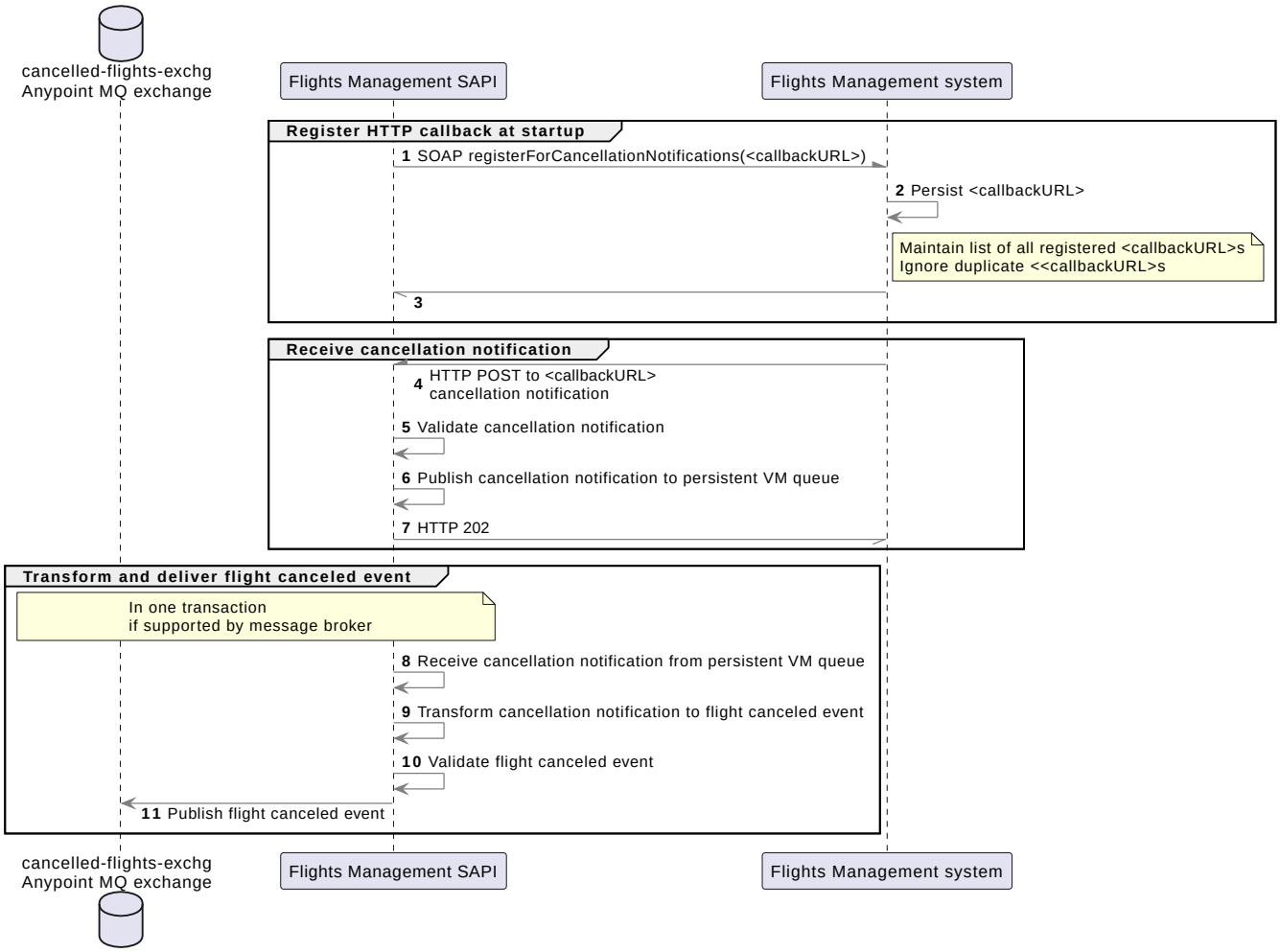

Figure 4. Sequence diagram visualizing the first part of the realization of US2: Flight Cancellation mobile Notifications: the registration of Flights Management SAPI with the Flights Management system to receive cancellation notifications, the delivery and transformation of cancellation notifications 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: Provisioning API-related artifacts

In this module, you use the Anypoint Platform web UI and Studio [Ref9] to create and configure all fundamental artifacts for exposing an API over HTTPS from a Mule app.

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

- Recap the fundamental API-related workflows in Anypoint Platform.
- Export, import, and publish an API specification.
- Manage an API instance in API Manager.
- Implement an API as a Mule app.
- Expose an HTTPS endpoint from a Mule app.
- Register an API implementation using autodiscovery.

Walkthrough 1-1: Export, import, and publish an API specification and manage an API instance in Anypoint Platform

In this walkthrough, you create the basic API-related artifacts for Check-In PAPI in Exchange [Ref4], and API Manager [Ref3]. This is a prerequisite for implementing this API in a later walkthrough. Instead of creating the API specification from scratch you export it from the AnyAirline public developer portal [Ref4] and publish it to Exchange using the API Catalog CLI.

You will:

- Export the Simplified Check-In PAPI API specification in OpenAPI Specification (OAS) format from the AnyAirline public developer portal.
- Modify the API specification and prepare it for publishing.
- Publish the Check-In PAPI API specification to Exchange.
- Create Anypoint Platform environments for dev, test, and prod.
- Apply an automated policy for logging.
- Create an API instance for Check-In PAPI in API Manager.

Export an API specification in OAS format from a public developer portal

In this section, you export the Simplified Check-In PAPI API specification from the AnyAirline public developer portal.

1. **Browse to API specification:** In the AnyAirline public developer portal at **https://anypoint.mulesoft.com/exchange/portals/anyairline/** locate **Simplified Check-In PAPI**.
2. **Download API specification:** From the main page of the Simplified Check-In PAPI entry, download the API specification as **OAS**; this should download a zip file.

   Note: The Simplified Check-In PAPI API specification was originally defined in RAML, so the OAS version is annotated as generated.

Modify the API specification and prepare it for publishing

In this section, you adapt the Check-In PAPI API specification for your organization and prepare to publish the API specification using the API Catalog CLI.

3. **Extract API specification:** Unpack the API specification package to a directory of your choosing:

```
mkdir check-in-papi-spec
unzip simplified-check-in-papi-1.0.2-oas.zip -d check-in-papi-spec
```

4. **Personalize Spec:** In a text editor, change the endpoint so that the CloudHub 2.0 **region** sub-domain is **usa-e2 (Ohio)** and it includes a temporary placeholder for a unique id and shard to the application name so that the URL resembles https://check-in-papi-uniqid.shard.usa-e2.cloudhub.io/api/v1 and change the API title to Check-In PAPI:

```
{
  "swagger": "2.0",
  "info": {
    "title": "Check-In PAPI"
  },
  "host": "check-in-papi-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 in a later walkthrough. You will update this value later.

Note: By convention, the endpoint URL stated in an API specification is understood to refer to the production deployment of the API and its implementation.

Note: CloudHub trial accounts can only deploy to usa-e2 (Ohio).

5. **Verify you have the API Catalog CLI installed:** In a command-line interface, run the API Catalog CLI help command:

```
api-catalog
```

Note: Installing API Catalog CLI is a part of the class setup guide steps. If the command is not recognized, install Git and Node.js, then install API Catalog CLI with the following command: npm install -g api-catalog-cli@latest

*Note: If there are any issues using the API Catalog CLI — create a new API specification project for **Check-In PAPI** using the code editor and language **OAS 2.0 (JSON)**, manually import your modified API specification, and publish to Exchange from Design Center instead.*

6. **Create an Exchange descriptor file:** In a command-line interface, navigate to the API specification directory, and run the API Catalog CLI create descriptor command to create a

named descriptor file in the current path:

```
cd check-in-papi-spec

api-catalog create-descriptor --file catalog.yaml
```

Note: Before you run the commands to publish your APIs, you must create a descriptor file that has the information Exchange needs to catalog the APIs. You can create the descriptor file using the CLI or you can create it manually.

Note: Typically you should create the file using the CLI and if you need to set more options to control how the assets are published, you can update the file manually.

7. **Modify Exchange descriptor file:** In a text editor, update the generated descriptor file, adapting the assetId to **check-in-papi** and adding a versionStrategy that starts the major version at 1.0.0:

```
#%Catalog Descriptor 1.0
projects:
  - main: api.json
    assetId: check-in-papi
    version: 1.0.0
    apiVersion: v1
    versionStrategy: majorIncrease
```

Note: Each API found is added to the apis element in the catalog. Each entry includes the path to the spec file to publish, and the assetId to create in Exchange.

Note: The original assetId is generated from the API title in API specification.

Note: The API Catalog CLI is ideal for when the sheer quantity of APIs are too large to catalog manually or when APIs are not developed using MuleSoft.

Note: There are many other YAML tags used to describe the APIs you want to catalog. Refer to the documentation for a comprehensive list: https://docs.mulesoft.com/exchange/apicat-create-descriptor-file-manually#descriptor-yaml

Note: The versionStrategy chosen here searches for the latest version that matches the version field in the descriptor and increases the major version. If the asset is in the development lifecycle state, the version is increased and the asset stays in development. If the asset is a stable version, a new stable version is published. This can also be configured for minor, patch

and snapshot development versions. More details can be found in the api-catalog documentation: https://docs.mulesoft.com/exchange/apicat-create-descriptor-file-manually

Note: There is also versionStrategyConditions configuration options that enable you to use the same api-catalog publish-asset command in your CI/CD scripts for each of your branches or environments and pass parameters to the command to control the version strategy of the APIs that are published.

8. **Reflect:** Compare the Check-In PAPI API specification to the solution design for US1: mobile Check-In in US1: mobile Check-In.

Publish the API specification to Exchange

In this section, you publish the Check-In PAPI API specification to Exchange using the API Catalog CLI, in your own Anypoint Platform organization using a Connected App.

9. **Create API Catalog Contributor 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 **API Catalog Contributor** and **View Environment** scopes and retrieve its client ID and secret. .
10. **Publish API specification:** Publish the modified Check-In PAPI API specification to Exchange using the API Catalog CLI publish asset command, providing your Anypoint Platform organization ID and the client id the Connected App created previously:

```
cd check-in-papi-spec

api-catalog publish-asset --organization=<insert-your-ap-org-id>
--client_id=<insert-your-ca-client-id> --client_secret
```

Note: When prompted, enter the client secret the Connected App created previously.

Note: You can also use Anypoint Platform credentials for an account of your Anypoint Platform organization with the same permission. However, it is best practice to use a Connected App for security.

Note: You can embed the publish asset command in your automation tools, such as a CI/CD pipeline or custom scripts, to automatically trigger the cataloging of your API assets.

11. **Study Exchange entry:** Inspect the newly created Check-In PAPI Exchange entry, comparing what Exchange displays to the API specification itself.

Create Anypoint Platform environments for the complete software development lifecycle

In this section, you create dev, test, and prod environments in your Anypoint Platform organization.

11. **Rename Sandbox:** In Anypoint Access Management, rename the Sandbox environment to **dev**.
12. **Create environments:** Create environments **test** and **prod** of type Sandbox.

    Note: The names of the environments must match Mule app configurations created in later walkthroughs.

    Note: The prod environment is created as a Sandbox environment to avoid Anypoint Platform vCore restrictions for trial accounts.

Apply an automated policy for logging

In this section, you apply Message Logging as an automated policy to the prod environment.

13. **Apply automated policy:** In the API Manager **prod** environment, apply a new automated policy.

    Note: You work in prod until you introduce environment-specific configuration in a later walkthrough. This is not typical but streamlines the workflow in this and later walkthroughs.

14. **Configure Message Logging:** Select the latest Message Logging API policy, logging all attributes before and after calling the API, and matching the widest possible range of Mule runtime versions:

    ```
    #[attributes]
    ```

Create an API instance in API Manager

In this section, you create an API instance for Check-In PAPI in the prod environment based on the API's Exchange entry.

15. **Manage API:** In the API Manager **prod** environment, manage an API from Exchange.
16. **Configure API instance:** Select the **Check-In PAPI** API specification version you have been working with so far and manage it as a **basic endpoint** deployed to a Mule 4 runtime; remember the **API ID** assigned by API Manager.
17. **Set consumer endpoint:** Copy the **implementation URL** into the API's consumer endpoint.

    Note: The implementation URL was taken from the API specification and therefore is suitable for

the prod environment.

Note: Because no API proxy is involved, the URL of the implementation and the endpoint seen by consumers are identical.

Walkthrough 1-2: Implement an HTTPs API

In the previous walkthrough you have published Check-In PAPI to Exchange, thereby assigning it a well-defined asset version, and you have created an API instance in API Manager.

In this walkthrough, you create a Mule app called check-in-papi that implements this version of Check-In PAPI by importing it from Exchange and registering it with the API instance. The Mule app is kept as simple as possible and does not yet implement real integration logic: it will be tidied-up and completed in later walkthroughs.

You will:

- Create a Mule app that implements the Check-In PAPI API specification published to Exchange.
- Create a self-signed certificate and keystore.
- Expose Check-In PAPI over HTTPS.
- Register the API implementation with an API instance in API Manager using autodiscovery.
- Manually deploy an API implementation to a CloudHub 2.0 shared space.

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/devprd/module01/wt1-2_solution.

Create a Mule app that implements an API specification available in Exchange

In this section, you use Studio to import the Check-In PAPI API specification from Exchange and implement it as a new Mule app check-in-papi. For now you keep almost all the code generated by Studio, such as for exposing the API over HTTP.

1. **Generate API implementation:** In Studio, create a new Mule app project named **check-in-papi** adding a dependency to the Exchange entry for the **Check-In PAPI** version published previously.

   *Note: You must be **logged-in** with an account of your Anypoint Platform organization to be able to search your Exchange.*

   Note: By convention, you chose the API implementation name (check-in-papi) to be identical to the asset ID (artifact ID) of the API specification (Check-In PAPI) it implements.

2. **Study Mule app:** Inspect the Mule app, its dependencies and its Mule flows.

   Note: Studio generates (scaffolds) Mule flows that implement the chosen API by returning hard-

coded sample data extracted from the API specification.

3. **Update Maven coordinates:** Update pom.xml adapting the Maven coordinates to match the **groupId** of your Anypoint Platform organization ID and appending **-app** to the **artifactId**:

   pom.xml of of check-in-papi

   ```
   <project>
     <groupId>your-AP-organization-id</groupId>
     <artifactId>check-in-papi-app</artifactId>
     <version>1.0.0-SNAPSHOT</version>
     ...
   </project>
   ```

   Note: You modify <groupId /> and <artifactId /> as this artifact will later be published to Exchange where the groupID is required to be the Anypoint Platform organization ID to which it is deployed and you have already deployed the API specification with the same artifactId which must be unique.

4. **Update listener paths:** Change the <http:listener /> paths for exposing the **API** and **API Console** to include the API major version:

   api.xml of check-in-papi

   ```
   <flow name="api-main">
     <http:listener config-ref="..."
       path="/api/v1/*">
       ...
     </http:listener>
   </flow>

   <flow name="api-console">
     <http:listener config-ref="..."
       path="/console/v1/*">
       ...
     </http:listener>
   </flow>
   ```

   Note: This follows a convention by which the API endpoint URL path (/api/v1) but not the API implementation name (check-in-papi) identifies the version of the API specification, and only the major version is announced in this way.

   Note: This allows more than one major version of the same API to be exposed by the same API

implementation — if this should be required in the future.

5. **Run and invoke:** Run the Mule app and invoke the API through API Console; confirm that the HTTP responses returned by the API invocations match the payload set in the respective Mule flows.

   Note: In all following walkthroughs, until you properly implement the integration logic of check-in-papi, these are the HTTP responses you will see when an API invocation was successful — so remember them.

Create a self-signed certificate and keystore for exposing an HTTPS endpoint

In this section, you create a self-signed certificate and keystore, which will be needed to expose Check-In PAPI over an HTTPS rather than an HTTP endpoint.

6. **Go to project resources directory:** In a command-line interface, navigate to the **src/main/resources** directory of **check-in-papi**:

```
cd $APDL2WS/check-in-papi/src/main/resources
```

7. **Create keypair in keystore:** Create a standard PKCS12 keystore **check-in-papi.p12** with a new public/private RSA keypair, using shell variables to explain the command configuration:

   Unix variants

```
PASS="mule12345"
APP="check-in-papi"
HOST="localhost"
ALTNAMES="DNS:$HOST,IP:127.0.0.1"
KEYSTORE="$APP.p12"
DNAME="cn=$HOST, ou=Training, o=MuleSoft, c=US"

keytool -v -genkeypair -keyalg RSA -dname "$DNAME" \
  -ext SAN="$ALTNAMES" -validity 365 -alias server \
  -keystore "$KEYSTORE" -storetype pkcs12 -storepass "$PASS"
```

   Windows

```
set PASS="mule12345"
set APP="check-in-papi"
set HOST="localhost"
```

```
set ALTNAMES="DNS:%HOST%,IP:127.0.0.1"
set KEYSTORE="%APP%.p12"
set DNAME="cn=%HOST%, ou=Training, o=MuleSoft, c=US"

keytool -v -genkeypair -keyalg RSA -dname %DNAME% ^
  -ext SAN=%ALTNAMES% -validity 365 -alias server ^
  -keystore %KEYSTORE% -storetype pkcs12 -storepass %PASS%
```

Note: Following PKCS12, both key and keystore share the same password.

Note: Only the keystore name depends on the Mule app name (check-in-papi) — the keypair and keystore contents, such as the distinguished and alternative names, do not include the Mule app name.

Note: This certificate is suitable for local development and deployment to CloudHub, although it uses simplistic host names.

Expose an API over HTTPS

In this section, you replace the Studio-generated HTTP Listener configuration such that Check-In PAPI is exposed only over HTTPS on port 8081, making use of the certificate and keystore created previously.

8. **Switch to HTTPS:** In Studio, change the HTTP Listener configuration to HTTPS on port **8081**, providing a global TLS configuration using the previously created keystore:

   api.xml of check-in-papi

```
<tls:context name="apiTLSContext">
  <tls:key-store type="pkcs12" path="check-in-papi.p12"
    password="mule12345" keyPassword="mule12345" alias="server" />
</tls:context>
<http:listener-config name="...">
  <http:listener-connection host="0.0.0.0"
    protocol="HTTPS" port="8081" tlsContext="apiTLSContext" />
</http:listener-config>
```

   Note: The hardcoded passwords and other configuration will be externalized in a later walkthrough.

   Note: Port 8081 follows CloudHub 2.0 conventions for publicly accessible HTTPS endpoints.

   Note: The TLS configuration belongs to an XML namespace that had so far not been used in this

Mule flow config file. This means that a new XML namespace declaration must be added to the root element of this Mule flow config file (api.xml). Adding this namespace declaration directly in the Mule config flow XML editor is tedious and error-prone in the current version of Studio. By contrast, the visual Mule config flow editor adds XML namespace declarations automatically and transparently. It is therefore best to always use the visual Mule config flow editor to add the first instance of any Mule flow configuration element to any given Mule flow config file, and then, if desired, switch to the Mule config flow XML editor — which will then also perform code-completion as expected.

9. **Run and invoke:** Run the Mule app and invoke the API through API Console; you may have to make your browser accept the self-signed certificate.
10. **Invoke:** In API Console, find the cURL command to invoke the API and use that as the starting point to invoke the API without the help of API Console, such as from a command-line interface, supplying an arbitrary PNR and example JSON request body:

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

Note: The `-i` flag instructs cURL to include HTTP response headers and status code in the output.

Note: The `-k` flag instructs cURL to not validate the certificate of the HTTPS endpoint.

Note: From now on, all invocations of APIs will be specified through cURL commands, but of course all other means of sending the same HTTP requests are equally valid.

Register an API implementation with an API instance using autodiscovery

In this section, you add autodiscovery to check-in-papi such that it automatically registers with the Check-In PAPI API instance created previously in API Manager in the prod environment. You then start check-in-papi in Studio with the client ID and secret Java system properties required by the uplink to the Anypoint Platform control plane.

11. **Add autodiscovery:** In Studio, add a global configuration for API autodiscovery using the **API ID** for the API instance created in API Manager earlier, pointing to the Mule flow with the <http:listener />:

api.xml of check-in-papi

```
<api-gateway:autodiscovery apiId="15678353" flowRef="api-main" />
```

Note: Use your API ID instead of the one shown above.

Note: An API ID implicitly refers to an environment (here: prod) in the Anypoint Platform organization for which it is valid.

Note: Ensure your Mule flow containing the <http:listener /> is indeed called api-main. The flowRef attribute must reference a Mule flow where an <http:listener /> is defined.

12. **Run, invoke, and check log:** Run the Mule app and invoke the API via cURL as before; this should fail with an HTTP 5xx error response and you should see a log entry similar to the following:

```
WARN  2019-04-08 17:55:17,885 [WrapperListener_start_runner]
com.mulesoft.mule.runtime.gw.client.provider.ApiPlatformClientProvider
: Client ID or Client Secret were not provided. API Platform client is
DISABLED.
```

13. **Get client ID and secret:** In **Anypoint Access Management**, navigate to Business Groups and click on the name of your Anypoint Platform organization and navigate to the Settings tab to retrieve its client ID and secret.

    Note: Following the principle of least privilege, it is typically recommended to use the client ID and secret for a specific environment rather than the entire Anypoint Platform organization: the latter gives access to all environments of that Anypoint Platform organization, whereas the former gives access only to that one environment. Nevertheless, to avoid confusion through multiple pairs of client ID and secret, this course will continue using organization-wide client ID and secret.

14. **Update run config:** In Studio, stop the Mule app and change its run configuration by adding the following **VM arguments** for client ID and secret:

```
-M-Danypoint.platform.client_id=<insert-your-client-id>
-M-Danypoint.platform.client_secret=<insert-your-client-secret>
```

    Note: This sets Java system properties with the given names and values.

15. **Run:** Run the Mule app; a **log entry** similar to the following should appear:

```
INFO  2019-04-08 18:39:02,120 [agw-policy-set-deployment.01]
com.mulesoft.mule.runtime.gw.policies.lifecyle.GateKeeper: API
ApiKey{id='15678353'} is now unblocked (available).
```

16. **Invoke:** Invoke the API via cURL as before; this should return a HTTP 200 OK response with whatever response body was extracted by APIkit from the API specification.
17. **Check log:** Inspect the log entries created by the **Message Logging** automated policy.
18. **Check API instance:** In **API Manager**, the status of the corresponding API instance should now be active/green.
19. **Stop:** Stop the Mule app.

Manually deploy an API implementation to a CloudHub 2.0 shared space

In this section, you use the Runtime Manager web UI to deploy check-in-papi to the CloudHub prod environment using a hostname that matches the endpoint URL in the Check-In PAPI API specification. You set the CloudHub properties for client ID and secret required by the uplink to the Anypoint Platform control plane.

20. **Locate deployable archive:** Locate the check-in-papi deployable archive in the **target** sub-directory of the check-in-papi base directory.
21. **Configure deployment to CloudHub 2.0 prod:** In **Runtime Manager**, launch the UI for deploying the check-in-papi archive to the prod environment, selecting the **CloudHub 2.0 US East Ohio Shared Space** as a deployment target and setting the application name to **check-in-papi**.

    Note: When deploying to the CloudHub 2.0 deployment target, the domain for the public endpoint is always app-name-uniq-id.shard.region.cloudhub.io.

    Note: To ensure that names are unique and avoid domain conflicts, CloudHub 2.0 adds a six-character unique id to the application name that you specify in the public endpoint URL.

    Note: The application name identifies your application not only in Runtime Manager but also in the public cloudhub.io domain. For example, an application named myapp is accessible at http://myapp-uniq-id.shard.usa-w2.cloudhub.io.

22. **Configure Last-mile security:** In the **Ingress** tab, check the Last-Mile Security option.

    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.

23. **Configure and trigger deployment:** In the **Properties tab**, enter client ID and secret of your Anypoint Platform organization as before, but in plain properties key=value format, then trigger the actual deployment:

```
anypoint.platform.client_id=<insert-your-client-id>
anypoint.platform.client_secret=<insert-your-client-secret>
```

Note: This sets Java system properties with the given names and values.

24. **Wait and observe:** Observe the Mule app's logs and wait for it to completely start up.
25. **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.
26. **Invoke:** Invoke the API via cURL using this endpoint URL; this should return a HTTP 200 OK response:

```
curl -i -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
:\"Smith\",\"numBags\":2}" https://check-in-papi-uniqid.shard.usa-
e2.cloudhub.io/api/v1/tickets/PNR123/checkin
```

Note: The `-k` cURL option is no longer needed as the certificate exposed by the HTTPS endpoint is that of the CloudHub Shared Space Load Balancer. The CloudHub Shared Space Load Balancer in turn invokes the API endpoint exposed by the Mule app.

Note: To test Runtime Fabric and CloudHub 2.0 applications via their API Console, ensure you append a trailing "/" to the console URL, for example: https://check-in-papi-uniqid.shard.usa-e2.cloudhub.io/api/v1/console/. Without the trailing "/", it will be redirected to the Runtime Fabric local service DNS.

27. **Check log:** Inspect the log entries created by the Message Logging automated policy.
28. **Check API instance:** In **API Manager**, the status of the corresponding API instance should (again) be active/green.
29. **Set consumer and implementation endpoints:** in API Manager, update the implementation URL and consumer endpoint to match your generated fully qualified domain name.

Note: These endpoints are optional and are used for informational purposes to let clients know

how to interact with your API, including surfacing these endpoints in Exchange.

Module 2: Applying basic software engineering principles

In this module, you apply these essential software engineering principles to a Mule app and its Maven build: repeatability through convention and automation, security through encryption, separation of concerns, and non-redundancy of code and configuration.

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

- Apply and follow coding conventions.
- Securely parameterize a Mule app and its Maven build for different runtime environments.
- Deploy to CloudHub from a Maven build.

Walkthrough 2-1: Apply and follow coding conventions

The code for check-in-papi at this point is essentially that generated by Studio. There are two fundamentally different approaches for dealing with generated code: One approach is to keep generated code clearly separate from hand-written code, making as few changes as possible to generated code, and applying coding conventions only to hand-written code. The other approach is to see code generation as a shortcut to writing "normal" code, and consequently treat generated code the same way as hand-written code, applying the same coding conventions to all code. It is this latter approach that AnyAirline follows and that you start enforcing in this walkthrough.

In this walkthrough, you establish the following basic coding conventions (a sub-set of those given in *Coding conventions*), apply them to check-in-papi, and lay the foundation for following them in all future development:

1. Don’t repeat yourself: avoid duplication.
2. Abbreviate when space is constrained: SAPI=System API, PAPI=Process API, EAPI=Experience API.
3. For Mule apps:
   a. Studio project name = Maven artifact ID = Maven project name, for example check-in-papi.
   b. Formatting: standard Eclipse XML editor settings with width 140 characters and do not format comments.
   c. Flows and global error handlers: for example check-in-by-pnr or api-error-handler.
   d. Global element names: for example apiHttpListenerConfig.
   e. Mule flow config files: global.xml, api.xml, main.xml, error.xml, health.xml.

You will add to these basic coding conventions in later walkthroughs as and when needed (see *Coding conventions*).

For conciseness, all doc:name attributes are omitted from Mule flow code shown in this document.

You will:

- Apply these coding conventions retroactively to check-in-papi.
- Deploy a reusable template to 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/devprd/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/devprd/module01/wt1-2_solution.

Apply coding conventions to a Studio-generated API implementation

In this section, you refactor check-in-papi: rename global configuration elements and move them to global.xml, rename global error handlers and move them to error.xml, extract all main integration logic to flows in main.xml, and separate flows related to exposing Check-In PAPI in api.xml.

1. **Rename Mule flow config file:** In Studio, rename your only Mule flow config file **api.xml**.
2. **Rename global elements:** In **api.xml**, change all global element names to follow the naming convention:

   api.xml of check-in-papi

   ```
   <tls:context name="apiTLSContext" />
   <http:listener-config name="apiHttpListenerConfig" />
   <apikit:config name="apiConfig" />
   ```

   Note: It is best to do this be search-replace so as not to miss any references to the old names, for example to the <apikit:config /> name from flow names that process API invocations.

3. **Move global elements:** Move all global elements from api.xml to **global.xml**; only Mule flows remain in api.xml.

   Note: The global.xml file only contains global configuration elements that are not error handlers.

   Note: To avoid XML namespace issues entirely, consider copying api.xml to global.xml and then deleting unwanted XML elements from both files.

4. **Extract and move error handlers:** In the **Mule config flow XML editor**, from **api.xml** extract the two local error handlers in the Studio-generated Mule flows **api-main** and **api-console** into two global error handlers called **api-error-handler** and **api-console-error-handler**, respectively, in **error.xml**:

   error.xml of check-in-papi

   ```
   <error-handler name="api-error-handler">
   ```

```
  ...
</error-handler>

<error-handler name="api-console-error-handler">
  ...
</error-handler>
```

api.xml of check-in-papi

```
<flow name="api-main">
  ...
  <error-handler ref="api-error-handler" />
</flow>

<flow name="api-console">
  ...
  <error-handler ref="api-console-error-handler" />
</flow>
```

Note: The error.xml file only contains global error handlers.

Note: The error handler references are currently neither visible nor editable in the visual Mule config flow editor.

Note: To avoid XML namespace issues entirely, consider copying api.xml to error.xml and then deleting unwanted XML elements from both files.

Note: Currently, the APIkit-generated api-error-handler does not contain a catch-all error handler: this is a shortcoming that you will correct in a later walkthrough.

5. **Extract and move main flows:** In **api.xml**, using the visual Mule config flow editor, locate the two Mule flows generated by APIkit to process the two HTTP requests and use the **extract to flow** refactoring to extract (only) the setting of the payload (the last Transform Message component in each flow) to two new private flows **check-in-by-pnr** and **payment-approval-by-pnr** in **main.xml**, such that they are called by <flow-ref /> elements from the request processing flows in api.xml; you will extend these extracted flows in future walkthroughs to perform the main integration logic of Check-In PAPI:

   api.xml of check-in-papi

```
<flow name="put:...">
  ...
  <flow-ref name="check-in-by-pnr" />
```

```
</flow>

<flow name="put:...">
  ...
  <flow-ref name="payment-approval-by-pnr" />
</flow>
```

main.xml of check-in-papi

```
<flow name="check-in-by-pnr">
  <ee:transform>
    <ee:message>
      <ee:set-payload>...</ee:set-payload>
    </ee:message>
  </ee:transform>
</flow>

<flow name="payment-approval-by-pnr">
  <ee:transform>
    <ee:message>
      <ee:set-payload>...</ee:set-payload>
    </ee:message>
  </ee:transform>
</flow>
```

Note: The Mule flows in main.xml are now essentially pure integration logic independent of the fact that this logic is exposed via an API, whereas the latter aspect is handled by the Mule flows in api.xml.

Note: It is good practice to set the doc:name attribute of <flow-ref /> elements to the name of the flow being invoked (see Coding conventions). But this document omits all doc:name attributes for conciseness.

Note: Each Mule flow is encapsulated and can be referenced by many calling Mule flows. Therefore it is necessary to define the interface of each invokable Mule flow via its metadata. This interface can be left implicit, set manually on each Mule flow, or set via the Metadata assistant when using the extract to flow option in the visual Mule config flow editor.

Note: The Metadata assistant automatically detects the actual metadata of your existing Mule flow and enables you to propagate the existing metadata to the Mule flow you are referencing.

Note: Once the metadata is defined for the child Mule flow, it can no longer be automatically updated if you modify the actual metadata in the calling parent Mule flow.

Note: Using the Metadata assistant encourages a coding style of developing large self-contained Mule flows and refactoring over time into smaller, reusable Mule flows. This coding style is supplemented by unit testing to make sure existing code doesn't break when later refactored. You will learn how to unit test Mule apps in later walkthroughs.

6. **Run:** Confirm that the Mule app still starts up without errors in Studio.

Deploy a reusable template to Exchange

In this section, you deploy check-in-papi as a reusable project template to Exchange — allowing other developers to use this canonical project to bootstrap other API projects in future.

7. **Bootstrap check-in-papi-template:** Copy **check-in-papi** to **check-in-papi-template** in the workspace directory:

```
cd $APDL2WS
cp -R check-in-papi check-in-papi-template
```

8. **Import into Studio:** In Studio, import the check-in-papi-template project as a Mule app, without copying it into the workspace, because it is already there; confirm that the import succeeded.
9. **Modify Maven coordinates:** Update pom.xml adapting the **artifactId** and **name** to the new project name: check-in-papi-template and increasing the **version** to a non-SNAPSHOT release version.

   pom.xml of check-in-papi-template

```
<project>
  <artifactId>check-in-papi-template</artifactId>
  <version>1.0.0</version>
  <name>check-in-papi-template</name>
  ...
</project>
```

   Note: Although Exchange supports SNAPSHOT versions for template projects, at the time of writing Studio does not. You can however use the Exchange Mule Maven plugin which will be discussed in a later walkthrough.

10. **Create Category metadata**: In **Exchange**, navigate to **Settings** and create a new category: **Tier** with values for each API layer: **System Tier**, **Process Tier** and **Experience Tier**.

    Note: Categories allow you to organize Exchange assets into groups to improve asset browsing

and discovery.

11. **Deploy template to Exchange:** In Studio, from the Anypoint menu: **Publish to Exchange**, choosing **Template** as the project type and click **Next**.
12. **Set metadata**: From the next screen, add a tag: **check-in**, and add the **Process Tier** category created previously.

    *Note: You must be **logged-in** with an account of your Anypoint Platform organization to be able to deploy to your Exchange.*

13. **Study pom.xml:** Notice the addition of the properties for the various metadata:

    pom.xml of check-in-papi-template

    ```
    <project>
      <artifactId>check-in-papi-template</artifactId>
      <version>1.0.0</version>
      <packaging>mule-application</packaging>
      <name>check-in-papi-template</name>
      ...
      <properties>
        <tags>check-in</tags>
        <categories>[{"key":"Tier","value":"Process Tier"}]</categories>
      </properties>
    </project>
    ```

    Note: You will look more in-detail at deploying various asset types to Exchange with Maven in a later walkthrough.

14. **Study Exchange asset:** Navigate to your Exchange, download the asset jar for check-in-papi-template, and extract the contents. Study the pom.xml file and notice the **Mule Maven plugin configuration** is configured to create an artifact with <classifier /> **mule-application-template**:

    pom.xml of check-in-papi-template

    ```
    <project>
      <artifactId>check-in-papi-template</artifactId>
      <version>1.0.0-SNAPSHOT</version>
      <packaging>mule-application</packaging>
      <name>check-in-papi-template</name>

      ...
    ```

```
  <plugin>
     <groupId>org.mule.tools.maven</groupId>
     <artifactId>mule-maven-plugin</artifactId>
     <version>${mule.maven.plugin.version}</version>
     <extensions>true</extensions>
     <configuration>
       <classifier>mule-application-template</classifier>
     </configuration>
   </plugin>
 </project>
```

Note: This configuration is not automatically added your pom.xml, you will need to manually add this configuration if you intend to deploy the template via Maven outside of Studio.

Note: You will look more in-detail at deploying various asset types to Exchange with Maven in a later walkthrough.

Walkthrough 2-2: Remove redundancy

When creating the check-in-papi Mule app in the previous walkthrough, you have hard-coded all configuration values directly in the Mule app source code. Furthermore, the Mule app utilizes a Maven build configuration generated by Studio, which also contains configuration values that partially overlap with values hard-coded in the Mule app source code.

In this walkthrough, you identify all configuration values of check-in-papi and its Maven build and extract them into suitable configuration files of the Maven build and in check-in-papi to reduce redundancy and improve maintainability and reuse.

You will:

- Extract Mule app configuration values into a properties file.
- Remove configuration redundancy using Maven resource filtering.
- Remove build redundancy and increase build reproducibility by introducing a parent POM.
- Centralize Maven dependency and plugin versions in a BOM.

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/devprd/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/devprd/module02/wt2-1_solution.

Extract Mule app configuration values into a properties file

In this section, you extract all configuration values from the check-in-papi Mule app code into a single configuration file properties.yaml.

1. **Identify config values:** In Studio, study the Mule flow config files of check-in-papi and identify all configuration values.

   Note: Consider all values that might change over time or that might differ between environments to be configuration values.

2. **Study APIkit api attribute:** Identify the constituent parts of the <apikit:config /> api attribute and determine their **meaning**:

global.xml of check-in-papi

```
<apikit:config api="resource::6ebe1206-ddb1-4f2a-b06e-
dca3244ebbad:check-in-papi:1.0.0:oas:zip:api.json"
```

Note: Some of these values will be different for you. This remains true for the entire walkthrough and will not be repeated.

3. **Copy config values to props file:** Create a new file **properties.yaml** in **src/main/resources** and add all configuration values, suitably named and organized:

   properties.yaml of check-in-papi

```
api:
  groupId:      "6ebe1206-ddb1-4f2a-b06e-dca3244ebbad"
  artifactId:   "check-in-papi"
  version:      "1.0.0"
  spec:
"resource::${api.groupId}:${api.artifactId}:${api.version}:oas:zip:api
.json"
  majorVersion: "v1"
  id:           "15678353"

https:
  port: "8081"

tls.keystore:
  type:        "pkcs12"
  path:        "check-in-papi.p12"
  alias:       "server"
  password:    "mule12345"
  keyPassword: "mule12345"
```

   Note: Break-down the complicated api.spec value into its constituent parts, as shown here.

4. **Read props file:** Near the top of **global.xml**, read **properties.yaml**:

   global.xml of check-in-papi

```
<configuration-properties file="properties.yaml" />
```

5. **Reference config props:** Replace all configuration values with references to the corresponding properties:

global.xml of check-in-papi

```
<tls:context name="apiTLSContext">
  <tls:key-store type="${tls.keystore.type}"
path="${tls.keystore.path}"
    password="${tls.keystore.password}"
    keyPassword="${tls.keystore.keyPassword}"
    alias="${tls.keystore.alias}" />
</tls:context>

<http:listener-config name="apiHttpListenerConfig">
  <http:listener-connection host="0.0.0.0"
    protocol="HTTPS" port="${https.port}" tlsContext="apiTLSContext"
/>
</http:listener-config>

<apikit:config name="apiConfig" api="${api.spec}"
  outboundHeadersMapName="outboundHeaders"
  httpStatusVarName="httpStatus" />

<api-gateway:autodiscovery apiId="${api.id}" flowRef="api-main" />
```

api.xml of check-in-papi

```
<flow name="api-main">
  <http:listener config-ref="apiHttpListenerConfig"
    path="/api/${api.majorVersion}/*">
    ...
  </http:listener>
  ...
</flow>

<flow name="api-console">
  <http:listener config-ref="apiHttpListenerConfig"
    path="/console/${api.majorVersion}/*">
    ...
  </http:listener>
  ...
</flow>
```

Note: You use the reserved property https.port for the port to expose the API on which is overriden by CloudHub. The value is assigned automatically by the platform services when deployed to CloudHub. CloudHub 2.0 assigns 8081 as the value, but you can use a different

value locally if needed.

6. **Run and invoke:** Run the Mule app from Studio and invoke the API as before; this should return a HTTP 200 OK response and the hard-coded response body:

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

Remove configuration redundancy using Maven resource filtering

In this section, you introduce Maven resource filtering to check-in-papi such that configuration values that must be defined in the Maven POM do not have to be repeated in the Mule app code.

Resource filtering is a feature of the standard Maven Resources plugin. A more appropriate name would be "resource preprocessing", because it consists in references to Maven property names in resource files being replaced with the values of these properties as defined in the Maven POM.

7. **Detect build and configuration repetition:** Inspect **pom.xml** in general and in particular the Maven dependency on the Check-In PAPI API specification in OAS format; note the duplication with **properties.yaml**:

pom.xml of check-in-papi

```
<dependency>
  <groupId>6ebe1206-ddb1-4f2a-b06e-dca3244ebbad</groupId>
  <artifactId>check-in-papi-app</artifactId>
  <version>1.0.0</version>
  <classifier>oas</classifier>
  <type>zip</type>
</dependency>
```

properties.yaml

```
api:
  groupId:      "6ebe1206-ddb1-4f2a-b06e-dca3244ebbad"
  artifactId:   "check-in-papi"
  version:      "1.0.0"
```

Note: This duplication means that every change to the API specification being implemented by

this Mule app requires a synchronized change to pom.xml and properties.yaml.

8. **Configure Maven Resources plugin resource filtering:** In pom.xml, add to the **<build />** element configuration for the standard Maven Resources plugin to perform Maven resource filtering of files in **src/main/resources** and **src/test/resources**, excluding known binary files:

   pom.xml of check-in-papi

```
<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
    </resource>
  </resources>
  <testResources>
    <testResource>
      <directory>src/test/resources</directory>
      <filtering>true</filtering>
    </testResource>
  </testResources>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-resources-plugin</artifactId>
      <version>3.3.1</version>
      <configuration>
        <nonFilteredFileExtensions>
          <nonFilteredFileExtension>p12</nonFilteredFileExtension>
          <nonFilteredFileExtension>crt</nonFilteredFileExtension>
          <nonFilteredFileExtension>pem</nonFilteredFileExtension>
        </nonFilteredFileExtensions>
      </configuration>
    </plugin>
    ...
  </plugins>
<build>
```

   Note: Merge the above into the existing <build /> element.

   Note: If binary files were not excluded they would garbled by the filtering process.

   Note: The Maven Resources plugin is a standard Maven plugin and unrelated to MuleSoft.

9. **Add Maven props:** Add to the **<properties />** element the configuration properties common to pom.xml and properties.yaml, copying their values from the <dependencies /> section:

   pom.xml of check-in-papi

   ```
   <api.groupId>6ebe1206-ddb1-4f2a-b06e-dca3244ebbad</api.groupId>
   <api.artifactId>check-in-papi</api.artifactId>
   <api.version>1.0.0</api.version>
   ```

   Note: Maven resource filtering replaces references to Maven properties in resource files with their values defined in the Maven build configuration, such as in pom.xml.

10. **Reference props in build:** Change the Maven dependency on the Check-In PAPI to use these properties:

    pom.xml of check-in-papi

    ```
    <dependency>
      <groupId>${api.groupId}</groupId>
      <artifactId>${api.artifactId}</artifactId>
      <version>${api.version}</version>
      <classifier>oas</classifier>
      <type>zip</type>
    </dependency>
    ```

11. **Reference props in config:** In **properties.yaml**, similarly replace the values with references to these Maven properties:

    properties.yaml of check-in-papi

    ```
    api:
      groupId:      "${api.groupId}"
      artifactId:   "${api.artifactId}"
      version:      "${api.version}"
    ```

    Note: Maven resource filtering replaces these property references with their values during the process-resources and process-test-resources life-cycle phases of the Maven build.

12. **Run and invoke:** Run the Mule app from Studio and invoke the API as before; this should return an HTTP 200 OK response.
13. **Study filtered props file:** Compare the filtered (pre-processed) contents of **properties.yaml** in **target/classes** to the source in src/main/resources.

Note: If target/classes does not exist then briefly run the Mule app in Studio, as this performs a Maven build. Or you can simply run a Maven build from a command-line interface, as will be done in later walkthroughs.

Remove build redundancy and increase build reproducibility by introducing a parent POM

In this section, you extract all Maven build configuration that is not specific to check-in-papi from pom.xml into a reusable parent POM within its own directory parent-pom/pom.xml. You then transform pom.xml into a child POM of parent-pom/pom.xml.

14. **Identify reusable build config:** In **pom.xml** and identify the configuration that is likely to be the same across related Mule apps.

    Note: This includes the versions of Maven dependencies and Maven plugins, the configuration of plugins like the Mule Maven plugin and Maven Resources plugin, the resource filtering configuration, and the declaration (location) of Maven repositories from which to download dependencies.

15. **Bootstrap parent POM:** Copy **pom.xml** to **parent-pom/pom.xml** in the parent directory:

    ```
    cd $APDL2WS
    mkdir parent-pom
    cp check-in-papi/pom.xml parent-pom/pom.xml
    ```

    Note: Typically, parent POMs are located in the parent directory of a Maven project. Because a parent POM is not related to a specific project, this approach is only viable if a monorepository is used, that is, if all related projects are located in the same source code (Git) repository [Ref14]. Then parent POMs can also be located in that monorepo, in a parent directory of the Maven projects. This is the approach taken by AnyAirline.

    Note: If every project is maintained in its own repository — the opposite of a monorepo, sometimes called polyrepository or multirepository — then parent POMs are typically also maintained in distinct repos. Because the relative filesystem location of these parent POMs to the Maven projects that use them can then no longer be guaranteed, parent POMs must in this case be installed in a Maven repository and loaded from there.

16. **Update dependencies:** In **parent-pom/pom.xml** update if necessary the versions of these dependencies: **Mule Maven plugin** to 3.8.2, APIkit to 1.9.1, and HTTP Connector to 1.7.3.
17. **Turn into parent POM:** Change parent-pom/pom.xml into a parent POM by adapting its Maven coordinates including setting the groupId to your Anypoint Platform organization ID and retaining only build configuration common to similar Mule apps, wrapping <dependencies /> into

<dependencyManagement />, and wrapping <plugins /> into **<pluginManagement />**, and also introducing an additional **property** for the **Mule runtime version**:

parent-pom/pom.xml (outline)

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>your-AP-organization-id</groupId>
  <artifactId>solutions-parent-pom</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>solutions-parent-pom</name>

  <properties>
    ...
    <app.runtime.semver>4.4.0</app.runtime.semver>
    <app.runtime>4.4.0-20230522</app.runtime>
    ...
  </properties>

  <build>
    <resources>...</resources>
    <testResources>...</testResources>
    <pluginManagement>
      <plugins>...</plugins>
    </pluginManagement>
  </build>

  <dependencyManagement>
    <dependencies>...</dependencies>
  </dependencyManagement>

  <repositories>...</repositories>
  <pluginRepositories>...</pluginRepositories>
</project>
```

parent-pom/pom.xml (more detail)

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>your-AP-organization-id</groupId>
  <artifactId>solutions-parent-pom</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>pom</packaging>
```

```
  <name>solutions-parent-pom</name>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>
      UTF-8
    </project.reporting.outputEncoding>
    <app.runtime.semver>4.4.0</app.runtime.semver>
    <app.runtime>4.4.0-20230522</app.runtime>
    <mule.maven.plugin.version>3.8.2</mule.maven.plugin.version>
  </properties>

  <build>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
      </resource>
    </resources>
    <testResources>
      <testResource>
        <directory>src/test/resources</directory>
        <filtering>true</filtering>
      </testResource>
    </testResources>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-resources-plugin</artifactId>
          <version>3.3.1</version>
          <configuration>
            <nonFilteredFileExtensions>
              <nonFilteredFileExtension>p12</nonFilteredFileExtension>
              <nonFilteredFileExtension>crt</nonFilteredFileExtension>
              <nonFilteredFileExtension>pem</nonFilteredFileExtension>
            </nonFilteredFileExtensions>
          </configuration>
        </plugin>
        <plugin>
          <groupId>org.mule.tools.maven</groupId>
          <artifactId>mule-maven-plugin</artifactId>
          <version>${mule.maven.plugin.version}</version>
          <extensions>true</extensions>
          <configuration>
```

```
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.mule.connectors</groupId>
        <artifactId>mule-http-connector</artifactId>
        <version>1.7.3</version>
        <classifier>mule-plugin</classifier>
      </dependency>
      <dependency>
        <groupId>org.mule.modules</groupId>
        <artifactId>mule-apikit-module</artifactId>
        <version>1.9.1</version>
        <classifier>mule-plugin</classifier>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <repositories>
    <repository>
      <id>anypoint-exchange-v3</id>
      ...
    </repository>
    <repository>
      <id>mulesoft-releases</id>
      ...
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>mulesoft-releases</id>
      ...
    </pluginRepository>
  </pluginRepositories>
</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.

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 does not support dynamic parameters, such as setting the asset version to 1.0.${revision}.

Note: In rare circumstances, such as when a hotfix to a Mule runtime is released, there is a need to distinguish between the full version of the runtime (4.4.0-20230522) and the simplified version of that same runtime (4.4.0, which follows semantic versioning (semver) rules). You anticipate this need by defining two separate Maven properties for the Mule runtime version.

Note: By wrapping <dependencies /> into <dependencyManagement /> you default the version of these dependencies for all child projects, while allowing child projects to bring in these dependencies as needed.

Note: By defining the Mule Maven plugin inside <pluginManagement /> you default the version and configuration of this plugin for all child projects, while allowing child projects to execute this plugins as needed. Mule apps will always want to execute this plugin, while other Maven projects will not.

Note: By defining the Maven Resources plugin inside <pluginManagement /> you default the version and configuration of this plugin for all child projects. But this is a standard Maven plugin that is always executed during a normal Maven build, so child projects do not have to explicitly request execution of this plugin, like they have to do for the Mule Maven plugin.

Note: This parent POM does not impose any actual dependencies or plugin executions on its child POMs.

Note: Do not manage the dependency on check-in-papi in the parent POM: there is currently no value in doing so. Shortly you will simply define the un-managed dependency in check-in-papi's pom.xml.

Note: Do not manage the dependency on the Sockets Connector in the parent POM, as this is added automatically as a transitive dependency of the HTTP Connector. Shortly you will also remove the Sockets Connector from check-in-papi's pom.xml.

18. **Turn POM into child POM:** Change **pom.xml** into a child POM by defining **parent-pom/pom.xml** as its parent POM and removing all configuration inherited from that parent POM, such as dependency versions:

pom.xml of check-in-papi

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>your-AP-organization-id</groupId>
    <artifactId>solutions-parent-pom</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <relativePath>../parent-pom/pom.xml</relativePath>
  </parent>
  <groupId>a63e6d25-8aaf-4512-b36d-d91b90a55c4a</groupId>
  <artifactId>check-in-papi-app</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>mule-application</packaging>
  <name>check-in-papi</name>

  <properties>
    <api.groupId>6ebe1206-ddb1-4f2a-b06e-dca3244ebbad</api.groupId>
    <api.artifactId>check-in-papi</api.artifactId>
    <api.version>1.0.0</api.version>
  </properties>

  <build>
    <plugins>
      <plugin>
        <groupId>org.mule.tools.maven</groupId>
        <artifactId>mule-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>

  <dependencies>
    <dependency>
      <groupId>${api.groupId}</groupId>
      <artifactId>${api.artifactId}</artifactId>
      <version>${api.version}</version>
      <classifier>oas</classifier>
      <type>zip</type>
    </dependency>
    <dependency>
      <groupId>org.mule.connectors</groupId>
      <artifactId>mule-http-connector</artifactId>
      <classifier>mule-plugin</classifier>
    </dependency>
    <dependency>
```

```
      <groupId>org.mule.modules</groupId>
      <artifactId>mule-apikit-module</artifactId>
      <classifier>mule-plugin</classifier>
    </dependency>
  </dependencies>
</project>
```

Note: Because you use a monorepo, the POM can safely load the parent POM through its relative path in the parent directory.

Note: You define a dependency on the Check-In PAPI API specification, which is not managed in the parent POM, and must therefore include the <version /> element.

Note: You define dependencies on APIkit and HTTP Connector, which are managed in the parent POM, and therefore do not require a <version /> element.

Note: Due to a current limitation in Mule Maven tooling, the dependency on the HTTP Connector must be defined before the dependency on APIkit.

Note: You define the execution of the Mule Maven plugin without repeating its configuration, because that is inherited from the parent POM.

19. **Run and invoke:** Run the Mule app from Studio and invoke the API as before; this should return an HTTP 200 OK response.

Centralize Maven dependency and plugin versions in a BOM

The parent POM introduced in the previous section combines plugin configuration and the definition ("management") of versions of dependencies and plugins. Configuration and version management typically change on a different timescale: once a project has matured, it is more common to update the version of dependencies or plugins, than it is to change the configuration of plugins.

In this section, you separate version management of Maven dependencies and Maven plugins into its own special kind of parent POM — a BOM. Instead of building the BOM by hand, however, you use AnyAirline's standard BOM for Mule apps. This is a BOM that manages the complete set of Maven dependencies and plugins used by AnyAirline and in all walkthroughs of this course.

A BOM — short for Bill Of Materials — is, by convention, a kind of parent POM that only defines the versions of Maven dependencies, including the versions of Maven plugin dependencies. A generic parent POM, on the other hand, has a broader purpose than a BOM and may also define configurations for Maven plugins and other elements of the build that go beyond simply managing versions. If a BOM is used together with a generic parent POM, like here, then it is good practice that the parent POM contains no version management at all, as this is done exclusively in the BOM.

20. **Identify dependency versions:** In **parent-pom/pom.xml** and identify all **<version />** elements of dependencies, either of a Maven plugin or normal Maven dependency.
21. **Copy AnyAirline's BOM:** Copy AnyAirline's complete **bom/pom.xml** into an new **bom** directory within the Studio workspace directory; from now on you will be using this BOM, which already manages all relevant dependencies:

```
cd $APDL2WS
mkdir bom
cp $APDL2DIST/walkthroughs/devprd/module02/wt2-2_solution/bom/pom.xml
bom
```

Note: This is AnyAirline's BOM, which manages all dependencies defined in your current parent POM — and many more.

Note: Both parent-pom/pom.xml and bom/pom.xml now reside in the parent directory of the check-in-papi project.

22. **Study BOM:** Browse the content of the bom/pom.xml, identifying the overlap with parent-pom/pom.xml in terms of the definition of Maven properties, <dependencyManagement />, <pluginManagement />, and the repositories where these dependencies can be retrieved from.

Note: Defining Maven properties for all versions makes changing these versions even easier.

Note: BOMs by convention never impose any actual dependencies or plugin executions on their child projects — they just tie down dependency versions.

23. **Update groupId:** Update bom/pom.xml adapting the Maven coordinates to match the groupId of your Anypoint Platform organization ID:

bom/pom.xml

```
<project>
  <groupId>your-AP-organization-id</groupId>
  <artifactId>solutions-bom</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  ...
</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.

24. **Turn parent POM into child POM of BOM:** Change **parent-pom/pom.xml** into a child POM of bom/pom.xml by defining **bom/pom.xml** as its parent POM and removing all configuration now inherited from the BOM: <dependencyManagement />, <repositories />, and all <version /> elements of Maven plugins:

parent-pom/pom.xml

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>your-AP-organization-id</groupId>
    <artifactId>solutions-bom</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <relativePath>../bom/pom.xml</relativePath>
  </parent>
  <artifactId>solutions-parent-pom</artifactId>
  <groupId>your-AP-organization-id</groupId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>solutions-parent-pom</name>

  <build>
    <resources>...</resources>
    <testResources>...</testResources>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-resources-plugin</artifactId>
          <configuration>...</configuration>
        </plugin>
        <plugin>
          <groupId>org.mule.tools.maven</groupId>
          <artifactId>mule-maven-plugin</artifactId>
          <extensions>true</extensions>
          <configuration />
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>
```

Note: This POM is both a parent POM of check-in-papi's POM as well as a child POM of the BOM. It has no <version /> elements for plugins or dependencies because it inherits all those from

the BOM.

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 configuration of the Mule Maven plugin will soon get much more involved, making it more obvious why it is beneficial to encapsulate it into this parent POM rather than repeating it in all child POMs.

25. **Run and invoke:** Run the Mule app from Studio and invoke the API as before; this should return an HTTP 200 OK response.

Walkthrough 2-3: Build and deploy Mule applications with Maven

The check-in-papi Mule app you have developed so far relies on Studio populating the local Maven repository with artifacts (libraries) that are needed for the Maven build of Mule apps. For many real-world scenarios, additional Maven configuration is needed to perform Maven builds reproducibly and independently of Studio. Another common feature of real-world Maven builds of Mule apps is automated deployment to CloudHub 2.0.

In this walkthrough, you address both of these common requirements for your Maven build of check-in-papi and future Mule apps.

You will:

- Install a Maven settings.xml.
- Deploy to CloudHub using the Mule Maven plugin.

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/devprd/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/devprd/module02/wt2-2_solution.

Install a Maven settings.xml

In this section, you create a global Maven settings.xml that allows a Maven build outside of Studio to retrieve artifacts — such as the Check-In PAPI API specification — from your Anypoint Platform organization's Exchange.

1. **Maven-build:** In a command-line interface, navigate to the **base directory** of check-in-papi and run a Maven build of this Mule app; this should succeed:

   ```
   cd $APDL2WS/check-in-papi
   mvn clean verify
   ```

   Note: If Studio had not populated the local Maven repository with all required artifacts, this build would have failed (unless you already have a suitable settings.xml installed). You will prove this

by deleting the local Maven repository at ~/.m2/repository and re-running the build.

2. **Discover remote Maven repos:** Inspect **bom/pom.xml** and note the Maven repositories which Studio added when creating the Mule app project; one of these repositories is Exchange, the other a MuleSoft public repository — the former requires authentication.
3. **Delete local Maven repo:** From your local Maven repository, delete some artifacts that were put there by Studio:

   ```
   rm -rf ~/.m2/repository/*-*-*-*-*
   ```

   Note: This command is likely to delete all artifacts downloaded from any Exchange.

   Note: You refrain from deleting the entire local Maven repository, because re-populating it just by downloading Maven dependencies would take too long. The above partial delete suffices to prove the point.

4. **Maven-build:** Re-run the Maven build from a command-line interface; this should now fail due to unresolvable dependencies.
5. **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.
6. **Create settings.xml:** In a text editor, create **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-capp-viewer-cid~?~your-capp-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.

Note: Maven allows passwords in settings.xml to be encrypted, but this is not done here for simplicity's sake.

Note: The server entry in settings.xml and the repository entry in the BOM are matched by the common ID.

Note: All Exchange Maven repositories use the same URL (and, by default, also the same ID); their content is determined by the credentials used.

Note: Because the above settings.xml entry uses the default Exchange repository ID, it is dangerous: whenever this user-wide Maven configuration is in use, all default Exchange repository definitions (which use the default repository ID) in any POM, regardless of what project the POM belongs to, will access your Anypoint Platform organization's Exchange. For this reason, it is typically recommended to change repository IDs to be unique to the actual Exchange being accessed, for example by appending the organization ID to the repository ID. You will do this in a later walkthrough.

7. **Maven-build:** In a command-line interface, re-run a Maven build of check-in-papi as before; this should now succeed.

Deploy to CloudHub using the Mule Maven plugin

In this section, you extend the existing Maven build configuration to perform automated deployments to CloudHub 2.0. To do so you add the relevant configuration for the Mule Maven plugin to the parent POM and create a Connected App for authentication. You then deploy check-in-papi to Exchange and the CloudHub 2.0 prod environment from the Maven command-line interface, passing the client ID and secret required by the uplink to the Anypoint Platform control plane as command-line arguments to Maven.

8. **Configure CloudHub 2.0 deployment:** Add to the **parent POM** the **Mule Maven plugin** configuration for a simple deployment to a CloudHub 2.0 Shared Space, utilising Object Store v2:

   parent-pom/pom.xml

```
<configuration>
  <cloudhub2Deployment>
    <businessGroup/>
    <environment>prod</environment>
```

```
    <muleVersion>${app.runtime.semver}</muleVersion>
    <target>Cloudhub-US-East-2</target>
    <provider>MC</provider>
    <replicas>1</replicas>
    <vCores>0.1</vCores>
    <applicationName>${project.name}</applicationName>
    <connectedAppClientId>${ap.ca.client_id}</connectedAppClientId>
    <connectedAppClientSecret>
${ap.ca.client_secret}</connectedAppClientSecret>
    <connectedAppGrantType>client_credentials</connectedAppGrantType>
    <properties>
      <anypoint.platform.client_id>
${ap.client_id}</anypoint.platform.client_id>
      <anypoint.platform.client_secret>
${ap.client_secret}</anypoint.platform.client_secret>
    </properties>
    <deploymentSettings>
      <http>
        <inbound>
          <lastMileSecurity>true</lastMileSecurity>
        </inbound>
      </http>
    </deploymentSettings>
    <integrations>
      <services>
        <objectStoreV2>
          <enabled>true</enabled>
        </objectStoreV2>
      </services>
    </integrations>
  </cloudhub2Deployment>
</configuration>
```

Note: This configuration applies by default to all Maven child projects.

Note: This configuration uses a Connected App client ID and secret for authentication, which you will create shortly.

Note: Instead of using Connected Apps, you can use a <server /> that refers to an entry in settings.xml to authenticate using an Anypoint Platform account. However, this does not support using Connected App client ID and secret as Maven credentials.

Note: You explicitly deploy to the master (root) Anypoint Platform organization, rather than a business group therein, by leaving <businessGroup /> empty.

Note: The Mule runtime version must be specified via the simplified, semantic version (4.4.0) rather than the full version (4.4.0-20230522). Of course, this only applies in the rare cases where there is a difference between these two version strings.

Note: client ID and secret are directly passed to the Mule app, but need to be set on the Maven command-line interface.

Note: The prod environment is hard-coded for the moment, and you will correct this in a later walkthrough.

Note: Anypoint Platform trial accounts can only deploy to 1 worker in usa-e2 (Ohio), so these values are hard-coded.

Note: To use as few vCores as possible, you hard-code deployment to 0.1.

9. **Study and update BOM distributionManagement:** Notice and update the Maven property: **student.deployment.ap.orgid**, replacing the entire value with your Anypoint Platform organization used to configure the **<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: Replace the entire contents of the property with your value and not any other properties the value currently references.

10. **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.
11. **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.

12. **Ensure parent POMs use release versions:** Ensure the **BOM**, **parent POM** and check-in-papi artifact version is a release version and not a SNAPSHOT:

    bom/pom.xml

    ```
    ...
    ```

```
<version>1.0.0</version>
...
```

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 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: 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.

13. **Deploy application to Exchange:** Deploy check-in-papi to your remote Exchange repository:

```
cd $APDL2WS/check-in-papi
mvn clean deploy
```

Note: With CloudHub 1.0 and Standalone Mule runtimes, applications are deployed to a Mule runtime directly. CloudHub 2.0 and Runtime Fabric are container based platforms and require the application binary to be deployed to Exchange first as a persistent store so the application can be retrieved and used to scale containers or recover failed containers.

14. **Create CloudHub Deployment Connected App:** In **Anypoint Access Management**, create a new **Connected App** that acts on its **own behalf (client credentials)**, for deployment to CH, adding the **Cloudhub Organization Admin**, **Create Applications**, **Delete Applications**, **Read Applications**, **View Environment**, **View Organization** and **Design Center Developer** scopes and applying to all environments and retrieve its client ID and secret.
15. **Maven-deploy:** In a command-line interface, deploy to CloudHub 2.0 using the Mule Maven plugin, supplying the same client ID and secret of your Anypoint Platform organization as previously and the client ID and secret of your CloudHub Deployment Connected App; this should result in a successful build:

```
mvn -DmuleDeploy deploy -Dap.client_id=<insert-your-client-id> -
Dap.client_secret=<insert-your-client-secret>
  -Dap.ca.client_id=<insert-your-ca-client-id> -Dap.ca.client_secret
=<insert-your-ca-client-secret>
```

Note: The supplied system properties match the Mule Maven plugin configuration.

Note: For the Mule Maven plugin to execute a deployment, the Maven lifecycle phase and the shown system property must both be specified.

Note: This triggers a deploy or re-deploy, depending on whether a CloudHub deployment with the given name already exists or not.

16. **Locate fully qualified domain name:** In **Runtime Manager**, verify the successful deployment of check-in-papi to the **prod** environment and copy the API's endpoint URL with the additional six-character uniq-id and shard.
17. **Invoke:** Invoke the API using this endpoint URL; this should return an HTTP 200 OK response:

```
curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
:\"Smith\",\"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.

18. **Check log:** In **Runtime Manager**, inspect the log entries as before.
19. **Check API instance:** In **API Manager**, the status of the corresponding API instance should (still) be active/green.

Walkthrough 2-4: Configure secure environment properties

The current implementation of check-in-papi supports configuration for and deployment to just one environment — and where necessary you have chosen this to be the prod environment. In this walkthrough, you extend check-in-papi and its Maven build to support additional environments. Furthermore, you will protect sensitive property values in Runtime Manager by stopping the cleartext display for these values.

You will:

- Configure a Mule app for different runtime environments.
- Encrypt sensitive Mule app configuration properties.
- Protect sensitive Mule app configuration properties in Runtime Manager.
- Deploy the same Mule app to different CloudHub environments.

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/devprd/module02/wt2-4_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/devprd/module02/wt2-3_solution.

Configure a Mule app for different runtime environments

In this section, you introduce environment-specific configuration to check-in-papi. You extract all environment-specific configuration properties, including the filenames and passwords of keystores, into their respective environment-specific properties file. You then create API instances for Check-In PAPI in the dev and test Anypoint Platform environments and complete the autodiscovery configuration of check-in-papi for these environments.

1. **Add env property:** In Studio, near the top of **global.xml**, define a property env that will at runtime always hold the identifier of the environment in which the Mule app currently executes, and default this value to dev:

   global.xml of check-in-papi

   ```
   <global-property name="env" value="dev" />
   ```

Note: Every Mule runtime in which check-in-papi executes has to identify the environment to which it belongs by setting env, otherwise it will be taken to be in the dev environment.

2. **Read env-dependent props file:** Just below, read from a properties file whose name contains the current environment, such as **dev-properties.yaml**; these files will contain environment-dependent properties:

   global.xml of check-in-papi

   ```
   <configuration-properties file="${env}-properties.yaml" />
   <configuration-properties file="properties.yaml" />
   ```

   Note: With this ordering, environment-dependent properties are defined before, and take precedence over, environment-independent properties. Also, the latter can refer to the former, but not the other way round.

3. **Create prod-dependent props file:** Create prod-dependent configuration file **prod-properties.yaml** in **src/main/resources** to add explicit support for the prod environment and move all environment-dependent properties from **properties.yaml** to **prod-properties.yaml**, because so far you have only made use of the prod environment:

   prod-properties.yaml of check-in-papi

   ```
   api:
     id: "15678353"

   tls.keystore:
     type:        "pkcs12"
     path:        "check-in-papi.p12"
     alias:       "server"
     password:    "mule12345"
     keyPassword: "mule12345"
   ```

   Note: You hereby assume — very realistically — that different environments use independent certificates and keystores for exposing HTTPS endpoints.

4. **Create keystores for dev and test:** Copy **check-in-papi.p12** to **check-in-papi-dev.p12** and **check-in-papi-test.p12**; this simulates the creation of independent certificates and keystores for all three environments.

   Note: This "simulation" leads to keystores and passwords (for keystores and keys) that are the same in all environments, which is of course not generally recommended.

Note: If time permits actually create independent keypairs and keystores as before.

Note: The naming convention for certificates, as applied here, is the same as for CloudHub hostnames: the dev and test environments use the respective environment suffix -dev and -test, respectively, while the prod environment does not use a suffix. This is just a convention.

5. **Move keystores:** Move all keystores to a new directory **src/main/resources/certs** to separate properties files from keystores.
6. **Update path property:** Correct the keystore path in **prod-properties.yaml**:

   prod-properties.yaml of check-in-papi

   ```
   tls.keystore:
     path:        "certs/check-in-papi.p12"
   ```

7. **Create dev-dependent props file:** Copy **prod-properties.yaml** to **dev-properties.yaml** and change the latter to add explicit support for the dev environment:

   dev-properties.yaml of check-in-papi

   ```
   api:
     id: ""

   tls.keystore:
     type:        "pkcs12"
     path:        "certs/check-in-papi-dev.p12"
     alias:       "server"
     password:    "mule12345"
     keyPassword: "mule12345"
   ```

   Note: You don't know the API ID in dev yet.

   Note: The passwords are the same in all environments because you just copied the keystores.

8. **Create test-dependent props file:** Copy **prod-properties.yaml** to **test-properties.yaml** and change the latter to add explicit support for the test environment:

   test-properties.yaml of check-in-papi

   ```
   api:
     id: ""

   tls.keystore:
   ```

```
  type:        "pkcs12"
  path:        "certs/check-in-papi-test.p12"
  alias:       "server"
  password:    "mule12345"
  keyPassword: "mule12345"
```

Note: You don't know the API ID in test yet.

Note: The passwords are the same in all environments because you just copied the keystores.

9. **Create API instance in test:** In **API Manager**, go to the **test** environment and create a new API instance for Check-In PAPI by promoting the existing instance from the **prod** environment and then fixing the implementation and consumer URLs by appending **-test** to the hostname, such as check-in-papi-test; remember the API ID of this API instance.

   Note: The -test hostname suffix is the same as for the respective keystore above, and its necessity will become clear shortly.

   Note: Promotion is typically done from a lower environment (test) to a higher environment (prod) not the other way round like here. You do "reverse promotion" here because you started in prod.

10. **Create API instance in dev:** In **API Manager**, go to the **dev** environment and create a new API instance for Check-In PAPI by promoting the existing instance from the **test** environment and then fixing the implementation and consumer URLs by appending **-dev** to the hostname, such as check-in-papi-dev; remember the API ID of this API instance.

    Note: The -dev hostname suffix is the same as for the respective keystore above, and its necessity will become clear shortly.

    Note: Promotion is typically done from a lower environment (dev) to a higher environment (test) not the other way round like here.

11. **Apply automated policies in dev and test:** Create **automated policies** in the dev and test environments, mirroring those in the prod environment.
12. **Update API IDs:** Update **dev-properties.yaml** and **test-properties.yaml** with the API IDs for the newly created API instances in the dev and test environments, respectively.
13. **Run and invoke:** Run the Mule app from Studio and invoke the API as before; this should return an HTTP 200 OK response:

```
curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
:\"Smith\",\"numBags\":2}"
```

```
https://localhost:8081/api/v1/tickets/PNR123/checkin
```

Note: This uses the default dev environment configuration.

14. **Switch to test config:** In Studio, change the **run configuration** by adding the following **VM argument** for **env**, to explicitly load the configuration for the test environment, then re-start the check-in-papi:

```
-M-Denv=test
```

15. **Check API instance:** In **API Manager**, the status of the corresponding API instance in the test environment, should now be active/green; this may take a few minutes; watch-out for the "unblocked" log entry in Runtime Manager, as before.
16. **Switch back to dev:** Change to Studio run configuration back to the dev environment either by deleting the **env** VM argument or by changing its value to dev:

```
-M-Denv=dev
```

Encrypt sensitive Mule app configuration properties

In this section, you add the Secure Configuration Properties module to the Maven build and check-in-papi and use it to protect the values of sensitive properties, such as passwords. To do so you introduce environment-specific secure properties files, which contain values encrypted with the Secure Properties tool, and which are read and decrypted by the Secure Configuration Properties module. You start check-in-papi in Studio with the encryption key required by the Secure Configuration Properties module supplied as a Java system property.

17. **Add Secure Configuration Properties module:** In Studio, install from Exchange the Secure Configuration Properties module; this adds support for encrypted (secure) values in properties files.
18. **Locate Maven dependency:** Locate in **pom.xml** the Maven dependency to Secure Configuration Properties module added by Studio.
19. **Use managed dependency:** Locate in **bom/pom.xml** the existing entry in <dependencyManagement /> that defines the (default) version of Secure Configuration Properties module, and remove the corresponding <version /> from pom.xml.

pom.xml of check-in-papi

```
<dependencies>
```

```
  ...
  <dependency>
    <groupId>com.mulesoft.modules</groupId>
    <artifactId>mule-secure-configuration-property-module</artifactId>
    <classifier>mule-plugin</classifier>
  </dependency>
</dependencies>
```

Note: You do not have to increment the version number of pom.xml itself, despite changing this file, because it uses a snapshot version.

20. **Download Secure Properties tool:** Download the Secure Properties tool from **https://docs.mulesoft.com/mule-runtime/4.4/_attachments/secure-properties-tool.jar** to a filesystem location of your choice, such as /tmp:

```
curl https://docs.mulesoft.com/mule-runtime/4.4/_attachments/secure-
properties-tool.jar -o /tmp/secure-properties-tool.jar
```

21. **Move sensitive props:** Extract all sensitive properties from **dev-properties.yaml** to a new file **dev-secure-properties.yaml**:

dev-secure-properties.yaml of check-in-papi

```
tls.keystore:
  password:    "mule12345"
  keyPassword: "mule12345"
```

22. **Encrypt secure props file for dev:** In a command-line interface, navigate to the **base directory** of check-in-papi and use **Secure Properties tool** to encrypt the values in **dev-secure-properties.yaml**, using Blowfish and CBC for encryption:

```
cd $APDL2WS/check-in-papi/src/main/resources
java -cp /tmp/secure-properties-tool.jar
com.mulesoft.tools.SecurePropertiesTool file encrypt Blowfish CBC
secure12345 dev-secure-properties.yaml tmp.yaml
mv tmp.yaml dev-secure-properties.yaml
```

Note: The Secure Properties tool does not allow you to change a properties file in place.

23. **Study encrypted values:** Inspect the encrypted values in **dev-secure-properties.yaml**, to

understand their syntax:

dev-secure-properties.yaml of check-in-papi

```
tls.keystore:
  password: "![c72bREVng+8ns4IMHA/S2Q==]"
  keyPassword: "![c72bREVng+8ns4IMHA/S2Q==]"
```

24. **Create secure props files for test and prod:** Copy dev-secure-properties.yaml to **test-secure-properties.yaml** and **prod-secure-properties.yaml**:

```
cp dev-secure-properties.yaml test-secure-properties.yaml
```

```
cp dev-secure-properties.yaml prod-secure-properties.yaml
```

Note: You use the same encryption key (secure12345) for all environments: this is purely for simpliciy's sake and not generally recommended.

Note: Because the encryption details and sensitive properties are the same in all environments, the encrypted secure properties files are also identical between environments.

25. **Read secure props file:** In Studio, in global.xml, add a global element to read the encrypted environment-dependent properties file for the current environment, using a new property **encrypt.key** for the encryption key:

global.xml of check-in-papi

```
<secure-properties:config file="${env}-secure-properties.yaml"
  key="${encrypt.key}" name="secureEnvPropsConfig">
  <secure-properties:encrypt algorithm="Blowfish" />
</secure-properties:config>
```

Note: You must specify Blowfish because it is not the default algorithm, whereas CBC is the default and does not need to be mentioned.

Note: Secure and normal properties cannot reference each other, so the order relative to each other does not matter.

26. **Reference secure props: Prefix** all references to secure properties with **secure::** as follows:

global.xml of check-in-papi

```
<tls:context name="apiTLSContext">
  <tls:key-store type="${tls.keystore.type}"
path="${tls.keystore.path}"
    password="${secure::tls.keystore.password}"
    keyPassword="${secure::tls.keystore.keyPassword}"
    alias="${tls.keystore.alias}" />
</tls:context>
```

27. **Update run config:** In Studio, change the **run configuration** of check-in-papi by setting the **VM arguments** for **encrypt.key**:

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

    Note: Because encrypt.key is not — and should not be — defaulted, the Mule app will not start unless this property is set by the Mule runtime in every environment — includig CloudHub, which will be addressed shortly.

    Note: Due to a current limitation in Studio, DataSense metadata support fails due to the Studio tooling instance not being able to locate the encrypt.key that's only passed at runtime. During development, there are temporary workarounds such as setting a temporary global env to provide the runtime property to the Studio tooling instance, but caution should be exercised to not deploy this change or commit to any SCM.

28. **Run:** Run check-in-papi in Studio and verify that it starts up correctly.

Protect sensitive Mule app configuration properties in Runtime Manager

In this section, you protect sensitive properties that must be set as Java system properties, such as encrypt.key and the Anypoint Platform client ID and secret, in the Runtime Manager web UI and by extending the Maven build configuration in the parent POM to automatically protect these value upon deployment.

29. **Review unprotected properties:** In the Runtime Manager dashboard, navigate to the Properties tab of check-in-papi. Note that all properties are visible in plain text.
30. **Protect sensitive properties:** Add the **encrypt.key** property, click **Protect** and apply the changes.
31. **Confirm protected properties:** On the Properties tab, confirm the value for property that you just protected is now no longer visible.

Note: These protected application values are encrypted and stored in the Anypoint Security secrets manager, which, in turn, is encrypted per user organization.

Note: Properties set only in properties files, including secure/encrypted properties, do not have to be included here, as they are not visible in Runtime Manager.

32. **Update CloudHub 2.0 deployment config:** Change the Mule Maven plugin configuration to protect the sensitive properties: **encrypt.key** and the Anypoint Platform **client ID and secret**, moving them to their own **secureProperties** configuration element:

parent-pom/pom.xml

```
<configuration>
  <cloudhub2Deployment>
    ...
    <properties>
    ...
    </properties>
    <secureProperties>
      <encrypt.key>${encrypt.key}</encrypt.key>
      <anypoint.platform.client_id>
${ap.client_id}</anypoint.platform.client_id>
      <anypoint.platform.client_secret>
${ap.client_secret}</anypoint.platform.client_secret>
    </secureProperties>
    ...
  </cloudhub2Deployment>
</configuration>
```

Note: You will validate these values are protected in Runtime Manager when you deploy via the Mule Maven plugin in the next section.

Deploy the same Mule app to different CloudHub environments

In this section, you extend the Maven build configuration in the parent POM to perform automated CloudHub deployments via Mule Maven plugin to different environments. You give deployments to the CloudHub dev and test environments a hostname suffix to denote the respective environment, and thereby make the hostnames unique. You also pass the environment name and encryption key as Java system properties to the Mule app, so that it can start-up in CloudHub with the appropriate environment-specific configuration. You then deploy check-in-papi to all environments using the Maven command-line interface.

33. **Add env config to parent POM:** Add **properties** to **parent-pom/pom.xml** that capture the environment and formalize the variations of CloudHub deployment/application names between different environments; this includes the same hostname prefix as before plus a new hostname suffix:

    parent-pom/pom.xml

```
<!-- Exchange demands a name for deployment -->
<name>solutions-parent-pom</name>
<properties>
  <!-- Not explicitly used but needed for deployment to Exchange,
otherwise Exchange cannot determine the asset type -->
  <type>custom</type>
  <!-- for automated deployments -->
  <deployment.env>set with -Ddeployment.env=...</deployment.env>
  <deployment.suffix>-${deployment.env}</deployment.suffix>
  <deployment.name>
${project.name}${deployment.suffix}</deployment.name>
  <!-- requires AP environments to have the same name as the env
property value in Mule apps (which determines the property files to
load) -->
  <ap.environment>${deployment.env}</ap.environment>
  <ap.client_id>set with -Dap.client_id=...</ap.client_id>
  <ap.client_secret>set with -Dap.client_secret=...</ap.client_secret>
  <ap.ca.client_id>set with -Dap.ca.client_id=...</ap.ca.client_id>
  <ap.ca.client_secret>set with -
Dap.ca.client_secret=...</ap.ca.client_secret>
  <!-- set encrypt.key for decrypting secure (encryped) properties
files (but don't set it here!) -->
</properties>
```

    Note: The same Mule app deployed to different CloudHub environments in the same region must be given distinct names to avoid duplicate fully qualified domain names.

    Note: In prod you will continue using no hostname suffix, as is common usage, by setting the suffix property to the empty string.

    Note: You define two separate Maven properties related to environments: Firstly, a Maven property for the name by which Mule apps identify the current environment, for the purpose of selecting environment-specific configuration; this will be passed to the Mule app through the env property. Secondly, a Maven property for the name of the Anypoint Platform (and therefore CloudHub) environment to which the Mule app is to be deployed. You also make explicit your chosen convention that these two properties hold the same value, while keeping the option of

departing from this convention if necessary, by maintaining two separate Maven properties.

34. **Update CloudHub 2.0 deployment config:** Change the Mule Maven plugin configuration to make use of the name and environment **properties**:

    parent-pom/pom.xml

    ```
    <configuration>
      <cloudhub2Deployment>
        ...
        <environment>${ap.environment}</environment>
        ...
        <applicationName>${deployment.name}</applicationName>
        ...
        <properties>
          ...
          <env>${deployment.env}</env>
        </properties>
      </cloudhub2Deployment>
    </configuration>
    ```

    Note: You instruct the Mule Maven plugin to deploy to the CloudHub environment identified by the respective Maven property, set previously.

    Note: You set the env property of the Mule app from the Maven property that identifies the environment. By default, as set above, this is the name of the CloudHub environment.

35. **Bump check-in-papi Maven artifact version:** In **pom.xml**, update the version to a release version:

    pom.xml of check-in-papi

    ```
    ...
    <version>1.0.1</version>
    ...
    ```

36. **Deploy application to Exchange:** Deploy check-in-papi to your remote Exchange repository, this time passing in the required encryption key:

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

37. **Maven-deploy:** In a command-line interface, deploy check-in-papi to all supported CloudHub environments, supplying the same client ID and secret of your Anypoint Platform organization as before, and only deploying to Exchange once, reusing the already packaged application from Exchange for subsequent deployments:

```
mvn -DmuleDeploy deploy -Dap.client_id=<insert-your-client-id> -
Dap.client_secret=<insert-your-client-secret> -Dap.ca.client_id
=<insert-your-ca-client-id> -Dap.ca.client_secret=<insert-your-ca-
client-secret>
  -Dencrypt.key=secure12345 -Ddeployment.env=dev
```

```
mvn -DmuleDeploy deploy -Dap.client_id=<insert-your-client-id> -
Dap.client_secret=<insert-your-client-secret> -Dap.ca.client_id
=<insert-your-ca-client-id> -Dap.ca.client_secret=<insert-your-ca-
client-secret>
  -Dencrypt.key=secure12345 -Ddeployment.env=test
```

```
mvn -DmuleDeploy deploy -Dap.client_id=<insert-your-client-id> -
Dap.client_secret=<insert-your-client-secret> -Dap.ca.client_id
=<insert-your-ca-client-id> -Dap.ca.client_secret=<insert-your-ca-
client-secret>
  -Dencrypt.key=secure12345 -Ddeployment.env=prod -Ddeployment.suffix=
```

Note: As per the chosen convention, the deployment/hostname suffix for prod is empty.

Note: If you were to use environment-level rather than organization-level client ID and secret, then the values for client ID and secret in each of these statements would be different.

38. **Wait and observe:** In **Runtime Manager** and **API Manager**, follow the deployment process and verify its success.
39. **Invoke:** Invoke the Check-In PAPI endpoints in all environments to verify the successful deployments:

```
curl -ik -X PUT -H "Content-Type: application/json" -d "{\"lastName\"
:\"Smith\",\"numBags\":2}" https://check-in-papi-uniqid.shard.usa-
e2.cloudhub.io/api/v1/tickets/PNR123/checkin
```

Note: The hostnames will be different for you. Note: Append the -dev or -test suffix to the application name in the URL for the corresponding environment.

40. **Confirm protected properties:** On the Properties tab, confirm the values for each property that you protected earlier are now no longer visible. and the Anypoint Platform client ID and secret.
41. **Set consumer and implementation endpoints:** in API Manager, update the implementation URL and consumer endpoint to match your generated fully qualified domain name.

    Note: These endpoints are optional and are used for informational purposes to let clients know how to interact with your API, including surfacing these endpoints in Exchange.

Module 3: Developing for Operational Concerns

In this module, you improve operational observability by implementing operational logging and monitorability of Mule apps and applying the previous non-redundancy of code principle to implement a reusable health check library for Mule apps.

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

- Implement operational logging
- Expose health check endpoints and monitor a Mule app from Anypoint Platform
- Extract reusable Mule app code into a library
- Extract reusable error handling and health check code into a library

Walkthrough 3-1: Implement operational logging

Logging should add minimal performance overhead to production operation yet provide enough context or information for traceability.

In this walkthrough, you tune the Log4J configuration generated by Studio for performance and enrich logging messages for traceability using the Tracing module and logging variables.

You will:

- Use asynchronous logging in production and synchronous logging at DEBUG level in unit tests.
- Enrich logging with Mapped Diagnostic Context and logging variables.

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/devprd/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/devprd/module02/wt2-4_solution.

Use asynchronous logging in production and synchronous logging at DEBUG level in unit tests

In this section, you inspect and tune the Log4J configuration generated by Studio when you created check-in-papi. You must ensure that logging in prod is asynchronous, so that it adds minimal latency to normal event processing. On the other hand, when running unit tests it is convenient to use synchronous logging, and so you change log4j2-test.xml to a synchronous configuration. Also, log entries written by unit tests should typically be down to DEBUG level, which would add prohibitive overhead in production.

1. **Study log configs:** Inspect **log4j2.xml** and **log4j2-test.xml**, the former is used in all normal deployments, while the latter is used only when running unit — such as, MUnit — tests; note that with the exception of the log appender (file versus console), both log configurations are essentially identical and perform logging asynchronously.
2. **Tune test log config:** Change **log4j2-test.xml** to use **synchronous** logging and show DEBUG log entries created by the **default logger**, and the **Message Logging** API policy:

log4j2-test.xml of check-in-papi

```
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%-5p %d [%t] %c: %m%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="INFO">
      <AppenderRef ref="Console"/>
    </Root>
    ...
    <Logger

name="org.mule.runtime.core.internal.processor.LoggerMessageProcessor"
      level="DEBUG" />
    <Logger
      name="org.mule.runtime.logging"
      level="DEBUG" />
  </Loggers>
</Configuration>
```

Note: The default logger, and Message Logging API policy all use different loggers/categories.

Note: You will see this synchronous logging configuration in action when you run MUnit tests in a later walkthrough.

3. **Study or tune log config:** In **log4j2.xml**, you need to make no changes, as the configuration is already asynchronous, but you may want to bring the structure of this file in line with the structure of log4j2-test.xml. Similarly, you may apply Maven resource filtering to reuse the Maven configuration in the log filename configuration:

   log4j2.xml of check-in-papi

```
<Configuration status="WARN">
  <Appenders>
    <RollingFile name="File"

fileName="${sys:mule.home}${sys:file.separator}logs${sys:file.separato
r}${project.name}.log"

filePattern="${sys:mule.home}${sys:file.separator}logs${sys:file.separ
ator}${project.name}-%i.log">
```

```
      <PatternLayout pattern="%-5p %d [%t] [event: %X{correlationId}]
%c: %m%n"/>
      <SizeBasedTriggeringPolicy size="10 MB"/>
      <DefaultRolloverStrategy max="10"/>
    </RollingFile>
  </Appenders>
  <Loggers>
    <AsyncRoot level="INFO">
      <AppenderRef ref="File"/>
    </AsyncRoot>
    ...
    <AsyncLogger

name="org.mule.runtime.core.internal.processor.LoggerMessageProcessor"
      level="INFO" />
    <AsyncLogger
      name="org.mule.runtime.logging"
      level="INFO" />
  </Loggers>
</Configuration>
```

Enrich logging with Mapped Diagnostic Context and logging variables

In this section, you inspect and configure the Log4J configuration generated by Studio and add a new Tracing module to check-in-papi that allows you to enrich log messages automatically by adding variables to the logging context for a given Mule event.

4. **Study log configs:** Inspect **log4j2.xml** and note the existing **File** appender and the pattern layout that currently logs the **correlationId**.
5. **Configure MDC config:** Change **log4j2.xml**, updating the **PatternLayout** for the existing **File** appender and temporarily adding a new **Console** appender with the same PatternLayout that instructs Mule to automatically add the MDC context:

   log4j2.xml of check-in-papi

```
<Configuration status="WARN">
  <Appenders>
    <RollingFile name="File">
      <PatternLayout pattern="%-5p %d [%t] [%MDC] %c: %m%n"/>
    </RollingFile>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%-5p %d [%t] [%MDC] %c: %m%n"/>
```

```
    </Console>
  </Appenders>
  <Loggers>
    ...
    <AsyncRoot level="INFO">
      <AppenderRef ref="File"/>
      <AppenderRef ref="Console"/>
      ...
    </AsyncRoot>
  </Loggers>
</Configuration>
```

Note: By default, Mule logs two MDC entries: processor, which shows the location of the current event, and event, which shows the correlation ID of the event.

Note: The Console appender is only used to make the log context available in the Studio console. This is not advised for production environments.

*Note: The Studio console is not completely controlled by the **log4j2.xml**. Studio itself only uses part of the **log4j2.xml** and does not use the configuration defined in the File appender. The full log file created by the File appender can be found by navigating to the <MULE_HOME>/logs directory of the Studio embedded Mule runtime. You can find the MULE_HOME location by searching for MULE_HOME in a running Studio console*

6. **Add managed library dependency:** Locate the existing dependency management of mule-tracing-module in the **BOM** and add a matching entry to the check-in-papi **POM**:

   pom.xml of check-in-papi

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

7. **Add logging variables to the context:** add a logging variable at the beginning of **check-in-by-pnr** for PNR:

main.xml of check-in-papi

```
<flow name="check-in-by-pnr">
  <tracing:set-logging-variable variableName="PNR" value="#[vars.PNR
default '']"/>
  ...
</flow>
```

Note: The Tracing module enables you to enhance your logs by adding, removing, and clearing variables from the logging context for a given Mule event.

Note: This logging context affects any output of a Logger component and any internal logging that the Mule runtime produces. Therefore, you can add more context to the event that Mule is processing if you need additional information for tracking down any potential issue, for example, when an unexpected error occurs.

Note: The logging context exists for the entire execution of the corresponding event. Once a variable is set, it will continue to be logged for an event until it is removed using the corresponding remove operation or until all variables are removed using the clear operation.

8. **Run, invoke, and check log:** Run the Mule app from Studio and invoke the API via cURL as before and study the log entries noting the **correlationId** and the new variables set in the logging context.

Walkthrough 3-2: Expose and monitor health check endpoints

It is common practice that Mule apps should expose health check endpoints that can be invoked from the outside to probe the functioning of the Mule app. CloudHub adds such health checks for its own purposes, to drive automated restart of Mule apps, for instance, but these health checks are only available when deploying to CloudHub and are generic, that is, without knowledge of any implementation details of a Mule app.

An independent but related concern is that Mule apps should also be represented in the graphical application network view provided by Visualizer, and show up in the correct architecture layer, following API-led connectivity.

In this walkthrough, you add two types of health check endpoints to check-in-papi, following a convention used, for example, by Kubernetes: Firstly, a liveness endpoint that can be invoked over HTTP/S to perform a liveness check of the Mule app. Mule apps that fail the liveness check are not in a runnable state and should be restarted. Secondly, a readiness endpoint that can be invoked over HTTP/S to perform readiness checks of the Mule app. Mule apps that fail the readiness check but not the liveness check are themselves running fine, but at least one of their required downstream dependencies — such as an API invoked by the Mule app — is not ready. Restarting such a Mule app is pointless, but it is nevertheless not ready to service requests. Furthermore, you extend check-in-papi and the Maven build to assign Mule apps to the correct architectural layer in Visualizer and have Visualizer use a concise name for the Mule app.

You will:

- Expose liveness and readiness endpoints.
- Make a Mule app announce its tier and concise name to Visualizer.
- Configure a Functional Monitor to invoke health check endpoints.

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/devprd/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/devprd/module03/wt3-1_solution.

Expose liveness and readiness endpoints

In this section, you add healtheck endpoints in the form of a liveness and a rudimentary readiness endpoint to check-in-papi, using the same HTTP Listener configuration also used for exposing Check-In PAPI. These health check endpoints respond to a HTTP GET with a HTTP 200 response (and a payload of UP) in case all is well, or a HTTP 500 response (and a payload of DOWN) if not.

1. **Create Mule flow config file:** In Studio, add to check-in-papi a Mule flow config file **health.xml**.
2. **Implement liveness endpoint:** Add to health.xml a new Mule flow **api-alive** for the liveness endpoint, referring to the same HTTP Listener configuration that is also used to expose the API, using a path of **/alive** and setting the response to 200 and UP under normal conditions, or 500 and DOWN in case of error:

   health.xml of check-in-papi

   ```
   <flow name="api-alive">
     <http:listener config-ref="apiHttpListenerConfig" path="/alive">
       <http:response statusCode="200" />
       <http:error-response statusCode="500">
         <http:body>DOWN</http:body>
       </http:error-response>
     </http:listener>
     <set-payload value="UP" />
   </flow>
   ```

   Note: Using the same HTTP Listener configuration for the health check endpoints as for exposing the API makes the health checks more meaningful.

   Note: health check endpoints are not part of the business-centric API defined in an API specification and therefore listen to an endpoint and path not covered by the API specification.

   Note: It is difficult to construct a scenario where the above liveness endpoint implementation would return the configured error response.

3. **Implement readiness endpoint:** Add to health.xml a new Mule flows **api-ready** for the readiness endpoint, which for the time being does not have any downstream dependencies to check:

   health.xml of check-in-papi

   ```
   <flow name="api-ready">
     <http:listener config-ref="apiHttpListenerConfig" path="/ready">
   ```

```
    <http:response statusCode="200" />
    <http:error-response statusCode="500">
      <http:body>DOWN</http:body>
    </http:error-response>
  </http:listener>
  <flow-ref name="check-all-dependencies-are-alive" />
  <set-payload value="UP" />
</flow>

<sub-flow name="check-all-dependencies-are-alive">
  <logger />
</sub-flow>
```

Note: Mule flows are not allowed to be empty, so to make check-all-dependencies-are-alive valid you add an empty logger.

4. **Run and invoke:** Run the Mule app from Studio and invoke the health check endpoints; this should return a HTTP 200 response with UP in both cases:

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

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

Make a Mule app announce its tier and concise name to Visualizer

In this section, you extend the Maven build to set a Java system property that informs Visualizer to which tier a given Mule app belongs. You assign check-in-papi to the Process tier. Visualizer calls tiers layers and predefines the Experience, Process, and System layers.

By default, Visualizer uses the CloudHub application name of a CloudHub-deployed Mule app (such as check-in-papi-dev) to represent that Mule app in its application network rendering. This is slightly longer than required and hence makes Visualizer renderings unnecessarily busy: you tell Visualizer to use a more concise display name for check-in-papi by setting a Java system property.

5. **Add Visualizer props to CloudHub deployment config:** To the **parent POM** add to the **Mule Maven plugin** configuration three properties, one for setting the Visualizer layer, one for setting the Visualizer display name, and another one to allow the Anypoint Platform control plane to collect relevant runtime analytics:

parent-pom/pom.xml

```
<cloudhub2Deployment>
  ...
  <properties>
    <anypoint.platform.config.analytics.agent.enabled>
      true
    </anypoint.platform.config.analytics.agent.enabled>
    <anypoint.platform.visualizer.displayName>
      ${project.name}
    </anypoint.platform.visualizer.displayName>
    <anypoint.platform.visualizer.layer>
      ${api.layer}
    </anypoint.platform.visualizer.layer>
    ...
  </properties>
  ...
</cloudhub2Deployment>
```

Note: The collection of runtime analytics may already be enabled for the CloudHub image you are deploying to, so the above is not always strictly necessary. But it is highly recommended.

Note: With this, Visualizer will render any Mule app using the <artifactId /> of the child project (such as check-in-papi) rather than the CloudHub application name (such as check-in-papi-dev). The former is sufficient because Visualizer only displays the application network for a given Anypoint Platform organization.

Note: This configuration relies on the api.layer property, which is fixed for every Mule app.

6. **Define layer name props:** To the **parent POM** add top-level properties that fix the allowed layer names:

parent-pom/pom.xml

```
<project>
    ...
    <name>solutions-parent-pom</name>
  <properties>
    <api.layer.eapi>Experience</api.layer.eapi>
    <api.layer.papi>Process</api.layer.papi>
    <api.layer.sapi>System</api.layer.sapi>
    <api.layer.backend>Backend</api.layer.backend>
    <api.layer.none>None</api.layer.none>
    ...
```

```
  </properties>
</project>
```

Note: The values Experience, Process, and System are well-known to Visualizer, but Visualizer accepts arbtirary layer names, such as Backend.

Note: These properties are available to all child POMs/projects.

7. **Set layer prop:** In the **POM** for check-in-papi, set the value of the **api.layer** property, such that it is available in the Mule Maven plugin configuration in the parent POM:

   pom.xml of check-in-papi

   ```
   <properties>
     ...
     <api.layer>${api.layer.papi}</api.layer>
   </properties>
   ```

   Note: Because the layer name is taken from a property inherited from the parent POM, it reduces redundancy and the risk of typos.

8. **Bump Maven artifact version:** In **pom.xml**, bump the patch version of check-in-papi:

   pom.xml of check-in-papi

   ```
   ...
   <version>1.0.2</version>
   ...
   ```

9. **Maven-deploy:** In a command-line interface, deploy the application to Exchange and subsequently the CloudHub **dev** environment using the Mule Maven plugin, supplying the same command-line arguments as before; this should result in a successful deployment:

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

   ```
   mvn -DmuleDeploy deploy -Dap.client_id=<insert-your-client-id> -
   Dap.client_secret=<insert-your-client-secret> -Dap.ca.client_id
   =<insert-your-ca-client-id> -Dap.ca.client_secret=<insert-your-ca-
   client-secret> -Dencrypt.key=secure12345 -Ddeployment.env=dev
   ```

10. **Invoke:** Invoke the health check endpoints; this should return a HTTP 200 response in both cases:

```
curl -i https://check-in-papi-uniqid.shard.usa-e2.cloudhub.io/alive
```

```
curl -i https://check-in-papi-uniqid.shard.usa-e2.cloudhub.io/ready
```

Note: The hostname will be different for you.

11. **Check Visualizer layer:** In Visualizer, under Sandbox environments, confirm that check-in-papi in the dev environment is indeed assigned to the Process layer.

Note: It can take several minutes for Visualizer to reflect the latest configuration.

Configure a Functional Monitor to invoke health check endpoints

In this module, you — using an account with sufficient entitlements, or an instructor, create a Functional Monitor in Anypoint Monitoring to implement a liveness and readiness probe on check-in-papi. The Functional Monitor sends HTTP GET requests on a regular basis to check-in-papi and raises an alert if either fails. 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.

12. **Navigate to Anypoint Monitoring:** Using credentials with sufficient permissions, log in an organization with sufficient entitlements or as instructor to the AnyAirline Anypoint Platform organization and navigate to Anypoint Monitoring.
13. **Create Functional Monitor:** Create a Functional Monitor for **check-in-papi** in the dev environment sending an HTTP **GET** request every **15 minutes** your CloudHub deployed check-in-papi endpoint, asserting that the HTTP status is **200** and notifying you via **email** to your email address of any failures.

Note: You can switch to the code editor, to create a monitor written in BAT, a Behavior Driven Development (BDD) language.

Note: Additionally, you can upload custom test suites written in BAT via the Anypoint Platform APIs.

Walkthrough 3-3: Extract reusable Mule app code into a library

Reflecting on the check-in-papi Mule app code, it becomes apparent that parts of it are likely to be re-usable in other similar Mule apps and API implementations. Examples are the global error handlers in error.xml and all of the health check endpoints in their current form in health.xml. In future walkthroughs you will identify other reusable Mule flows or configuration elements.

In this walkthrough, you extract reusable Mule app code from check-in-papi into a separate project apps-commons, which builds a library-style Mule plugin, that can be imported into any Mule app that needs this functionality. The apps-commons project is the first of many projects that can reuse parts of the Maven build configuration encapsulated in the parent POM and BOM.

You will:

- Identify reusable Mule app code elements.
- Extract reusable Mule app flows and configuration elements into a new Maven project.
- Import Mule app code elements from a library.
- Deploy Maven artifacts to Exchange as custom assets.

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/devprd/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/devprd/module03/wt3-2_solution.

Identify reusable Mule app code elements

In this section, you inspect check-in-papi and identify code elements that could be reused by other, similar Mule apps.

1. **Reflect:** In Studio, inspect **error.xml** and think on how the global error handlers therein were created and under what circumstances they could be reused in other Mule apps.
2. **Reflect:** Inspect **health.xml** and think on how api-alive and api-ready differ between check-in-papi and other similar Mule apps.

Extract reusable Mule app flows and configuration elements into a new Maven project

In this section, you create a new Maven project apps-commons with the code from error.xml and health.xml in check-in-papi. The apps-commons Maven project is structured like a Mule app, with Mule flow config files in src/main/mule, but builds an artifact with Maven classifier mule-plugin. This type of Mule plugin could be called a "library-style Mule plugin" to distinguish it from other, more sophisticated Mule plugins like connectors or XML SDK modules — all of which use the Maven classifier mule-plugin. By contrast, a library-style Mule plugin just contains simple resources like Mule flow config files and is built from a Maven project with <packaging /> mule-application and <classifier /> mule-plugin.

3. **Create new library project:** In Studio, create a new standard Mule app project named **apps-commons**; this creates the apps-commons project directory in your Studio workspace directory.
4. **Delete auto-created Mule flow config file:** Delete the Mule flow config file automatically created by Studio in the new apps-commons project.
5. **Copy reusable Mule flow config files:** In a command-line interface or Studio, copy the previously identified reusable Mule flow config files from check-in-papi to apps-commons, appending **-common** to their filenames:

   ```
   cd $APDL2WS
   cp check-in-papi/src/main/mule/health.xml  apps-
   commons/src/main/mule/health-common.xml
   cp check-in-papi/src/main/mule/error.xml  apps-
   commons/src/main/mule/error-common.xml
   ```

   Note: By convention, you append -common to all Mule flow config files in apps-commons, so that these filenames are unlikely to conflict with filenames in any Mule app that includes apps-commons.

   Note: Studio might show an error that the HTTP Listener configuration used in health-common.xml is missing. This will be corrected shortly.

6. **Adapt POM:** Change the pom.xml of apps-commons into a child POM by defining **parent-pom/pom.xml** as its parent POM, adapting the Maven coordinates to match the **groupId** of your Anypoint Platform organization ID, removing **repositories** and adding the **Mule Maven plugin configuration** to create an artifact with <classifier /> **mule-plugin**:

   pom.xml of apps-commons

   ```
   <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>
<!-- students: replace with your AP org ID -->
<groupId>a63e6d25-8aaf-4512-b36d-d91b90a55c4a</groupId>
<artifactId>apps-commons</artifactId>
<version>1.0.0-SNAPSHOT</version>
<packaging>mule-application</packaging>
<name>apps-commons</name>
<build>
  <plugins>
    <plugin>
      <groupId>org.mule.tools.maven</groupId>
      <artifactId>mule-maven-plugin</artifactId>
      <configuration>
        <classifier>mule-plugin</classifier>
      </configuration>
    </plugin>
  </plugins>
</build>
<dependencies>
  <dependency>
    <groupId>org.mule.connectors</groupId>
    <artifactId>mule-http-connector</artifactId>
    <classifier>mule-plugin</classifier>
    <!-- must be provided by enclosing Mule app -->
    <scope>provided</scope>
  </dependency>
</dependencies>
```

Note: You must ensure to remove the repository definitions as they will conflict with the BOM and cause deployment failures later.

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 <packaging /> element remains mule-application but the supplied Mule Maven plugin configuration changes the created artifact to be a Mule plugin via the <classifier /> element.

Note: The Mule Maven plugin configuration is inherited in its entirety from the parent POM, and only the <classifier /> value is overridden in this child POM.

Note: The Mule flow config files in apps-commons — and therefore apps-commons itself — currently depend on the HTTP Connector, just like check-in-papi from which they originate. You make this fact explicit by defining these two dependencies in provided scope, meaning they must be provided (packaged) by all Mule apps that include apps-commons. To define these dependencies, you make use of the already configured version management in the BOM.

7. **Remove non-reusable flow:** In Studio, open **health-common.xml** and cut **check-all-dependencies-are-alive** because this Mule flow cannot be implemented in a common library; Studio should now show an error that this Mule flow is invoked from a <flow-ref /> in api-ready but missing.
8. **Create stubs:** Create a new Mule flow config file called **stubs.xml** in **src/test/resources** (drag it from src/main/mule if necessary) that contains stubs (placeholders) for configuration elements expected to exist in the including Mule app but missing from apps-commons, paste **check-all-dependencies-are-alive** into it, and add a placeholder HTTP Listener configuration:

   stubs.xml of apps-commons

   ```
   <http:listener-config name="apiHttpListenerConfig">
     <http:listener-connection host="0.0.0.0" protocol="HTTP" port="8081"
   />
   </http:listener-config>

   <sub-flow name="check-all-dependencies-are-alive">
     <logger />
   </sub-flow>
   ```

   Note: The stubs.xml file is in the test source branch and is therefore available at development time but is not packaged in the resulting apps-commons library. This serves merely to eliminate the error(s) shown in Studio.

   Note: The details of these configuration elements do not matter, only their type and name. They will never be instantiated.

9. **Maven-build:** Navigate to the base directory of apps-commons and run a Maven build to install apps-commons into your local Maven repo; this should succeed:

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

   Note: Installing apps-commons into your local Maven repository with this statement is essential, otherwise Maven will not be able to resolve dependencies to apps-commons in later

walkthroughs.

10. **Install parent POMs:** Install the **BOM** and **parent POM** used by apps-commons and check-in-papi 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
```

Note: This is necessary because the POM of apps-commons depends on both the parent POM and BOM, yet in the previous step only apps-commons was installed into the local Maven repository. When a dependency on apps-commons is resolved, the POM of apps-commons is loaded from the local Maven repo, and so the parent POM and BOM — which are required to interpret the POM of apps-commons — must also be loaded from the local Maven repo.

Import Mule app code elements from a library

In this section, you import apps-commons as a Maven dependency into check-in-papi to reuse the Mule app code from that library.

11. **Add managed library dependency:** Locate the existing dependency management of apps-commons 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>apps-commons</artifactId>
    <classifier>mule-plugin</classifier>
  </dependency>
</dependencies>
```

Note: The BOM defines a Maven property, available in all child POMs, to hold your Anypoint Platform organization ID, which is the <groupId /> for the apps-commons retrieved from the Exchange of that Anypoint Platform organization.

12. **Refresh Studio project:** In Studio, close and reopen the **check-in-papi** project; Studio should resolve the apps-commons dependency and show it as a project dependency.

Note: If Studio does not find apps-commons 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
```

13. **Replace error handlers with import:** In **error.xml** of check-in-papi replace the code that has been extracted to apps-commons — which is all Mule app code in that file — with an import of the relevant Mule flow config file from apps-commons:

    error.xml of check-in-papi

    ```
    <mule>
      <import file="error-common.xml" />
    </mule>
    ```

    Note: Do not delete the XML namespace-related attributes from the root element in this Mule flow config file — they are omitted here just for brevity!

    Note: Mule flow config files from the Mule app itself are always implicitly available in that Mule app, but files packaged in a library must be imported explicitly, such as here.

14. **Replace health check endpoints with import:** In **health.xml** of check-in-papi replace the code that has been extracted to apps-commons with an import of the relevant Mule flow config file from apps-commons:

    health.xml of check-in-papi

    ```
    <mule>
      <import file="health-common.xml" />

      <sub-flow name="check-all-dependencies-are-alive">
        <logger />
      </sub-flow>
    </mule>
    ```

    Note: Do not delete the XML namespace-related attributes from the root element in this Mule flow config file — they are omitted here just for brevity!

    Note: The check-all-dependencies-are-alive Mule flow inoked from api-ready in apps-commons but missing from that library must be implemented in health.xml, otherwise check-in-papi would

fail at runtime and Studio should also show an error at development time.

15. **Run and invoke:** Run the Mule app from Studio and invoke the health check endpoints; this should return a HTTP 200 response in both cases:

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

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

Deploy Maven artifacts to Exchange as custom assets

Mule 4 Extensions that declare dependencies that don't exist in Maven Central or the MuleSoft Maven repositories, are not currently supported. Therefore any extension type artifact deployed to Exchange that relies on a custom asset such as the parent POM and BOM introduced in the previous sections, are required to deployed to Exchange prior.

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 Exchange Mule Maven plugin each POM as a custom asset.

16. **Study BOM:** Browse the content of the bom/pom.xml, identifying the **Exchange Mule Maven plugin** configuration located in a Maven profile **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: The exchange-pre-deploy goal must be run to pre-validate the asset with Exchange where it checks various pre-conditions such as unique asset versions. Without explicitly running this goal, the Exchange API will return a status code of 412 (Precondition Failed). It 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: This plugin is applied directly to the BOM and configured not to be inherited by Maven child projects so that the Exchange pre-validation steps are not applied to projects that will not be deployed to Exchange.

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.

17. **Add Maven property to parent POMs:** Add to the parent POMs a property to instruct the Exchange Mule Maven plugin plugin that this is a custom asset:

    bom/pom.xml

```
...
<properties>
  ...
  <type>custom</type>
  ...
```

```
<properties>
...
```

parent-pom/pom.xml

```
...
<properties>
  ...
  <type>custom</type>
  ...
<properties>
...
```

Note: This property is not explicitly used but is used internally by the Exchange Mule Maven plugin and needed for deployment to Exchange, otherwise Exchange cannot determine the asset type.

18. **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.

19. **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.

20. **Refresh Studio project:** In Studio, close and reopen the **check-in-papi** project; Studio should resolve the apps-commons dependency and show it as a project dependency.

    Note: If Studio does not find apps-commons 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
    ```

Module 4: Automating unit-testing with MUnit

In this module, you develop a unit test suite for a Mule app using the MUnit framework, and execute these tests interactively from Studio and automatically from every Maven build.

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

- Enable a Mule app for unit testing with MUnit.
- Perform basic unit testing of integration functionality.
- Mock external dependencies.
- Spy on the data exchanged with external dependencies.

Walkthrough 4-1: Enable MUnit

The check-in-papi Mule app so far implements no real integration logic. Before you start adding integration logic, you must enable unit testing support, so that integration logic can be unit tested with minimal friction.

In this walkthrough, you add support for MUnit testing to check-in-papi and its Maven build. You also have Studio generate some basic MUnit tests for the code in check-in-papi. At the end of this walkthrough your parent POM will be sufficiently mature to be reusable in all Mule apps and Maven projects created in the remainder of this course.

You will:

- Add fundamental MUnit support and basic MUnit tests to check-in-papi and its parent POMs.
- Analyze MUnit test coverage.

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/devprd/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/devprd/module03/wt3-3_solution.

Add fundamental MUnit support and basic MUnit tests to a Mule app

In this section, you use Studio features to add MUnit support to check-in-papi and generate a basic MUnit test suite for the Mule flows in main.xml. You also pass the encryption key when MUnit tests are run, so that secure properties files can be decrypted.

1. **Add MUnit Maven config:** In Studio, on the **check-in-papi** project, execute **Configure MUnit Maven support**; this adds the MUnit Maven plugin and supporting Maven dependencies to the POM.
2. **Use managed dependencies:** Locate in the **BOM** the already existing <dependencyManagement /> entries for all MUnit-related dependencies, then delete their **<version />** elements from the **POM**:

pom.xml of check-in-papi

```
<build>
  <plugins>
    ...
    <plugin>
      <groupId>com.mulesoft.munit.tools</groupId>
      <artifactId>munit-maven-plugin</artifactId>
      <executions>...</executions>
      <configuration>...</configuration>
    </plugin>
  </plugins>
</build>

<dependencies>
  ...
  <dependency>
    <groupId>com.mulesoft.munit</groupId>
    <artifactId>munit-runner</artifactId>
    <classifier>mule-plugin</classifier>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>com.mulesoft.munit</groupId>
    <artifactId>munit-tools</artifactId>
    <classifier>mule-plugin</classifier>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.mule.weave</groupId>
    <artifactId>assertions</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Note: These MUnit Maven dependencies must be defined in test scope, otherwise the Mule app will no longer execute successfully.

3. **Pull-up refined plugin config:** Copy the **MUnit Maven plugin** configuration to the **parent POM** under **<pluginManagement />** so that it is available for all related Mule apps and specify MUnit tests to run on **4.4.0-20230522 EE** with the correct **encrypt.key**:

parent-pom/pom.xml

```
<plugin>
  <groupId>com.mulesoft.munit.tools</groupId>
  <artifactId>munit-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>test</id>
      <phase>test</phase>
      <goals>
        <goal>test</goal>
        <goal>coverage-report</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <runtimeVersion>${app.runtime}</runtimeVersion>
    <runtimeProduct>MULE_EE</runtimeProduct>
    <environmentVariables>
      <!-- this implies that all tests run in the same Maven build
must use the same encrypt.key -->
      <encrypt.key>${encrypt.key}</encrypt.key>
    </environmentVariables>
    <coverage>
      <runCoverage>true</runCoverage>
      <formats>
        <format>console</format>
        <format>html</format>
      </formats>
    </coverage>
  </configuration>
</plugin>
```

Note: You omit the version of the MUnit Maven plugin because it is inherited from the BOM.

Note: You configure the MUnit Maven plugin in <pluginManagement /> so that each child project can decide whether to execute it or not.

Note: Runtime version and product used to run MUnit tests have defaults taken from mule-artifact.json, but can also be specified explicitly, like here. This is recommended when a precise version of the Mule runtime is targeted, as is often the case.

Note: The Mule runtime version must be specified via its full version (4.4.0-20230522) rather than the simplified, semantic version (4.4.0). Of course, this only applies in the rare cases

where there is a difference between these two version strings.

Note: MUnit tests would fail at launch if the encrypt.key property were not supplied to the embedded Mule runtime executing the tests.

Note: Test coverage reports will be produced in HTML format and printed to the console.

4. **Simplify plugin config in POM:** In the POM, delete everything now inherited from the parent POM and BOM, retaining just what is needed to execute the already configured **MUnit Maven plugin** (in addition to the Mule Maven plugin), and keeping the MUnit-related dependencies without version in test scope:

   pom.xml of check-in-papi

   ```
   <build>
     <plugins>
       <plugin>
         <groupId>org.mule.tools.maven</groupId>
         <artifactId>mule-maven-plugin</artifactId>
       </plugin>
       <plugin>
         <groupId>com.mulesoft.munit.tools</groupId>
         <artifactId>munit-maven-plugin</artifactId>
       </plugin>
     </plugins>
   </build>
   ```

5. **Generate MUnit tests:** Using the New MUnit test wizard (**File > New > MUnit Test**), generate MUnit test skeletons for all Mule flows in **main.xml** into a new file **main-test-suite.xml**.
6. **Study and clean tests:** Inspect and clean-up the generated MUnit tests to follow your coding conventions.
7. **MUnit-test:** In Studio, run the test suite; this should fail.
8. **Update run config:** Edit the run configuration generated by Studio for the MUnit test suite by adding the **encryption key** as a **VM argument**:

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

   Note: This is a direct argument to the JVM, not to the Mule runtime.

9. **Locate repos:** In **BOM**, find in **<pluginRepositories />** the existing entry for the Maven

repository to download the **EE** Mule runtime from.

Note: The normal MuleSoft releases repository is always required both under <repositories /> and <pluginRepositories />.

Note: The MuleSoft EE releases repository is required under <pluginRepositories /> because the MUnit Maven plugin has been configured to run tests on the EE version of the Mule runtime. If EE-only Mule plugins or connectors were used, which is currently not the case, then it would also be required under <repositories />.

10. **Install parent POMs:** Install the **BOM** and **parent POM** into your local Maven repository:

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

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

Note: You have already installed the parent POM in a previous walkthrough, for different reasons, but parent-pom/pom.xml has changed in the meantime.

11. **Add repository credentials:** In **settings.xml**, add the credentials required to access the MuleSoft EE releases repository:

settings.xml

```
<server>
  <id>releases-ee</id>
  <username>muletraining.nexus</username>
  <password>eApHMgTm</password>
</server>
```

12. **Maven-build:** In a command-line interface, run a full Maven build including the MUnit test suite, passing the encryption key; this should succeed, all unit tests should pass, and a coverage report should be output to the console:

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

13. **MUnit-test:** In Studio, run the MUnit test suite, using the run configuration that includes the encryption key; this should succeed and all unit tests should pass.

Analyze MUnit test coverage

In this section, you briefly compare the visual test coverage indicators in Studio and the textual test coverage report produced by the MUnit Maven plugin.

14. **Study coverage report:** Inspect the test coverage report output to the console in the previous step.
15. **Study Studio coverage:** In Studio, open **main.xml** and, for example, **api.xml**, and confirm that the visual test coverage indicators match the coverage report.
16. **Reflect:** Discuss desired coverage limits for the different Mule flow config files and Mule flows, which could be enforced by the MUnit Maven plugin.

Walkthrough 4-2: Perform basic unit testing

The MUnit tests generated for check-in-papi in the previous walkthrough are just skeletons.

In this walkthrough, you first realize that there is value in simply asserting that a Mule app deploys and starts successfully. You then proceed to implement simple unit tests for the check-in functionality of check-in-papi, ignoring the payment approval functionality of that API implementation.

You will:

- Test deployment and startup of check-in-papi.
- Prepare and pass test data to Mule flows under test.
- Assert the output of executing these Mule flows.

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/devprd/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/devprd/module04/wt4-1_solution.

Test deployment and startup of the Mule app under test

In this section, you implement the simplest possible MUnit test and realize that even this serves an important purpose, namely the verification that the Mule app is syntactically correct, has all required dependencies, and is starts up without errors. Every Mule app that does not implement a proper MUnit test suite should at least provide such a "start-up test".

1. **Study test execution:** Inspect **main-test-suite.xml** and match it with the log output of executing this test suite.

   Note: The Mule app under test — check-in-papi — was deployed to an embedded Mule runtime and started.

   Note: When a test case and tests generated by Studio executes, it invokes its respective flow-under-test without preparing test data or asserting any end state.

   Note: Invoking flows-under-test without preparing test data cannot in general be expected to

succeed, so the fact that these tests pass is a fortunate conincidence. This is why the generated tests are only skeletons — starting points for implementing tests.

2. **Implement startup test only:** Temporarily comment-out all generated tests and add the simplest possible test case:

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

   ```
   <munit:test name="start-up-test" description="...">
     <munit:execution>
       <logger />
     </munit:execution>
   </munit:test>
   ```

   Note: The <munit:execution /> section must not be empty.

   Note: This test case does nothing (but log the current message) and is completely independent of check-in-papi.

3. **MUnit-test:** Run the test suite; this should pass.

   Note: If check-in-papi were misconfigured — a missing properties file, an invalid Mule flow config file, a missing or incorrect encrypt.key, etc. — the test suite would have failed.

   Note: Even the simplest possible MUnit test, whose code is completely independent of the Mule app under test, verifies that the Mule app is syntactically correct, not missing any dependencies, and deploys and starts successfully.

4. **Revert to generated tests:** Undo the changes to **main-test-suite.xml**.

Prepare and pass test data to Mule flows under test

In this section, you prepare test data as input for check-in-by-pnr, the flow-under-test. Because this flow is in main.xml, and not in api.xml, it does not expose an HTTP-based interface and the input test data is therefore not a HTTP request. Rather than hard-coding complex test data in the MUnit tests, you load it from a file.

5. **Determine expected test input:** Open **main.xml** and explore the input expected by **check-in-by-pnr** by tracing back to its calling flow in api.xml and the API specification for Check-In PAPI.

   Note: check-in-by-pnr requires a variable PNR and a JSON payload with last name and number of bags, as defined by the API specification.

6. **Create test data file:** Create a new directory and file **json/check-in-by-pnr-request.json** under **src/test/resources** to hold the payload test data for check-in-by-pnr:

   json/check-in-by-pnr-request.json of check-in-papi

   ```
   { "lastName": "Mule", "numBags": 2 }
   ```

7. **Set test input data from file:** In **main-test-suite.xml** edit the tests generated before to implement a happy-path test that sets the input test data for check-in-by-pnr using <munit:set-event />, reading the payload from the previously created data file:

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

   ```
   <munit:test name="check-in-by-pnr-happy-path-test"
     description="...">
     <munit:execution>
       <munit:set-event>
         <munit:payload
           value="#[output application/json ---
   read(MunitTools::getResourceAsString('json/check-in-by-pnr-
   request.json'), 'application/json')]"
         />
         <munit:variables>
           <munit:variable key="PNR" value="N123" />
         </munit:variables>
       </munit:set-event>
       <flow-ref name="check-in-by-pnr" />
     </munit:execution>
   </munit:test>
   ```

   Note: The content-type application/json of the payload data read from file must be stated repeatedly, once when reading and parsing the file content and once when specifying the output media type of the DataWeave expression.

   Note: Setting the output media type of the DataWeave expression is sufficient for setting the payload value in MUnit test suites. The same is not true when setting the value of a variable or attribute to a non-primitive value (that is, other than here), where the media type must be (redundantly) specified as an XML attribute even if the DataWeave expression for the value already specifies the output media type.

   Note: <munit:set-event /> can be used in both <munit:behavior /> and <munit:execution />. When used in <munit:behavior />, it must be the last action configured, as mocks and spies defined after can interfere.

Note: The above message payload and variable are passed as input to check-in-by-pnr.

Note: By default, MUnit does not start any of the event sources (flow sources such as triggers, and listeners). If the Mule flow under test requires interacting via a listener, then flow sources must be enabled via munit:enable-flow-sources.

8. **MUnit-test and check log:** Run the MUnit test; confirm in the log output the successful setting of the input data.

Assert the output of executing Mule flows

In this section, you add assertions to your unit test to confirm that the output of check-in-by-pnr, the flow-under-test, is as expected.

9. **Determine expected test output:** Open **main.xml** and explore the output created by **check-in-by-pnr**.
10. **Assert test output:** In **main-test-suite.xml** add to the happy-path test for check-in-by-pnr an assertion of the expected payload:

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

```
<munit:test name="..." description="...">
  <munit:execution>...</munit:execution>
  <munit:validation>
    <munit-tools:assert-that
      expression="#[payload]"
      is="#[MunitTools::equalTo({paymentID: 'PAY-
1AKD7482FAB9STATKO'})]"
    />
  </munit:validation>
</munit:test>
```

Note: You construct in-line an object to compare to the payload. This could also have been read from a file.

11. **MUnit-test:** Run the MUnit test; this should succeed and all tests should pass.
12. **Maven-build:** In a command-line interface, run a Maven build including the MUnit test suite as before; this should succeed and all unit tests should pass:

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

Walkthrough 4-3: Mock and spy external dependencies

The current implementation of check-in-papi does not actually invoke any external dependencies, such as downstream APIs. But before this will be done in later walkthroughs, it is important that the check-in-papi unit tests are first prepared so that external dependencies are not actually invoked from unit tests. Instead, it is good practice that unit tests substitute all external dependencies with MUnit mocks, so that the unit tests run independently of external dependencies, and therefore also run in environments that do not provide these dependencies. Furthermore, unit tests should also validate data that would have been sent to, and — to a lesser degree — returned from external dependencies, but is instead actually exchanged with the corresponding mock: this validation is achieved using MUnit spies.

In this walkthrough, you implement a basic MUnit test suite for the core check-in functionality of check-in-papi, encapsulating the future invocation of downstream System APIs in private Mule flows and then mocking and spying on the invocation of those Mule flows. You do not only cover the happy path but also a simple exception path of this use case.

Importantly, if integration logic is properly encapsulated in small Mule flows, then mocking and spying on the invocation of Mule flows is by far the most important mocking technique needed.

You initially follow a Test-Driven Development (TDD) approach. This means you assume that integration logic has been implemented and first write the unit test that depends on that integration logic. After the unit test fails as expected, you actually implement the integration logic in the simplest possible way, such that the unit test then passes. In later parts of this walkthrough, and in future walkthroughs, you take a more pragmatic, not strictly TDD-based approach, where unit tests are implemented after the integration logic they test.

You will:

- Mock and verify the invocation of Mule flows.
- Validate data exchanged with invoked Mule flows.
- Refactor MUnit tests to remove code smells.
- Mock and require errors raised by the invocation of Mule flows.

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/devprd/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/devprd/module04/wt4-2_solution.

Mock and verify the invocation of Mule flows

In this section, you assume the existence of three private Mule flows that invoke the three System APIs on which check-in-papi depends. You set-up MUnit mocks for the invocation of those Mule flows and assert that the mocks are indeed called. You then confirm that the unit test fails as expected, add the Mule flows and their invocations, and finally confirm that the unit test now passes.

1. **Study System APIs invoked by check-in-papi:** Review 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-by-pnr.
2. **Assume flows invoking System APIs:** Assume that check-in-by-pnr invokes the following three Mule flows, which in turn invoke the aforementioned System APIs: check-in-flights-management, register-passenger-data, and create-payment-for-bags.
3. **Mock flow invocations:** Add to **main-test-suite.xml** in the **<munit:behavior />** section of the happy-path test for check-in-by-pnr three MUnit mocks for the invocation of these three private Mule flows via <flow-ref /> elements, returning empty payloads for now:

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

```
<munit:behavior>
  <munit-tools:mock-when processor="flow-ref">
    <munit-tools:with-attributes>
      <munit-tools:with-attribute
        attributeName="name"
        whereValue="check-in-flights-management" />
    </munit-tools:with-attributes>
    <munit-tools:then-return>
      <munit-tools:payload
        value="#[output application/json --- {}]"
      />
    </munit-tools:then-return>
  </munit-tools:mock-when>

  <munit-tools:mock-when processor="flow-ref">
    <munit-tools:with-attributes>
      <munit-tools:with-attribute
        attributeName="name"
        whereValue="register-passenger-data" />
```

```
    </munit-tools:with-attributes>
    <munit-tools:then-return>
      <munit-tools:payload
        value="#[output application/json --- {}]"
      />
    </munit-tools:then-return>
  </munit-tools:mock-when>

  <munit-tools:mock-when processor="flow-ref">
    <munit-tools:with-attributes>
      <munit-tools:with-attribute
        attributeName="name"
        whereValue="create-payment-for-bags" />
    </munit-tools:with-attributes>
    <munit-tools:then-return>
      <munit-tools:payload
        value="#[output application/json --- {}]"
      />
    </munit-tools:then-return>
  </munit-tools:mock-when>
</munit:behavior>
```

Note: The MUnit mock configuration mirrors the invocation of a private Mule flow with <flow-ref />.

Note: When a <flow-ref /> matching one of these mocks is executed, the mock directly returns the configured payload instead of invoking the Mule flow.

Note: The mocked payloads are not currently used so you can return empty placeholder JSON payloads.

4. **Assert number of flow invocations:** Add to the **<munit:validation />** section of the happy-path test for check-in-by-pnr three **<munit-tools:verify-call />** elements that assert that each Mule flow is called exactly once:

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

```
<munit:validation>
  <munit-tools:verify-call times="1" processor="flow-ref">
    <munit-tools:with-attributes>
      <munit-tools:with-attribute
        attributeName="name"
        whereValue="check-in-flights-management" />
    </munit-tools:with-attributes>
```

```
  </munit-tools:verify-call>

  <munit-tools:verify-call times="1" processor="flow-ref">
    <munit-tools:with-attributes>
      <munit-tools:with-attribute
        attributeName="name"
        whereValue="register-passenger-data" />
    </munit-tools:with-attributes>
  </munit-tools:verify-call>

  <munit-tools:verify-call times="1" processor="flow-ref">
    <munit-tools:with-attributes>
      <munit-tools:with-attribute
        attributeName="name"
        whereValue="create-payment-for-bags" />
    </munit-tools:with-attributes>
  </munit-tools:verify-call>

  <munit-tools:assert-that expression="#[payload]" is="..." />
</munit:validation>
```

Note: The mocks and <munit-tools:verify-call /> elements are formally independent, but follow the same logic and syntax and augment each other naturally. Each could be used without the other, though.

5. **MUnit-test:** Run the MUnit test suite; all tests should run without error but the happy-path test for check-in-by-pnr should fail due to an assertion violation.

   Note: The flow-under-test (check-in-by-pnr) is invoked, but its implementation does not actually call the three Mule flows verified by the <munit:validation /> section of the MUnit test.

6. **Make test pass:** To **main.xml** add the three missing Mule flows and invoke them from check-in-by-pnr without any special consideration for inputs and outputs:

   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>...</ee:set-payload>
    </ee:message>
  </ee:transform>
</flow>
<flow name="check-in-flights-management">
  <logger />
</flow>
<flow name="register-passenger-data">
  <logger />
</flow>
<flow name="create-payment-for-bags">
  <logger />
</flow>
```

Note: You leave the setting of the payload in check-in-by-pnr to a hard-coded value in place.

Note: In true TDD style, this is the simplest possible integration logic that will make the test pass, but it is not a functional implementation of check-in-by-pnr.

7. **MUnit-test:** Run the MUnit test suite; this should succeed and all tests should now pass.

   Note: The test so far makes no use of the placeholder payloads returned by the three mocks.

Validate data exchanged with invoked Mule flows

In this section, you spy on and validate the data that would be passed by check-in-by-pnr to one of the private Mule flows, create-payment-for-bags. In actual fact, the invocation of create-payment-for-bags is intercepted by the corresponding mock set-up previously. You start by capturing the payload sent to check-in-by-pnr in a variable which will be passed to create-payment-for-bags, and then validate the content of that variable upon invocation of create-payment-for-bags using an MUnit spy. Using the same spy you could also validate the data returned by create-payment-for-bags — or rather, the corrsponding mock.

8. **Store input payload:** In **main.xml**, store the payload sent to **check-in-by-pnr** into a variable:

   main.xml of check-in-papi

   ```
   <set-variable variableName="checkIn" value="#[payload]" />
   ```

   Note: This makes it explicit that the payload sent to check-in-by-pnr is a check-in command, and it also isolates that input data from future changes to the payload.

Note: This variable will be the input to create-payment-for-bags — it forms part of the contract with create-payment-for-bags.

9. **Assert input to flow invocation:** Add an MUnit spy to the **<munit:behavior />** section of the happy-path test for check-in-by-pnr, after the mocks, to assert this variable value at the point of invoking create-payment-for-bags:

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

```
<munit:behavior>
  ...
  <munit-tools:spy processor="flow-ref">
    <munit-tools:with-attributes >
      <munit-tools:with-attribute
        attributeName="name"
        whereValue="create-payment-for-bags" />
    </munit-tools:with-attributes>
    <munit-tools:before-call>
      <munit-tools:assert-that
        expression="#[vars.checkIn]"

is="#[MunitTools::equalTo(read(MunitTools::getResourceAsString('json/c
heck-in-by-pnr-request.json'), 'application/json'))]"
      />
    </munit-tools:before-call>
  </munit-tools:spy>
</munit:behavior>
```

   Note: The assertion value is the exact same expression that was used to set the payload before invoking check-in-by-pnr from the unit test.

   Note: In the same spy you could also validate the output returned by the create-payment-for-bags mock.

   Note: The mock and spy elements are formally independent, but follow the same logic and syntax and augment each other naturally. Each could be used without the other, though.

   Note: MUnit spies contain assertions but are configured in the <munit:behavior /> section, not the <munit:validation /> section.

10. **Homework: Assert input to remaining flow invocations:** Set-up spies with input data validation for the invocation of the other two mocked Mule flows, check-in-flights-management and register-passenger-data.

11. **MUnit-test:** Run the MUnit test suite; this should succeed and all tests should now pass.

Refactor MUnit tests to remove code smells

In this section, you inspect the MUnit test suite of check-in-papi and discover that it is not of the desired code quality. Recognizing that test code must be maintained just like production code, you refactor the test code to raise its quality while keeping its functionality unchanged — the definition of refactoring. You apply two refactoring techniques: shortening a long Mule flow by extracting parts of its integration logic into one or more private Mule flows or subflows; and extracting duplicate DataWeave variables and functions into a DataWeave module.

12. **Identify code smells:** Inspect **main-test-suite.xml** of check-in-papi and identify code smells.

    Note: At the very least, you should identify duplicate Mule app code, including duplicate DataWeave code, and overly long Mule flows.

13. **Extract mock config:** Extract the set-up of all mocks into a new subflow:

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

```
<munit:test name="check-in-by-pnr-happy-path-test">
  <munit:behavior>
    <flow-ref name="setup-happy-sapi-mocks" />
    ...
  </munit:behavior>
  <munit:execution>...</munit:execution>
  <munit:validation>...</munit:validation>
</munit:test>

<sub-flow name="setup-happy-sapi-mocks">
  <munit-tools:mock-when processor="flow-ref">
    ...
  </munit-tools:mock-when>
  ...
</sub-flow>
```

14. **Extract spy config:** Extract the set-up of all spies into a new subflow:

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

```
<munit:test name="check-in-by-pnr-happy-path-test">
  <munit:behavior>
    ...
    <flow-ref name="spy-all-mocks" />
```

```
    ...
  </munit:behavior>
  <munit:execution>...</munit:execution>
  <munit:validation>...</munit:validation>
</munit:test>

<sub-flow name="spy-all-mocks">
  <munit-tools:spy processor="flow-ref">
    ...
  </munit-tools:spy>
  ...
</sub-flow>
```

15. **Extract setting input data:** Extract the setting of the input data for check-in-by-pnr into a new subflow:

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

```
<munit:test name="check-in-by-pnr-happy-path-test">
  <munit:behavior>...</munit:behavior>
  <munit:execution>
    <flow-ref name="set-check-in-event" />
    <flow-ref name="check-in-by-pnr" />
  </munit:execution>
  <munit:validation>...</munit:validation>
</munit:test>

<sub-flow name="set-check-in-event">
  <munit:set-event>
    ...
  </munit:set-event>
</sub-flow>
```

16. **Extract call verifications:** Extract the verification of all invocations of mocked Mule flows into a new subflow:

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

```
<munit:test name="check-in-by-pnr-happy-path-test">
  <munit:behavior>...</munit:behavior>
  <munit:execution>...</munit:execution>
  <munit:validation>
    <flow-ref name="verify-all-mocks-are-called-once" />
```

```
    <munit-tools:assert-that expression="#[payload]" is="..." />
  </munit:validation>
</munit:test>

<sub-flow name="verify-all-mocks-are-called-once">
  <munit-tools:verify-call times="1" processor="flow-ref">
    ...
  </munit-tools:verify-call>
  ...
</sub-flow>
```

17. **Load test data in DataWeave module:** Copy all occurrences of test data — especially when that data is read from a file — into variables in a new DataWeave module **TestData.dwl** in **src/test/resources**:

    TestData.dwl of check-in-papi

```
import getResourceAsString from MunitTools

var pnr              = "N123"
var checkIn          = read(getResourceAsString('json/check-in-by-pnr-
request.json'), 'application/json')
var checkInByPNRResp = { paymentID: "PAY-1AKD7482FAB9STATKO" }
```

    Note: This defines a DataWeave module with the name given by the filename TestData.dwl and with global availability in the containing Mule app.

    Note: Such a DataWeave module not only encapsulates the provisioning of test data but also ensures that test data is initialized only once — including files being read only once.

    Note: Other promising ways of storing this data, such as in variables set in <munit:before-suite /> or <munit:before-test /> do not reliably work in all execution scenarios of the MUnit test suite and the flow-under-test!

18. **Replace test data with DataWeave references:** Replace all occurrences of test data in main-test-suite.xml with references to variables from this DataWeave module:

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

```
<munit:test name="check-in-by-pnr-happy-path-test">
  ...
  <munit:validation>
    <flow-ref name="verify-all-mocks-are-called-once" />
```

```
    <munit-tools:assert-that
      expression="#[payload]"
      is="#[MunitTools::equalTo(TestData::checkInByPNRResp)]"
    />
  </munit:validation>
</munit:test>

<sub-flow name="spy-all-mocks">
  ...
  <munit-tools:spy processor="flow-ref">
    <munit-tools:with-attributes>
      <munit-tools:with-attribute
        attributeName="name"
        whereValue="create-payment-for-bags" />
    </munit-tools:with-attributes>
    <munit-tools:before-call>
      <munit-tools:assert-that
        expression="#[vars.checkIn]"
        is="#[MunitTools::equalTo(TestData::checkIn)]"
      />
    </munit-tools:before-call>
  </munit-tools:spy>
</sub-flow>

<sub-flow name="set-check-in-event">
  <munit:set-event>
    <munit:payload
      value="#[output application/json --- TestData::checkIn]"
    />
    <munit:variables>
      <munit:variable
        key="PNR"
        value="#[TestData::pnr]" />
    </munit:variables>
  </munit:set-event>
</sub-flow>
```

Note: All custom DataWeave modules on the Java classpath — such as those located in src/main/resources and src/test/resources — are automatically available in DataWeave code blocks even without an explicit import. An import is only needed when references to members of these DataWeave modules should be shortened, such as by omitting the package name, or omitting the DataWeave module when referring to a member. None of this is helpful for the usages of this test data DataWeave module, so import statements are not required and therefore omitted.

19. **MUnit-test:** Run the MUnit test suite; this should succeed and all tests should still pass.

Mock and require errors raised by the invocation of Mule flows

In this section, you start testing an exception path in the invocation of create-payment-for-bags. Because this Mule flow will ultimately invoke a remote API, PayPal SAPI, it is essential that the caller of create-payment-for-bags, check-in-by-pnr, is prepared to deal with errors in create-payment-for-bags. For now you just assume that create-payment-for-bags abstracts from all underlying API invocation errors by raising a custom application error, APP:CANT_CREATE_PAYMENT, and that check-in-by-pnr propagates that error. You therefore mock the invocation of create-payment-for-bags to raise that error and define the MUnit test of check-in-by-pnr so that it requires that error to be thrown. (In a later walkthrough you will implement retries after transient errors, so that APP:CANT_CREATE_PAYMENT will be refined to signal a permanent, non-retryable error in creating a payment.)

It turns out that you can't write an MUnit test for a custom error type that is never defined (mentioned) in the Mule app under test. You therefore extend create-payment-for-bags by actually raising APP:CANT_CREATE_PAYMENT if any error occurs in this Mule flow.

20. **Add exception path test:** Add a new MUnit test to **main-test-suite.xml** for testing the exception path of **check-in-by-pnr** and require it to raise APP:CANT_CREATE_PAYMENT:

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

    ```
    <munit:test name="check-in-by-pnr-exception-path-test"
      expectedErrorType="APP:CANT_CREATE_PAYMENT">
      <munit:behavior>
        <flow-ref name="setup-happy-sapi-mocks"/>
      </munit:behavior>
      <munit:execution>
        <flow-ref name="set-check-in-event" />
        <flow-ref name="check-in-by-pnr" />
      </munit:execution>
    </munit:test>
    ```

    Note: You create mocks and set the input data for the flow-under-test as before.

    Note: With these mocks this test would not raise an error and hence fail because it expects an error.

    Note: There is no need for spies in this test scenario.

    Note: There is no need for an <munit:validation /> section because the flow-under-test will not

return normally but raise an error.

21. **Mock flow invocation to raise error:** Override the mock for the invocation of **create-payment-for-bags** to raise **APP:CANT_CREATE_PAYMENT**:

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

    ```
    <munit:behavior>
      <flow-ref name="setup-happy-sapi-mocks"/>
      <munit-tools:mock-when processor="flow-ref">
        <munit-tools:with-attributes>
          <munit-tools:with-attribute
            attributeName="name"
            whereValue="create-payment-for-bags" />
        </munit-tools:with-attributes>
        <munit-tools:then-return>
          <munit-tools:error typeId="APP:CANT_CREATE_PAYMENT" />
        </munit-tools:then-return>
      </munit-tools:mock-when>
    </munit:behavior>
    ```

    Note: This defines two mocks for the <flow-ref /> invoking create-payment-for-bags and the last one defined (the one raising the error) overrides the previous one.

    Note: This test would fail because the check-in-papi Mule app never defines APP:CANT_CREATE_PAYMENT, that is, never raises or even mentions that error type.

22. **Define error:** In **main.xml**, change **create-payment-for-bags** by making it raise APP:CANT_CREATE_PAYMENT whenever any error occurs in the Mule flow:

    main.xml of check-in-papi

    ```
    <flow name="create-payment-for-bags">
      <logger />
      <error-handler>
        <on-error-continue>
          <raise-error type="APP:CANT_CREATE_PAYMENT"/>
        </on-error-continue>
      </error-handler>
    </flow>
    ```

    Note: In its current form, create-payment-for-bags never actually raises APP:CANT_CREATE_PAYMENT, but it defines it as a custom application error.

Note: Now that the custom application error APP:CANT_CREATE_PAYMENT is defined in this Mule app it can be used in the MUnit tests implemented earlier.

Note: You don't have to change check-in-by-pnr because by default it propagates all errors.

23. **MUnit-test and check log:** Run the MUnit test suite; this should succeed, all tests should now pass, and the logs should indicate that the exception path test of check-in-by-pnr succeeded because APP:CANT_CREATE_PAYMENT was indeed raised (by the mock).

Walkthrough 4-4: Automate the creation of MUnit Tests using MUnit Test Recorder

In the current main-test-suite.xml, you focused solely on the check-in-by-pnr Mule flow, manually creating each test — including all input/output data and assertions.

In this walkthrough, you implement an MUnit test for the core functionality of the payment-approval-by-pnr Mule flow, by automatically generating the test — including its test data and assertions — using the MUnit Test Recorder.

You will:

- Setup the required Mule flows and stubs.
- Generate a pre-configured MUnit test using MUnit Test Recorder.
- Study and refactor generated artifacts.

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/devprd/module04/wt4-4_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/devprd/module04/wt4-3_solution.

Setup the required Mule flows and stubs

In this section, you create two private Mule flows that invoke the downstream System API and implement internal logic on which payment-approval-by-pnr depends.

In a previous walkthrough, you followed a modified Test-Driven Development (TDD) approach, whereby you assumed that integration logic had been implemented (although that was not actually the case) and you first wrote the unit tests that depended on that integration logic. However, MUnit Test Recorder requires the integration logic has already been implemented (at least to a certain extent) in order to generate MUnit tests. Therfore it is required to setup the intitial skeleton implementation and stub data first, in order to take advantage of MUnit Test Recorder.

1. **Study System APIs invoked by check-in-papi:** Review the Solution architecture and in particular 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 payment approval functionality being implemented in payment-approval-by-pnr and the logic required to display a boarding pass is

contained within check-in-papi without dependencies.

2. **Decide flows to implement:** Assume that payment-approval-by-pnr invokes the following two Mule flows, which in turn invoke the aforementioned PayPal SAPI or internal logic: **update-approvals** and **get-boarding-pass** respectively.
3. **Implement flows:** To **main.xml** add two Mule flows and invoke them from **payment-approval-by-pnr** moving the setting of the payload (the Transform Message component in the flow) to the new **get-boarding-pass** Mule flow and adding a new Transform Message component to return the expected status response from PayPal SAPI to the **update-approvals** Mule flow:

   main.xml of check-in-papi

```
<flow name="payment-approval-by-pnr">
  ...
  <flow-ref name="update-approvals" />
  <flow-ref name="get-boarding-pass" />
</flow>
<flow name="update-approvals">
  <ee:transform>
    <ee:message>
      <ee:set-payload><![CDATA[%dw 2.0
      output application/json
      ---
      {
        status: "Payment Executed"
      }]]></ee:set-payload>
    </ee:message>
  </ee:transform>
</flow>
<flow name="get-boarding-pass">
  <ee:transform>
    <ee:message>
      <ee:set-payload>...</ee:set-payload>
    </ee:message>
  </ee:transform>
</flow>
```

   Note: You move the setting of the payload from payment-approval-by-pnr to get-boarding-pass as this transform represents the final boarding pass to be returned.

   Note: This is the simplest possible integration logic that will allow you to generate a meaningful test, but it is not a functional implementation of payment-approval-by-pnr.

4. **Run:** Run the Mule app in Studio.
5. Invoke the API via cURL, supplying an arbitrary PNR and example JSON request body; this should return a 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
```

Generate a pre-configured MUnit test using MUnit Test Recorder

In this section, you use Studio features to record real input/output data and generate an MUnit test for the payment-approval-by-pnr Mule flow.

6. **Use the MUnit Test Recorder:** In Studio, on the **payment-approval-by-pnr** Mule flow, execute **Record test for this flow**; this should fail.
7. **Update run config:** Edit the **MUnit Test Recording** run configuration generated by Studio for the MUnit Test Recorder by adding the **encryption key** as a **VM argument**:

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

Note: These are arguments to a temporary embedded Mule runtime that executes check-in-papi for the purpose of recording MUnit tests.

Note: It is recommended to disable gatekeeper, because the uplink to API Manager is irrelevant for recording MUnit tests and might interfere with MUnit Test Recorder.

Note: Disabling gatekeeper does not stop any previously downloaded policies from being applied. If you need to, you can remove the locally cached policies from the <MULE_HOME>/policies directory.

Note: The policies directory contains any API policies automatically downloaded from Exchange when using autodiscovery or any custom offline API policies manually added to this directory.

Note: You can find the MULE_HOME location by searching for MULE_HOME in a running Studio console.

*Note: Due to a current limitation in MUnit Test Recorder, you should **apply and close** the run configuration dialogue and **not run** the Mule app from that dialogue.*

8. **Record test:** In Studio, on the **payment-approval-by-pnr** Mule flow, execute **Record test for this flow** again; this should now succeed, deploy check-in-papi to the embedded Mule runtime configured above, and start it, waiting for the payment-approval-by-pnr Mule flow to be executed.

   Note: Any trigger of Mule flow execution is supported, not just an <http:listener />.

9. **Invoke:** Invoke the API as before; this should return an 'Empty reply from server'.

   Note: Even though the invocation of Check-In PAPI and the subsequent execution of payment-approval-by-pnr succeed, the resulting HTTP response is consumed by MUnit Test Recorder and not returned to the API client.

10. **MUnit Test Recorder stops:** In Studio notice the request has been captured and the embedded Mule runtime stopped.
11. **Generate MUnit test:** Follow the MUnit Test Recorder wizard back in Studio to create **payment-approval-by-pnr-happy-path-test** in **main-test-suite.xml**.

    Note: If you want to create an exception path test for payment-approval-by-pnr, MUnit Test Recorder cannot generate tests for flows with Mule errors raised inside the Mule flow or already existing in the incoming event, not even if they are handled by an on-error-continue error handler.

12. **Mock flow invocations that depend on System APIs:** During the wizard, select the <flow-ref /> element for **update-approvals** Mule flow and choose to **Mock this processor** selecting payload to be mocked with the captured value.

    Note: The mock data is defaulted to what was captured by the MUnit Test Recorder.

    Note: You can only choose one test behaviour per processor at this point, others can be added manually as seen in previous walkthroughs.

    Note: The mock uses the doc:id attribute automatically to decide which processor to mock. It is best practice to refactor mocks to use another attribute such as the name attribute for <flow-ref /> as seen in previous walkthroughs.

13. **MUnit-test:** In Studio, run the test suite; this should pass.

Study and refactor generated artifacts

In this section, you study the generated artifacts and optionally refactor them to follow the coding conventions laid out in previous walkthroughs.

14. **Study tests:** Inspect the generated MUnit test added to **main-test-suite.xml**. Review the Behavior, Execution and Validation scopes of the test — paying close attention to the type of assertion and the input/output test data configuration.
15. **Study test resources:** Review the **src/test/resources** directory, in particular the newly created package named after the generated test.

    Note: In previous walkthroughs, you created test data files using the raw JSON payload. In contrast, the test data generated by MUnit Test Recorder are DataWeave objects.

16. **Homework: Refactor tests:** Clean-up the generated test for payment-approval-by-pnr to follow your coding conventions similar to that of check-in-by-pnr. Look at adding spies and verifications, extracting mock config and setting of input data into new or existing Mule flows.

    Note: Review your use of the previous DataWeave module to not only encapsulate the provisioning of test data but also ensure that test data is initialized only once — including files being read only once. Refactor the new generated test data into the existing DataWeave module.

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	API Manager CH2 support;upgraded dependencies;minor bug fixes;
2022-11-23	4.3.0	upgraded dependencies;Minor bug fixes;
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;Use new api-catalog cli to publish spec to Exchange;
2022-05-04	3.0.0	restructured Applying basic software engineering principles into two modules: Applying basic software engineering principles and Developing for Operational Concerns; removed domains module; 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;New step to include new metadata support in Studio when publishing template to Exchange in walkthrough 2-1;
2022-03-09	2.3.0	upgraded dependencies;Exchange Lifecycle support for SNAPSHOT dependencies; New step to deploy template in walkthrough 2-1;
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;
2021-11-08	2.1.0	upgraded dependencies;added Windows compatible commands;changed client keystore passwords;
2021-10-05	2.0.0	upgraded dependencies, incl. Studio 7.11.0; added Mule 4.4 tracing module and logging variable section to walkthrough 3-1; Removed json-logger from walkthrough 3-1; Added Exchange V3 and deployment of apps-commons and parent POMs to walkthrough 3-3; Switch to use Connected Apps instead of credentials in walkthrough 3-3 and WT 2-3;
2021-08-03	1.9.1	fixed student files

2021-07-30	1.9	emphasized extract-to-flow refactoring to capitalize on metadata assistant in walkthrough 2-1; 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
2021-03-29	1.7	upgraded dependencies, removed step in walkthrough 4-4 to manually remove local API Policy from embedded MULE_HOME, simplified domain walkthroughs, introduced new Exchange location for json logger wt3-1
2020-10-30	1.6	upgraded dependencies, added step to walkthrough 4-4 to manually remove local API Policy from embedded MULE_HOME
2020-07-31	1.5	added a wt3-1-log-performance-and-security-in-prod; new formatting; upgraded dependencies, incl. to Studio 7.6.0
2020-06-05	1.4	new naming pattern for source code distribution package affecting the setup walkthrough; simplified WT solution installation instructions and related directory layout; added walkthrough 4-4; upgraded dependencies, incl. to Studio 7.5.1
2020-05-06	1.3	corrected steps in walkthrough 3-3; adapted to new way of running Secure Properties tool in walkthrough 2-4; added <businessGroup /> to Mule Maven plugin config in WT 2-3; upgraded dependencies, incl. to Mule runtime 4.3.0 and Studio 7.5.0
2020-04-23	1.2.1	adapted the setup walkthrough to Studio bringing its own JDK; downgraded Mule runtime to 4.2.2 and APIkit to 1.3.9 because of known issues; fixed PDF formatting issues; upgraded dependencies
2020-04-09	1.2	renamed shell variable APDL2REPO to {GitRepoDirEnvVar}; put parent POM in parent directory right from the start, affecting sections of walkthroughs 2-2 and <3-3; using provided BOM rather than building it up, affecting sections of walkthroughs 2-2, 2-4, 4-1; using `mvn install` only where necessary, `mvn verify` otherwise; new walkthrough solutions/starters directory layout and build approach affecting the setup walkthrough and the installation of all walkthrough solutions and starters; set Visualizer display name in walkthrough 3-2; set <applyLatestRuntimePatch /> in walkthrough 2-3; <munit:set-event /> now done in <munit:execution /> instead of <munit:behavior /> in walkthroughs 4-2 and 4-3; apps-commons is now library-style Mule plugin instead of simple JAR, affecting sections of walkthroughs 3-3; upgraded dependencies
2020-01-15	1.1	renamed course; renamed ; upgraded dependencies
2019-12-06	1.0	first public release, based on Mule runtime 4.2.2 and Studio 7.4.1