Application Networks

Anypoint Platform Architecture

MuleSoft Training

2018-05-02

Table of Contents

Welcome To Anypoint Platform Architecture: Application Networks

Introducing the course

Course prerequisites

The *target audience* of this course are architects, especially Enterprise Architects and Solution Architects, new to Anypoint Platform, API-led connectivity and the application network approach, but experienced in other integration approaches (e.g., SOA) and integration technologies/platforms.

Prior to attending this course, students are required to get an overview of Anypoint Platform and its constituent components. This can be achieved by various means, such as

- attending the short Getting Started with Anypoint Platform course
- attending the much more thorough developer-centric courses Anypoint Platform Development: Fundamentals or MuleSoft.U Development Fundamentals
- participating in the 1-day hands-on "API-Led Connectivity Workshop" organized by MuleSoft Presales upon request

Course goals

The overarching goal of this course is to enable students to

- direct the *emergence of an effective application network* out of individual integration solutions following API-led connectivity, working with all relevant stakeholders on all levels of the organization
- create credible *high-level architecture models* for integration solutions on Anypoint Platform such that functional and non-functional requirements are likely to be met and the principles of API-led connectivity and application networks are followed

Course outline

- Module 5
- Module 6
- Module 7
- Module 8
- Module 9
- Module 10
- *Wrapping-Up Anypoint Platform Architecture: Application Networks*

How the course will work

This is a course on *Enterprise Architecture with elements of Solution Architecture*:

- It discusses topics on the scale of Enterprise Architecture, touching lightly on Business Architecture, and heavily on Application Architecture and Technology Architecture.
- It motivates and builds Enterprise Architecture from strategically important integration solutions and therefore elaborates on parts of their high-level Solution Architecture.
- It stays away from architecturally insignificant design and implementation discussions:
  - As a rule, these are all topics whose repercussions are confined to individual application components and are therefore not apparent from outside these application components.
  - When a decision affects the large-scale properties of the application network, however, it becomes architecturally significant. This is the reason why the course contains a fairly detailed discussion on strategies for invoking APIs in a fault-tolerant way (7.2).
- No code is shown, neither implementation code nor code for API specifications such as RAML definitions. However, the topic of API specifications and the features offered by RAML in this space are touched upon in several places, because they *are* important for the functioning of an application network.

This course is primarily driven by a single *case study, Acme Insurance*, and two imminent strategically important change initiatives that need to be addressed by Acme Insurance. These change initiatives provide the background and motivation for most discussions in this course.

As various aspects of the case study are addressed, the discussion naturally elaborates on the central topic of the course, i.e., *how to architect and design application networks using API-led connectivity and Anypoint Platform*.

However, the course cannot jump into architecting without any prior knowledge about Anypoint Platform, what terms like "API-led connectivity" and "application network" actually mean, and how MuleSoft and MuleSoft customers typically approach integration challenges like those faced by Acme Insurance. Therefore, Module 1 and Module 2 provide this context-setting

and introduction. Acme Insurance itself is briefly introduced already in Introducing Acme Insurance and becomes the focus of the discussion from Module 3 onwards.

As the architectural and design discussions in this course unfold, it is inevitable that opinions are expressed, solutions presented and decisions made that are somewhat ambiguous, *without a clear-cut distinction between correct or false*: such is the nature of architecture and design. A good example of this is the discussion on Bounded Context Data Model versus Enterprise Data Model in section 6.3. Students are of course encouraged to challenge the decisions made, and to decide differently in similar real-world situations. The crucial point is that the thought processes behind these architectural and design decisions are elaborated-on in the course, which creates awareness of the topic and increases understanding of the tradeoffs involved in making a decision.

Exercises, typically in the form of group discussions, are an important element of this course. But these exercises are never in the form of actually doing something, on the computer, with Anypoint Platform or any of its components. Instead, they are simply discussions that invite to reflect, with the intention of validating and deepening the understanding of a topic addressed in the course.

All architecture diagrams use ArchiMate 3 notation. A summary of that notation can be found in Appendix B.

Course materials and certification

Students receive the *Course Manual* (this document), a PDF of more than 200 pages, which contains all material presented on slides plus additional discussions and explanations.

The course manual is somewhere between a bound edition of the slides and a standalone book: it contains all slide content and enough context around this content to be much easier to consume than the slides alone would be. On the other hand, the course manual lacks some of the explanations and elaborations that a full-fledged book would be expected to contain: this additional depth is provided by the instructor when teaching the course!

MuleSoft offers a *certification* based on this course. For students fulfilling the course's prerequisites, attending class and studying the Course Manual should be sufficient for passing the exam.

Introducing Acme Insurance

The Acme Insurance organization

Acme Insurance is a *well-established, medium-sized, regional* insurance provider. They have

two lines of business (LoBs): *personal motor* insurance and *home* insurance.

Acme Insurance has recently been *acquired* by an international competitor: The New Owners. As one consequence, Acme Insurance is currently being rebranded as a *subsidiary* of The New Owners. As another important consequence, Acme Insurance's *strategy* is increasingly being *defined by The New Owners*.

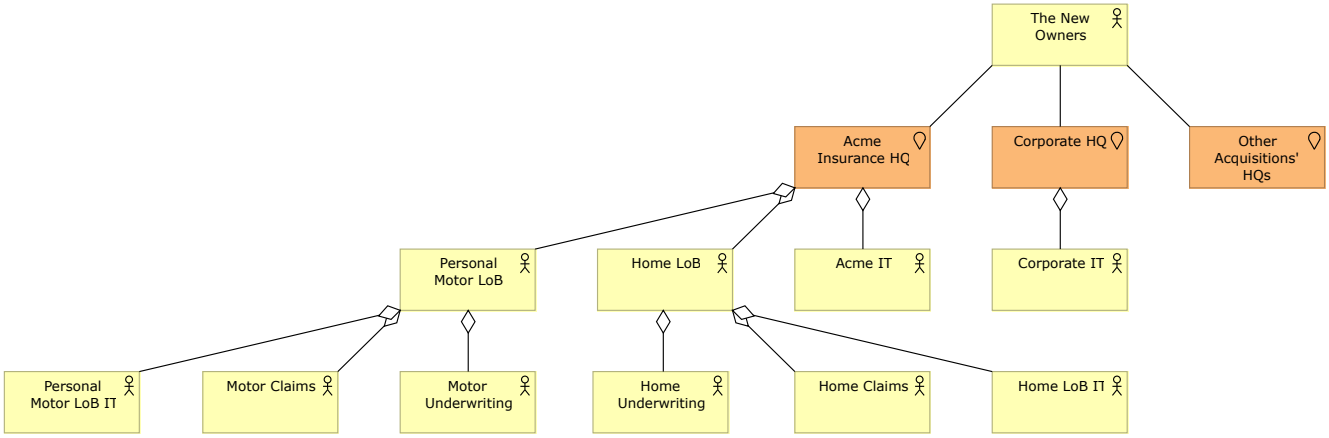

Figure 1. Relevant elements of the organizational structure of Acme Insurance.

A glimpse into Acme Insurance's baseline Technology Architecture

The baseline Technology Architecture of Acme Insurance can, at a very high level, be characterized as follows (Figure 2):

- Acme Insurance operates an IBM-centric data center with the Acme Insurance Mainframe and clusters of AIX machines
- The Policy Admin System runs on the Mainframe and is used by both Motor and Home Underwriting. However, support for Motor and Home policies was added to the Policy Admin System in different phases and so uses different data schemata and database tables
- The Motor Claims System is operated in-house on a WebSphere application server cluster deployed on AIX
- The Home Claims System is a different system, operated by an external hosting provider and accessed over the web
- Both claims systems are accessed by Acme Insurance's own Claims Adjudication team from their workstations
- Simple claims administration is handled by an outsourced call center, also via the Motor Claims System and Home Claims System

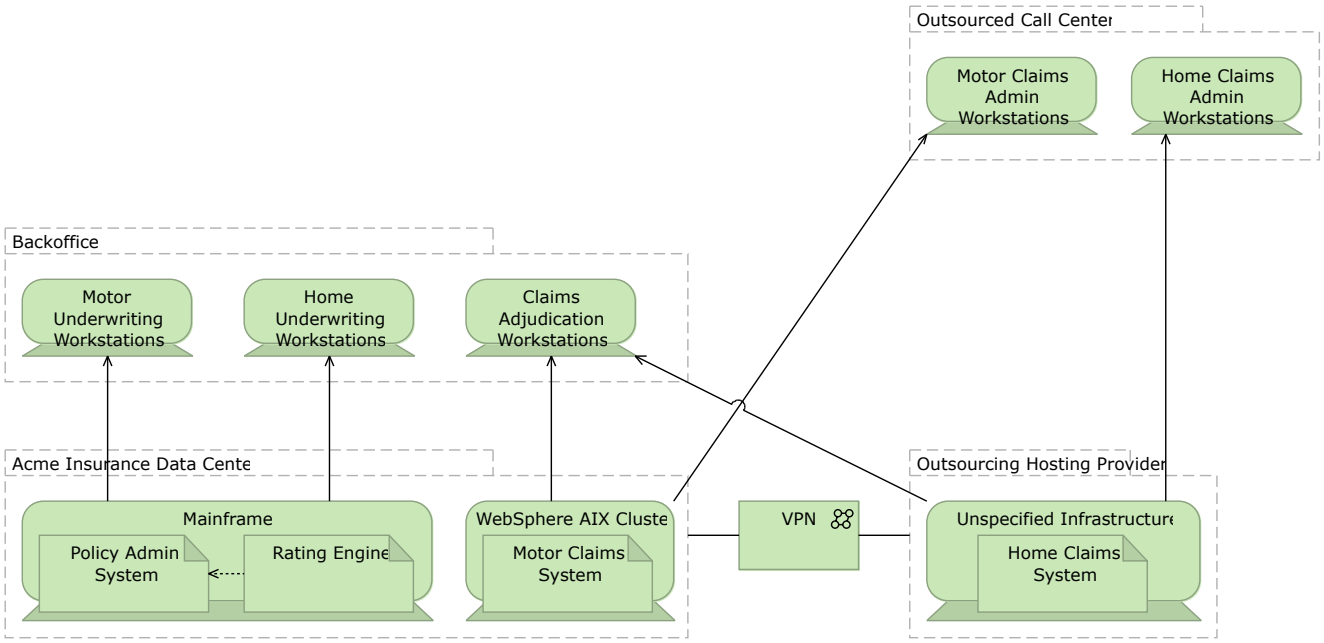

Figure 2. A small sub-set of the baseline Technology Architecture of Acme Insurance.

Beware of the two completely distinct meanings of the term *"policy"* in this course: *insurance policy* on the one hand and *API policy* on the other hand. To avoid confusion, the latter will always be referred to using the complete term *"API policy"*.

Acme Insurance's motivation to change

Under pressure from The New Owners, Acme Insurance executives embark on two immediate strategic initiatives (Figure 3):

- Open-up to external price comparison services (*"Aggregators"*) for motor insurance: This contributes to the goal of establishing new sales channels, which in turn (positively) influences the driver of increasing revenue, which is important to all management stakeholders
- Provide (minimal) *self-service* capabilities to customers: This contributes to the goal of increasing the prevalence of customer self-service, which in turn (positively) influences the driver of reducing cost, which is important to all management stakeholders as well as to Corporate IT

Not immediately relevant, but clearly on the horizon, are the following far-reaching changes:

- Replace the two bespoke *claims handling systems*, the Motor Claims System and Home Claims System, with one COTS product: This contributes to the principle of preferring COTS

solutions, which in turn contributes to Corporate IT's goal of standardizing IT systems across all subsidiaries of The New Owners

- Replace the legacy *policy rating engine* with a corporate standard: This contributes to the principle of reusing custom software (such as the corporate standard policy rating engine) where possible, which in turn contributes to Corporate IT's goal of standardizing IT systems

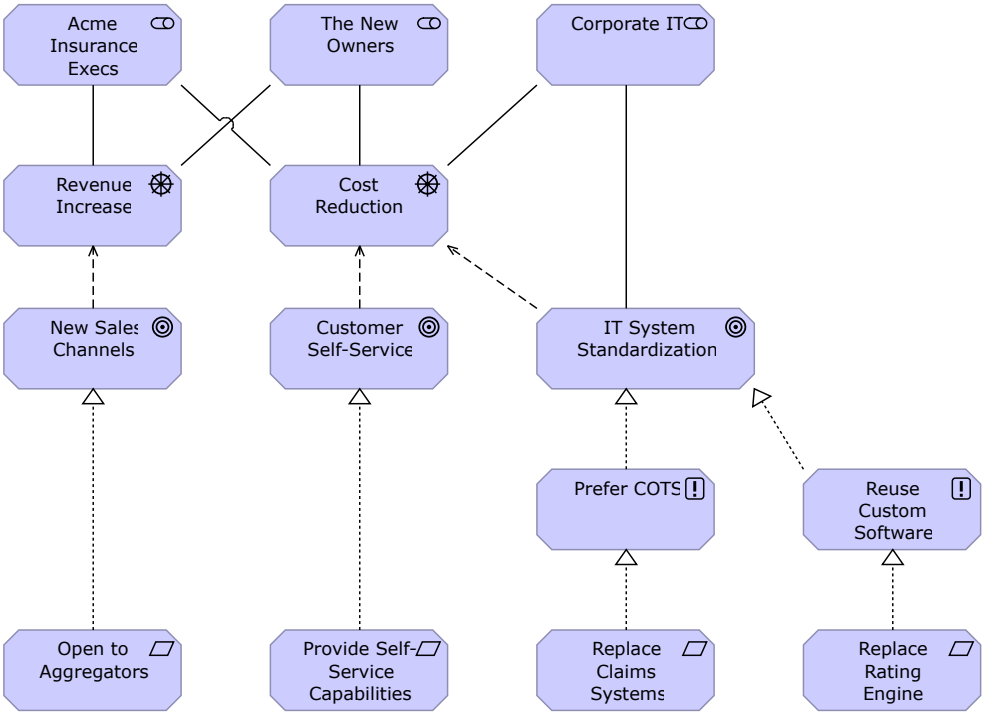

Figure 3. Acme Insurance's immediate and near-term strategic change initiatives, their rationale and stakeholders.

Module 1. Putting the Course in Context

Objectives

- Define Outcome-Based Delivery (OBD)
- Describe how this course is aligned to parts of OBD
- Use essential course terminology correctly
- Recognize the ArchiMate 3 notation subset used in this course

1.1. Understanding the course organization

1.1.1. Introducing Outcome-Based Delivery - OBD

OBD is a framework and methodology for enterprise integration delivery proposed by MuleSoft. It takes a holistic view of delivering integration capabilities, addressing

- *Business outcomes*, incl. alignment and governance of integration capability delivery
- *Technology delivery*, separating
  - cross-project *platform concerns*
  - from the *delivery of projects*
- *Organizational enablement*, specifically
  - the establishment of a *Center for Enablement* in the organization
  - the professional *IT support for Anypoint Platform* with the help of the MuleSoft support organization
  - *training* of all staff involved in delivering integration capabilities

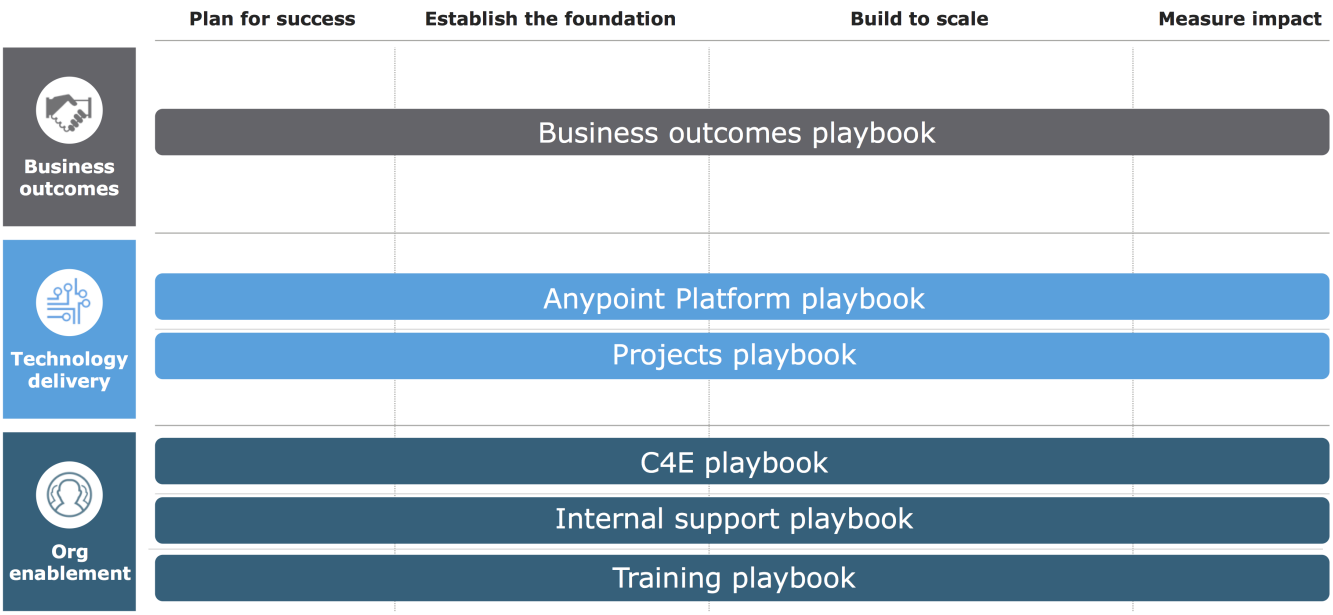

Figure 4. OBD holistically addresses all aspects of integration capability delivery into an organization.

1.1.2. Course content in the context of OBD

OBD is materialized by "playbooks", where each playbook addresses one of the dimensions of OBD and identifies activites along that dimension. This course is aligned with the *principles* of OBD and the OBD playbooks, but it does not follow the exact naming or sequential order of the playbooks' activities.

Being an *architecture course*, it focuses on these dimensions of OBD:

- Technology delivery, both from an *Anypoint Platform and projects* perspective
- Organizational enablement through the *C4E*

However:

- Iteration is at the heart of OBD, but *this course does not iterate*
  - *Every topic is discussed once*, in the light of different aspects of the case study, which would in the real world be addressed in different iterations
- OBD stresses planning, but *this course does not simulate planning activities or present plans*
- Discussion of organizational enablement and the C4E is light and mainly introduces the concept and a few ideas on how to measure the C4E's impact with KPIs

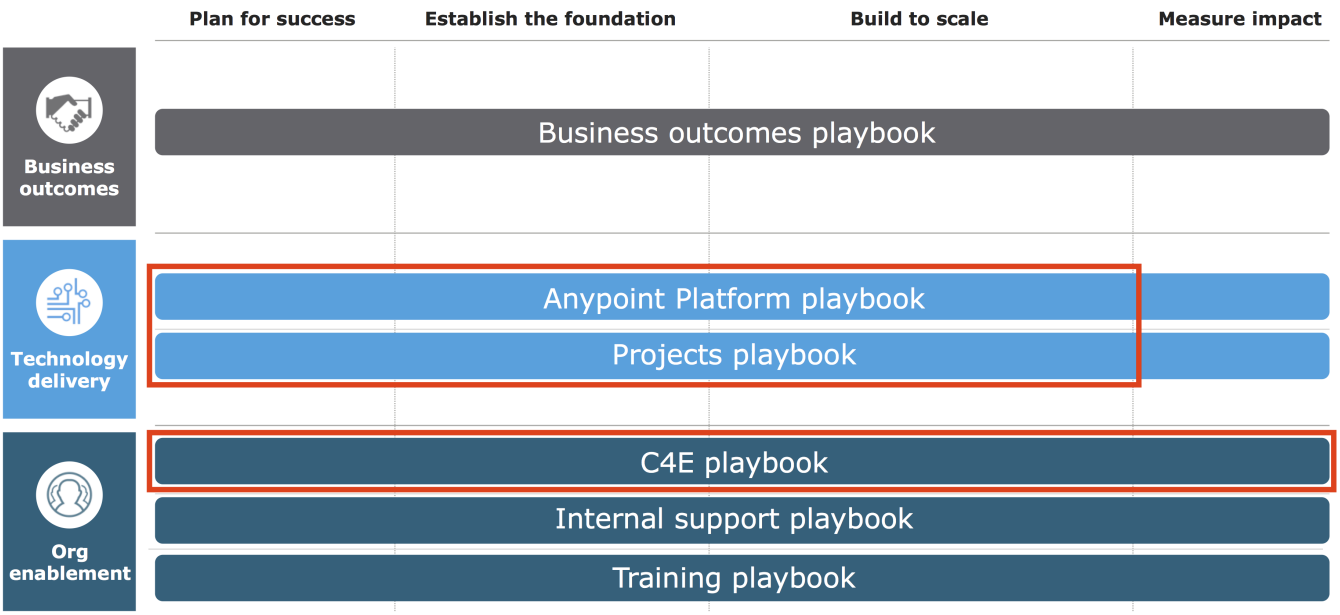

Figure 5. This course focuses on architectural aspects of technology delivery and introduces the C4E.

1.2. Understanding essential course terminology

1.2.1. Defining API

See the corresponding glossary entry, cf. Figure 6.

1.2.2. Defining API client

See the corresponding glossary entry.

1.2.3. Defining API consumer

See the corresponding glossary entry.

1.2.4. Defining API implementation

See the corresponding glossary entry.

1.2.5. Defining API-led connectivity

See the corresponding glossary entry.

1.2.6. Simplified notion of API

In colloquial usage of the term API, departing from the precise - and somewhat confining - definition given in 1.2.1, API often refers not just to the application interface but to the *combination* of

- a programmatic application interface
  - i.e., the precise meaning of API
- the application service to which this application interface provides access
- the business service realized by that application service
- the HTTP-based technology interface realizing the application interface in concrete technical terms
- and, importantly, *the application component implementing the functionality* exposed by the application interface
  - i.e., the API implementation
- See Figure 6, cf. Figure 138

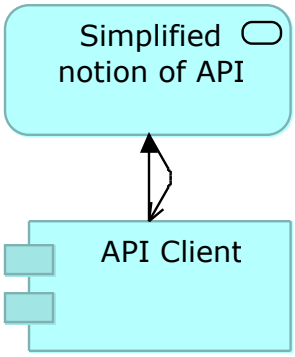

Figure 6. The simplified notion of API merges the aspects of application interface, technology interface, API implementation, application service and business service and is here represented visually as an application service element with the name of the API. An API in this simplified sense directly serves an API client and is invoked (triggered) by that API client.

This simplified notion of API is justified because in a very significant number of cases there is *exactly one instance of each of these elements per API*. Indeed, striving for a 1-to-1 relationship between API as application interface and API implementation in particular is usually advisable as it helps combat the complexity of large application networks. This is also the approach followed in this course.

Using this simplified notion of API:

- Experience APIs are shown invoking Process APIs and Process APIs are shown invoking System APIs, although, in reality, it is only the API implementation of the Experience API that depends on the technology interface of the Process API and, at runtime, through that interface, invokes the API implementation of the Process API; and it is only the API implementation of the Process API that depends on the technology interface of the System API and, at runtime, through that interface, invokes the API implementation of the System API.
- It is possible to say that an “API is deployed to a runtime” when in reality it is only the API implementation (the application component) that is deployable.

In other contexts (not in this course), the terms "service" or "microservice" are used for the same purpose as the simplified notion of API.

When the simplified notion of API is dominant then the pleonasm "API interface" is sometimes used to specifically address the interface-aspect of the API in contrast to its implementation-aspect.

For instance:

- If the Auto policy rating API were just exposed over one HTTP-based interface, e.g., a JSON/REST interface, then the simplified notion of this API would comprise:
  - Technology interface: Auto policy rating JSON/REST programmatic interface
  - Application interface: Auto policy rating
  - Business service: Auto policy rating
  - The application component (API implementation) implementing the exposed functionality
- However, since the Auto policy rating API (in the strict sense of application interface) is also realized by a second technology interface, the Auto policy rating SOAP programmatic interface, it is not clear whether the simplified notion of API comprises both technology interfaces or not. It is therefore preferred, in complex cases such as this, to use the term API only in its precise sense, i.e., as a special kind of application interface as defined in 1.2.1.

1.2.7. Defining application network

See the corresponding glossary entry.

Summary

- Outcome-based Delivery (OBD) is a holistic framework and methodology for enterprise integration delivery proposed by MuleSoft, addressing

    - Business outcomes
    - Technology delivery in the form of platform delivery and delivery of projects
    - Organizational enablement through the Center for Enablement (C4E), support for Anypoint Platform and training
- This course is aligned with the technology delivery and C4E dimensions of OBD
- An API is an application interface, typically with a business purpose, to programmatic API clients using HTTP-based protocols
- Sometimes the term "API" also denotes the API implementation, i.e., the underlying application component that implements the API’s functionality
- API-led connectivity is a style of integration architecture that prioritizes APIs and assigns them to three tiers
- Application network is the state of an Enterprise Architecture that emerges from the application of API-led connectivity and fosters governance, discoverability, consumability and reuse

Module 2. Introducing MuleSoft, the Application Network Vision and Anypoint Platform

Objectives

- Articulate MuleSoft’s mission
- Explain MuleSoft’s proposal for closing the increasing IT delivery gap
- Describe the capabilities and high-level components of Anypoint Platform

2.1. Introducing MuleSoft

2.1.1. MuleSoft's mission

MuleSoft has been on a mission since 2003 to connect the world’s applications, data and devices. MuleSoft started solving the classic on-premises backend integration problems with an open source, light-weight ESB: Mule ESB. Since then, MuleSoft has grown to provide a comprehensive platform to help companies with today’s business challenges.

MuleSoft mission statement:

> To connect the world’s applications, data and devices to transform business

MuleSoft’s mission is to enable companies to transform and compete in a vastly changing digital world.

2.1.2. The story behind the name

Frustrated by integration "donkey work", Ross Mason, VP Product Strategy, founded the open source Mule project in 2003. He created a new platform that emphasized ease of development with quick and efficient assembly of components, instead of custom-coding by hand.

2.1.3. MuleSoft customers

- More than 175,000 developers worldwide
- More than 1100 customers
- 3 of the top 10 global auto companies
- 5 of the top 10 global banks
- 2 of the top 5 global retailers

2.1.4. Company overview

- Company
  - Founded in 2006, HQ: San Francisco
  - 22 offices world-wide
  - 1300+ employees worldwide
  - Nearly 70% new subscription bookings driven by APIs, mobile and SaaS integration
- Products
  - MuleSoft's Anypoint Platform addresses on-premises, cloud and hybrid integration use cases with scale and ease-of-use
  - Subscription model with various packaged options to serve different use cases

2.2. Introducing the application network vision

This module aims to give a simplified, easily understandable introduction to concepts that will be developed and applied in much more detail and depth in the remainder of this course - API-led connectivity and application networks. As such, this section contains a somewhat superficial comparison to SOA, does not make use of the Acme Insurance case study and intentionally glosses over more subtle but important aspects - which will be discussed in later course modules.

2.2.1. 10-year turnover in Fortune 500 companies

There is a convergence of forces – mobile, SaaS, Cloud, Big Data, IoT and Social - which is causing a major need for a change in speed to remain competitive and/or lead the market.

It used to be that 80% of companies on the Fortune 500 would still be there after a decade. Today, with these forces, enterprises have no better than a 1-in-2 chance of remaining in the Fortune 500.

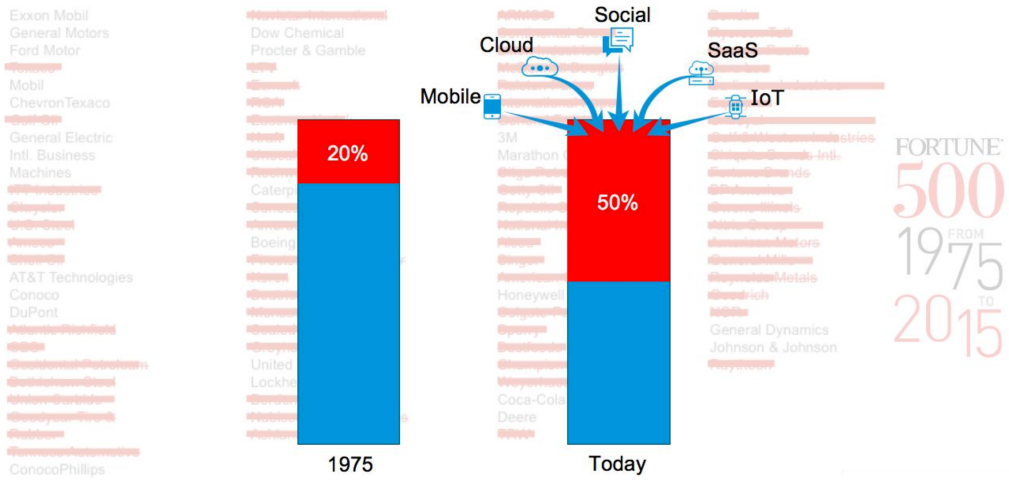

Figure 7. Fortune 500 10-year turnover.

To succeed, companies need to be driving a very different clock speed and embrace change; change has become a constant. Successful companies are leveraging the aforementioned forces to be competitive and in some cases dominate their markets.

Business is pushing to move at much faster speeds than IT and technology are able to. Technology and IT are holding back business rather than enabling it.

MuleSoft is helping 1100+ enterprise customers undertake transformative changes, and so have a unique perspective on the market. MuleSoft's customers' CIOs say that they have to achieve 3-5 times higher speed as a business within the next 2-3 years to compete.

2.2.2. Digital pressures create a widening IT delivery gap

While IT delivery capacity remains nearly constant, the compound effect of the aforementioned forces (mobile, SaaS, Cloud, Big Data, etc.) leads to ever-increasing demands on IT. The result is a widening IT delivery gap.

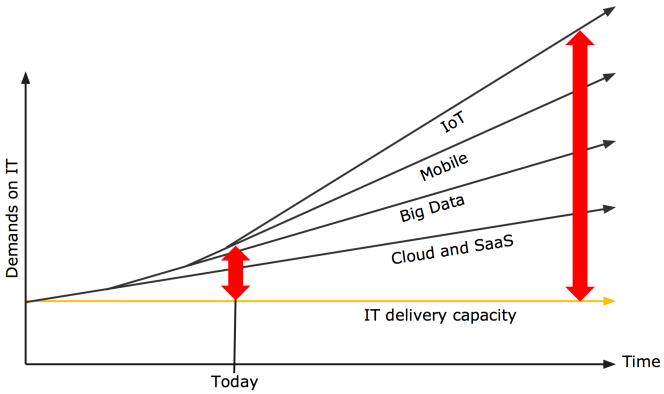

Figure 8. Illustration of the widening IT delivery gap caused by various forces.

2.2.3. Current responses are insufficient

Current responses to that widening IT delivery gap are not sufficient:

- Working harder is not sustainable
- Over-outsourcing exacerbates the situation
- Agile and DevOps are important and helpful but not sufficient

2.2.4. Learning from other industries

Companies such as

- McDonald’s
- Subway
- Marriot
- Amazon

have shown the value of these approaches:

- Create scale through *reuse*
- Enable *self-service*
- Encourage *innovation "at the edge"*
- Promote *quality*
- Retain *visibility and control*

2.2.5. Closing the delivery gap with constant IT delivery capacity

MuleSoft proposes an *IT operating model* that takes these successful approaches from other industries on board:

- Even with constant IT delivery capacity, IT can empower "the edge" - i.e., LoB IT and developers - by creating assets and helping to create assets they require.
- Consumption of those assets and the innovation enabled by those assets can then occur outside of IT, at the edge, and therefore grow at a considerably faster rate than IT delivery capacity itself.
- In this way, the ever-increasing demands on IT can be met even though IT delivery capacity stays approximately constant.

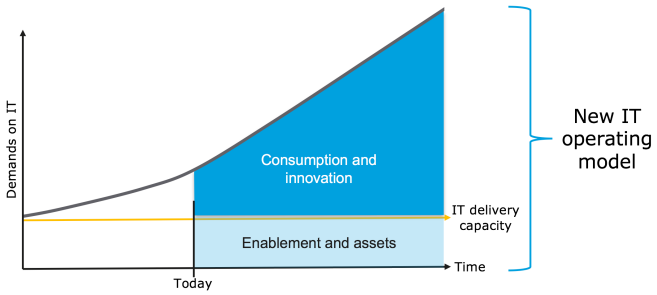

Figure 9. How MuleSoft's proposal for an IT operating model that distinguishes between enablement and asset production on the one hand, and consumption of those assets and innovation on the other hand, allows the increasing demands on IT to be met at constant IT delivery capacity.

2.2.6. An operating model that emphasizes consumption

IT is now the steward of this operating model with a virtuous cycle in which IT produces reusable assets and enables consumption of those assets by making them discoverable and self-served, by LoB IT and developers, while monitoring feedback and usage.

Central IT needs to move away from trying to deliver all IT projects themselves and start building reusable assets to enable the business to deliver some of their own projects.

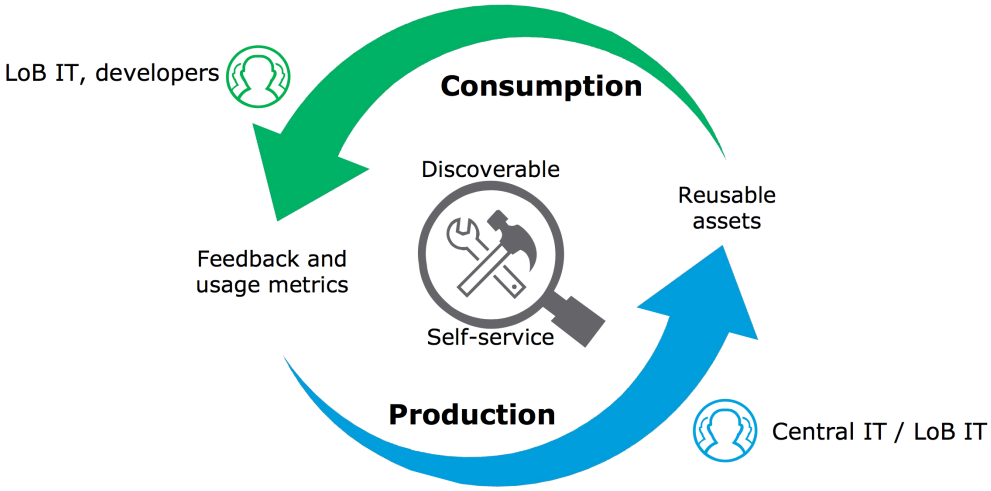

Figure 10. An IT operating model that emphasizes the consumption of assets by LoB IT and developers as much as the production of these assets.

- The key to this strategy is to *emphasize consumption* as much as production
- Traditional IT approaches (for example, SOA) focused exclusively on production for the delivery of projects
- In this operating model, IT changes its mindset to think about *producing assets that will be consumed by others in lines of business*

- The *assets need to be discoverable* and *developers need to be enabled* to self-serve them in projects
- The virtuous part of the cycle is to get active feedback from the consumption model along with usage metrics to inform the production model

2.2.7. The modern API as a core enabler of this operating model

In the proposed operating model, the organization packages-up core IT assets and capabilities in the modern API.

The modern *API is a product* and it has its own software development lifecycle (SDLC) consisting of design, test, build, manage, and versioning and it comes with thorough documentation to *enable its consumption*.

- Modern APIs adhere to standards (typically HTTP and REST), that are developer-friendly, easily accessible and understood broadly
- They are treated more like products than code
- They are designed for consumption for specific audiences (e.g., mobile developers)
- They are documented and versioned
- They are secured, governed, monitored and managed for performance and scale

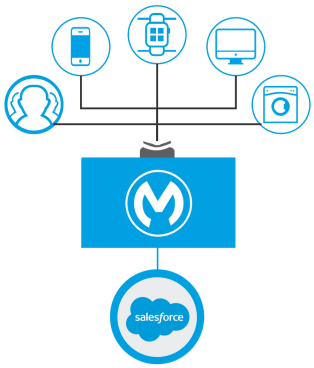

Figure 11. Visualization of how a modern API, productized with the API consumer in mind, gives various types of API clients access to backend systems.

This approach is different from SOA in these respects:

- Modern APIs are easier to consume than WS-* web services
- SOA was built by IT and for the consumption by IT only
- The technology was hard: One can’t just give the extended organization access to WS-* web services, they wouldn’t be able to use it

- Not discoverable and consumable by broad developer teams within the ecosystem, including mobile developers
- This left organizations with IT still being the bottleneck
- However, when an organization has a good SOA strategy in place, it will accelerate the value of what API-led connectivity provides: after all, both SOA and API-led connectivity revolve around services

2.2.8. The API-led connectivity approach

API-led connectivity is a methodical way to connect data to applications through a series of reusable and purposeful modern APIs that are each developed to play a specific role – unlock data from systems, compose data into processes, or deliver an experience.

API-led connectivity provides an approach for connecting and exposing assets through APIs. As a result, these assets become discoverable through self-service without losing control.

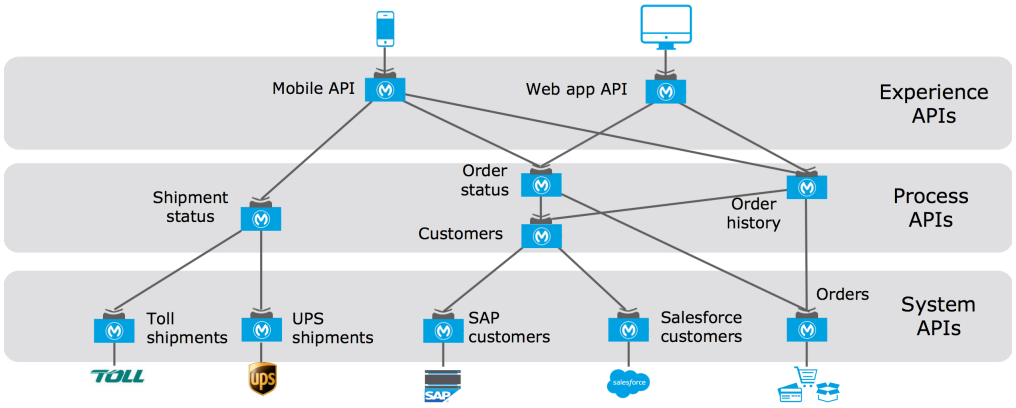

Figure 12. An example of an integration architecture following API-led connectivity principles by assigning APIs to the tiers of System APIs, Process APIs and Experience APIs.

- System APIs: In the example, data from SAP, Salesforce and ecommerce systems is unlocked by putting APIs in front of them. These form a System API tier, which provides *consistent, managed, and secure access to backend systems*.
- Process APIs: Then, one builds on the System APIs by combining and streamlining customer data from multiple sources into a "Customers" API (breaking down application silos). These Process APIs take core assets and combines them with some business logic to *create a higher level of value*. Importantly, these higher-level objects are now useful assets that can be further reused, as they are APIs themselves.
- Experience APIs: Finally, an API is built that brings together the order status and history, delivering the data specifically needed by the Web app. These are Experience APIs that are *designed specifically for consumption by a specific end-user app or device*. These APIs *allow app developers to quickly innovate* on projects by consuming the underlying assets without

having to know how the data got there. In fact, if anything changes to any of the systems of processes underneath, it may not require any changes to the app itself.

With API-led connectivity, when tasked with a new mobile app, there are now reusable assets to build on, eliminating a lot of work. *It is now much easier to innovate.*

2.2.9. Focus and owners of APIs in different tiers

The organizational approach to API-led connectivity empowers the entire organization to access their best capabilities in delivering applications and projects through modern APIs that are *developed by the teams that are best equipped to do so* due to their roles and knowledge of the systems they unlock, or the processes they compose, or the experience they'd like to offer in the application.

- *Central IT* produces reusable assets, and in the process *unlocks key systems*, including legacy applications, data sources, and SaaS apps. Decentralizes and democratizes access to company data. These assets are created *as part of the project delivery process*, not as a separate exercise.
- *LOB IT and Central IT* can then reuse these API assets and *compose process level information*
- *App developers* can *discover and self-serve* on all of these reusable assets, *creating the experience-tier* of APIs and ultimately the end-applications

It is critical to connect the three tiers as driving the production and consumption model with reusable assets, which are discovered and self-served by downstream IT and developers.

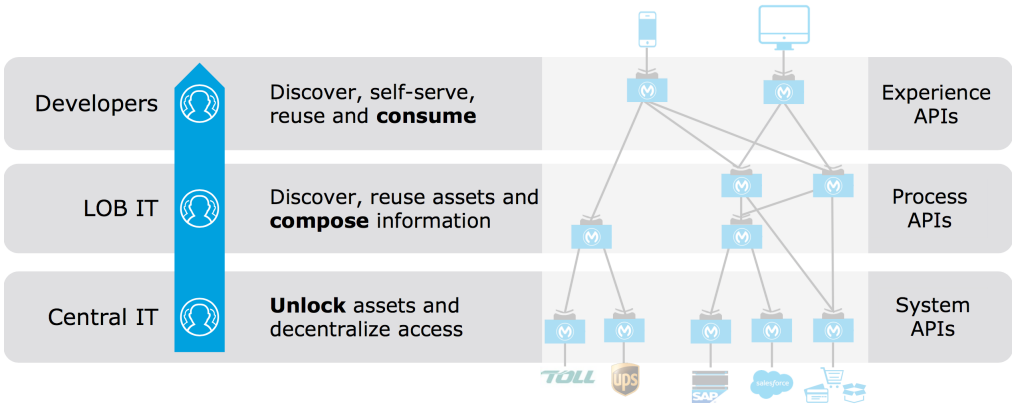

Figure 13. The APIs in each of the three tiers of API-led connectivity have a specific focus and are typically owned by different groups.

- API-led connectivity is not an architecture in itself, it is an approach to encourage to unlock assets and drive reuse and self service

- API-led connectivity is not just about technology, but is a way to organize people and processes for efficiencies within the organization
- The APIs depicted in those tiers are actually building blocks that encapsulate connectivity, business logic, and an interface through which others interact. These building blocks are productized, fully tested, automatically governed, and fully managed with policies.
- Easy publishing and discovery of APIs is crucial

2.2.10. Anypoint Platform and organizational enablement

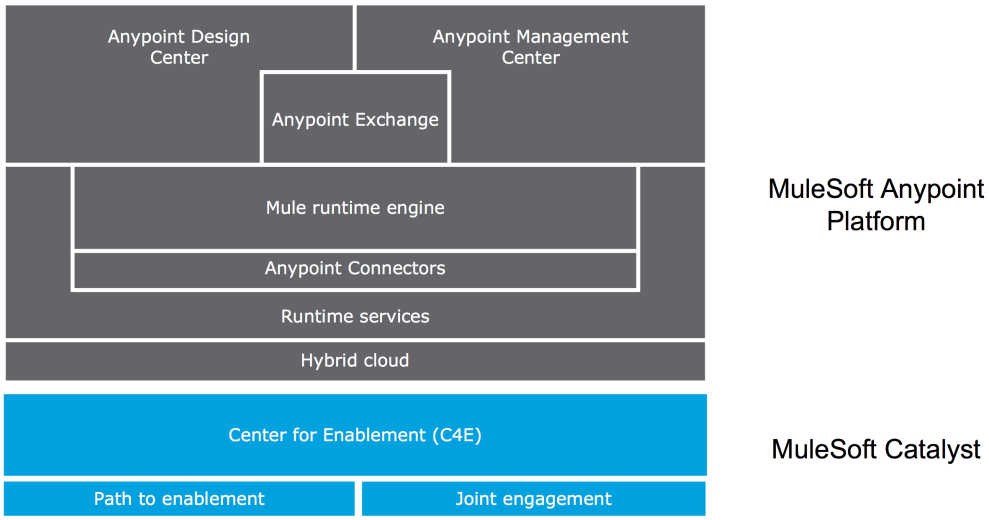

Figure 14. The components of Anypoint Platform sit on a foundation of organizational enablement, of which the C4E is one important element. MuleSoft offers various engagement models to help enable organizations.

2.2.11. Application landscape at the start of the journey towards the application network

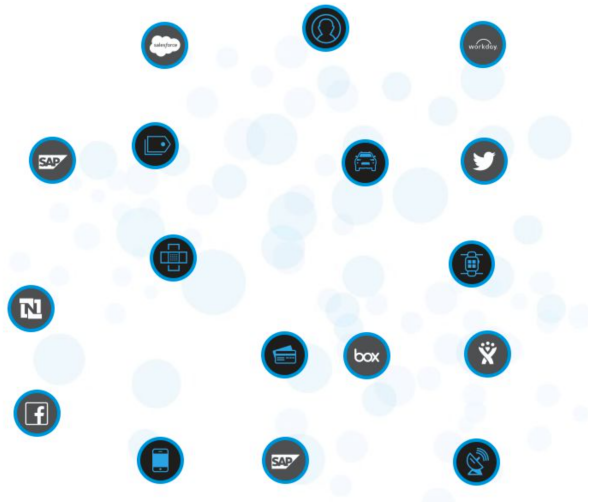

Figure 15. Isolated backend systems before the first project following API-led connectivity.

2.2.12. Every project adds value to the application network

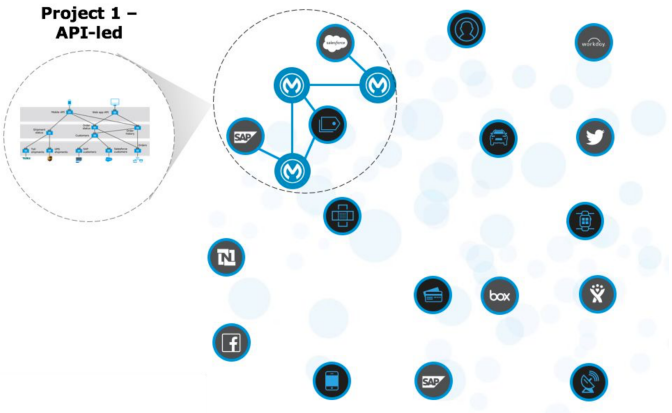

Figure 16. Every project following API-led connectivity not only connects backend systems but contributes reusable APIs and API-related assets to the application network.

2.2.13. The application network emerges

An application network is built ground-up over time and *emerges as a powerful by-product of API-led connectivity*. See Figure 141.

The assets in an application network should be

- discoverable
- self-service
- consumable by the broader organization

Nodes in the application network create new business value.

The application network is *recomposable*: it is built for change because it "bends but does not break".

2.3. Highlighting important capabilities needed to realize application networks

2.3.1. High-level technology delivery capabilities

To perform API-led connectivity an organization has to have a certain set of capabilities, some of which are provided by Anypoint Platform:

- API design and development
  - i.e., the design of APIs and the development of API clients and API implementations
- API runtime execution and hosting
  - i.e., the deployment and execution of API clients and API implementations with certain runtime characteristics
- API operations and management
  - i.e., operations and management of APIs and API policies, API implementations and API invocations
- API consumer engagement
  - i.e., the engagement of developers of API clients and the management of the API clients they develop

As was also mentioned in the context of OBD in 1.1.1, these capabilities are to be deployed in such a way as to contribute to and be in alignment with the organization’s (strategic) goals, drivers, outcomes, etc..

These technology delivery capabilities are furthermore used in the context of an (IT) operating model that comprises various functions.

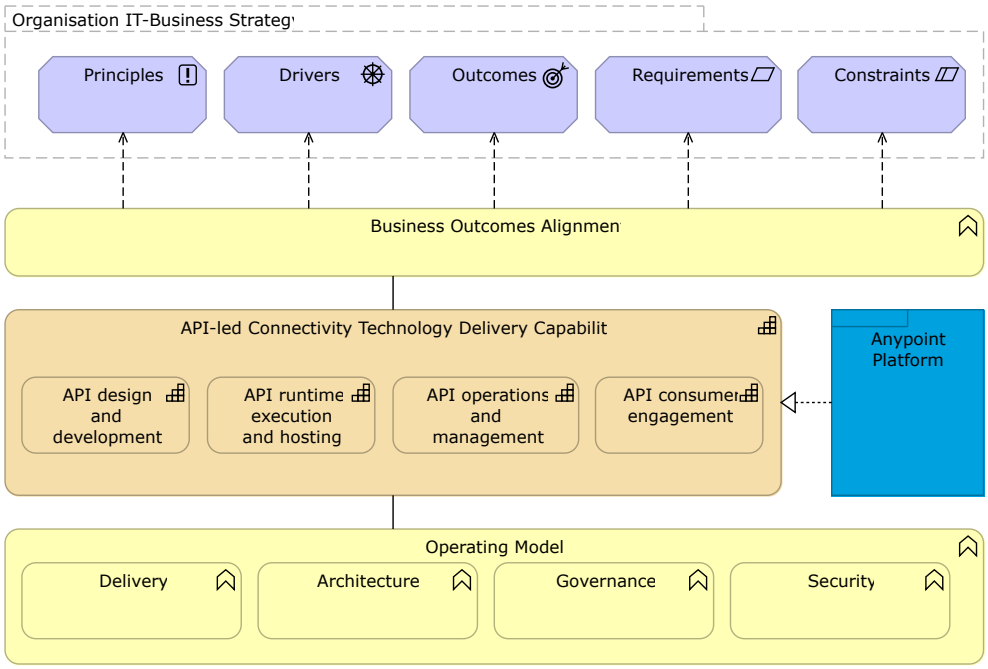

Figure 17. A high-level view of the technology delivery capabilities provided by Anypoint Platform in the context of various relevant aspects of the Business Architecture of an organization.

2.3.2. Medium-level technology delivery capabilities

Figure 18 unpacks the technology delivery capabilities introduced at a high level in Figure 17.

Rather than going into a lot of detail now, it is best to just browse through these and revisit them at the end of the course, matching what was discussed during the course against these capabilities.

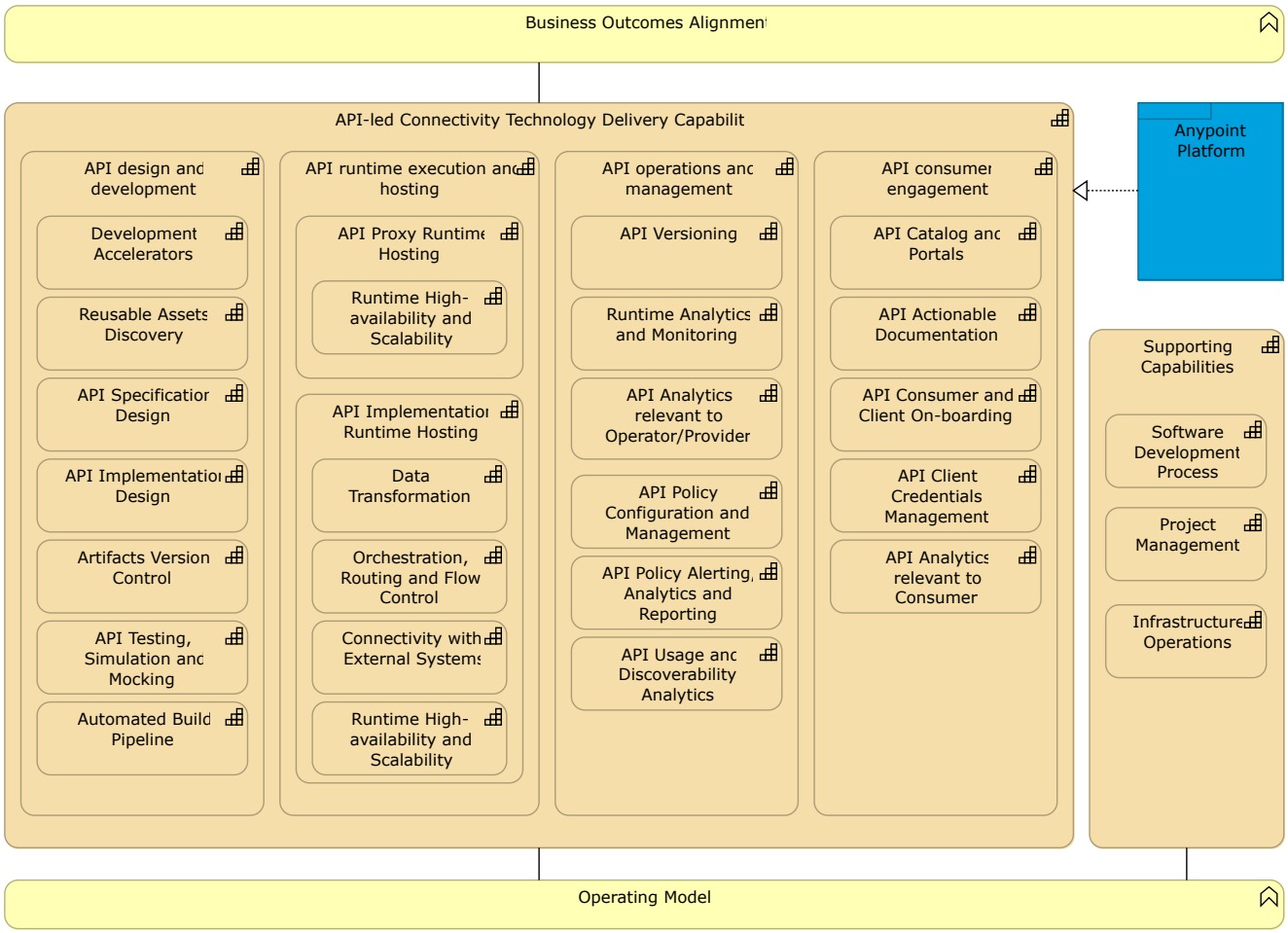

Figure 18. A medium-level drill-down into the technology delivery capabilities provided by Anypoint Platform, and some generic supporting capabilities needed for performing API-led connectivity.

2.3.3. Introducing important derived capabilities related to API clients and API implementations

API clients and API implementations are application components. For application components, the capabilities provided by Anypoint Platform enable the following important specific features, properties and activities:

- Backend system integration
- Fault-tolerant API invocation
- HA and scalable execution
- Monitoring and alerting of API implementations and, if possible, API clients

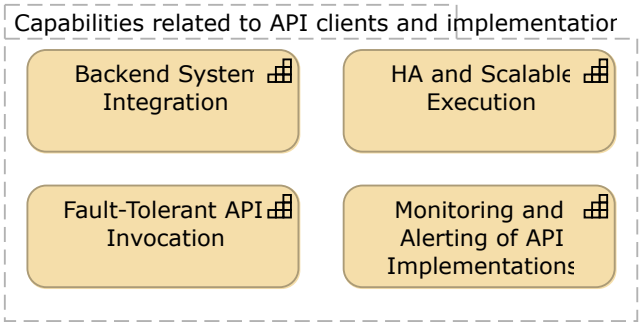

Figure 19. Important derived capabilities related to API clients and API implementations.

2.3.4. Introducing important derived capabilities related to APIs and API invocations

For APIs and API invocations themselves, rather than for the underlying application components that implement or invoke these APIs, the capabilities provided by Anypoint Platform enable the following important specific features, properties and activities:

- API design
- API policy enforcement and alerting
- Monitoring and alerting of API invocations
- Analytics and reporting of API invocations, including the reporting on meeting of SLAs
- Discoverable assets for the consumption of anyone interested in the application network, such as API consumers and API providers
- Engaging documentation, primarily for the consumption of API consumers
- Self-service API client registration for API consumers

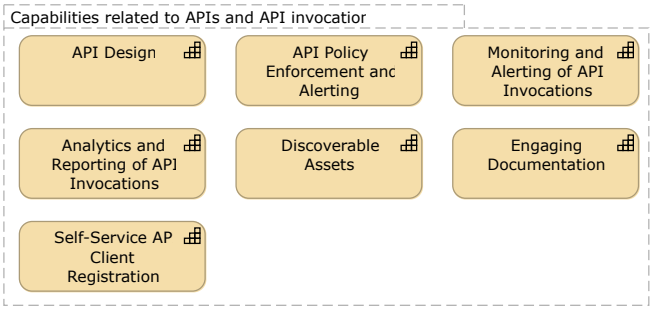

Figure 20. Important derived capabilities related to APIs and API invocations.

2.4. Introducing Anypoint Platform

2.4.1. Revisiting Anypoint Platform components

For resources that introduce Anypoint Platform see Course prerequisites.

What follows is a brief recap of the components of Anypoint Platform:

- Anypoint Design Center: Development tools to design and implement APIs, integrations and connectors [Ref2]:
  - API designer
  - Flow designer
  - Anypoint Studio
  - Connector DevKit
  - APIKit
  - MUnit
  - RAML SDKs
- Anypoint Management Center: Single unified web interface for Anypoint Platform administration:
  - Anypoint API Manager [Ref3]
  - Anypoint Runtime Manager [Ref5]
  - Anypoint Analytics [Ref7]
  - Anypoint Access management [Ref6]
- Anypoint Exchange: Save and share reusable assets publicly or privately [Ref4]. Preloaded content includes:
  - Anypoint Connectors
  - Anypoint Templates
  - Examples
  - WSDLs
  - RAML APIs
  - Developer Portals
- Mule runtime and Runtime services: Enterprise-grade security, scalability, reliability and high availability:
  - Mule runtime [Ref1]
  - CloudHub
  - Anypoint MQ [Ref8]
  - Anypoint Enterprise Security

    - Anypoint Fabric
        - Worker Scaleout
        - Persistent VM Queues
    - Anypoint Virtual Private Cloud (VPC)
- Anypoint Connectors:
    - Connectivity to external systems
    - Dynamic connectivity to APIs
    - Build custom connectors using Connector DevKit/SDK
- Hybrid cloud

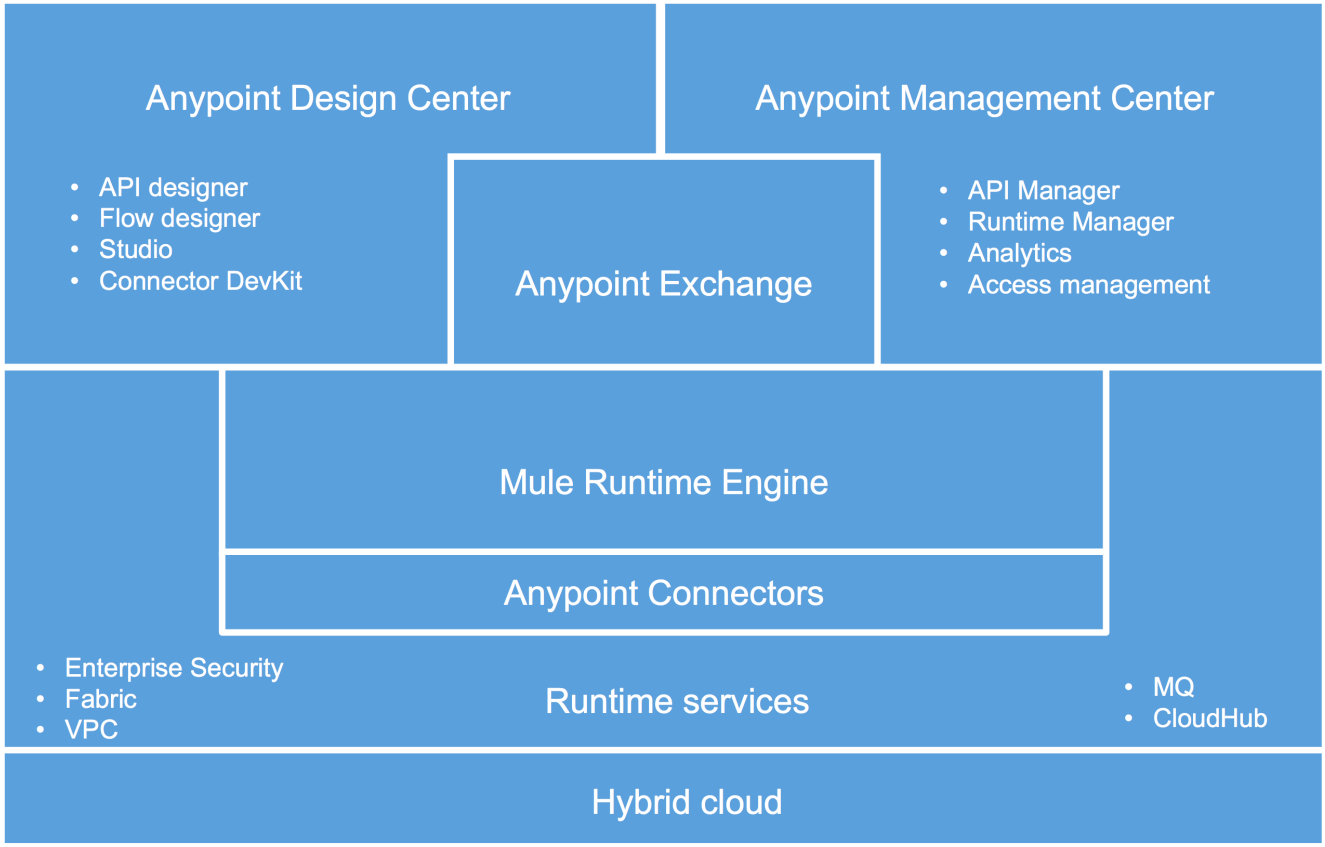

Figure 21. Anypoint Platform and its main components.

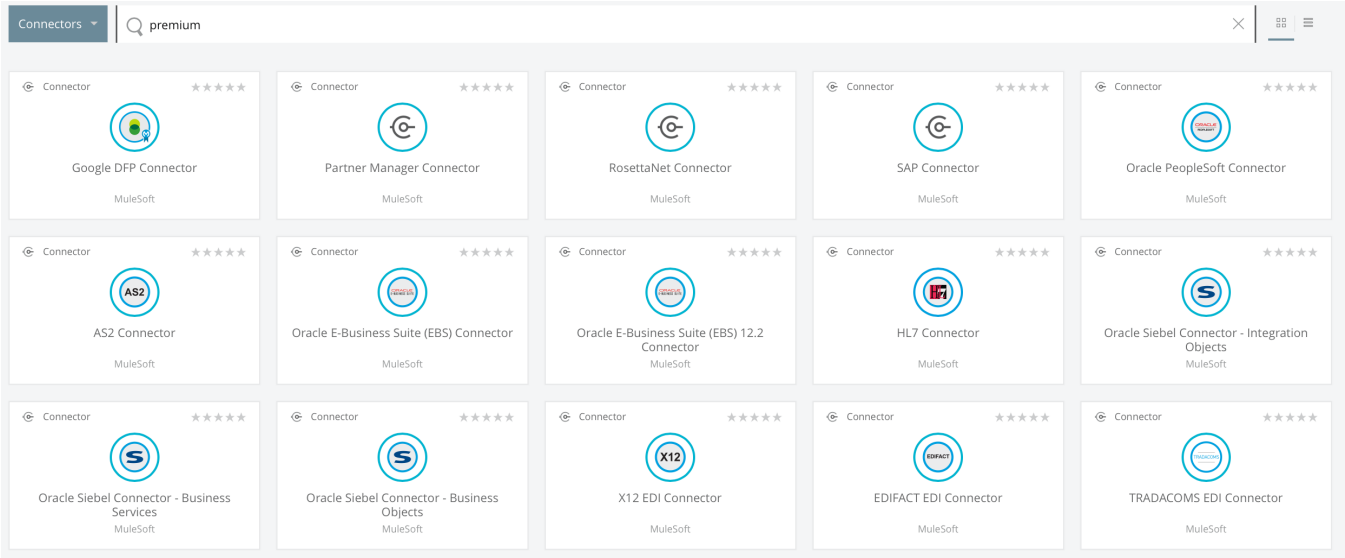

Figure 22. Some of the Anypoint Connectors published in Anypoint Exchange.

2.4.2. Understanding automation on Anypoint Platform

Anypoint Platform exposes a consolidated *web UI* for easy interaction with users of all levels of expertise. Screenshots in this course are from that web UI.

All functionality exposed in the web UI is *also available via Anypoint Platform APIs*: these are JSON REST APIs which are also invoked by the web UI. Anypoint Platform APIs enable *extensive automation* of the interaction with Anypoint Platform.

MuleSoft also provides higher-level automation tools that capitalize on the presence of Anypoint Platform APIs:

- *Anypoint CLI*, a command-line interface providing a user-friendly interactive layer on top of Anypoint Platform APIs
- *Mule Maven Plugin*, a Maven plugin automating packaging and deployment of Mule applications (including API implementations) to all kinds of Mule runtimes, typically used in CI/CD (9.1.2)

Related to this discussion is the observation that Anypoint Exchange is also accessible as a Maven repository. This means that a Maven POM can be configured to deploy artifacts into Anypoint Exchange and retrieve artifacts from Anypoint Exchange, just like with any other Maven repository (such as Nexus).

Summary

- MuleSoft's mission is "To connect the world's applications, data and devices to transform

business"

- MuleSoft proposes to close the increasing IT delivery gap through a consumption-oriented operating model with modern APIs as the core enabler
- API-led connectivity defines tiers for Experience APIs, Process APIs and System APIs with distinct stakeholders and focus
- An application network emerges from the repeated application of API-led connectivity and stresses self-service consumption, visibility, governance and security
- Anypoint Platform provides the capabilities for realizing application networks
- Anypoint Platform consists of these high-level components: Anypoint Design Center, Anypoint Management Center, Anypoint Exchange, Mule runtime, Anypoint Connectors, Runtime services, Hybrid cloud
- Interaction with Anypoint Platform can be extensively automated

Module 3. Establishing Organizational and Platform Foundations

Objectives

- Advise on establishing a C4E and identify KPIs to measure its success
- Choose between options for hosting Anypoint Platform and provisioning Mule runtimes
- Describe the set-up of organizational structure on Anypoint Platform
- Compare and contrast Identity Management and Client Management on Anypoint Platform

3.1. Establishing a Center for Enablement (C4E) at Acme Insurance

3.1.1. Assessing Acme Insurance's integration capabilities

The New Owners contract a well-known management consultancy to assess Acme Insurance's capabilities to embark on the strategic initiatives introduced in Acme Insurance's motivation to change. One focus of this assessment is Acme Insurance's IT capabilities in general and integration capabilities in particular. The findings are:

- LoBs (personal motor and home) have a long history of independence, also in IT
- LoBs have strong IT skills, medium integration skills but no know-how in API-led connectivity or Anypoint Platform
- Acme IT is small but enthusiastic about application networks and API-led connectivity
- DevOps capabilities are present in LoB IT and Acme IT
- Corporate IT lacks the capacity and desire to involve themselves directly in Acme Insurance's Enterprise Architecture. All they care about is that corporate principles are being followed, as summarized in Figure 3

3.1.2. A decentralized C4E for Acme Insurance

Based on the above analysis of Acme Insurance's integration capabilities and organizational characteristics, Acme Insurance and Corporate IT agree on the establishment of a C4E at Acme Insurance with the following guiding principles:

enable

Enables LoBs to fulfil their integration needs

API-first

Uses *API-led connectivity* as the main architectural approach

asset-focused

Provides directly valuable *assets* rather than just documentation contributing to this goal

self-service

Assets are to be *self-service consumed* (initially) and (co-) created (ultimately) by the LoBs

reuse-driven

Assets are to be *reused* wherever applicable

federated

Follows a decentralized, *federated operating model*

The decentralized nature of the C4E matches well

- Acme Insurance's division into LoBs
- the IT skills available in the LoBs
- Acme Insurance's relationship with The New Owners

Remarks:

- The "enable" principle defines an *Outcome-Based Delivery model (OBD)* for the C4E
- The principles of "self-service", "reuse-driven" and "federated" promise *increased integration delivery speed*
- Overall the C4E aims for a *streamlined, lean engagement model*, as also reflected in the "asset-focused" principle
- This discussion omits questions of funding the C4E and whether C4E staff is fully or partially assigned to the C4E

The following *technical business roles* are assigned to the C4E at the positions in their organizational structure as shown in Figure 23:

- *Platform Architect*:
  - Responsibilities: Platform vision and it's evolution to meet tactical and strategic objectives of the business
  - Profile: Solid understanding of the technical foundation of the platform; a technology enthusiast with a strong track record of building and running high volume, reliable architectures, preferably including cloud, As-A-Service infrastructure and applications

- *API Architect*, *Integration Architect/Developer*:
  - Responsibilities: Provides “enough” guidance over the design and operations of the C4E assets; thought leadership, leading key stakeholders in designing and adopting API architectures; architecting integrations solutions for C4E internal customers working close with them through the whole lifecycle; gathering requirements from both the business and technical teams to determine the integration solutions/APIs
  - Profile: Deep experience of integration and API architecture, thorough understanding of industry trends, vendors, frameworks and practices
- *DevOps Architect/Engineer*

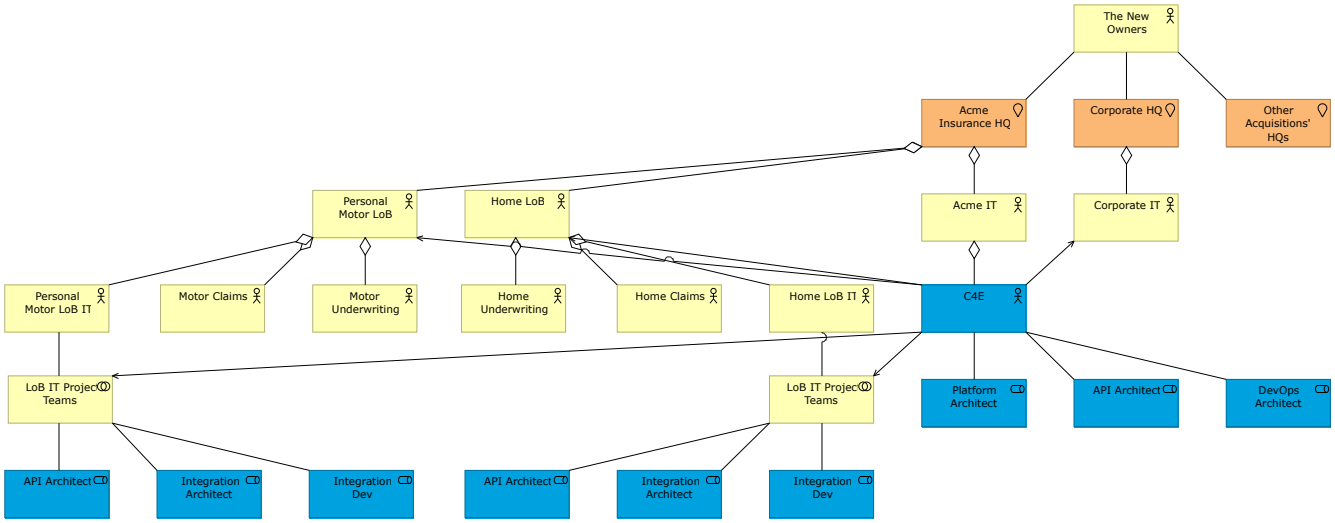

Figure 23. Organizational view into Acme Insurance's target Business Architecture with a C4E supporting LoBs and their project teams with their integration needs. Technical C4E business roles are shown in blue, managerial C4E business roles are not shown.

3.1.3. Exercise 1: Measuring success of the C4E

Thinking back on the application network vision on the one hand, and the principles of Acme Insurance’s' C4E on the other hand:

1. Compile a *list of statements* which, if largely true, allow the conclusion that the *C4E is successful*
2. Compile a similar list that allows the conclusion that the *application network vision is being realized*
3. From the above lists, extract a list of corresponding *metrics*

Solution

See 3.1.4.

3.1.4. Key Performance Indicators measuring the success of Acme Insurance's C4E and the growth of its application network

Acme Insurance uses the following key performance indicators (KPIs) to measure and track the success of the C4E and its activities, as well as the growth and health of the application network.

All of the metrics can be extracted automatically, through REST APIs, from Anypoint Platform.

- # of assets published to Anypoint Exchange
- # of interactions with Anypoint Exchange assets
- # of APIs managed by Anypoint Platform
- # of System APIs managed by Anypoint Platform
- # of API clients registered for access to APIs
- # of API implementations deployed to Anypoint Platform
- # of API invocations
- # or fraction of lines of code covered by automated tests in CI/CD pipeline
- Ratio of info/warning/critical alerts to number of API invocations

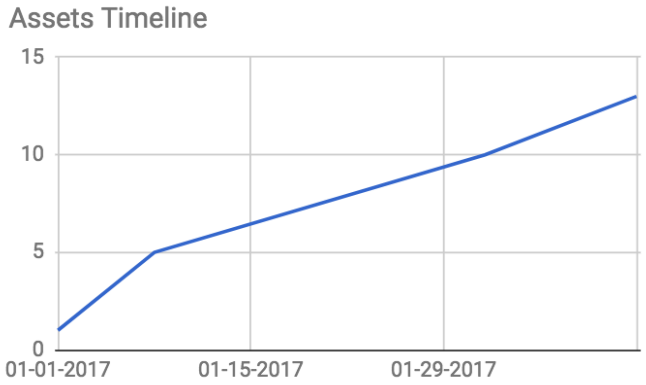

Figure 24. Number of assets published to Anypoint Exchange over time.

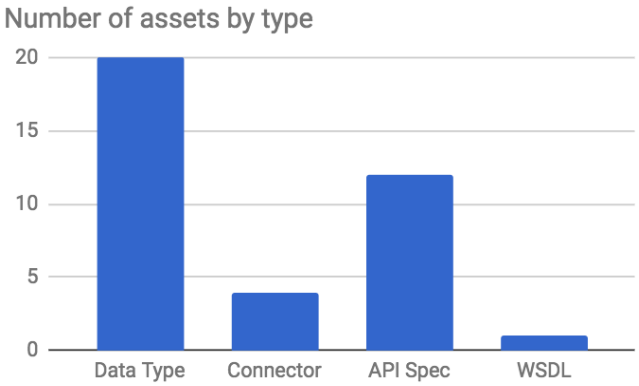

Figure 25. Current number of assets published to Anypoint Exchange grouped by type of asset.

3.2. Understanding Anypoint Platform deployment scenarios

3.2.1. Separating Anypoint Platform control plane and runtime plane

Anypoint Platform components (2.4.1) can be distinguished according to whether they are directly involved in the execution of Mule applications or not:

- The *Anypoint Platform runtime plane* comprises the Mule runtime itself, Anypoint Connectors used by Mule applications executing in the Mule runtime, and all supporting Runtime services, incl. Anypoint Enterprise Security, Anypoint Fabric, Anypoint VPCs, CloudHub Dedicated Load Balancers, Object Store and Anypoint MQ. Simply put, the runtime plane is where integration logic executes.
- The *Anypoint Platform control plane* comprises all Anypoint Platform components that are involved in managing/controlling the components of the runtime plane (i.e., Anypoint Management Center, incl. Anypoint API Manager and Anypoint Runtime Manager), the design of APIs or Mule applications (i.e., Anypoint Design Center, incl. Anypoint Studio and API designer) or the documentation and discovery of application network assets (i.e., Anypoint Exchange)

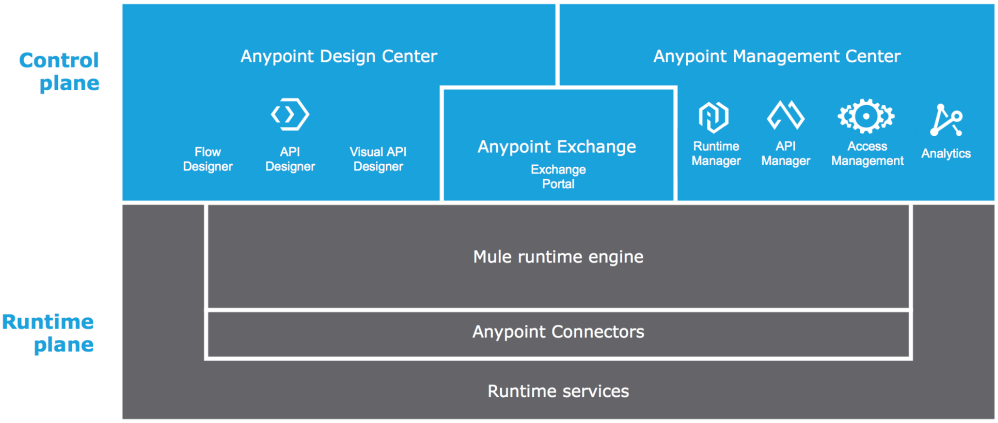

Figure 26. Components of the Anypoint Platform belong to either the control plane or the runtime plane.

3.2.2. Introducing MuleSoft-hosted and customer-hosted Anypoint Platform deployments

See [Ref5], [Ref9], [Ref10].

		Runtime Plane / Mule runtimes				
		MuleSoft-hosted		Customer-hosted		
		iPaaS-provisioned				Manually provisioned
		AWS public cloud	AWS VPC	Pivotal Cloud Foundry	Kubernetes Docker	-
Control Plane	MuleSoft-hosted	**Anypoint Platform** with **CloudHub**	**Anypoint VPC** with **CloudHub**	-	**Anypoint Runtime Fabric**	**Hybrid**
	Customer-hosted	-	-	**Anypoint Platform for PCF**	-	**Anypoint Platform Private Cloud Edition**

Figure 27. Overview of Anypoint Platform deployment options and MuleSoft product names (in bold) for each supported scenario. (At the time of this writing, Anypoint Runtime Fabric is not yet generally available.)

Options for the deployment of the Anypoint Platform *control plane*:

- MuleSoft-hosted:
  - Product: Anypoint Platform
  - AWS regions: US East (N Virginia, https://anypoint.mulesoft.com) or EU (Frankfurt, https://eu1.anypoint.mulesoft.com)
- Customer-hosted:
  - Product: Anypoint Platform Private Cloud Edition

Options for the deployment of the Anypoint Platform *runtime plane*, incl. provisioning of Mule runtimes:

- MuleSoft-hosted:
  - In the public AWS cloud: CloudHub
  - In an AWS VPC: CloudHub with Anypoint VPC
  - AWS regions:
    - Under management of the US East control plane: US East, US West, Canada, Asia Pacific, EU (Frankfurt, Ireland, London, …), South America
    - Under management of the EU control plane: EU (Frankfurt and Ireland)
- Customer-hosted:
  - *Manually provisioned* Mule runtimes on bare metal, VMs, on-premises, in a public or private cloud, …
  - *iPaaS-provisioned* Mule runtimes:
    - MuleSoft-provided software appliance: Anypoint Runtime Fabric
    - Customer-managed Pivotal Cloud Foundry installation: Anypoint Platform for Pivotal Cloud Foundry

3.2.3. MuleSoft-hosted Anypoint Platform control plane and runtime plane with iPaaS functionality in public cloud

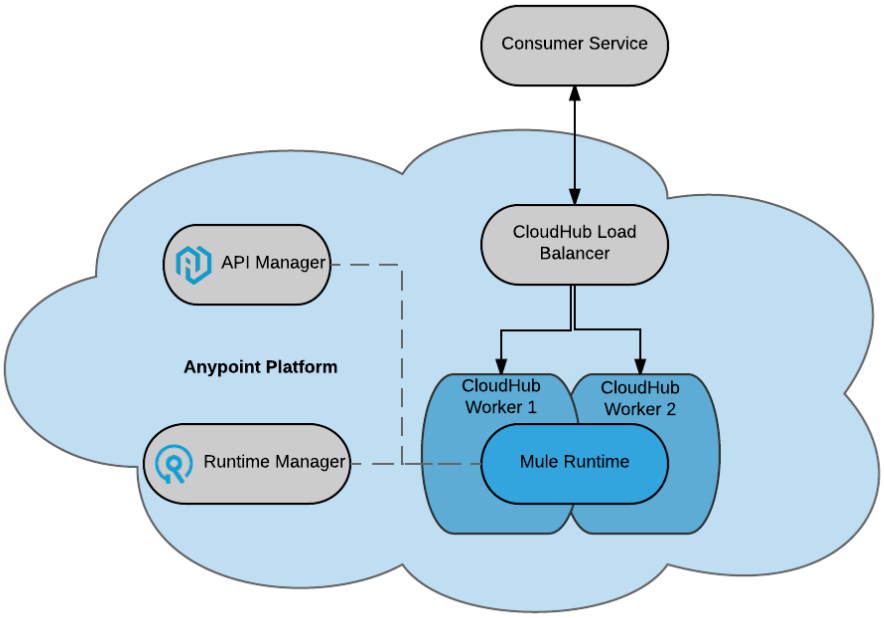

Figure 28. MuleSoft-hosted Anypoint Platform control plane managing MuleSoft-hosted Anypoint Platform runtime plane with iPaaS-provisioned Mule runtimes on CloudHub in the public AWS cloud.

3.2.4. MuleSoft-hosted Anypoint Platform control plane and runtime plane with iPaaS functionality in Anypoint VPC

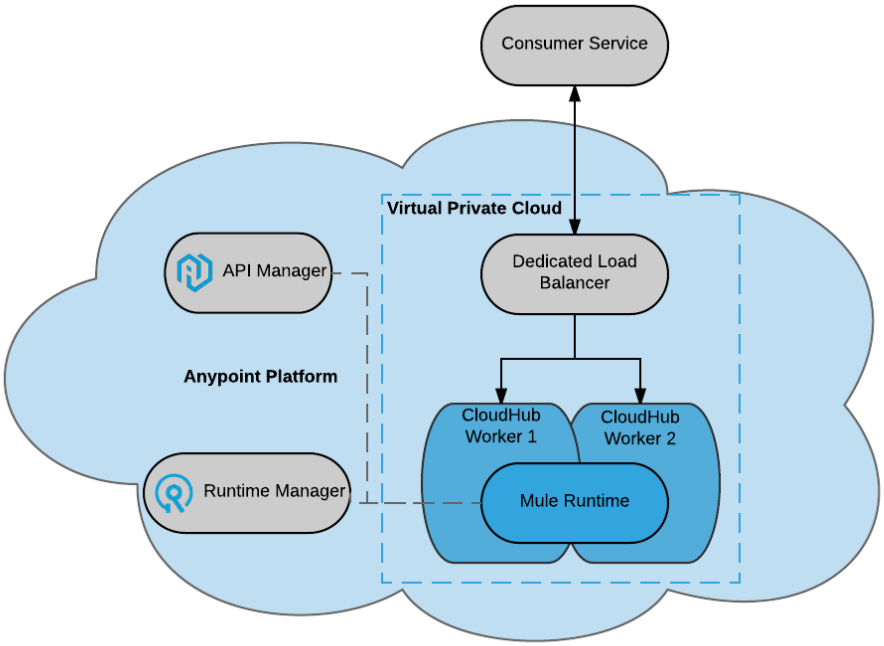

Figure 29. MuleSoft-hosted Anypoint Platform control plane managing MuleSoft-hosted Anypoint Platform runtime plane with iPaaS-provisioned Mule runtimes on CloudHub in an Anypoint VPC.

3.2.5. MuleSoft-hosted Anypoint Platform control plane and customer-hosted runtime plane without iPaaS functionality

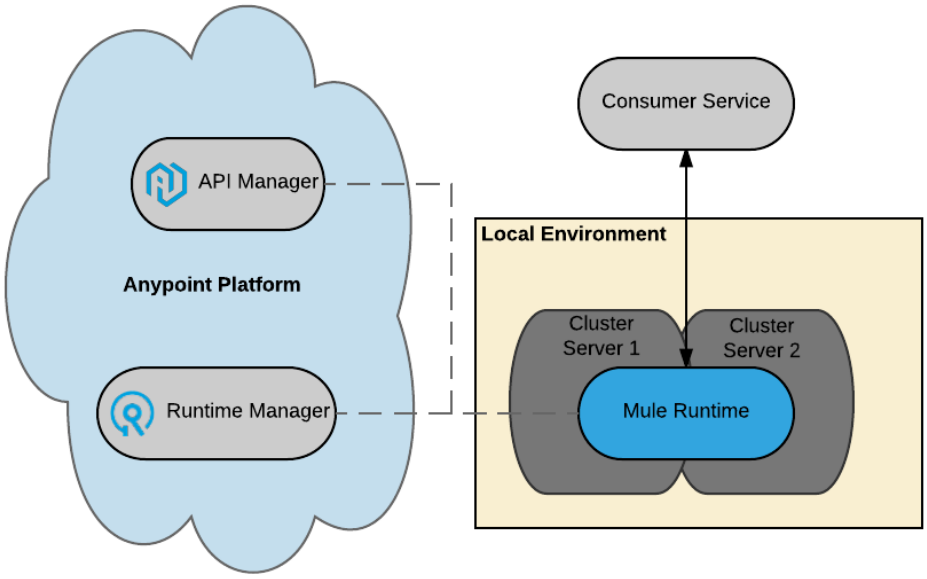

Figure 30. MuleSoft-hosted Anypoint Platform control plane managing customer-hosted Anypoint Platform runtime plane with manually provisioned Mule runtimes.

3.2.6. Customer-hosted Anypoint Platform control plane and runtime plane without iPaaS functionality

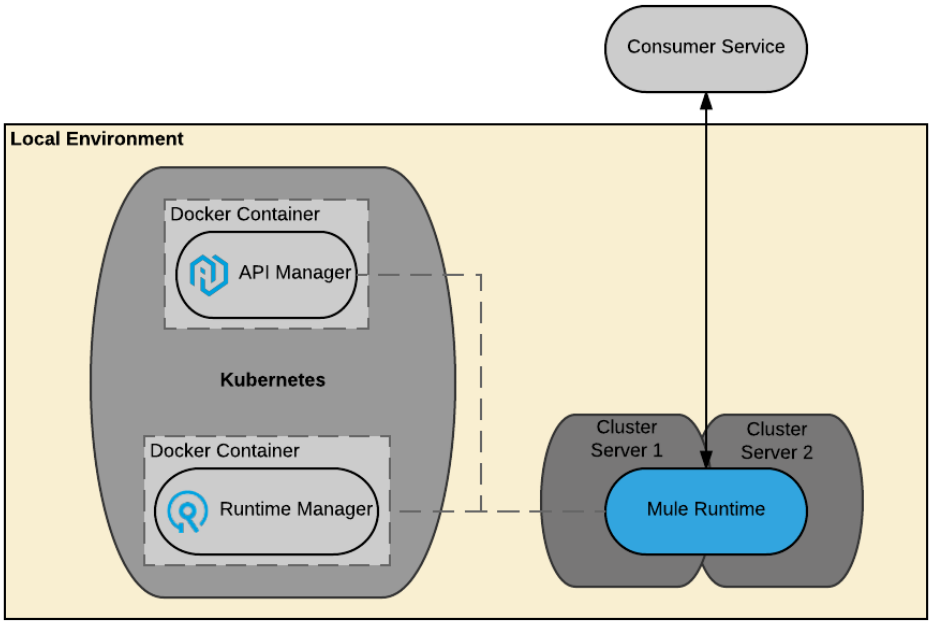

Figure 31. Anypoint Platform Private Cloud Edition: Customer-hosted Anypoint Platform control plane managing customer-hosted Anypoint Platform runtime plane with manually provisioned Mule runtimes.

3.2.7. Customer-hosted Anypoint Platform control plane and runtime plane with iPaaS functionality on Pivotal Cloud Foundry

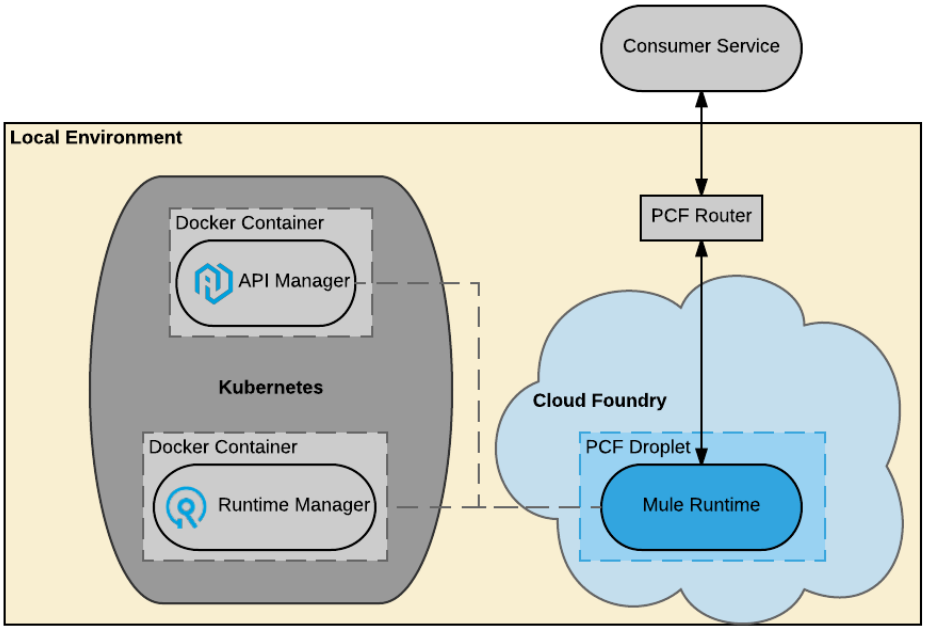

Figure 32. Anypoint Platform for Pivotal Cloud Foundry: Customer-hosted Anypoint Platform control plane managing customer-hosted Anypoint Platform runtime plane with iPaaS-provisioned Mule runtimes on PCF.

3.2.8. Anypoint Platform differences between deployment scenarios

Anypoint Platform deployment options differ in various ways:

- At the highest level, not all Anypoint Platform components are currently available for every Anypoint Platform deployment scenario.
- Some features of Anypoint Platform components that *are* available in more than one deployment scenario differ in their details, typically due to the different technical characteristics and capabilities available in each case.

Table 1. Availability of Anypoint Platform components and high-level features in different deployment scenarios.

Component, Feature	MuleSoft-hosted Anypoint Platform	Hybrid	Anypoint Platform Private Cloud Edition	Anypoint Platform for Pivotal Cloud Foundry
API designer	**yes**	**yes**	**yes**	**yes**
Flow designer	**yes**	**yes**	**no**	**no**
Anypoint Access management	**yes**	**yes**	**yes**	**yes**

Component, Feature	MuleSoft-hosted Anypoint Platform	Hybrid	Anypoint Platform Private Cloud Edition	Anypoint Platform for Pivotal Cloud Foundry
LDAP for Identity Management	no	no	yes	yes
Anypoint Runtime Manager	**yes**	**yes**	**yes**	**yes**
External Runtime Analytics	no	yes	yes	yes
Schedules UI	yes	yes	yes	yes
Anypoint Runtime Manager monitoring dashboards	yes	yes	no	no
Insight	yes	yes	no	no
Anypoint Runtime Manager alerts	yes	yes	no	no
Anypoint API Manager	**yes**	**yes**	**yes**	**yes**
Anypoint API Manager alerts	yes	yes	no	no
Anypoint Analytics	**yes**	**yes**	**no**	**no**
External API Analytics	no	yes	yes	yes
Anypoint Exchange	**yes**	**yes**	**yes**	**yes**
Anypoint MQ	**yes**	**yes**	**no**	**no**
iPaaS	**yes** (CloudHub)	**no**	**no**	**yes**
Mule runtime load balancing	yes (Fabric)	no	no	yes (PCF)
Mule runtime persistent VM queues	yes (Fabric)	yes	yes	yes
Mule runtime auto-restart	yes	no	no	yes
Mule runtime cluster	no	yes	yes	yes
Zero downtime deployments	yes	no	no	yes

Notes:

- External Analytics refers to Splunk, ELK or similar software
- Persistent VM queues in a non-CloudHub deployment scenario have many more options, such as cluster-wide in-memory replication or persistence on disk or in a database, but may

require management of persistent storage

- The Tanuki Software Java Service Wrapper offers limited auto-restart capabilities also for customer-managed Mule runtimes, but this is not comparable to CloudHub's, Anypoint Runtime Fabric's or PCF's sophisticated health-check-based auto-restart feature
- Anypoint Runtime Manager alerts on CloudHub (and only on CloudHub) also include the option of custom alerts and notifications (via the CloudHub connector)
- Anypoint Platform Private Cloud Edition supports minimal Anypoint Runtime Manager alerts related to the Mule application being deployed, undeployed, etc., but no alerts based on Mule events or API events
- Anypoint Runtime Manager monitoring dashboards include load and performance metrics for Mule applications and the servers to which Mule runtimes are deployed
- Insight is a troubleshooting tool that gives in-depth visibility into Mule applications by collecting Mule events (business events and default events) and the transactions to which these events belong. Only CloudHub supports the replay of transactions (which causes re-processing the involved message)
- Clustering Mule runtimes when deploying through Anypoint Platform for Pivotal Cloud Foundry requires a PCF Hazelcast service. Mule runtime clusters in Hybrid and Anypoint Platform Private Cloud Edition deployment scenarios only need and use the Hazelcast features built into the Mule runtime itself.

3.2.9. Exercise 2: Choosing between deployment scenarios

Reflecting on the various deployment scenarios supported by Anypoint Platform:

1. Discuss the characteristics of each scenarios
2. For each deployment scenario, identify requirements that would clearly require that scenario

Solution

Anypoint Platform deployment scenarios can be evaluated along the following dimensions:

- *Regulatory or IT operations requirements* that mandate on-premises processing of every data item, including meta-data about API invocations and messages processed within Mule applications: requires Anypoint Platform Private Cloud Edition or Anypoint Platform for Pivotal Cloud Foundry
- *Time-to-market*, assuming the effort to deploy Anypoint Platform must be included in the elapsed time: favors MuleSoft-hosted Anypoint Platform
- *IT operations effort*: favors MuleSoft-hosted Anypoint Platform over all other deployment scenarios; favors Anypoint Runtime Fabric over Anypoint Platform for Pivotal Cloud Foundry

over Anypoint Platform Private Cloud Edition

- Latency and throughput when accessing *on-premises data sources*: favors scenarios where Mule runtimes can be deployed close to these data sources, i.e., Anypoint Runtime Fabric, Anypoint Platform Private Cloud Edition and Anypoint Platform for Pivotal Cloud Foundry over CloudHub with geographically close runtime plane
- *Isolation* between Mule applications: favors scenarios where each Mule application is assigned to its own Mule runtime; favors bare metal over VMs over containers for Mule runtimes
- *Control* over Mule runtime characteristics like JVM and machine memory, garbage collection settings, hardware, etc.: favors Hybrid and Anypoint Platform Private Cloud Edition over Anypoint Platform for Pivotal Cloud Foundry over MuleSoft-hosted Anypoint Platform
- *Scalability* of runtime plane: consider horizontal and vertical scaling; consider static and dynamic (load-based, automatic) scaling; favors cloud-deployments of runtime plane, including CloudHub; favors iPaaS functionality over manually provisioned Mule runtimes
- *Roll-out of new releases*: continuously (weekly) in the MuleSoft-hosted control plane versus quarterly releases of Anypoint Platform Private Cloud Edition

3.2.10. Anypoint Platform data residency

The deployment scenarios supported by Anypoint Platform cover a large number of options of who (MuleSoft or the customer) controls and manages the various components of Anypoint Platform, where these components reside, and where Mule applications execute as a result. As Mule applications execute integration logic, data flows through the various systems of Anypoint Platform and beyond. This section describes in detail where what kind of data is sent and stored. This is particularly relevant in the light of various regulatory (legal) requirements such GDPR (https://www.eugdpr.org) and the Patriot Act.

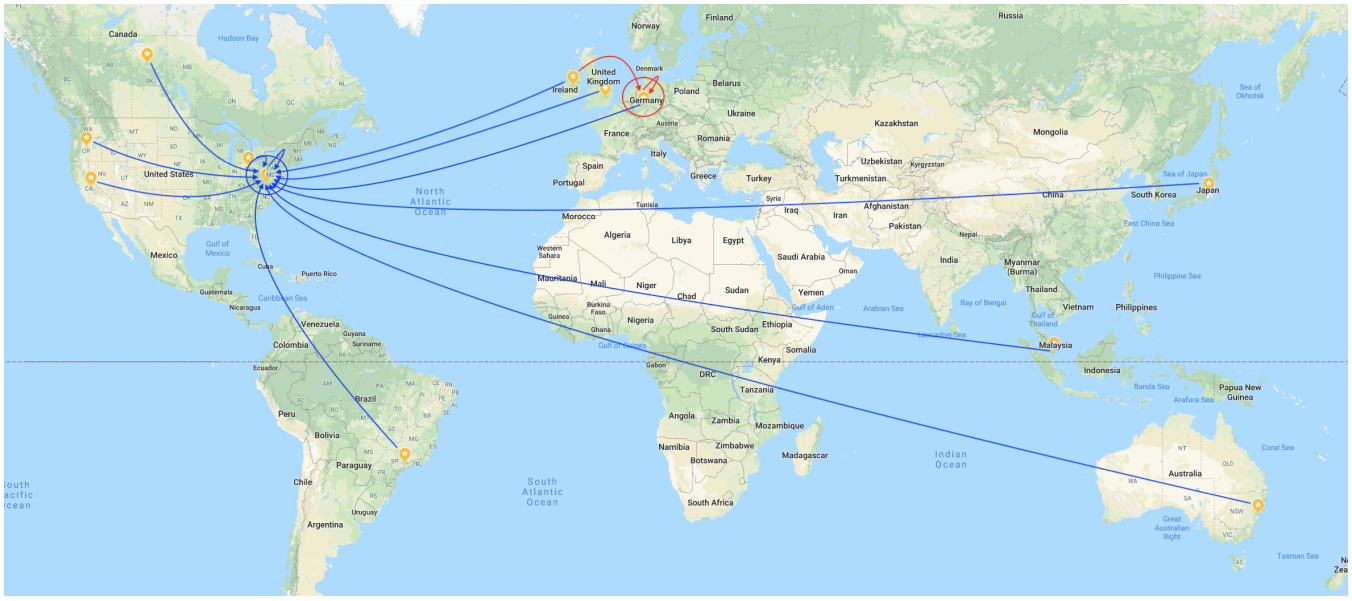

Figure 33. AWS regions in which MuleSoft hosts the Anypoint Platform control plane (large colored circles) and runtime plane (small yellow marks). Data resides within the runtime plane, while limited amounts of metadata are sent from there to the managing control plane (arrows in the color of the managing control plane).

The fundamental principle is that the *location of Mule runtimes*, together with the *integration logic implemented by Mule applications* executing in these Mule runtimes, determine the *location and residency of all data*. In addition to the data itself, limited amounts of *metadata and metrics* are exchanged with the Anypoint Management Center components of the *Anypoint Platform control plane*. Specifically:

- Mule runtimes and Mule applications can either be managed entirely by the customer or execute in one of the *AWS regions supported by CloudHub* as described in 3.2.2: this determines where all *data*, in the form of Mule messages, is processed.
- Mule applications can make use of *Object Store, persistent VM queues* or other resilience features. In a CloudHub deployment, all of these features store data in the *same AWS region as the Mule runtime* itself. For customer-managed deployments the customer fully controls where Mule applications send or store data.
- Mule applications may also make use of *Anypoint MQ* (8.3.1): Anypoint MQ queues/exchanges are explicitly created to reside in an AWS region. The available regions are typically those of the Anypoint Platform runtime plane.
- *Metadata, incl. metrics*, about Mule messages and API invocations is sent by the Mule runtime to Anypoint Management Center (10.1.1). Thus by selecting either a customer-hosted control plane, or one of the supported AWS regions of the MuleSoft-hosted Anypoint Platform (3.2.2), customers determine where this metadata is sent.

    - Example of metadata: CPU/memory usage, message/error count, API name and version, geodata about the API client, HTTP method, violated API policy name, etc. (10.2.1)
- *Logs* produces by Mule applications deployed to CloudHub have residency characteristics similarly to those of metadata.
- Mule applications may produce business events and Anypoint Runtime Manager may use the Insight feature to analyze and replay business events. Depending on the details of the configuration, events may also include the data (payload) of the Mule message, although this is not the case by default. If this feature is chosen, then message data (and not just metadata) is sent to Anypoint Runtime Manager.
- Mule applications are typically deployed to Mule runtimes using Anypoint Runtime Manager: this means that the Mule applications themselves, incl. all code and resources packaged within the Mule applications, are stored where the Anypoint Platform control plane resides.

These rules are typically applied as follows to meet regulatory requirements through jurisdiction-local deployments (assuming suitably implemented Mule applications):

- A combination of the *MuleSoft-hosted Anypoint Platform EU/US control plane* and a matching *MuleSoft-hosted EU/US runtime plane* (i.e., using CloudHub and Anypoint MQ in a matching AWS region) keeps all *data and metadata* in the EU/US.
- A combination of the *MuleSoft-hosted Anypoint Platform EU/US control plane* and a *customer-hosted runtime plane* (i.e., Hybrid deployment , optionally with Anypoint Runtime Fabric) keeps all *data* on customer-hosted infrastructure and all *metadata* in the EU/US.
- A combination of a *customer-hosted Anypoint Platform control plane* and a matching *customer-hosted runtime plane* (i.e., using Anypoint Platform Private Cloud Edition or Anypoint Platform for Pivotal Cloud Foundry) keeps all *data and metadata* on customer-hosted infrastructure.

3.3. Onboarding Acme Insurance onto Anypoint Platform

3.3.1. Anypoint Access management

- Controls access to entitlement areas in Anypoint Platform
- Manage
    - Business groups, users, roles and permissions
    - Environments
    - Other Resources

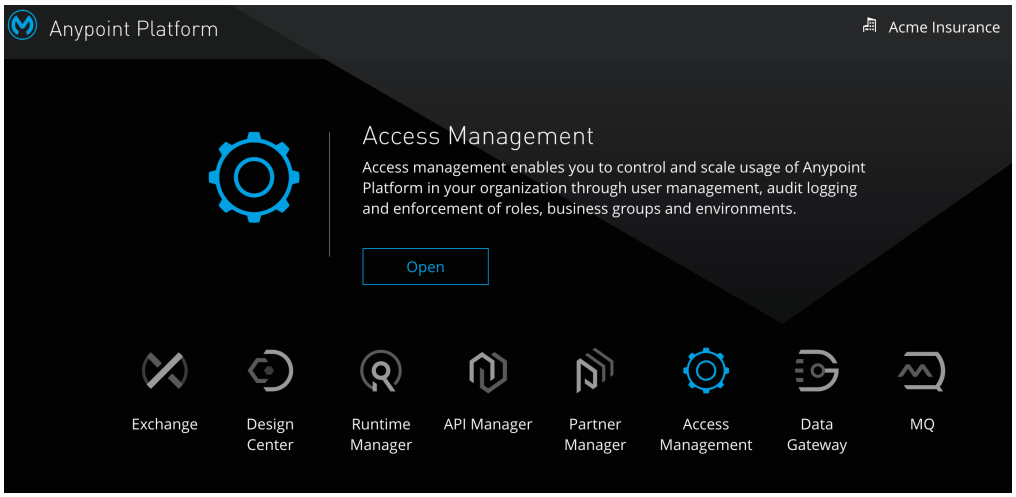

Figure 34. Anypoint Access management and the Anypoint Platform entitlements.

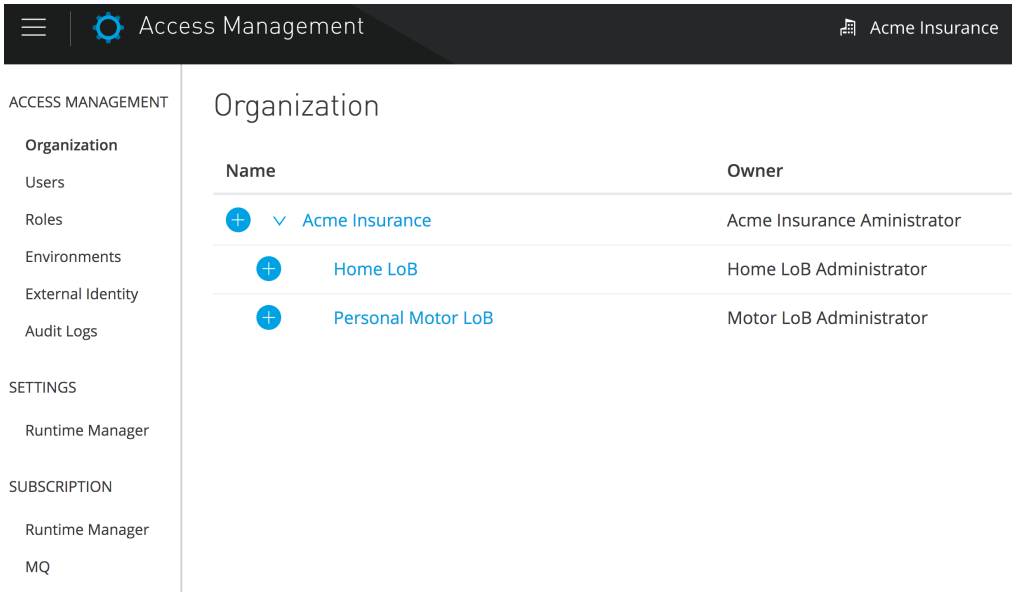

Figure 35. Anypoint Access management controls access to and allocation of various resources on Anypoint Platform.

3.3.2. Anypoint Platform organizations and business groups

- Organization: An administrative collection of resources and users
- Business group: A sub-organization at any level

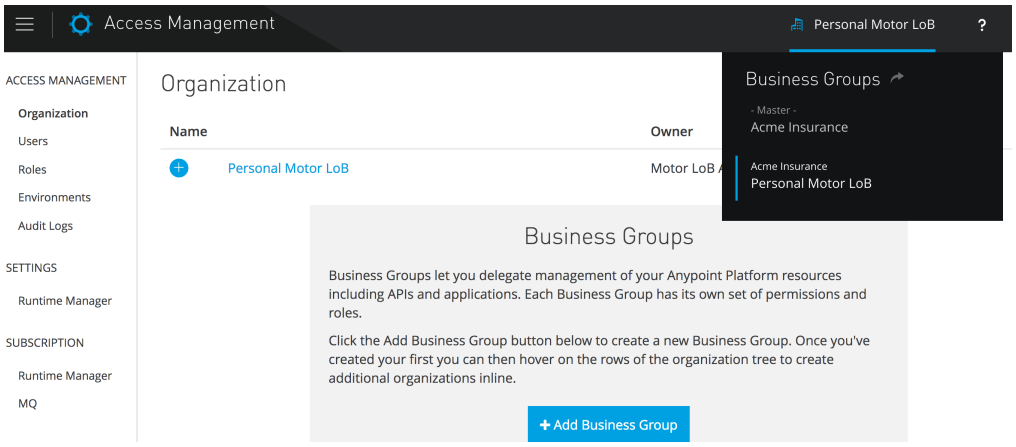

Figure 36. Anypoint Access management at the level of the Personal Motor LoB business group.

3.3.3. Identity Management vs Client Management in Anypoint Platform

- Identity Management is concerned with users of Anypoint Platform
  - includes users of the Anypoint Platform web UI and the Anypoint Platform APIs
  - enables Single Sign-On (SSO)
- Client Management is concerned with API clients using OAuth 2.0

By default, Anypoint Platform acts as an Identity Provider for Identity Management. But Anypoint Platform also supports configuring one external Identity Provider for each of these two uses, independently of each other.

If an external Identity Provider is configured for Identity Management, then *SAML 2.0 bearer tokens* issued by that Identity Provider can be used *for invocations of the Anypoint Platform APIs*. Optionally, before configuring an external Identity Provider for Identity Management, *setup administrative users* for invoking the Anypoint Platform APIs. These will remain valid after the external Identity Provider has been configured.

3.3.4. Supported Identity Provider standards and products

For *Identity Management*, Anypoint Platform supports:

- The mapping of Anypoint Platform roles to groups in an external Identity Provider
- OpenID Connect
  - a standard implemented by Identity Providers such as PingFederate, OpenAM, Okta
  - does not support single log-out

- SAML 2.0
  - a standard implemented by Identity Providers such as PingFederate, OpenAM, Okta, Shibboleth, Active Directory Federation Services (AD FS), onelogin, CA Single Sign-On
  - supports single log-out
- LDAP
  - a standard that is supported for Identity Management only on Anypoint Platform Private Cloud Edition

For *Client Management*, Anypoint Platform supports the following Identity Providers as OAuth 2.0 servers:

- OpenAM
- PingFederate
- OpenID Connect Dynamic Client Registration (DCR)
  - a standard implemented by Identity Providers such as Okta and OpenAM

3.3.5. Selecting an Identity Provider for Acme Insurance

Acme Insurance currently uses Microsoft Active Directory (AD) to store user accounts.

After a brief evaluation period Acme Insurance chooses PingFederate as an Identity Provider ontop of AD. They configure their Anypoint Platform organization in the MuleSoft-hosted Anypoint Platform to access their on-premises PingFederate instance for Identity Management.

Acme Insurance is currently unsure whether they will need OAuth 2.0, but if they do, they plan to use the same PingFederate instance also for Client Management.

Summary

- A federated C4E is established at Acme Insurance to facilitate API-led connectivity and the growth of an application network
  - Federation plays to the strength of Acme Insurance’s LoB IT
  - KPIs to measure the C4E’s success are defined and monitored
- Anypoint Platform control plane and runtime plane can both be hosted by MuleSoft or customers
- Mule runtimes can be provisioned manually or through iPaaS functionality
- iPaaS-provisioning of Mule runtimes is supported via CloudHub, Anypoint Platform for Pivotal Cloud Foundry and Anypoint Runtime Fabric

- Not all Anypoint Platform components are available in all deployment scenarios
- Acme Insurance and its LoBs and users are onboarded onto Anypoint Platform using an external Identity Provider
- Identity Management and Client Management are clearly distinct functional areas, both supported by Identity Providers

Module 4. Identifying, Reusing and Publishing APIs

Objectives

- Map Acme Insurance’s planned strategic initiatives to products and projects
- Identify APIs needed to implement these products
- Assign each API to one of the three tiers of API-led connectivity
- Reason in detail composition and collaboration of APIs
- Reuse APIs wherever possible
- Publish APIs and related assets for reuse

4.1. Productizing Acme Insurance's strategic initiatives

4.1.1. Translating strategic initiatives into products, projects and features

Acme Insurance has committed to realize the two most pressing strategic initiatives introduced ealier (Acme Insurance's motivation to change):

- Open-up to Aggregators for motor insurance
- Provide self-service capabilities to customers

All relevant stakeholders come together under the guidance of the C4E to concretize these strategic initiatives into two minimally viable products and their defining features:

- The "Aggregator Integration" product
  - with the "Create quote for aggregators" feature as the defining feature
- The "Customer Self-Service App" product
  - with the "Retrieve policy holder summary" feature as one defining feature
  - and the "Submit auto claim" feature as the other defining feature

The products' features realize the requirements defined by the strategic initiatives.

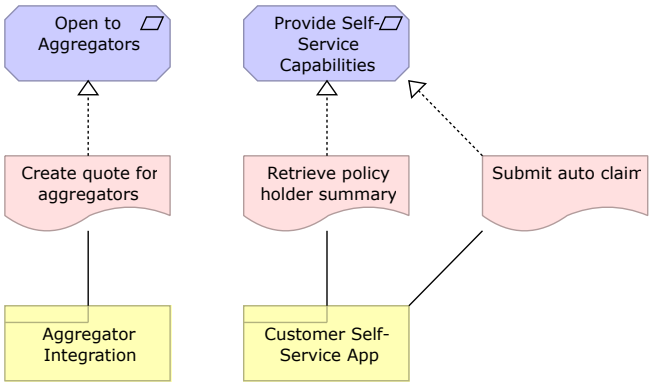

Figure 37. Architecturally significant features of the immediately relevant strategic initiatives, and the products they are assigned to.

The "Aggregator Integration" product and "Customer Self-Service App" product are assigned to two project teams. The project for the "Aggregator Integration" product is kicked-off immediately, while the project for the "Customer Self-Service App" product starts with some delay.

This project for the "Aggregator Integration" product is the first to use API-led connectivity at Acme Insurance, and is also the one etablishing the foundation of what will become the Acme Insurance application network.

4.2. Identifying APIs for the "Aggregator Integration" product

4.2.1. Towards an application network

The "Aggregator Integration" product is Acme Insurance's immediate priority. It has just one defining feature, the "Create quote for aggregators" feature (Figure 37).

The project to implement the "Aggregator Integration" product kicks off at the Personal Motor LoB, and is actively supported by the newly established C4E within Acme IT (3.1.2). In particular, the C4E's *Platform Architect* spends 50% of his time contributing exclusively to this project, and an experienced architect from the Personal Motor LoB who is assigned full-time to this project also reports to the C4E Lead in his new role of C4E *API Architect*.

This is the first API-led connectivity project at Acme Insurance, so it must establish an Enterprise Architecture compatible with an application network. The resulting application network will at first be minimal, just enough to sustain the "Aggregator Integration" product, but it will grow subsequently when the "Customer Self-Service App" product is realized.

Within the application network and API-led connectivity frameworks, you first architect for the

functional and later, in 5.1, for the non-functional requirements of this feature.

4.2.2. "Create quote for aggregators" feature business process view

Analyzing the "Create quote for aggregators" feature, you observe that it can be realized by one end-to-end, fully-automated business process, the "Create Aggregator Quotes" business process:

1. The business process is triggered by the receipt of a policy description from the Aggregator
2. First it must be established whether the policy holder for whom the quote is to be created is an existing customer of Acme Insurance, i.e., whether they already hold a policy at Acme Insurance
3. Applicable policy options (collision coverage, liability coverage, comprehensive insurance, …) must be retrieved based on the policy description
4. Policy options must be ranked such that options most likely to be attractive to the customer and, at the same time, most lucrative to Acme Insurance appear first
5. One policy quote must be created for each of the top-5 policy options
6. The business process ends with the delivery (return) of the top-5 quotes to the Aggregator

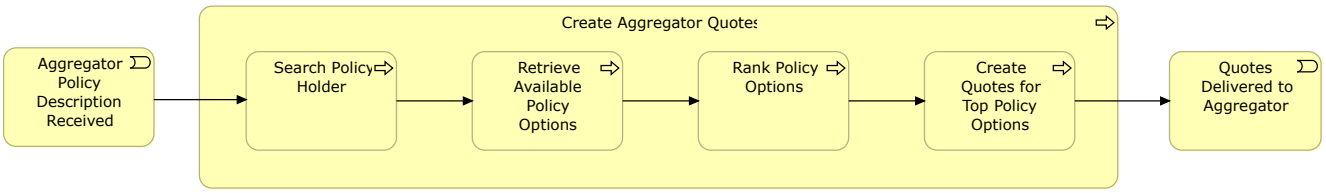

Figure 38. High-level view of the "Create Aggregator Quotes" business process.

4.2.3. Looking ahead to the NFRs for the "Create quote for aggregators" feature

To give a bit more context for the following discussion, it is helpful to briefly inspect the non-functional requirements (NFRs) that will have to be fulfilled for the "Create quote for aggregators" feature: 5.1.1.

4.2.4. Exercise 3: Identify APIs for the "Create quote for aggregators" feature in all tiers

Using the "Create Aggregator Quotes" business process and knowledge of the capabilities of the Policy Admin System (Figure 2), break down the required functionality of the "Create quote for aggregators" feature into APIs in the 3 tiers of API-led connectivity:

- Experience APIs are defined by the Aggregator as the "user-visible app"
  - Custom-designed for a specific user interaction
  - May change comparatively often as user-visible apps change often
- Process APIs implement and orchestrate pure business/process logic
  - Serve Experience APIs but are independent of the concrete top-level API clients that determine the Experience APIs
- System APIs are defined by the needs of the Process APIs and the capabilities of the Policy Admin System
  - Typically many System APIs in-front of the same backend system
  - Change comparatively rarely as backend systems change rarely

Solution

The following APIs could serve to realize the functional requirements of the "Create quote for aggregators" feature with an API-led connectivity approach (Figure 39):

- *"Aggregator Quote Creation EAPI"*: (Aggregator policy description) -> (0-5 ranked motor policy quotes)
  - Receives a description of the details of the motor policy desired by the customer, incl. essential customer data, as specified by the Aggregator. Returns up to 5 matching quotes, ranked (ordered) such that the most preferred quote is first
  - Determines whether the customer is an existing policy holder or new to Acme Insurance
  - Retrieves a ranked list of policy options to be offered
  - Creates one quote for each of the top 5 policy options and returns these, as specified by the Aggregator
- *"Policy Holder Search PAPI"*: (personal identifiers) -> (matching policy holders)
  - Based on personal identifiers (name, dob, SSN, …) returns a description of all matching policy holders of any policies at Acme Insurance, for any LoB
- *"Policy Options Ranking PAPI"*: (policy holder properties, policy properties) -> (policy options ranked from highest to lowest)
  - Given essential properties of a (future) policy holder (age, standing with Acme Insurance, …) and a some aspects of the policy to be offered to that policy holder (type of vehicle/home, value, …) retrieves a list of available options for this policy (collision coverage, liability coverage, comprehensive insurance, theft, …) and ranks them in the order in which they should be offered
- *"Motor Quote PAPI": create*: (policy description, policy holder description) -> (motor policy quote)

    - Given a complete description of a desired motor policy and the (future) policy holder, creates a matching quote and returns its description
- *"Motor Policy Holder Search SAPI"*: (personal identifiers) -> (matching motor policy holders)
    - Searches the Policy Admin System for policy holders of motor policies matching the given personal identifiers (name, dob, SSN, …) and returns matching policy holder's data
- *"Home Policy Holder Search SAPI"*: (personal identifiers) -> (matching home policy holders)
    - Searches the Policy Admin System for policy holders of home policies matching the given personal identifiers (name, dob, SSN, …) and returns matching policy holder's data
- *"Policy Options Retrieval SAPI"*: (policy properties) -> (policy options)
    - Given some aspects of a policy to be created (type of vehicle/home, value, …) returns all options that can possibly be offered for this policy (collision coverage, liability coverage, comprehensive insurance, theft, …)
- *"Motor Quote Creation New Business SAPI"*: (policy description, policy holder description) -> (new business motor policy quote)
    - Given a complete description of a desired motor policy and a future policy holder new to Acme Insurance, creates a "new business" quote for such a motor policy and returns its description
- *"Motor Quote Creation Addon Business SAPI"*: (policy description, policy holder identifier) -> (addon business motor policy quote)
    - Given a complete description of a desired motor policy and an identifier for an existing policy holder, creates an "addon business" quote for such a motor policy and returns its description

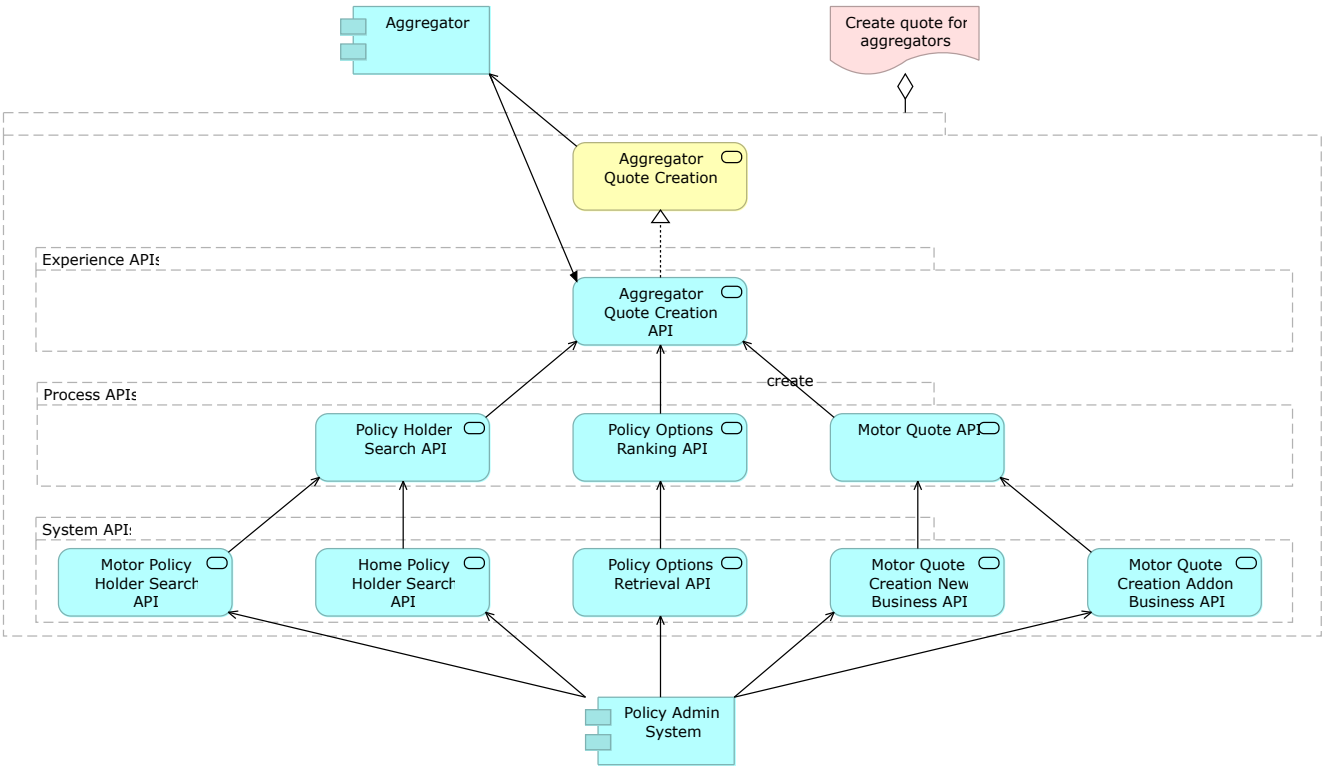

Figure 39. Experience API, Process APIs and System APIs collaborating for the "Create quote for aggregators" feature and ultimately serving the needs of the Aggregator.

You observe that the "Create quote for aggregators" feature can be implemented by one synchronous invocation of the "Aggregator Quote Creation EAPI" which in turn triggers apparently synchronous invocations of several APIs in the other tiers of the architecture, ultimately leading to multiple invocations of the Policy Admin System.

This serves the *functional requirements* of this feature, but will need to be *revisited when NFRs are discussed*.

Summary

- Essential aspect of "Create quote for aggregators" feature implemented by one synchronous invocation of the "Aggregator Quote Creation EAPI"
- In turn triggers invocations of several APIs in all 3 tiers of the architecture
- Ultimately leads to invocations of the Policy Admin System

4.2.5. Exercise 4: Pros and cons of fine-grained APIs and API implementations

Figure 39 is an example of a *pronouncedly fine-grained API architecture*. A break-down of the

same functionality (that of the "Create quote for aggregators" feature) into much more coarse-grained APIs is equally possible:

- Compare and contrast coarse-grained and fine-grained APIs and API implementations

Solution

The granularity of the decomposition of functionality into APIs and API implementations can be analyzed along the following dimensions:

- *Deployability*: Each API and API implementation is independently deployable from all other APIs and API implementations (assuming proper version management (6.2) to not break API dependencies): finer-grained APIs and API implementations allow finer-grained evolution and rollout of functionality (in the form of newer versions of APIs and API implementations)
- *Management*: Each API implementation and API is monitored and managed independently, where the latter includes the enforcement of access control, QoS, etc. (Module 5), which can hence be more finely tailored with more fine-grained APIs and API implementations
- *Scalability*: Resources (memory, number of CPUs, number of machines, …) are allocated to each API implementation independently, and can therefore be tuned (scaled) to each API implementation's specific needs
- *Resources*: Each API implementation consumes a minimum set of resources (CPUs/vCores, CloudHub workers, …) and more API implementations - even if they are smaller - typically means higher resource usage overall
- *Complexity*: Smaller APIs and API implementations are simpler and therefore more easily understood and maintained. Compared to larger and hence fewer APIs and API implementations they also result in more API-related assets visible in the application network and more and more complex interactions (API invocations). In other words, as the complexity of each node in the application network is reduced the complexity of the entire application network increases
- *Latency*: Each additional API invocation adds latency, and smaller APIs therefore cause higher overall latency - which often must be mitigated through caching, etc. (Module 7)
- *Failure modes*: Each additional API invocation is an additional remote interaction between application components, the potential failure of which must be addressed (7.2)
- *Team organization*: Each API can be implemented independently of all other API implementations, assuming that the application interfaces between API implementations - in the form of API specifications - have been agreed. This means that team organization and parallelization of implementation effort are more flexible with fine-grained APIs and API implementations: Business Architecture follows Application Architecture [Ref14]

- *Agility and innovation*: Following from most of the above, smaller APIs and API implementations typically result in shorter innovation cycles because changes and new features can be deployed into production more swiftly

4.2.6. Details of the "Aggregator Quote Creation EAPI"

To be explicit about some of the components of the "Aggregator Quote Creation EAPI":

- The technology interface of the "Aggregator Quote Creation EAPI" is an XML/HTTP API that is invoked by the Aggregator
- The API implementation of the "Aggregator Quote Creation EAPI" invokes various Process APIs, such as the "Policy Holder Search PAPI"

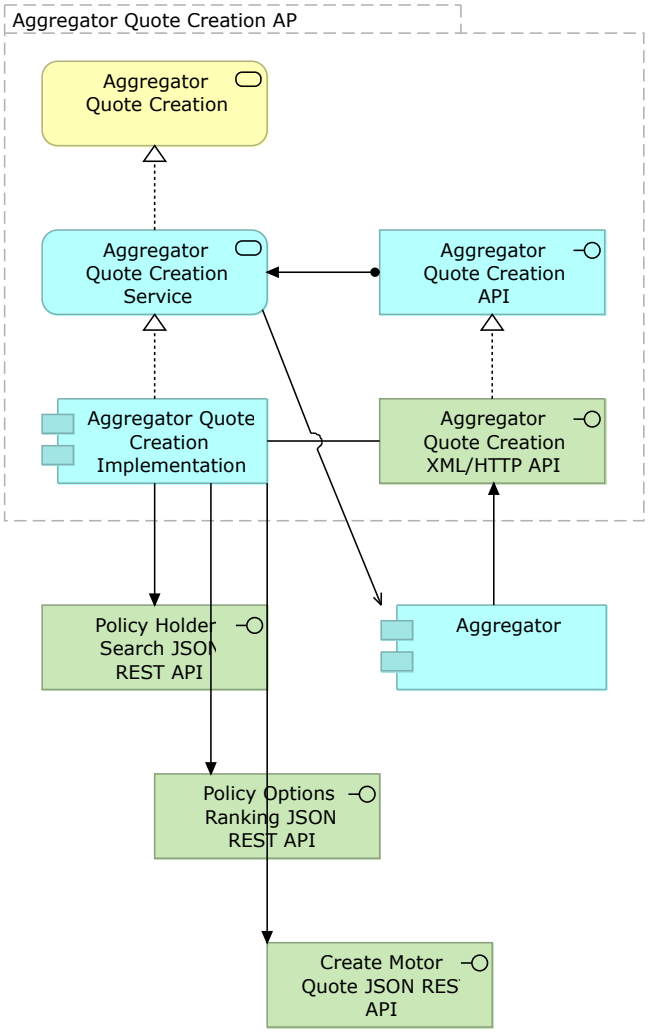

Figure 40. "Aggregator Quote Creation EAPI", serving the Aggregator.

4.2.7. Details of the "Policy Holder Search PAPI"

"Policy Holder Search PAPI" is a Process API with the following important characteristics:

- Its technology interface is a JSON REST API that is invoked by the API implementation of the "Aggregator Quote Creation EAPI"
- Its API implementation invokes two System APIs, one of them being the "Motor Policy Holder Search SAPI"

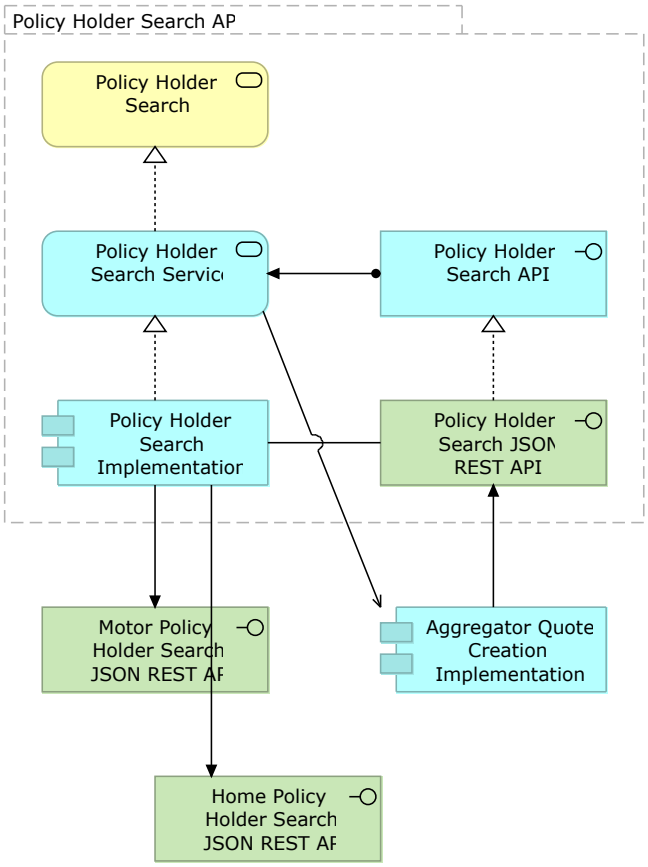

Figure 41. "Policy Holder Search PAPI", initially serving the API implementation of the "Aggregator Quote Creation EAPI".

4.2.8. Details of the "Motor Policy Holder Search SAPI"

"Motor Policy Holder Search SAPI" is a System API with the following important characteristics:

- Its technology interface is a JSON REST API that is invoked by the API implementation of the "Policy Holder Search PAPI"
- Its API implementation invokes the Policy Admin System over an unidentified technology interface (MQ-based, 7.1.3)

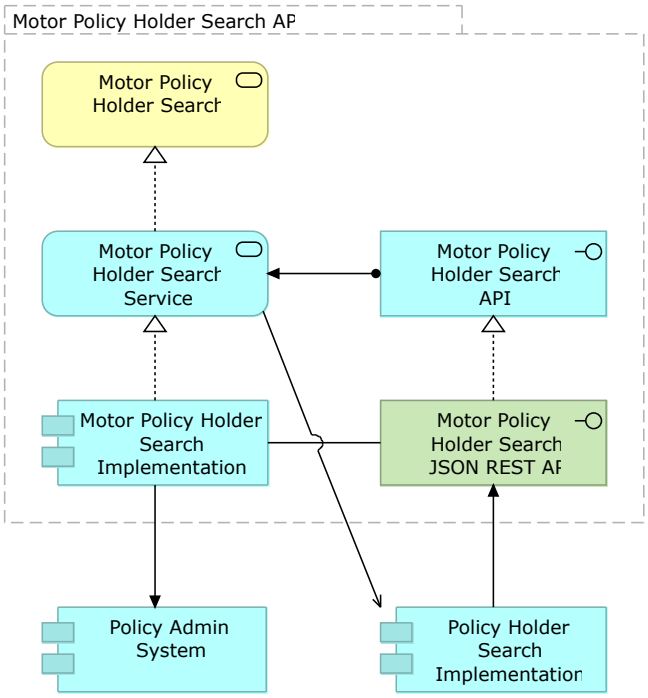

Figure 42. "Motor Policy Holder Search SAPI", exposing existing functionality in the Policy Admin System, and initially serving the API implementation of the "Policy Holder Search PAPI".

4.2.9. API-business alignment

You confirm that this proposal for the APIs realizing the "Create quote for aggregators" feature is aligned with the "Create Aggregator Quotes" business process.

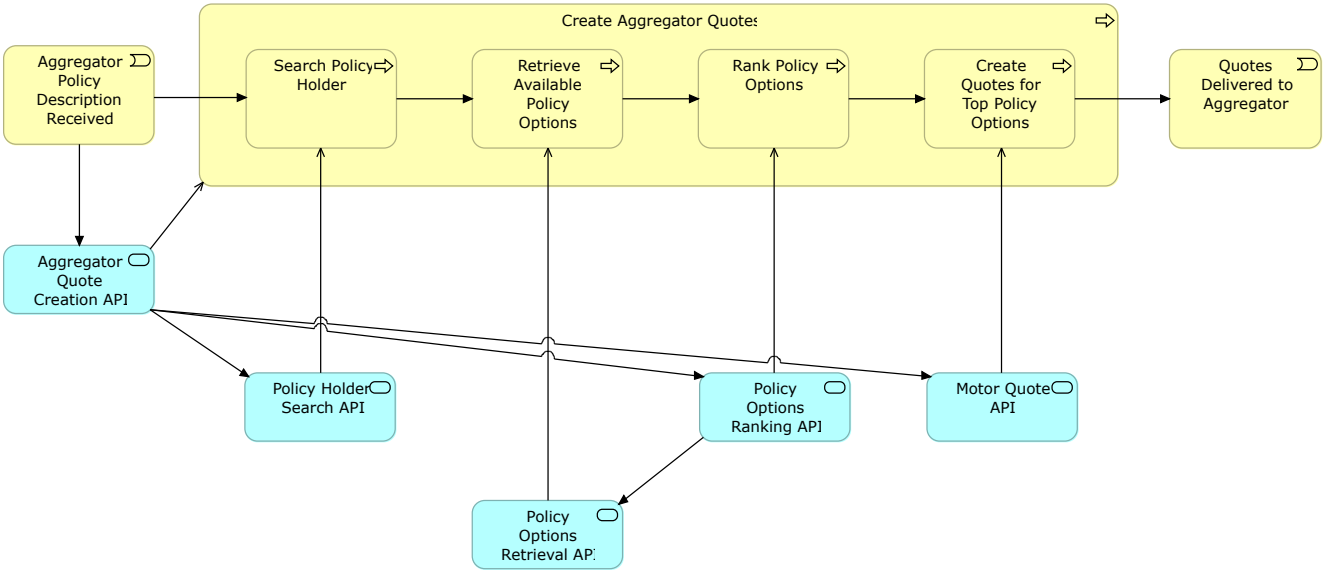

Figure 43. How APIs in all tiers serve the "Create Aggregator Quotes" business process.

4.2.10. Exercise 5: Improve reusability of "Create quote for aggregators" feature APIs

As you better understand the "Create quote for aggregators" feature, you discover that other Aggregators use different data formats for communicating with insurance providers:

1. Analyze the APIs identified for the realization of the "Create quote for aggregators" feature with respect to their dependency on the data format exchanged with the Aggregator
2. Identify new APIs and refine existing APIs to maximize reuse when other Aggregators will need to be supported in the future
3. Describe as clearly as possible which elements of the currently identified APIs will have to change to accommodate your proposed changes

Solution

- Only the "Aggregator Quote Creation EAPI" depends on the Aggregator-defined data format
- In the future there will be one Experience API per Aggregator, similar to "Aggregator Quote Creation EAPI"
- The common functionality of these Experience APIs should be encapsulated in a new Process API, e.g. the "One-Step Motor Quote Creation Process API"
  - Accepts and returns Aggregator-neutral description of policy and quotes
- Changes:
  - "Aggregator Quote Creation EAPI": Interface unchanged, implementation changed significantly to mainly delegate to "One-Step" Process API
  - New "One-Step" Process API with implementation similar to the orchestration logic in the current "Aggregator Quote Creation EAPI"
  - New Experience APIs, one per Aggregator, delegating to "One-Step" Process API
  - Other APIs remain unchanged

This scenario, i.e., the refactoring of an existing Experience API to adapt to an improved understanding of an integration scenario, is a concrete realization of the claim that application networks are *recomposable* and "bend but don’t break" under change (2.2.13). The current Aggregator as an existing client of the "Aggregator Quote Creation EAPI" does not experience any change as the "Aggregator Quote Creation EAPI" API implementation is refactored to use the new "One-Step" Process API. At the same time, technical debt for the existing, misguided implementation of the "Aggregator Quote Creation EAPI" is paid back immediately by the creation of the new "One-Step" Process API and the re-use of the orchestration logic hidden in "Aggregator Quote Creation EAPI".

4.3. Reusing and publishing API-related assets for the "Aggregator Integration" product

4.3.1. Steps to reusing API-related assets

You have identified APIs that will need to be designed and implemented in later stages of the project. But maybe someone in the organization has already provided these or sufficiently similar APIs? You make it a routine to *always check first for the possiility of reusing exisiting APIs by searching Anypoint Exchange*.

As the application network is just being established, Anypoint Exchange currently contains no APIs that can be reused for this feature.

In order to announce the fact that the chosen APIs will be implemented, you immediately *create and publish an Anypoint Exchange entry for each API*:

1. A basic API specification, preferably in the form of a *RAML definition*, is required for each API
2. The creation of the RAML definition must *start in Anypoint Design Center*, from where the API specification can be *exported to Anypoint Exchange*
3. The version of each API should clearly indicate that it is not production-ready yet, e.g., by using `v0` as the version of the RAML definition and `0.0.1` for the corresponding first Anypoint Exchange asset (6.2)
4. A rudimentary *API portal* for each API is then automatically rendered in Anypoint Exchange

The C4E provides guidance and support with these activities. Importantly, the *C4E defines naming conventions for all assets*, including for those to be published in Anypoint Exchange. The following examples illustrate the *naming conventions used by Acme Insurance* for these first steps:

- *Anypoint Design Center projects*: "Policy Holder Search PAPI", similarly for Experience APIs (EAPI) and System APIs (SAPI)
- *RAML version*: `v0`
- *API specification Anypoint Exchange entry*: name: "Policy Holder Search PAPI", asset ID: `policy-holder-search-papi` (group ID is set by Anypoint Exchange to be the Anypoint Platform organization ID), version: `0.0.1` (group/asset ID and version form the "Maven coordinates" of an Anypoint Exchange asset)

4.3.2. Defining RAML

See the corresponding glossary entry.

4.3.3. "Policy Holder Search PAPI" documentation

API documentation and assets need to be created for all APIs identified so far. The discussion here picks the "Policy Holder Search PAPI" as an example.

API documentation for the "Policy Holder Search PAPI" is a form of contract for all elements of the API, i.e., its business service, application service, application interface and technology interface. The RAML definition of the API is the most important way of expressing that contract.

API documentation must be discoverable and engaging for it to be effective: two capabilities that are provided by Anypoint Platform as discussed shortly.

You document various aspects of the API as follows:

- Details of the JSON/REST interface to the API should be exhaustively specified in the RAML definition of the API
- The same is true for security constraints like required HTTPS protocol and authentication mechanisms
  - Currently unknown information can be added later to the RAML definition, for instance when NFRs are addressed
- Other NFRs, like throughput goals are not part of the RAML definition but the wider API documentation, specifically the API’s Anypoint Exchange entry

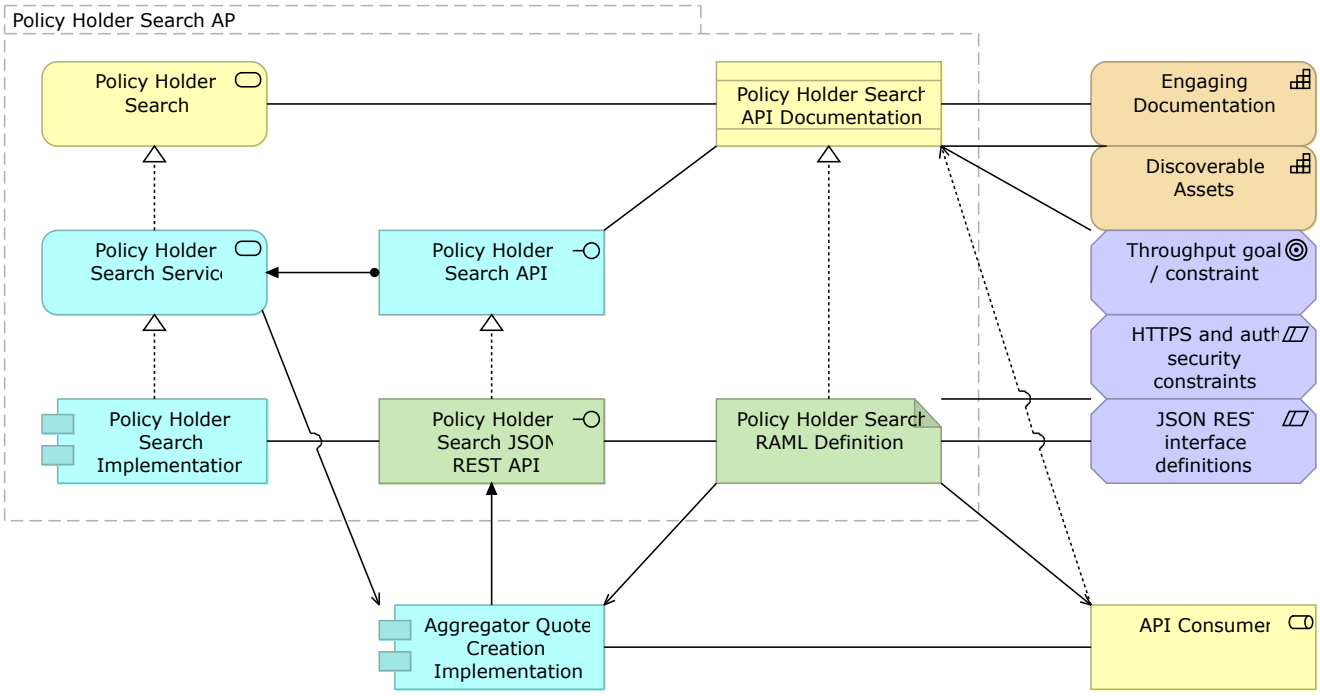

Figure 44. Documentation for the "Policy Holder Search PAPI", including its RAML definition, documents the business service realized by the API, its SLA and non-functional aspects. API documentation must also be discoverable and engaging, as it must be easy to access by API consumers.

4.3.4. Using Anypoint Design Center to sketch and simulate a RAML definition for "Policy Holder Search PAPI"

Using API designer, a feature of Anypoint Design Center, you sketch a first draft of the API specification of "Policy Holder Search PAPI" in the form of a RAML definition such that

- its purpose and
- the essential elements of its interface
  - name, version and description of API
  - preliminary resources and methods

can be communicated widely within the application network.

The RAML definition should capture all of these aspects that are currently known about the API, but may well be at first more of a stub than a complete API specification: it will be amended as the project progresses and the understanding of the API improves.

Using the *mocking feature* of API designer you confirm that the interaction with the API is sound from the API client's perspective.

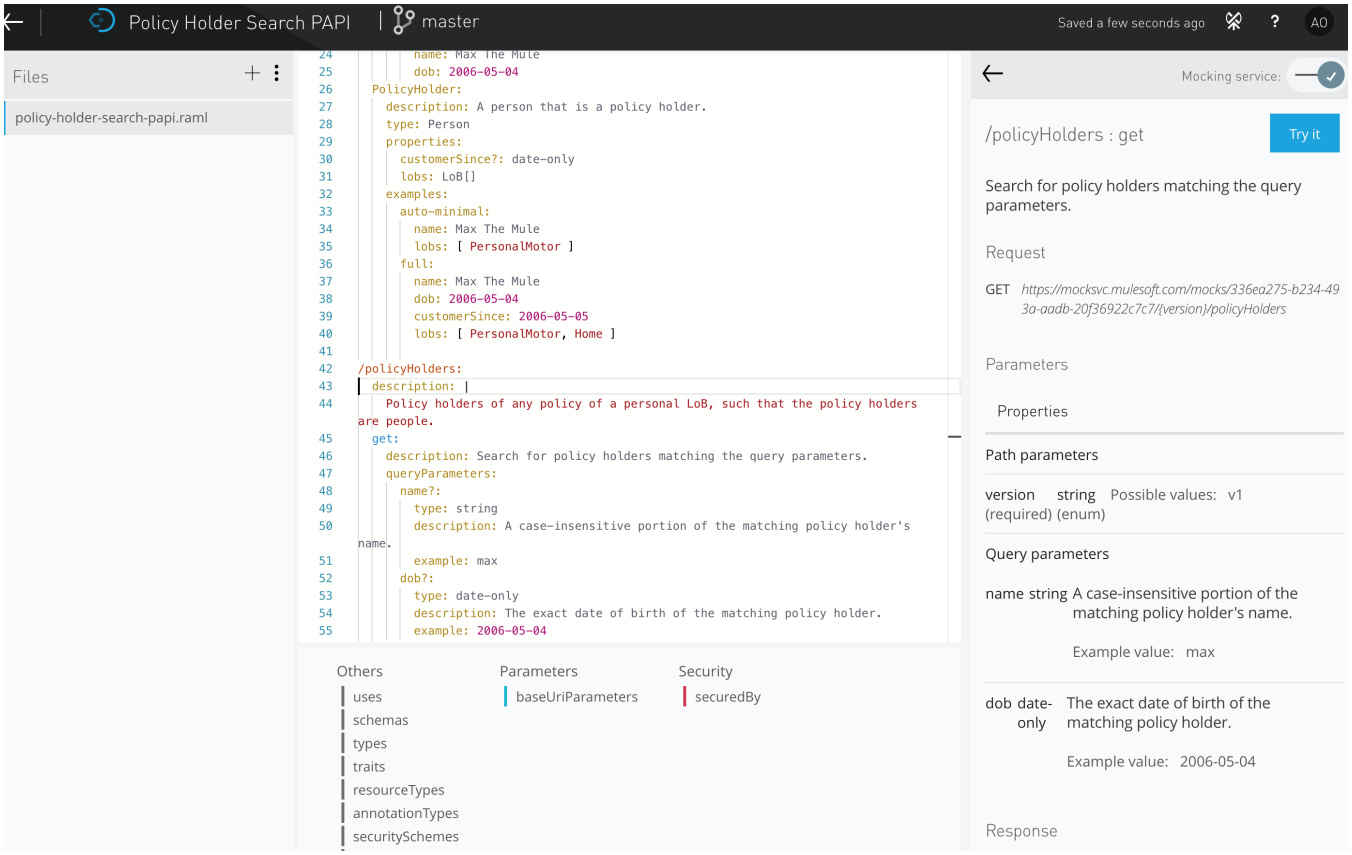

Figure 45. Using the API designer feature of Anypoint Design Center to sketch and try-out (mock) "Policy Holder Search PAPI".

4.3.5. Creating engaging and discoverable documentation and assets for "Policy Holder Search PAPI"

- Anypoint Platform provides two main features for making the documentation for an API engaging: API Notebooks and API Consoles
- Both, plus optional auxiliary documentation, are part of the Anypoint Exchange entry for an API
- Anypoint Exchange entries for an API are linked to the API specification for that API and provide the documentation for the API contract
- Anypoint Exchange entries are discoverable within Acme Insurance and may be included in the organization’s Public (Developer) Portal (Exchange Portal), which makes them discoverable on the public internet
- Anypoint Exchange entries serve API consumers, i.e. the users of the API, amongst other audiences (such as operations teams, 10.5.1)

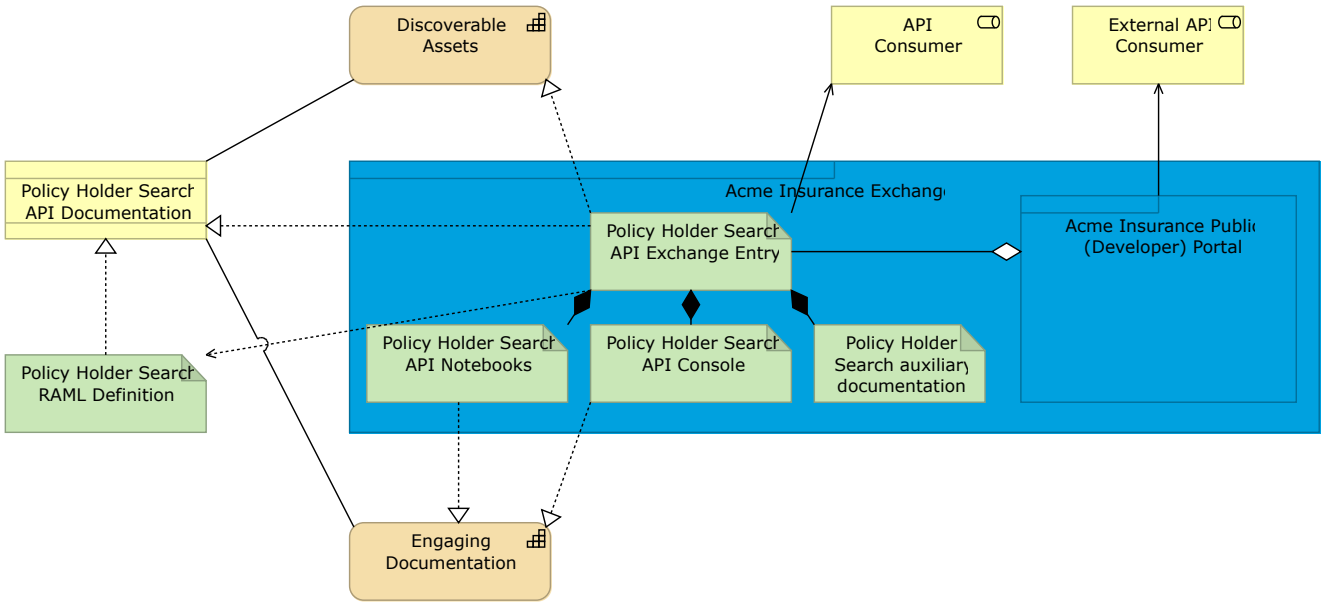

Figure 46. Publishing an Anypoint Exchange entry for "Policy Holder Search PAPI", including API Notebooks and an API Console, and optionally including it in Acme Insurance's Public (Developer) Portal (Exchange Portal), makes for engaging and discoverable documentation of that API serving internal and optionally external API consumers.

4.3.6. Publishing an Anypoint Exchange entry for "Policy Holder Search PAPI"

Anypoint Exchange is a kind of Content-Management System specifically optimized for supporting application networks. Anypoint Exchange can store many kinds of assets, including *RAML definitions, connectors, SOAP web services, and more*.

From within the Anypoint Design Center project for "Policy Holder Search PAPI" *publish to Anypoint Exchange*, thereby creating an Anypoint Exchange entry of type "REST API" for that API.

The Anypoint Exchange entry for an API is the main entry point to the documentation for that API. As soon as a first draft of the RAML definition of "Policy Holder Search PAPI" has been created in Anypoint Design Center, it should be published to Anypoint Exchange to announce its addition to the application network. Use versioning to reflect the maturity (or lack thereof) of a RAML definition.

Strictly speaking, an Anypoint Exchange entry of type "REST API" is for a RAML definition. But Anypoint Exchange provides several features that turn Anypoint Exchange entries for APIs into comprehensive portals for these APIs:

- It separates consumer- and client-facing version (major version) from asset version (full semantic version of the RAML definition artifact): 6.2
- It parses the RAML definition and renders an API Console called "API summary", which allows the exploration and mocking of the API
- It keeps track of API instances, i.e., API endpoints on which API implementations accept requests
- It allows the addition of arbitrary content and specifically supports that creation of API Notebooks

The manually-added content of an Anypoint Exchange entry for each API should at the very least document what cannot be expressed in the RAML definition, such as the HTTPS mutual authentication requirement for the "Aggregator Quote Creation EAPI".

Every change to the content of that RAML definition triggers an asset version increase in the corresponding Anypoint Exchange entry. This behavior is consistent with the fact that Anypoint Exchange is also a Maven-compatible artifact repository - storing, in this case, a RAML definition. See 6.2 for a discussion of versioning API-related artifacts.

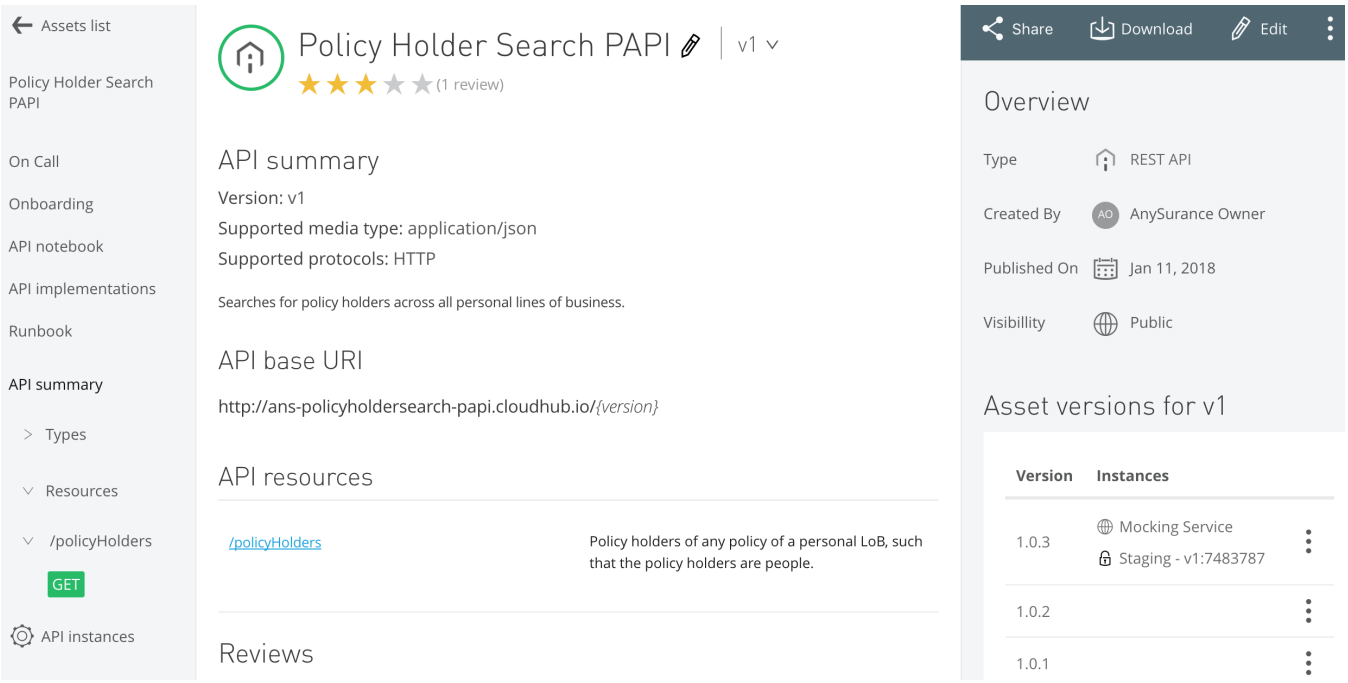

Figure 47. The Anypoint Exchange entry for "Policy Holder Search PAPI", providing a portal for this API, including such views as the auto-generated API Console ("API summary") and instances/endpoints for this and previous versions of this API.

"Policy Holder Search PAPI" can now be discovered in Acme Insurance's Anypoint Exchange, when browsing or searching for any kind of asset, including APIs.

You note that when an API specification is published to Anypoint Exchange then a corresponding Anypoint Connector for Mule applications to invoke the API is also created as a separate Anypoint Exchange entry (Figure 48).

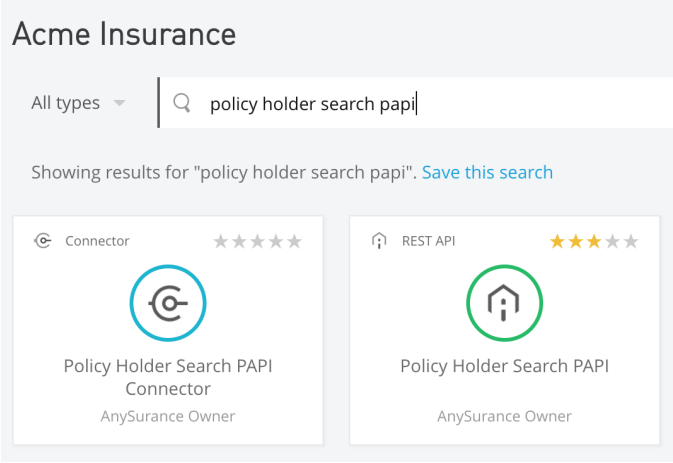

Figure 48. Publishing an Anypoint Exchange entry based on an API specification auto-generates an Anypoint Connector to invoke that API from Mule applications.

4.3.7. Understanding the API Console for "Policy Holder Search PAPI"

Anypoint Platform can fully automatically create a web UI to browse and trigger API invocations of an API with an API specification. This feature is available

- when designing an API in Anypoint Design Center
- as part of an API's Anypoint Exchange entry, where it is called "API summary"
- when implementing the API in Anypoint Studio

The API Console for the "Policy Holder Search PAPI" is automatically included in its Anypoint Exchange entry, based on the preliminary RAML definition created earlier. It allows the invocation of the API against the `baseUri` from the RAML definition as well as any of its known instances/endpoints, including the mocking service.

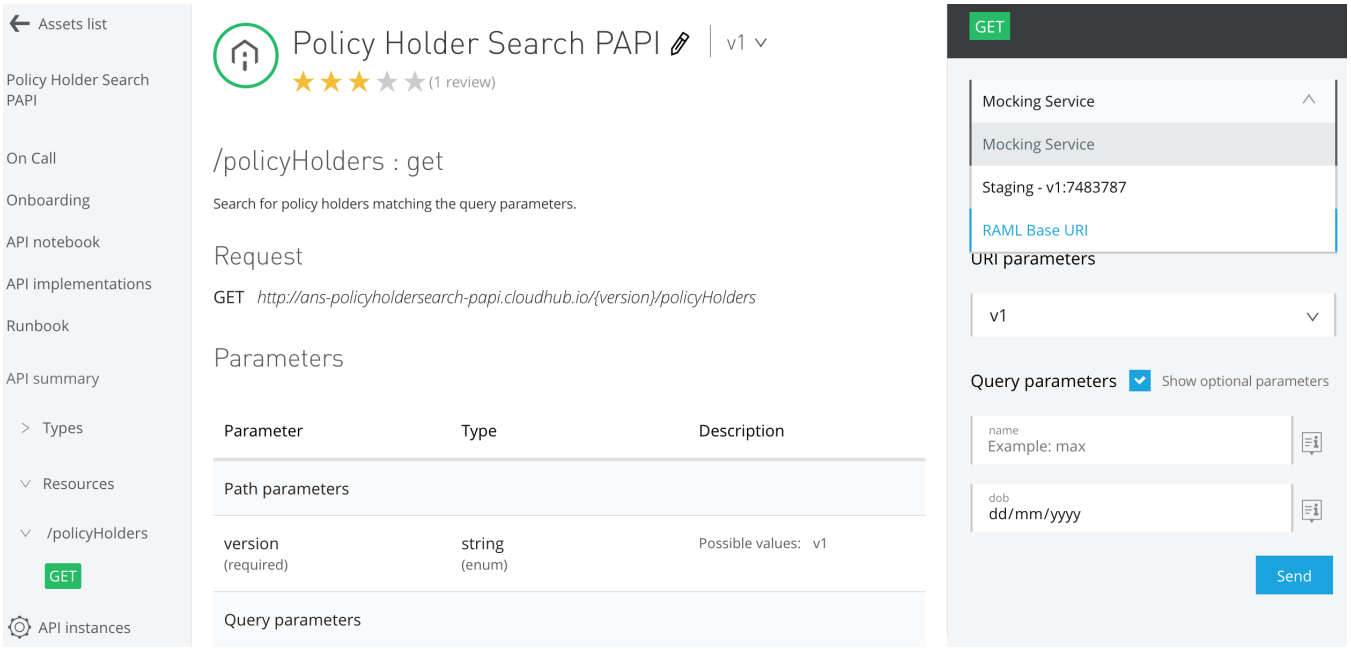

Figure 49. The API Console automatically included in the Anypoint Exchange entry for "Policy Holder Search PAPI".

4.3.8. Including an API Notebook for "Policy Holder Search PAPI"

The API Notebook for an API makes use of the RAML definition for that API and provides an interactive JavaScript-based coding environment that can be used to document interactions with the API from the point of view of an API client. You create an API Notebook for the "Policy Holder Search PAPI" to demonstrate how to invoke the features it provides.

Include an API Notebook for the "Policy Holder Search PAPI" in its Anypoint Exchange entry. This API Notebook makes use of the API's preliminary RAML definition created earlier.

The API Notebook can be created as a new, distinct page or API Notebook code cells can be included in any of the existing (editable) pages. The former is typically preferred, as it makes the API Notebook stand out. In any case, the essence of an API Notebook are its code cells, which are demarcated as Markdown fenced code blocks with the `notebook` info-string (https://github.github.com/gfm/) and contain JavaScript code.

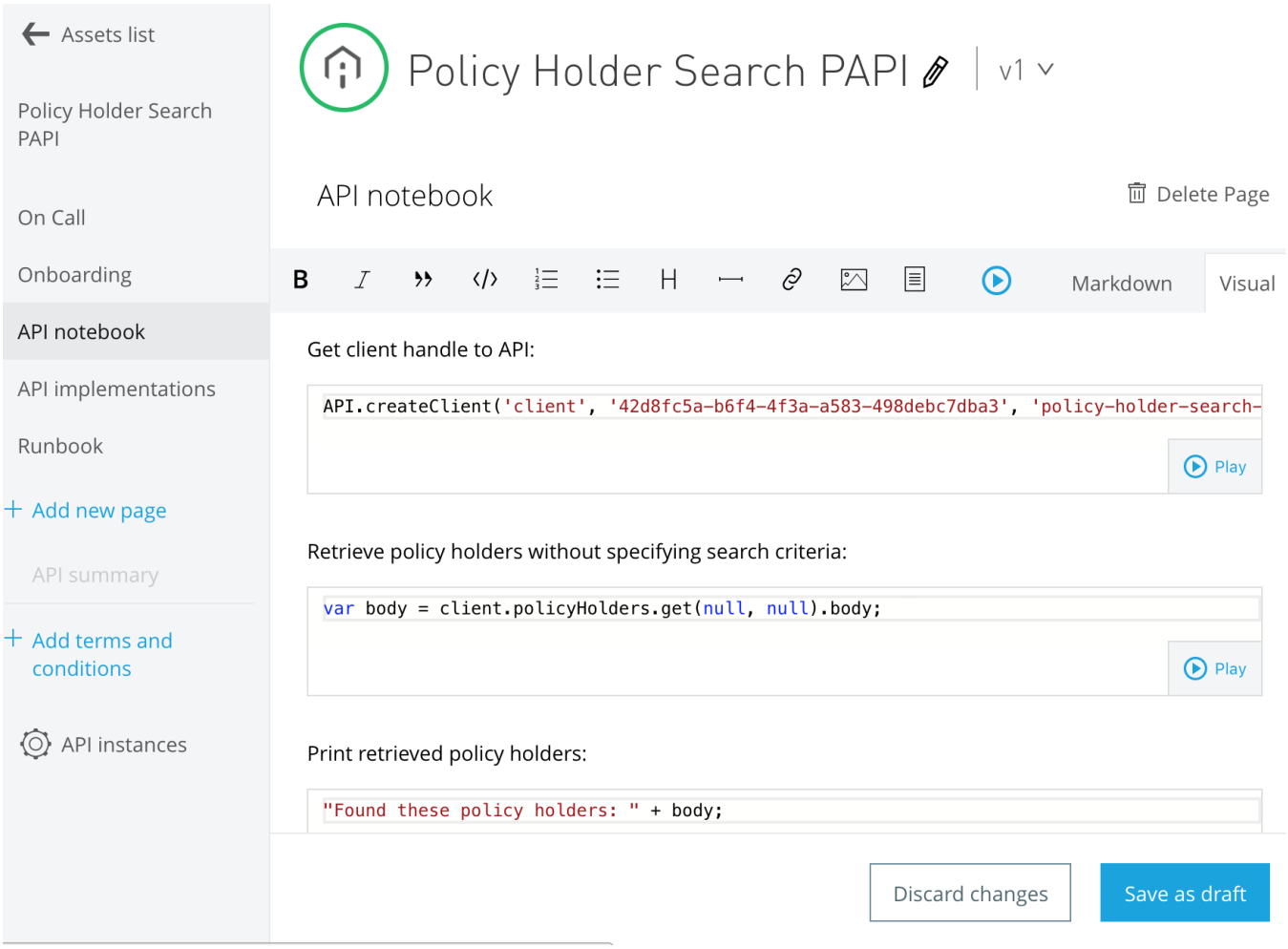

Figure 50. Creating an API Notebook for "Policy Holder Search PAPI".

4.3.9. Publishing the Anypoint Exchange entry for "Policy Holder Search PAPI" to Acme Insurance's Public (Developer) Portal (Exchange Portal)

The newly created portal for "Policy Holder Search PAPI" in the form of its Anypoint Exchange entry can trivially be included in Acme Insurance's Public (Developer) Portal (Exchange Portal) through the sharing functionality in Anypoint Exchange (Figure 51).

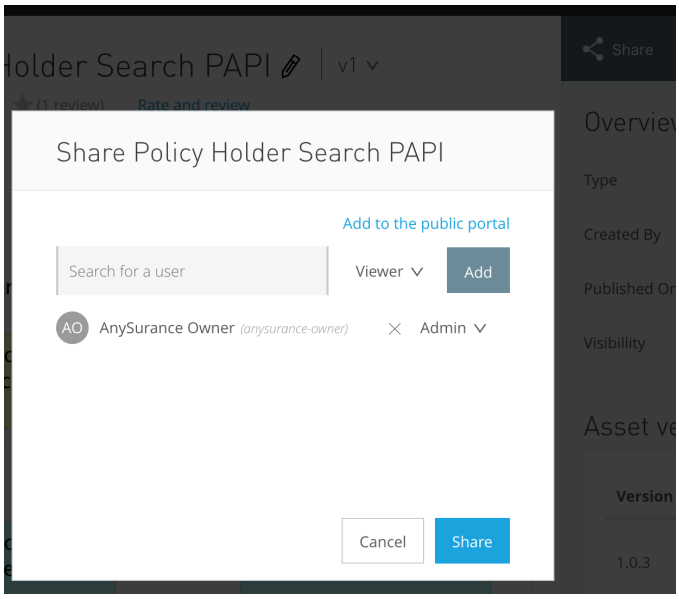

Figure 51. Sharing the Anypoint Exchange entry for "Policy Holder Search PAPI" allows it to be published in Acme Insurance's Public (Developer) Portal (Exchange Portal).

Accessing the Public (Developer) Portal (Exchange Portal) does not require authentication and authorization and hence only publicly visible APIs should be exposed in this way. (This is clearly not the case for the "Policy Holder Search PAPI" used here to illustrate the Public (Developer) Portal (Exchange Portal) - apologies.)

The Public (Developer) Portal (Exchange Portal) can be styled to reflect the Corporate Identify of an organization.

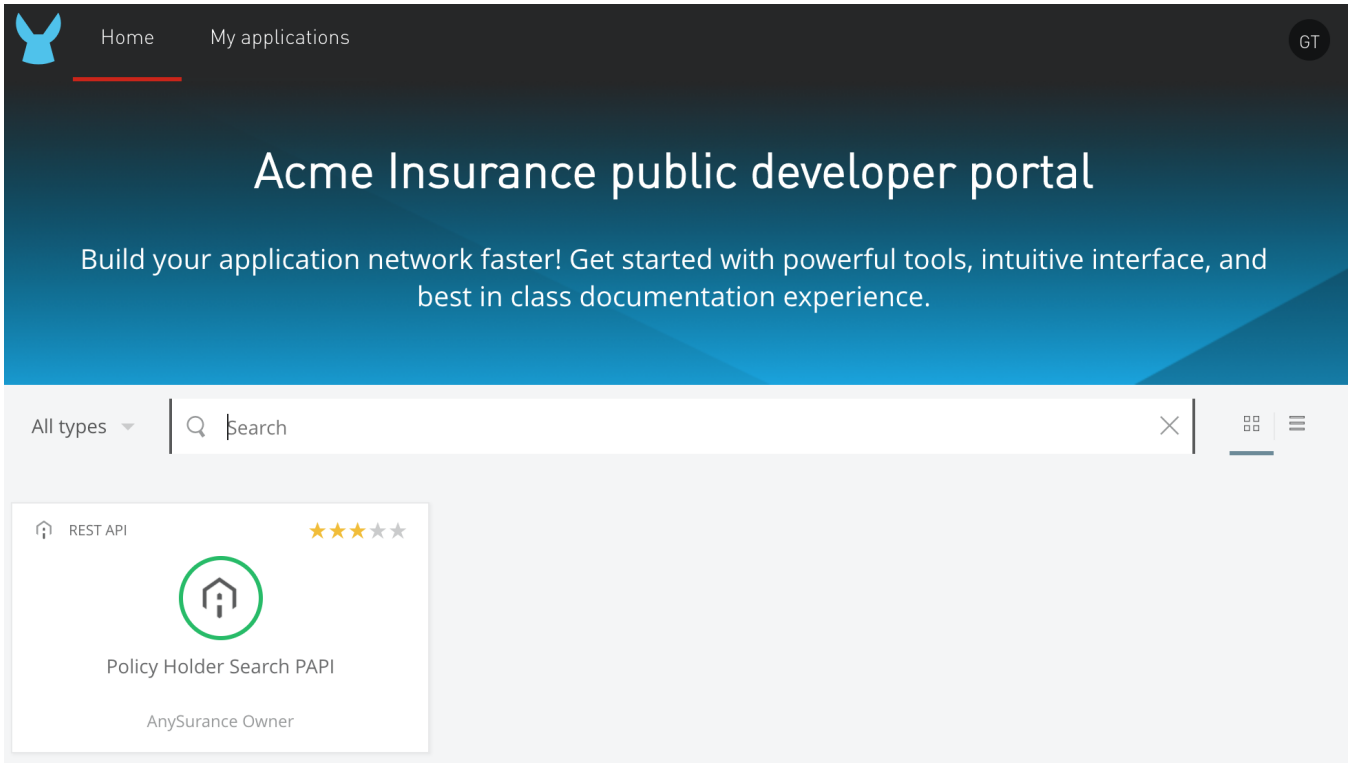

Figure 52. Acme Insurance's lightly styled Public (Developer) Portal (Exchange Portal) showing only publicly visible APIs from Acme Insurance's application network.

The API instances (endpoints) visible in the Anypoint Exchange entry of an API accessed from the Public (Developer) Portal (Exchange Portal) are only those that have been marked with *public visibility*. By default, this includes the mocking service instance.

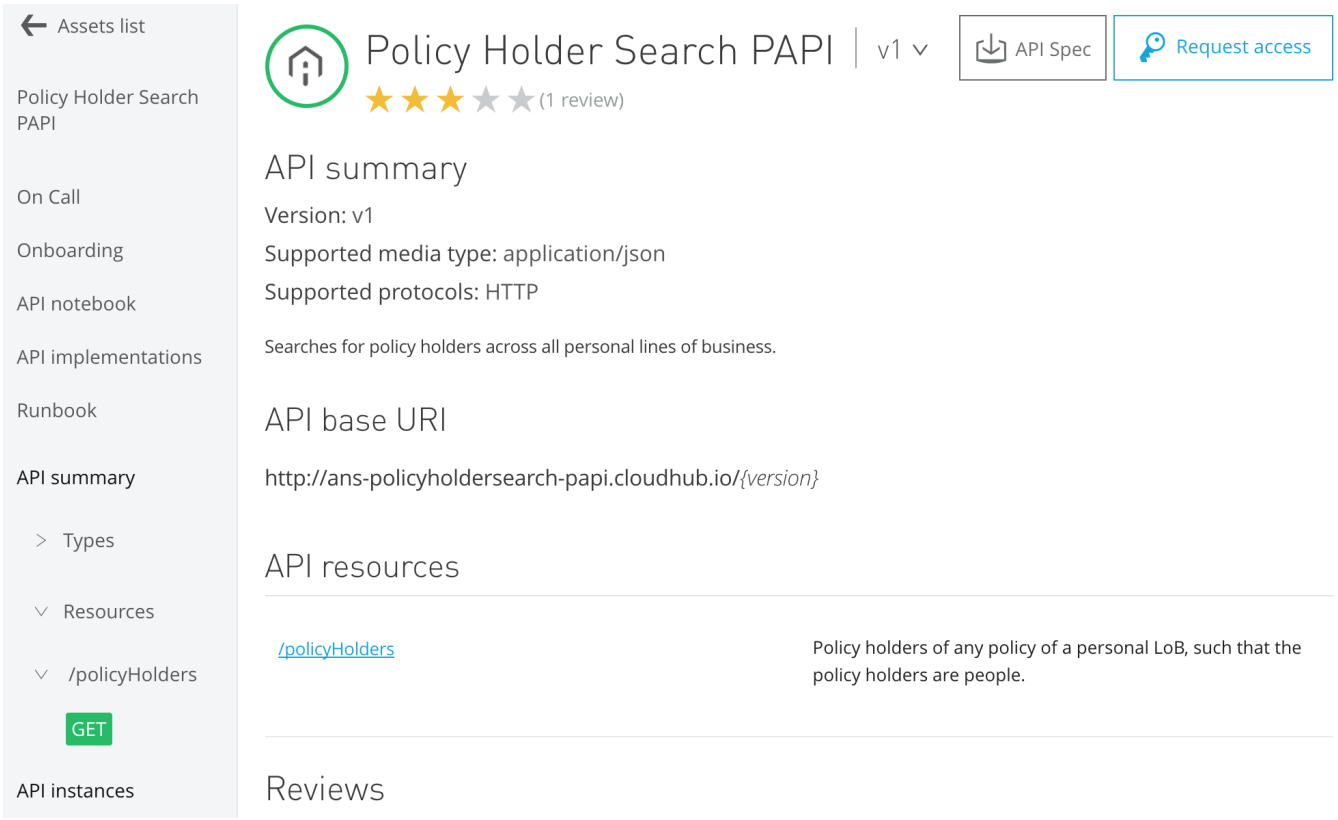

Figure 53. The API consumer's view of the Anypoint Exchange entry for "Policy Holder Search PAPI" when accessed from Acme Insurance's Public (Developer) Portal (Exchange Portal).

4.3.10. Repeat for all APIs for the "Create quote for aggregators" feature

Create rudimentary RAML definitions and corresponding Anypoint Exchange entries with API Notebooks and API Consoles for all APIs needed for the "Create quote for aggregators" feature.

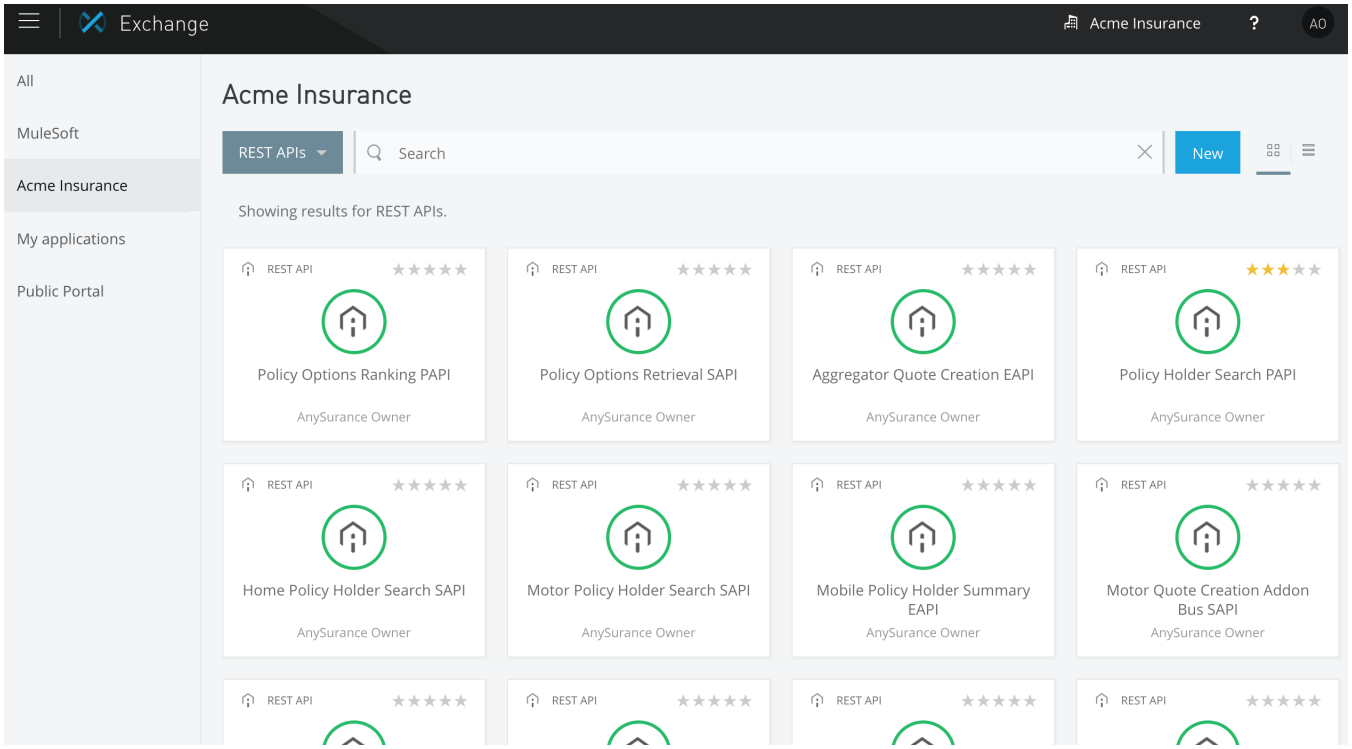

Figure 54. Acme Insurance's Anypoint Exchange showing some of the APIs available in Acme Insurance's application network.

4.4. Identifying, reusing and publishing APIs and API-related assets for the "Customer Self-Service App" product

4.4.1. Growing the application network for the "Customer Self-Service App" product

The "Customer Self-Service App" product (which is defined as a minimally viable product) has just two defining features, the "Retrieve policy holder summary" feature and the "Submit auto claim" feature (Figure 37).

This is the second API-led connectivity project at Acme Insurance, so it can already build on an Enterprise Architecture compatible with a nascent application network.

The project team realizing the "Customer Self-Service App" product is again located at the Personal Motor LoB. However, it is assumed that the "Retrieve policy holder summary" feature will require access to information typically handled by the Home LoB. This product therefore has a wider business scope than the very focused "Aggregator Integration" product addressed earlier. The contribution of the C4E, as a cross-LoB, Acme Insurance-wide entity, is therefore particularly important. The federated nature of the Acme Insurance C4E should come as an

advantage here, because it means that there are C4E-aligned and -assigned roles such as *API Architects in both Personal Motor LoB IT and Home LoB IT*.

Within the application network and API-led connectivity frameworks, you first architect for the functional and then for the non-functional requirements of the two features of the "Customer Self-Service App" product, in turn.

4.4.2. APIs for the "Retrieve policy holder summary" feature in all tiers

The "Retrieve policy holder summary" feature is the first feature of the "Customer Self-Service App" product to be analyzed. It is, however, part of the second API-led connectivity project in Acme Insurance and therefore can build on a foundation of reusable assets.

You analyze the "Retrieve policy holder summary" feature, trying to break it down into APIs in the three tiers of API-led connectivity, checking against Acme Insurance's Anypoint Exchange as you do so:

- You discover the existing "Policy Holder Search PAPI" and decide that it fits the first step in the "Retrieve policy holder summary" feature, so you reuse it from the new "Policy Holder Summary PAPI"
- You define the new "Policy Search PAPI" to support searching for policies across lines of business (motor and home)
- The "Claims PAPI" currently only needs to support searching for claims across LoBs, but is envisioned to ultimately grow to support other operations on claims

All-in-all, the following *new APIs* could serve to realize the functional requirements of the "Retrieve policy holder summary" feature with an API-led connectivity approach (Figure 55):

- *"Mobile Policy Holder Summary EAPI"*: (policy holder identifier) -> (policy holder status summary)
  - Given an identifier of an Acme Insurance policy holder (SSN, customer number, …), returns a concise, mobile-friendly summary of its status as a customer of Acme Insurance, incl. concise summary data about policies held, claims open or recently updated, etc.
  - A particular (strictly speaking non-functional) security aspect this Experience API is that the policy holder identifier shall not be passed as a normal input parameter but must always be taken to be the currently authenticated user (OAuth 2.0 resource owner) on whose behalf the API invocation is made
- *"Policy Holder Summary PAPI"*: (policy holder identifier) -> (policy holder status summary)

    - Given an identifier of an Acme Insurance policy holder (SSN, customer number, …), returns a summary of its status as a customer of Acme Insurance, incl. summary data about all policies held and claims known, etc.
    - It is assumed that the available policy holder identifier is acceptable input to "Policy Holder Search PAPI"
    - The policy holder identifier is passed as a normal input parameter
- *"Policy Search PAPI"*: (policy properties) -> (matching policies)
    - Based on available data describing a policy (data about policy holder, vehicle, insured home address, …) returns a description of all matching policies at Acme Insurance, for any LoB
- *"Claims PAPI": search*: (claim properties) -> (matching claims)
    - Based on available data describing a claim (data about policy, claimant, vehicle, burgled home address, …) returns a description of all matching policies at Acme Insurance, for any LoB
- *"Motor Policy Search SAPI"*: (motor policy properties) -> (matching motor policies)
    - Searches the Policy Admin System for motor policies matching the given policy data (data about policy holder, vehicle, …)
- *"Home Policy Search SAPI"*: (home policy properties) -> (matching home policies)
    - Searches the Policy Admin System for home policies matching the given policy data (insured home address, …)
- *"Motor Claims Search SAPI"*: (claim properties) -> (matching motor claims)
    - Searches the Motor Claims System for motor claims matching the given claim properties (data about policy holder, vehicle, …)
- *"Home Claims Search SAPI"*: (claim properties) -> (matching home claims)
    - Searches the Home Claims System for home claims matching the given claim properties (data about policy holder, burgled home address, …)

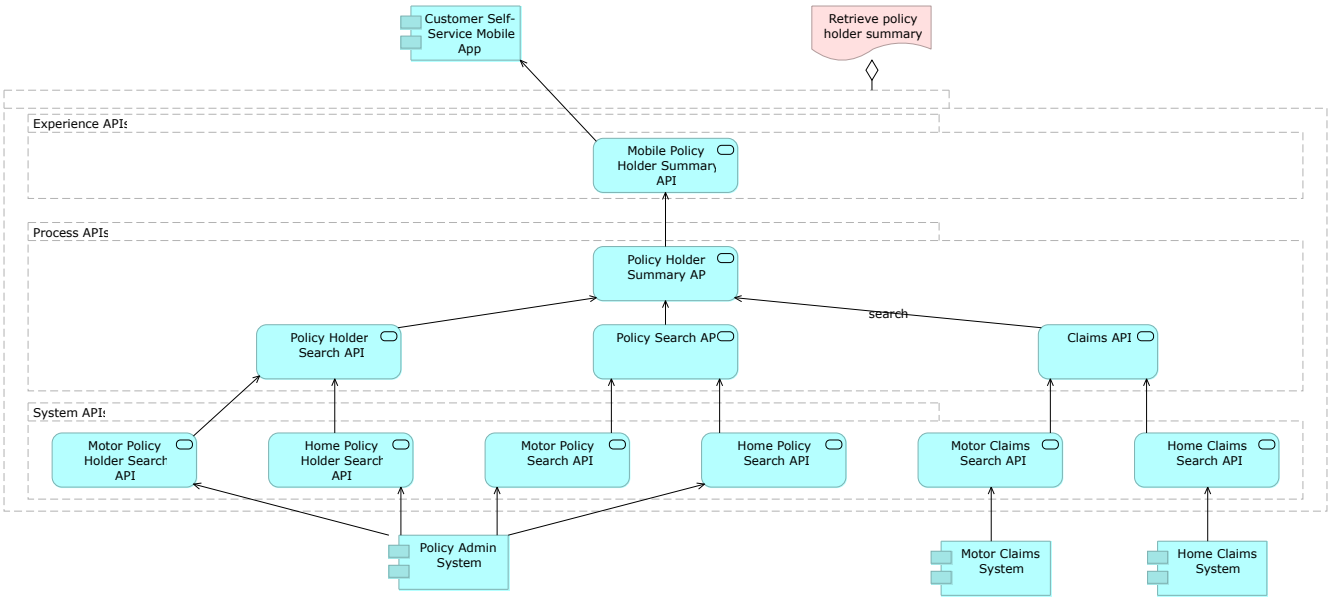

Figure 55. Experience API, Process APIs and System APIs collaborating for the "Retrieve policy holder summary" feature of the "Customer Self-Service App" product.

4.4.3. APIs for the "Submit auto claim" feature in all tiers

You define the "Mobile Auto Claim Submission EAPI" as the interface for the Customer Self-Service Mobile App and the "Motor Claims Submission SAPI" for the interaction with the Motor Claims System.

You *tentatively* define the "Motor Claims Submission PAPI", to insulate the Experience API from the System API. This is because

- it is possible that the Process API will have to perform as-yet undiscovered coordination in order to invoke the System API
- the Process API will likely need to validate the claim submission before passing it on to the System API

All-in-all, the following APIs could serve to realize the functional requirements of the "Submit auto claim" feature with an API-led connectivity approach (Figure 56):

- *"Mobile Auto Claim Submission EAPI"*: (claim description) -> (acknowledgement)
  - Accepts the complete description of a claim to be submitted to Acme Insurance and returns an acknowledgement of the submission.
  - A particular (strictly speaking non-functional) security aspect of this Experience API is that the claim being submitted must be against a policy whose policy holder must always be taken to be the currently authenticated user (OAuth 2.0 resource owner) on whose

behalf the API invocation is made
    - Processing of the claim submission itself is performed asynchronously and its status can be retrieved with the submission ID contained in the acknowledgement.
- *"Motor Claims Submission PAPI"*: (claim description) -> (acknowledgement)
    - Accepts the complete description of a claim to be submitted to Acme Insurance and returns an acknowledgement of the submission.
    - The claim submission can be against any policy and policy holder.
    - Processing of the claim submission itself is performed asynchronously and its status can be retrieved with the submission ID contained in the acknowledgement.
- *"Motor Claims Submission SAPI"*: (claim description) -> (acknowledgement)
    - Accepts the complete description of a claim to be submitted to Acme Insurance and returns an acknowledgement of the submission.
    - Processing of the claim submission itself is performed asynchronously and its status can be retrieved with the submission ID contained in the acknowledgement.

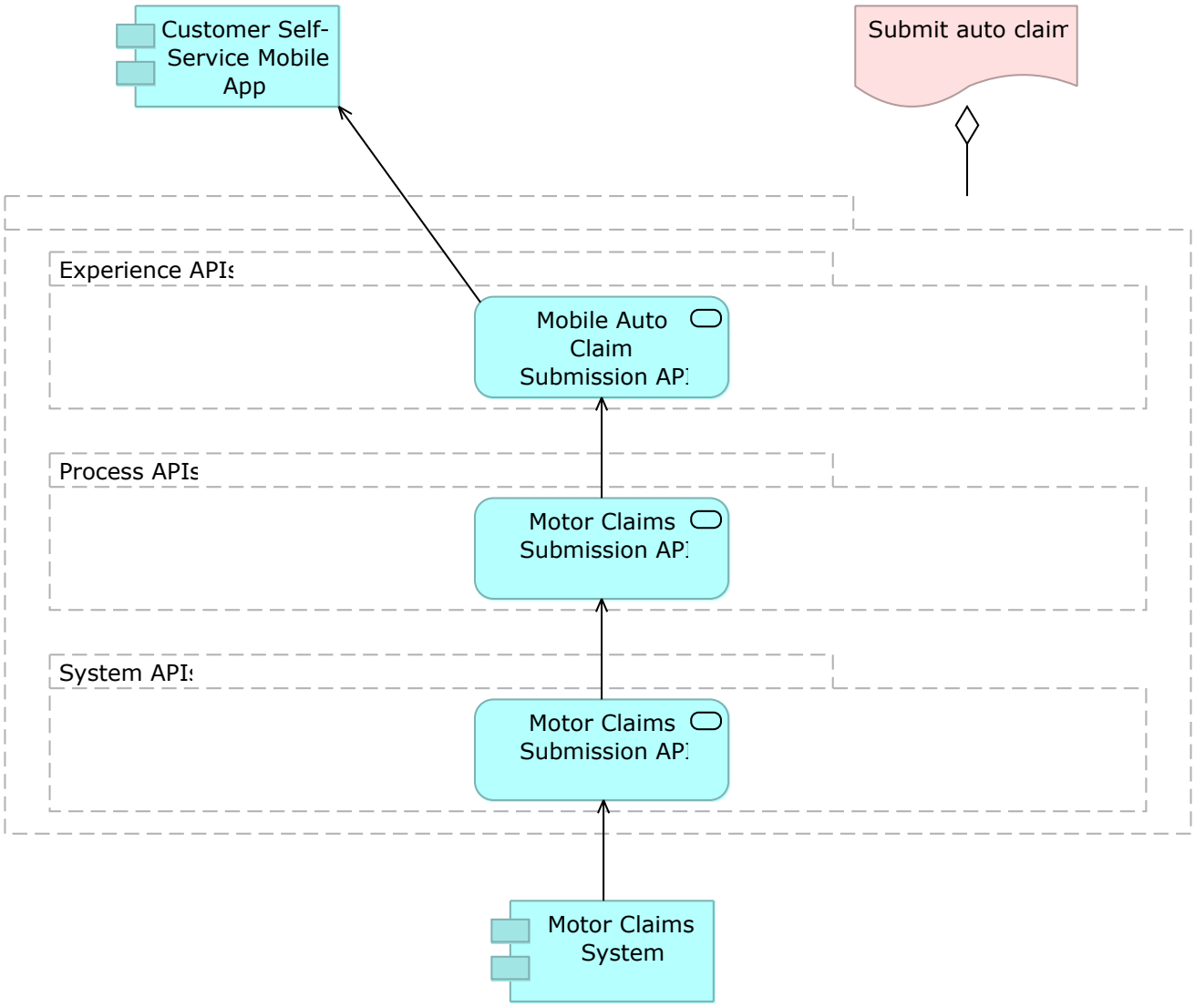

Figure 56. Experience API, Process APIs and System APIs collaborating for the "Submit auto claim" feature of the "Customer Self-Service App" product.

Note that the asynchronicity of the interaction (5.2.3) is not visible in Figure 56.

4.4.4. Publishing API-related assets for the "Customer Self-Service App" product

At this point Acme Insurance's application network has been populated with assets for all APIs needed for the "Aggregator Integration" product and "Customer Self-Service App" product:

- The RAML definitions for the APIs capture the important functional and some non-functional aspects in a preliminary fashion

- An entry in Acme Insurance's Anypoint Exchange has been created based on each API's RAML definition, including an API Console and a rudimentary API Notebook for that API, and pointing to the API's instances/endpoints (which at this point only comprise those from the mocking service)
- The Acme Insurance Public (Developer) Portal (Exchange Portal) gives external API consumers access to all public APIs: these are typically only (some of) the Experience APIs in the application network
- No NFRs have been addressed
- No API implementations and no API clients have been developed

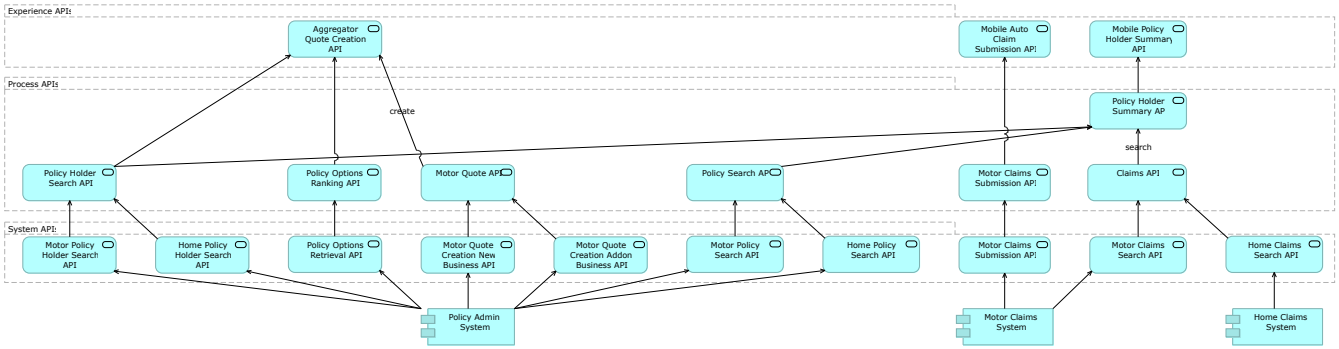

Figure 57. All APIs in the Acme Insurance application network after addressing the functional requirements of the "Aggregator Integration" product and "Customer Self-Service App" product.

See also Figure 52 and Figure 54.

Summary

- Acme Insurance's immediate strategic initiatives require the creation of an "Aggregator Integration" product and a "Customer Self-Service App" product
- The functional requirements of these products have been analyzed:
  - Require 3 Experience APIs, 7 Process APIs and 10 System APIs
  - Aggregator and Customer Self-Service Mobile App invoke Experience APIs
  - API implementations of Experience APIs invoke Process APIs
  - API implementations of Process APIs invoke other Process APIs or System APIs
  - System APIs access the Policy Admin System, the Motor Claims System and the Home Claims System, respectively
- 1 Process API and 2 System APIs originally identified for the "Aggregator Integration" product have been reused in the "Customer Self-Service App" product

- Using Anypoint Design Center, RAML definitions for each API were sketched and simulated
- Anypoint Exchange entries with API Console and API Notebook were created and published for each API

Module 5. Enforcing NFRs on the Level of API Invocations Using Anypoint API Manager

Objectives

- Describe how Anypoint API Manager controlls API invocations
- Use API policies to enforce non-functional constraints on API invocations
- Choose between enforcement of API policies in an API implementation and an API proxy
- Register an API client for access to an API version
- Describe when and how to pass client ID/secret to an API
- Establish guidelines for API policies suitable for System APIs, Process APIs and Experience APIs

5.1. Addressing the NFRs of the "Aggregator Integration" product

5.1.1. NFRs for the "Create quote for aggregators" feature

Aggregators define *strict SLAs* for all insurance providers: they follow a commoditized business model that capitalizes on high traffic volume to their site, with little or no willingness for special treatment of individual insurance providers.

Consequently, the NFRs for the "Create quote for aggregators" feature are dictated primarily by the Aggregator:

- *Synchronous creation of* up to 5 *quotes*:
  - *Aggregator-defined XML*-formatted policy description is sent in *HTTP POST* request
  - Up to 5 quotes may be returned in Aggregator-defined XML format in HTTP response
- Performance:
  - Throughput: up to *1000 requs/s*
  - Response time: *median = 200 ms, maximum = 500 ms* at 1000 requs/s
    - Invocations that exceed the maximum response time are timed-out by the Aggregator
- Security: HTTPS mutual authentication
- Reliability: *quotes* are legally binding and *must not be lost*

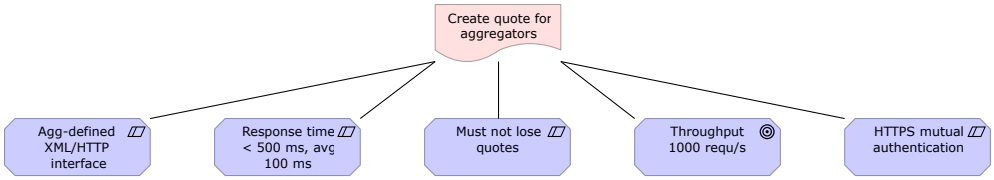

Figure 58. Essential NFRs for the "Create quote for aggregators" feature.

5.1.2. Meeting the NFRs for the "Create quote for aggregators" feature using an Anypoint Platform-based Technology Architecture

The implementation of the "Create quote for aggregators" feature must meet the NFRs listed above. At this point you make a *first attempt* at selecting a Technology Architecture rooted in Anypoint Platform features that addresses these NFRs:

- XML/HTTP interface:
  - Not architecturally significant, should be captured in API specification
- Throughput and response time:
  - Very demanding
  - Must be broken-down to APIs on all tiers
  - Must be *enforced, monitored and analyzed: Anypoint API Manager, Anypoint Analytics*
  - Anticipate the need for caching
  - Select *highly performant runtime plane* for API implementations: *CloudHub*
  - Need to carefully manage load on Policy Admin System: Anypoint API Manager
- Must not lose quotes
  - All-synchronous chain of API invocations, hence reliability requirement can be met by a transactional (ACID) operation on Policy Admin System
    - If the Aggregator receives a quote then that quote must have been persisted in the Policy Admin System
    - If the Aggregator does not receive a quote due to a failure then a quote may still have been persisted in the Policy Admin System, but the Aggregator user cannot refer to that quote and it is therefore "orphaned"
- HTTPS mutual authentication:
  - Possible with *CloudHub Dedicated Load Balancers in Anypoint VPC*
  - Should *add client authentication* on top of HTTPS mutual auth: Anypoint API Manager

5.2. Addressing the NFRs of the "Customer Self-Service App" product

5.2.1. NFRs for the "Retrieve policy holder summary" feature

Initially, this feature is only part of Acme Insurance's own "Customer Self-Service App" product. But it has great potential for re-use, such as opening it up to external API consumers. This would change the NFRs significantly.

- *Synchronous HTTP* request-response chain
- Performance:
  - Currently ill-defined NFRs
  - Aim for *100 requs/s*
  - Aim for avg response time of *2 s* at 100 requs/s
- Security: *HTTPS*, *OAuth 2.0*-authenticated customer
- Consistency: Claims submitted from the Customer Self-Service Mobile App through the "Submit auto claim" feature should be included as soon as possible in the summary returned by the "Retrieve policy holder summary" feature

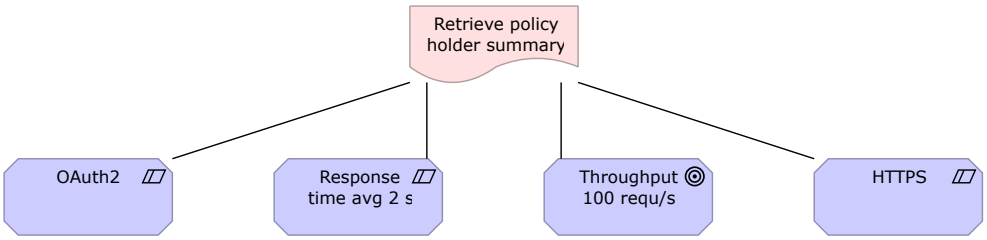

Figure 59. Essential NFRs for the "Retrieve policy holder summary" feature.

5.2.2. Augmenting the Technology Architecture to support the NFRs for the "Retrieve policy holder summary" feature

The implementation of the "Retrieve policy holder summary" feature must meet the NFRs listed earlier - you augment the Technology Architecture selected earlier to try and address these NFRs:

- Throughput and response time:
  - Do not seem overly challenging
  - But future use may change requirements significantly
  - Select *highly scalable runtime plane* for API implementations: *CloudHub*: fits with existing Technology Architecture

- HTTPS:
  - Document in RAML definition
  - Ensure in API implementation
- OAuth 2.0:
  - *Enforce with Anypoint API Manager*
  - Requires Identity Provider for Client Management: PingFederate
- Consistency: to be addressed through event notifications, Module 8

This means that Acme Insurance's PingFederate instance, in addition to serving as an Identity Provider for Identity Management, also assumes the responsibilities for OAuth 2.0 Client Management. The C4E in collaboration with Acme IT configures the MuleSoft-hosted Anypoint Platform accordingly.

5.2.3. NFRs for the "Submit auto claim" feature

Again, this feature is initially only used by Acme Insurance's own "Customer Self-Service App" product, but it has great potential for opening it up to external API consumers, which would change the NFRs significantly.

Processing claim submissions entails numerous automated downstream validation and processing steps, for example:

- Storing images (typically of the accident) sent with the claim submission in a Document Management System
- Checking coverage of the vehicle and driver involved in the accident with the Policy Admin System

Performing these steps synchronously with the claim submission would take too long. Processing claim submissions must therefore be done *asynchronously*.

- *Request over HTTP* with claim submission, with *asynchronous processing* of the submission
- Performance:
  - Currently ill-defined NFRs
  - Aim for *10 requs/s*
  - No response time requirement because processing is asynchronous
- Security: *HTTPS*, *OAuth 2.0*-authenticated customer
- Reliability: *claim submissions must not be lost*
- Consistency: Claims submitted from the Customer Self-Service Mobile App through the "Submit auto claim" feature should be included as soon as possible in the summary

returned by the "Retrieve policy holder summary" feature

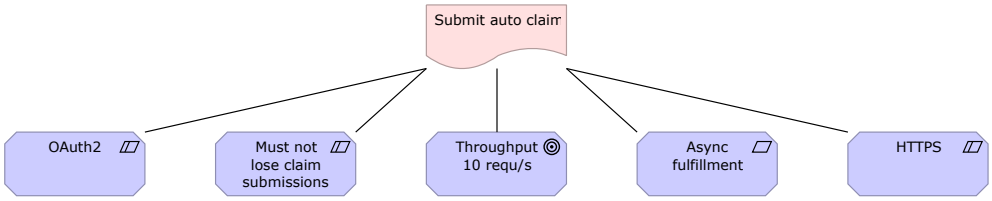

Figure 60. Essential NFRs for the "Submit auto claim" feature.

5.2.4. Augmenting the Technology Architecture to support the NFRs for the "Submit auto claim" feature

The implementation of the "Submit auto claim" feature must meet the NFRs listed earlier - you add to the Technology Architecture accordingly:

- Performance and security requirements: as before for "Retrieve policy holder summary" feature
- Async processing of claim submission and no claim submission loss:
  - Select suitable *messaging system* to trigger asynchronous processing without message loss:
    - *Anypoint MQ or Mule runtime persistent VM queues* as implemented in CloudHub
      - Anypoint MQ would be a new component in Acme Insurance's Technology Architecture (8.3.1)
  - Select suitable *persistence mechanism* to store correlation information for asynchronous processing:
    - *Mule runtime Object Store* as implemented in CloudHub (7.1.13)
- Consistency: to be addressed through event notifications, Module 8

The consistency requirement cannot be met just through communication with the Motor Claims System alone, because once a claim submission is passed to the Motor Claims System it goes through a sequence of transitions that are not visible from outside the Motor Claims System, i.e., are not accessible through the "Motor Claims Search SAPI". Only after some considerable time has passed becomes the newly submitted claim visible to the "Motor Claims Search SAPI" and can therefore be returned via the normal interaction with the Motor Claims System for the "Retrieve policy holder summary" feature. This requirement will be addressed separately in Module 8.

5.3. Using Anypoint API Manager and API policies to manage API invocations

5.3.1. Reviewing types of APIs

Building on the definition of API, what types of APIs are there?:

- REST APIs
  - With API specification in the form of a RAML definition or OAS definition
  - Without formal API specification
  - Hypermedia-enabled REST APIs
- Non-REST APIs
  - GraphQL APIs
  - SOAP web services (APIs)
  - JSON-RPC, gRPC, …

5.3.2. API management on Anypoint Platform

- Using Anypoint API Manager and API policies
- On the level of HTTP
- Applicable to all types of HTTP/1.x APIs
  - Hence not applicable to WebSocket APIs or HTTP/2 APIs like gRPC APIs
- Special support for RAML-defined APIs
  - Allow definition of resource-level instead of just endpoint-level API policies

5.3.3. Defining API policy

See the corresponding glossary entry.

5.3.4. Enforcement of API policies

On Anypoint Platform, API policies are always *enforced from within a Mule application* executing in a Mule runtime:

- An API implementation implemented as a Mule application can *embed* the feature of enforcing API policies
- Alternatively, a separate Mule application called an *API proxy* can be deployed infront of the API implementation proper to enforce API policies for the API exposed by that API

implementation

The API policies themselves are *not* included into any of these Mule applications, just the capability of enforcing API policies. This is true for both the API policy template (code) and API policy definition (data). *API policies are downloaded at runtime from Anypoint API Manager into the Mule application that enforces them*.

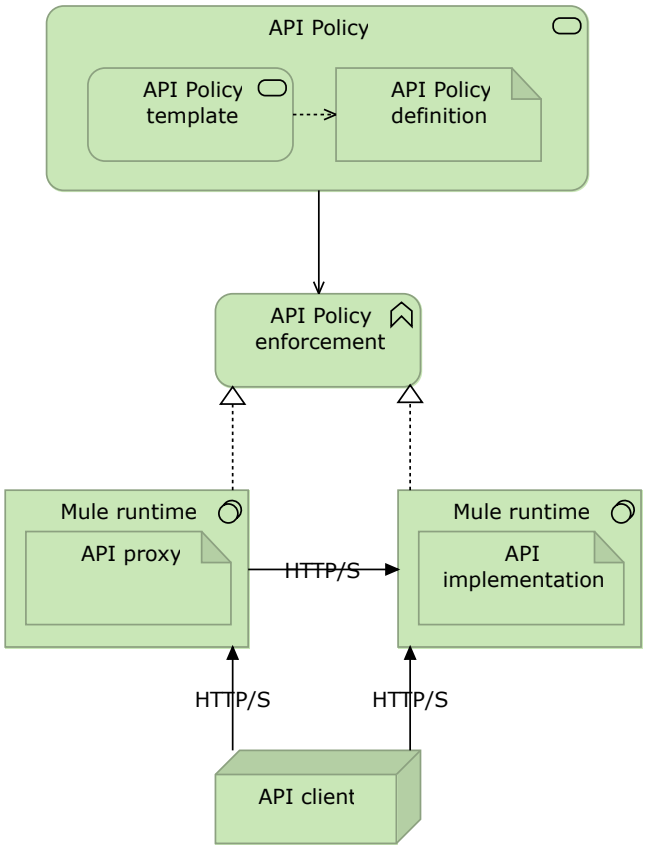

Figure 61. API policies, their structure and enforcement. The API implementation to which an API proxy delegates the API invocation need not necessarily be a Mule application executing in a Mule runtime as shown here.

5.3.5. Exercise 6: Pros and cons of API policy enforcement embedded in the API implementation versus in an API proxy

Compare the characteristics of the two sites of API policies enforcement available in Anypoint Platform:

- List scenarios/requirements that would be best addressed by API policy enforcement embedded in the API implementation or in an API proxy, respectively

Solution

The following scenarios call for either API proxies or embedded API policy enforcement:

- *API implementations are not Mule applications* and hence do not execute in Mule runtimes: API proxies are required
- *Resource-usage must be minized*: embeeded API policy enforcement is preferred because number of nodes approx. doubles when API proxies are used
- Deployment architecture and CI/CD must be as *simple* as possible: embeeded API policy enforcement is preferred
- API policies with *special resource requirements* are applied: API proxies preferred because they allow these API policies to be deployed to dedicated machines (both in number and size)
  - E.g., a caching API policy is best served by few machines with large amounts of RAM, and API proxies can be configured accordingly
  - E.g., a security API policy may require access to a local Hardware Security Module (HSM), and API proxies can be deployed to machines that have this available
- API policies require *special network configuration*, such as access to a highly secure service that is only accessible from a particular subnet: API proxies are preferred because they can be deployed to a different network than the API implementations
- *Security sensitive APIs*, such as sensitive externally accessible Experience APIs: API proxies preferred because they can be deployed to a *DMZ* and they can also shield API implementations from *DoS* or similar attacks, which would be rejected by the API proxy and therefore wouldn't even reach the API implementations.
  - However, because all API invocations to an API implementation go through the API proxies for that API, the DoS attack still has the potential to disrupt the service offered by that API simply by swamping the API proxies with requests

5.3.6. Managing APIs with Anypoint API Manager

Anypoint API Manager is an Anypoint Platform component which provides the following capabilities:

- Management of APIs using the concept of *API instances*
  - An API instance is an entry in Anypoint API Manager that represents a concrete API endpoint for a specific major version of an API in a specific environment (Staging, Production, …)
- Configuration of API policies for a given *API instance*
  - by selecting an API policy template and parameterizing it with an API policy definition

- Configuration of custom API policies in addition to the ones provided by Anypoint API Manager out-of-the-box (5.3.9)
- Is contacted from the site of API policy enforcement to download all API policies that must be enforced
- Definition of alerts based on the characteristics of API invocations
- Admin of API clients ("Client Applications") that have requested access to APIs
  - API consumers use Anypoint Exchange to request access to an API
- Gives access to Anypoint Analytics

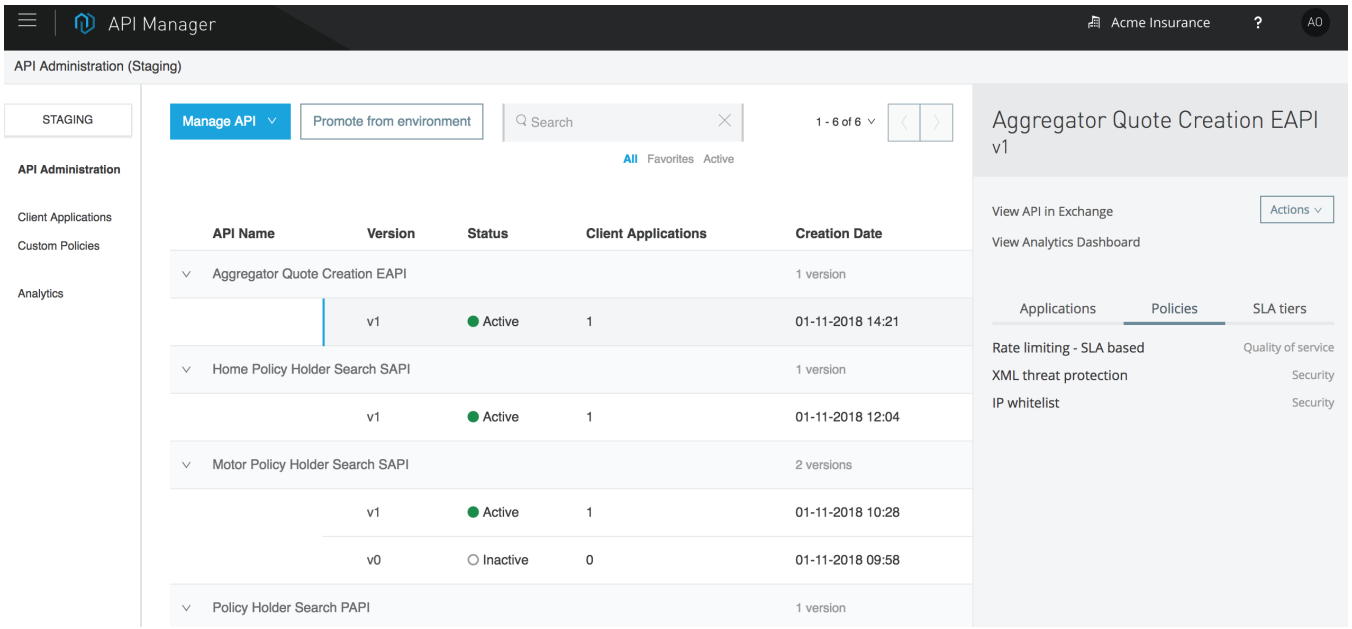

Figure 62. Anypoint API Manager displaying some of the APIs in the Acme Insurance application network.

An API instance must also declare the asset version (full semantic version) currently being exposed by this endpoint. The asset version (e.g., `2.3.4`) must be compatible with the major API version (e.g., `v2`) and can be updated during the lifetime of the API instance

5.3.7. Selectively applying an API policy to some resources and methods of an API

APIs defined with a RAML definition can apply API policies not just to the entire API endpoint (represented in Anypoint API Manager as an API instance) but to selected combinations of API resources and HTTP methods. This configuration is performed when configuring the API policy and is then applied at the time when the API policy is enforced. Because OpenAPI documents can be converted to RAML definitions, this option is also available for APIs defined with OpenAPI API specifications.

5.3.8. API policies available on Anypoint Platform

Anypoint Platform currently provides the following API policies (API policy templates, to be precise) for managing non-functional cross-cutting concerns on APIs:

- Compliance-related API policies
  - Client ID enforcement
  - Cross-Origin Resource Sharing (CORS) (control thereof)
- Security-related API policies
  - API policies performing HTTP Basic Authentication
    - Basic Authentication - LDAP
    - Basic Authentication - Simple
  - IP blacklist
  - IP whitelist
  - JSON threat protection
  - XML threat protection
  - OAuth 2.0 access token enforcement using external provider
  - OpenAM access token enforcement
  - PingFederate access token enforcement
  - OpenId Connect access token enforcement
- QoS-related API policies
  - SLA-based
    - Rate Limiting - SLA-based
  - non-SLA-based
    - Rate Limiting
    - Spike Control
- Transformation
  - Header Injection
  - Header Removal
- Troubleshooting
  - Message Logging

The following API policies were available previously when being enforced on Mule 3 runtimes and/or older versions of Anypoint Platform:

- Replaced by or included in newer versions of HTTP Basic Authentication API policy:
  - HTTP Basic Authentication
  - LDAP security manager (injection thereof)
  - Simple security manager (injection thereof)
- Replaced by Spike Control API policy:
  - Throttling
  - Throttling - SLA-based
- Replaced by Header Injection/Removal API policies:
  - Add/remove request/response headers

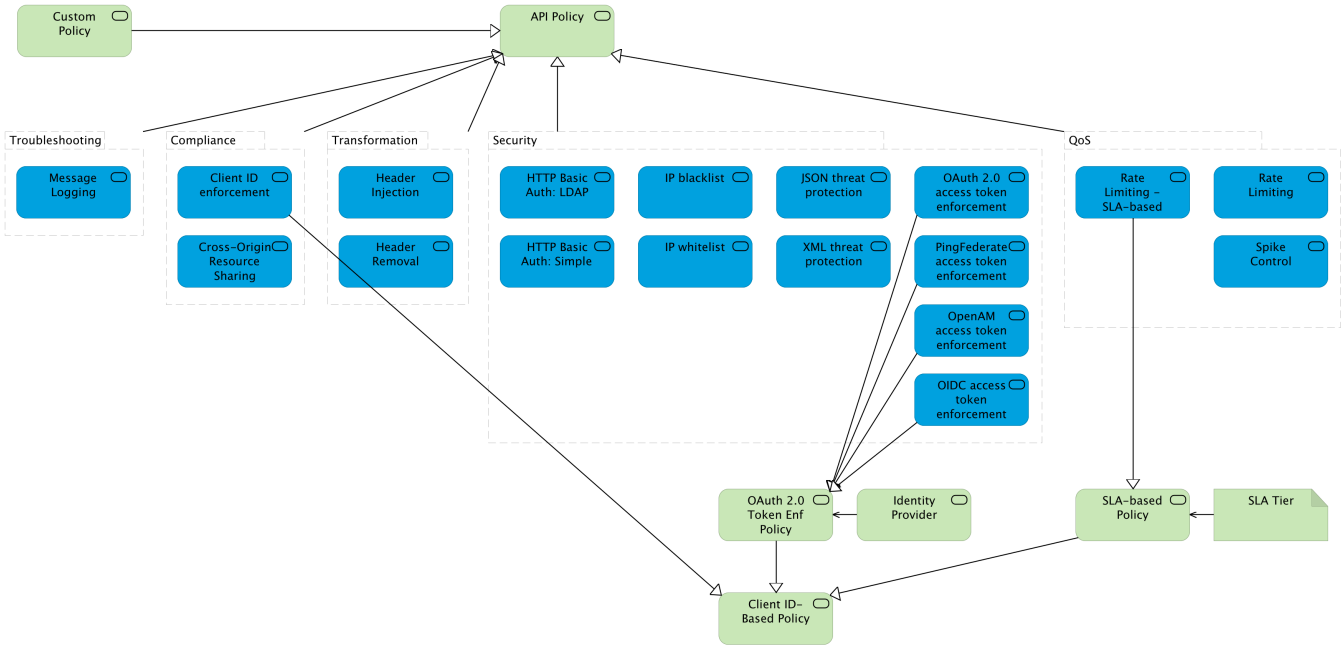

Figure 63. Classification of API policies (templates) available out-of-the-box in Anypoint Platform and the ability to define custom API policies. Only blue API policies are concrete, the others elements are included for clarification.

5.3.9. Understanding custom API policies

API policies can be seen as a form of Aspect-Oriented Programming (AOP) applied to API invocations:

- API policies are ordered in a chain, with the API implementation or API proxy as the last element
- An *incoming HTTP request* for an HTTP endpoint exposed by the API is passed down this chain, and the *outgoing (returning) HTTP response* is passed up the chain

- API policies implement what is called an "around advice" in AOP, i.e., they execute code before handing control to the next element in the chain and after the next element in the chain has handed-back control, altering the HTTP request or response if desired
- In Mule 4 runtimes, API policies can also be applied to *outgoing HTTP requests*, i.e., these API policies can define a separate "around advice" that applies to HTTP requests sent by the API implementation or API proxy and incoming (returning) HTTP responses subsequently received

The mechanics of implementing and applying custom API policies is as follows:

- Custom API policies must be implemented very similar to Mule applications
- They must be packaged specifically as API policies and deployed to Anypoint Exchange
- This package contains the API policy template, i.e., both the code for the API policy as well as a YAML file that describes the API policy's configuration data, i.e., the parameters to be specified when the policy is applied to an API
- When applying an API policy to an API instance, Anypoint API Manager retrieves the API policy template from Anypoint Exchange and renders a configuration UI to enter the definition (parameter values)
- The API policy template and definition are then downloaded as usual to any Mule runtime that registers as that API instance

5.3.10. Introducing compliance-related API policies

Two of Anypoint Platform's API policies can be categorized as related to compliance:

- Client ID enforcement
- CORS control

Client ID enforcement will be discussed in 5.3.19.

The CORS policy participates in interactions with API clients defined by CORS (Cross-Origin Resource Sharing):

- Rejects HTTP requests whose `Origin` request header does not match configured origin domains
- Sets `Access-Control-` HTTP response headers to match configured cross-origins, usage of credentials, etc.
- Responds to CORS pre-flight HTTP OPTIONS requests (containing `Access-Control-Request-` request headers) as per the policy configuration (setting `Access-Control-` response headers)

The CORS policy can be important for Experience APIs invoked from a browser.

See https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS for a good discussion of CORS.

5.3.11. Introducing security-related API policies

Anypoint Platform provides security-related API policies in the following categories:

- Authentication/Authorization
- IP-based access control
- Payload threat protection

5.3.12. Introducing OAuth 2.0 token enforcement API policies

OAuth 2.0-based API policies have a dependency on a suitable Identity Provider for Client Management:

- OpenAM access token enforcement requires OpenAM as an Identity Provider
- PingFederate access token enforcement requires PingFederate as an Identity Provider
- OpenId Connect access token enforcement requires an Identity Provider compatible with OIDC (incl. Dynamic Client Registration), such as Okta
- OAuth 2.0 access token enforcement using external provider requires an external OAuth 2.0 provider that just validates access tokens and is not configured in Anypoint Platform Client Management
  - Client IDs/secrets of API clients registered with Anypoint Platform not kept in sync with such an external OAuth 2.0 provider as would be the case if Client Management were configured at the Anypoint Platform-level
  - The Mule OAuth 2.0 provider is a custom-developed application component that can serve as such an external OAuth 2.0 provider: see template in Anypoint Exchange
  - Use of this API policy is discouraged other than for testing and exploration

5.3.13. Understanding the interaction between Anypoint Platform, PingFederate and the access token enforcement policy

When an Identity Provider such as PingFederate is configured for Client Management on Anypoint Platform, then API clients who register with Anypoint Platform for access to an API, and therefore receive client ID and secret, are kept in sync between Anypoint Platform and the Identity Provider. This is in addition to the Identity Provider validating OAuth 2.0 access tokens

for every API invocation to an API that is protected by the matching access token enforcement API policy - such as PingFederate access token enforcement.

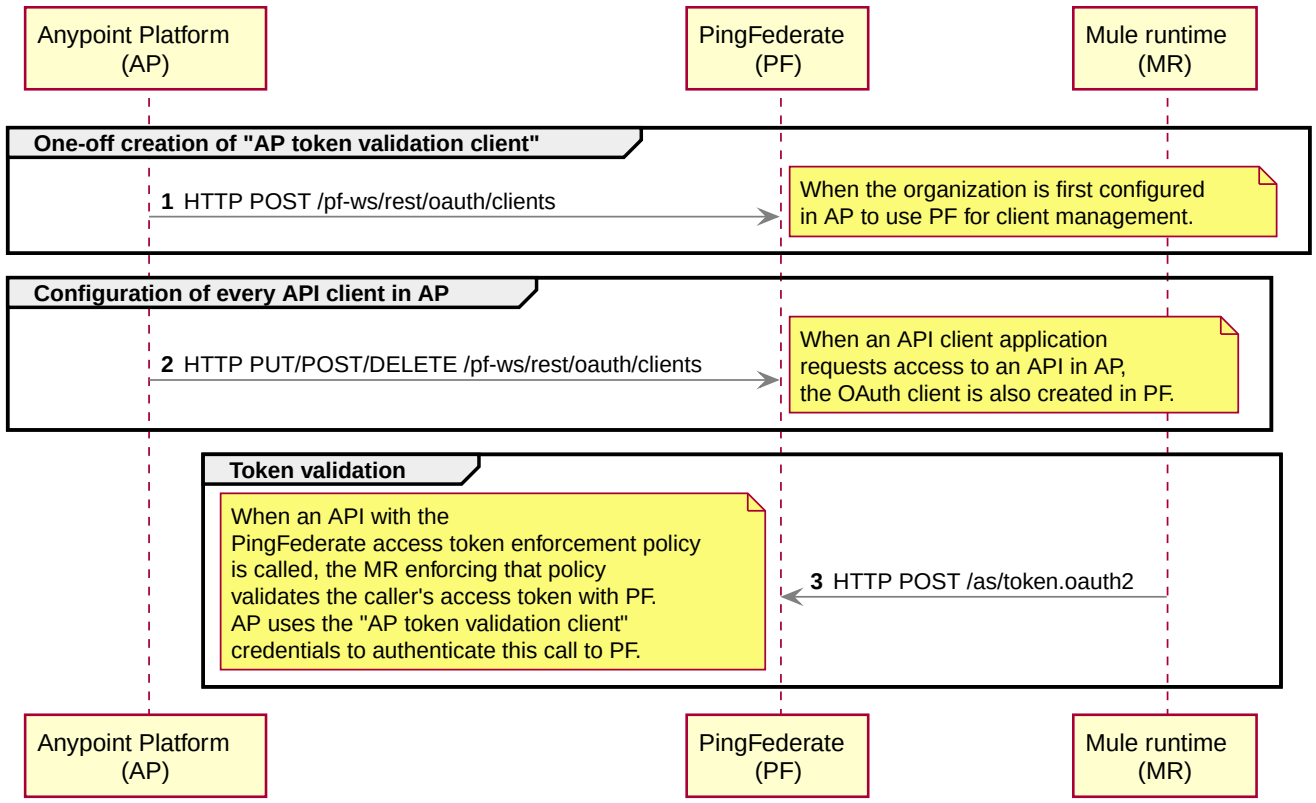

Figure 64. The interaction between Anypoint Platform, PingFederate as an Identity Provider configured for Client Management, and a Mule runtime enforcing the PingFederate access token enforcement API policy.

5.3.14. Introducing API policies for HTTP Basic Authentication

In addition to the above OAuth 2.0-based API policies, Anypoint Platform also supports API policies that enforce HTTP Basic Authentication.

The "HTTP Basic Authentication" API policy must be backed by one of these Security Managers:

- Simple security manager (for testing only)
- LDAP security manager

The Security Manager is made available to the HTTP Basic Authentication API policy through its own "Security Manager Injector" API policy.

Alternatively, if the API implementation is deployed to a Mule 4 runtime, the

- Basic Authentication: LDAP
- Basic Authentication: Simple

API policies configure the functionality of a Simple or LDAP security manager inside the API policy itself and therefore do not require a separate "Security Manager Injector" API policy.

5.3.15. Introducing API policies protecting against JSON and XML threats

These policies guard against attacks that work by sending over-sized HTTP request bodies to an API. They work by *limiting the size of XML and JSON bodies* by setting upper limits on

- nesting levels
- string length
- number of elements

etc.

5.3.16. Introducing QoS-related API policies

Anypoint Platform currently provides two types of API policies related to QoS (Quality of Service) of APIs:

- Rate Limiting (SLA-based and not)
- Spike Control

These API policies *enforce a throughput limit* defined in number of API invocations per unit of time:

- Rate Limiting rejects requests when the throughput limit has been reached
- Spike Control queues requests beyond the throughput limit and delays and limits reprocessing of these requests

Anypoint Platform provides two different ways to define the throughput limit enforced by the Rate Limiting API policy:

- Non-SLA-based, where a throughput limit is defined on the API policy definition associated with a particular API instance
  - Limit is enforced for that API instance and the sum of all its API clients, ignoring the identity of the API clients
- SLA-based, where a throughput limit is defined in an SLA tier

- API clients must register with the API instance at a particular SLA tier
- Limit is enforced separately for each registered API client

Spike Control is only available non-SLA-based.

An SLA-based API policy requires the API client to identify itself with a client ID: 5.3.19. On the other hand, the API clients of APIs without client ID-based API policies can remain anonymous.

When an API client invokes an API that has any QoS-related API policy defined, then the HTTP response from the API invocation may contain HTTP response headers that inform the API client of the remaining capacity as per the QoS-related API policy:

- `X-RateLimit-Reset`: remaining time in milliseconds until the end of the current limit enforcement time window
- `X-RateLimit-Limit`: overall number of API invocations allowed in the current limit enforcement time window
- `X-RateLimit-Remaining`: actually remaining number of API invocations in the current limit enforcement time window

Returning these HTTP response headers is optional (configurable) and should only be done if API clients are internal to the organization, such that external API clients do not become privy to how QoS is enforced for the API.

5.3.17. Introducing Anypoint Platform SLA tiers for APIs

Anypoint Platform (and, specifically Anypoint API Manager) supports the notion of SLA tiers (Service Level Agreement tiers) to enable different classes of API clients to receive different degrees of QoS.

If an API instance has SLA tiers defined then every API client that registers for access to that API instance is assigned to exactly one SLA tier and is thereby promised the QoS offered by that SLA tier.

An SLA tier for an API instance managed on Anypoint Platform

- defines one or more *throughput limits*, i.e., limits on the number of API invocations per time unit
  - E.g., 100 requs per second and simultaneously 1000 requs per hour
  - These limits are *per API client and API instance*
- requires either manual approval or supports automatic approval of API clients requesting

usage of that SLA tier

- typically, SLA tiers that confer high QoS guarantees require manual approval

To enforce the throughput limits of an SLA tier, an SLA-based Rate Limiting API policy needs to be configured for that API instance. The violation of the QoS defined by an SLA tier can be monitored and reported with Anypoint Analytics and can also be the source of alerts.

API clients sending API invocations to an API with enforced SLA tiers must identify themselves via a client ID/client secret pair sent in the API invocation to the API.

5.3.18. Registering API clients with an Anypoint Platform-managed API

API clients that wish to invoke an API that has client ID-based API policy defined, must be registered for access to the API instance. Access must be requested by the API consumer for that particular API client through the Anypoint Exchange entry for that API instance, accessed either directly from Anypoint Exchange or via the Public (Developer) Portal (Exchange Portal). In either case, requesting access to an API requires an Anypoint Platform user account.

In Anypoint Platform, an API client requesting access or having been granted access to an API is called "application" or "client application".

Once the registration request is approved - either automatically or manually - the API consumer receives a client ID and client secret that must be supplied by the nominated API client in subsequent API invocations to that API instance.

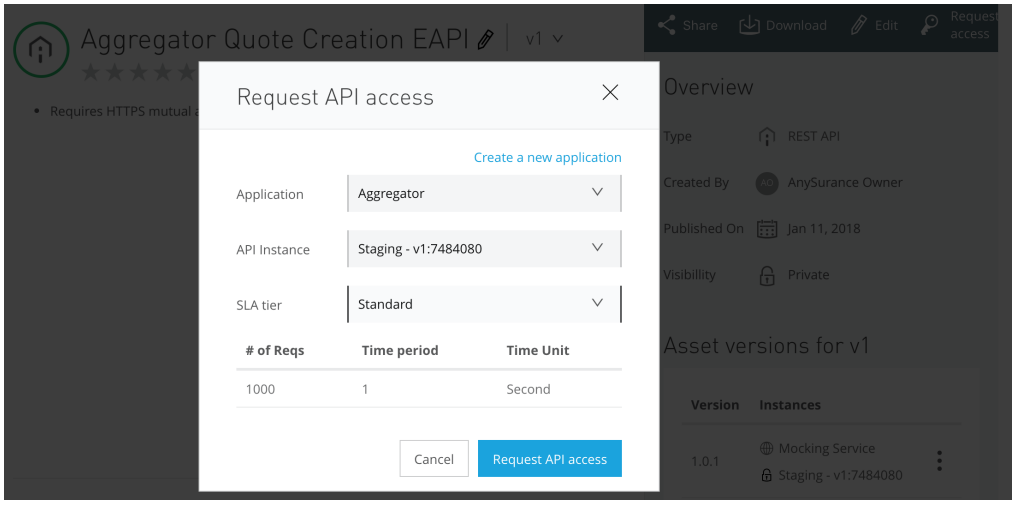

Figure 65. An API consumer is using the Anypoint Exchange entry for a particular (major) version of "Aggregator Quote Creation EAPI" to request access to an API instance of that API for an API client (application) called Aggregator.

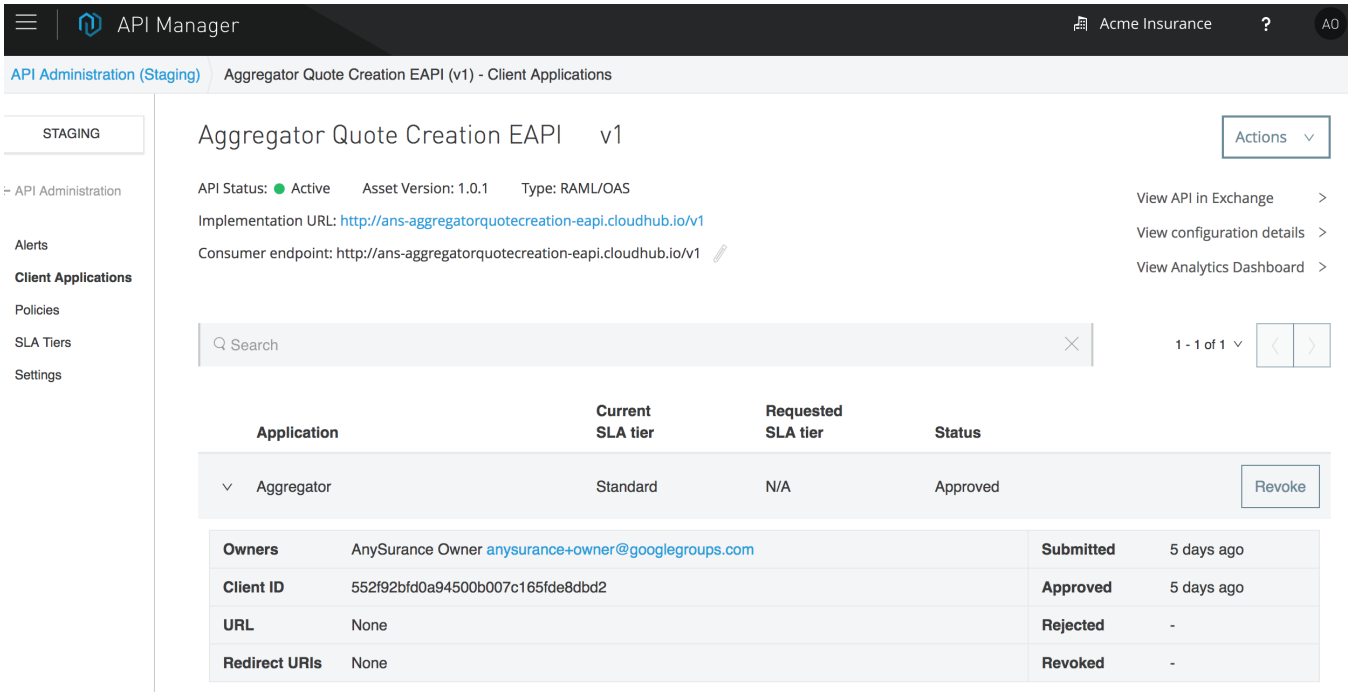

Figure 66. Anypoint API Manager web UI showing the Aggregator as the only API client (application) registered for access to this particular API instance of "Aggregator Quote Creation EAPI". The Aggregator is registered with the "Standard" SLA tier.

5.3.19. Client ID-based API policies

Anypoint Platform provides several API policies that require API clients to identify themselves with a client ID. By default, API clients are also required to send a client secret.

Client ID and secret must be supplied in the API invocation as defined by the API policy. Available options are:

- as query parameters
  - by default `client_id` and `client_secret`
- as custom request headers
- in the standard `Authorization` header as defined by HTTP Basic Authentication
  - where client ID takes the role of username and client secret that of password

The client ID-based API policies currently available in Anypoint Platform are:

- Client ID enforcement
  - Enforces presence and validity of client ID (and typically also client secret)
- Rate Limiting - SLA-based

    - Rejects requests when the throughput limit defined in the SLA tier for the API client has been reached

SLA-based Rate Limiting requires the SLA tier of the API client making the current API invocation to be retrieved by the client ID supplied in the API invocation. This API policy therefore implicitly also enforces the presence and validity of the client ID, and, as a convenience, optionally, also the client secret. It therefor can *subsume the functionality of the Client ID enforcement API policy*.

OAuth 2.0 access tokens implicitly carry the identity of the API client and its client ID, because:

- The API client (and its client ID/secret) is known to both Anypoint Platform and the OAuth 2.0 server (the Identity Provider registered for Client Management with Anypoint Platform)
- When the API client retrieves a token from the OAuth 2.0 server, it does so by identifying itself with its client ID
- The OAuth 2.0 access token enforcement API policy exchanges the token for the client ID and passes it to any downstream SLA-based API policy

Therefore, client secret validation is often turned off if the SLA-based API policy is combined with an OAuth 2.0 access token enforcement API policy. In this case, the OAuth 2.0 API policy exchanges the token for the client ID and makes the client ID available to the SLA-based API policy.

5.3.20. Introducing transformation API policies

Anypoint Platform provides API policies to manipulate HTTP headers in HTTP requests and HTTP responses:

- Header Injection
- Header Removal

The values for the injected headers are expressions, and hence can be dynamically evaluated.

They are useful, for instance, as one part of a solution for propagating transcation IDs as HTTP headers along chains of API invocations.

5.3.21. Exercise 7: Select API policies for all tiers in Acme Insurance's application network

1. Revisit the API policies supported out-of-the-box by Anypoint Platform

2. Select one Experience API, one Process API and one System API from the APIs involved in the "Aggregator Integration" product
3. For each of these APIs, select all API policies that you would recommend applying to that API
4. Also define the order for these API policies
5. Are there any API policies missing that you would want to apply?

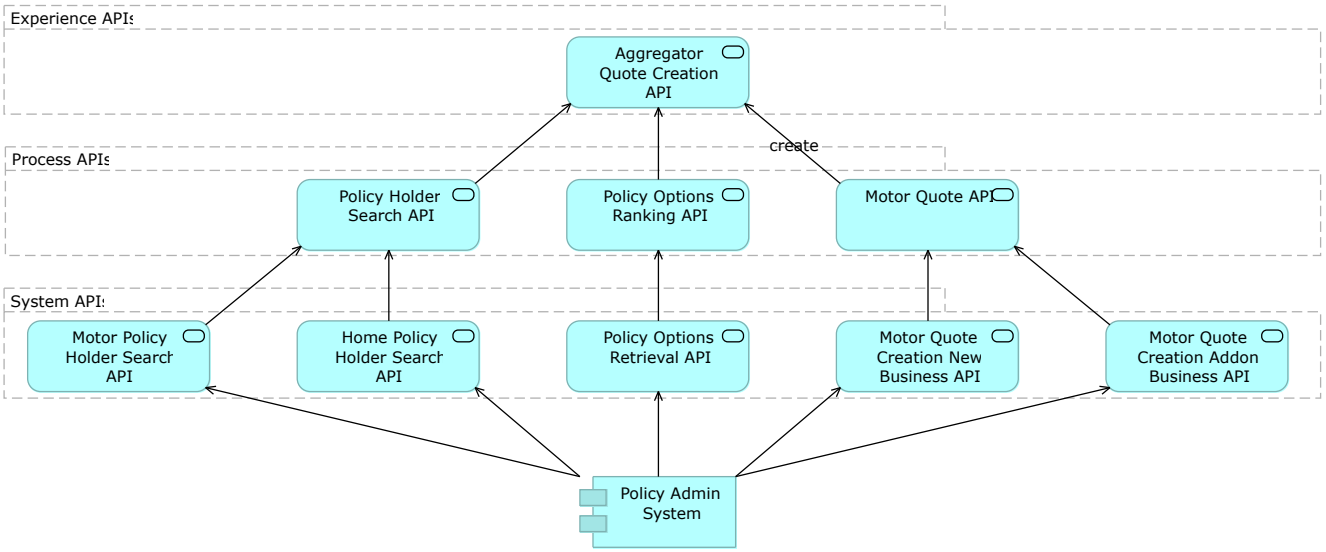

Figure 67. All APIs collaborating for the "Aggregator Integration" product.

See Figure 63.

Solution

See 5.3.22, 5.3.23, 5.3.24 and 5.3.25.

5.3.22. Choosing appropriate API policies for System APIs

Acme Insurance's C4E defines the following guidelines for defining API policies on System APIs:

- *IP whitelisting* to the IP address range of the API implementations of Process APIs (assuming they execute in a subnet that assigns well-defined source IPs to them)
- Must always define *SLA tiers that require manual approval*
- *SLA-based Rate Limiting* API policy must enforce the QoS of the chosen SLA tier
- *Spike Control* API policy must protect the backend system from temporary API invocation bursts by evening them out and enforces the published overall throughput guarantee

- `X-RateLimit-` HTTP response headers should not be exposed to API clients from this API policy but from the SLA-based API policy

- These API policies should be applied in this order and thereby enforce strict compliance for this critical class of APIs

Acme Insurance applies these guidelines to all System APIs in their application network.

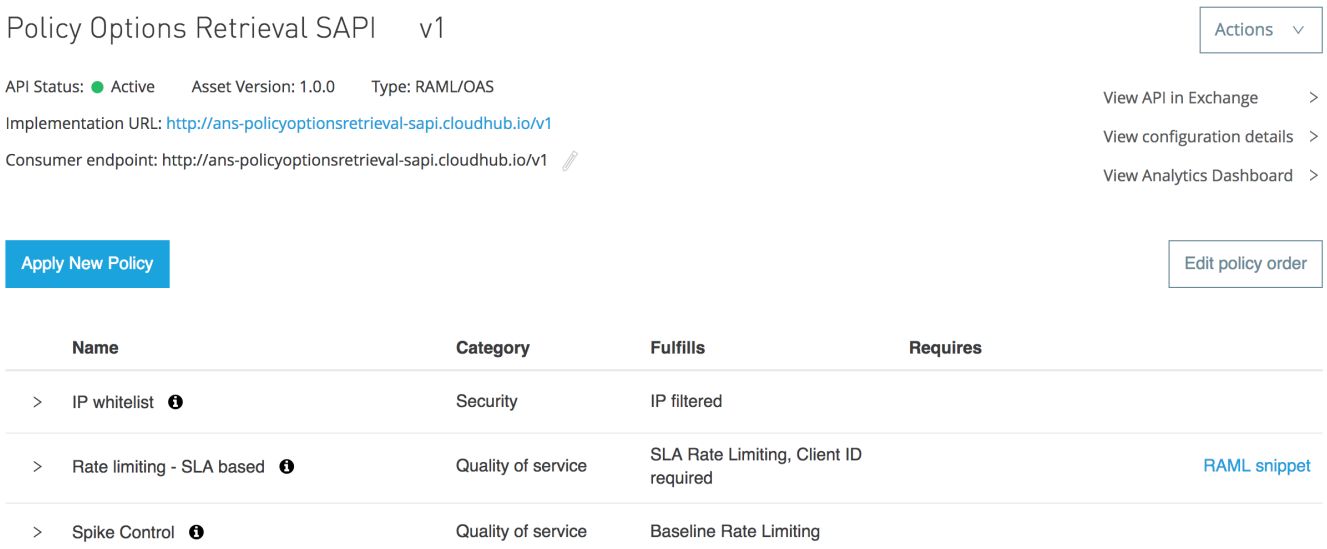

Figure 68. API policies defined for a particular API instance of the "Policy Options Retrieval SAPI".

5.3.23. Choosing appropriate API policies for Process APIs

Acme Insurance's C4E defines the following guidelines for defining API policies on Process APIs:

- *IP whitelisting* to the IP address range of the API implementations of Process APIs and Experience APIs (assuming they execute in a subnet that assigns well-defined source IPs to them)
- May define *non-SLA-based API policies* but should then use *Client ID enforcement*, such that the identity of API clients is always known and analyses per API client can be performed
- If SLA tiers are defined then *SLA-based Rate Limiting* API policy must enforce the QoS of the chosen SLA tier
- *Spike Control* API policy must protect from temporary API invocation bursts by evening them out and enforces the published overall throughput guarantee
  - `X-RateLimit-` HTTP response headers should not be exposed to API clients from this API policy

- These API policies should be applied in this order

Acme Insurance applies these guidelines to all Process APIs in their application network.

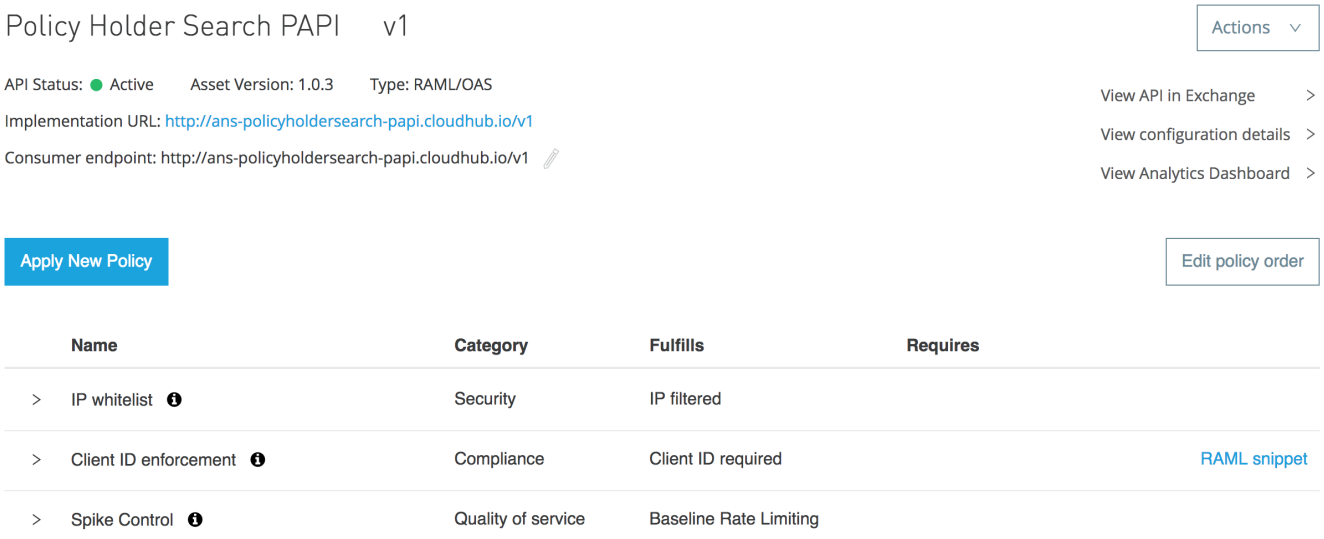

Figure 69. API policies defined for a particular API instance of the "Policy Holder Search PAPI".

5.3.24. Choosing appropriate API policies for Experience APIs

API policies on Experience APIs depend critically on the nature of the top-level API client for which an Experience API is intended. Acme Insurance's C4E therefore first defines API policies for concrete Experience APIs, generalizing to Acme Insurance-wide guidelines in a second step.

For the "Aggregator Quote Creation EAPI" consumed by the Aggregator you define the following:

- *IP whitelisting* to the IP address of the Aggregator, to complement TLS mutual authentication
- *XML threat protection*
- One SLA tier for the required 1000 requs/s, which makes this SLA explicit and allows monitoring and reporting on the SLA
- *SLA-based Rate Limiting* (not Spike Control)
  - `X-RateLimit-` HTTP response headers should not be exposed to API clients from this API policy because API clients are external
- No Spike Control API policy, to limit the resource consumption incurred by queuing requests
- These API policies should be applied in this order

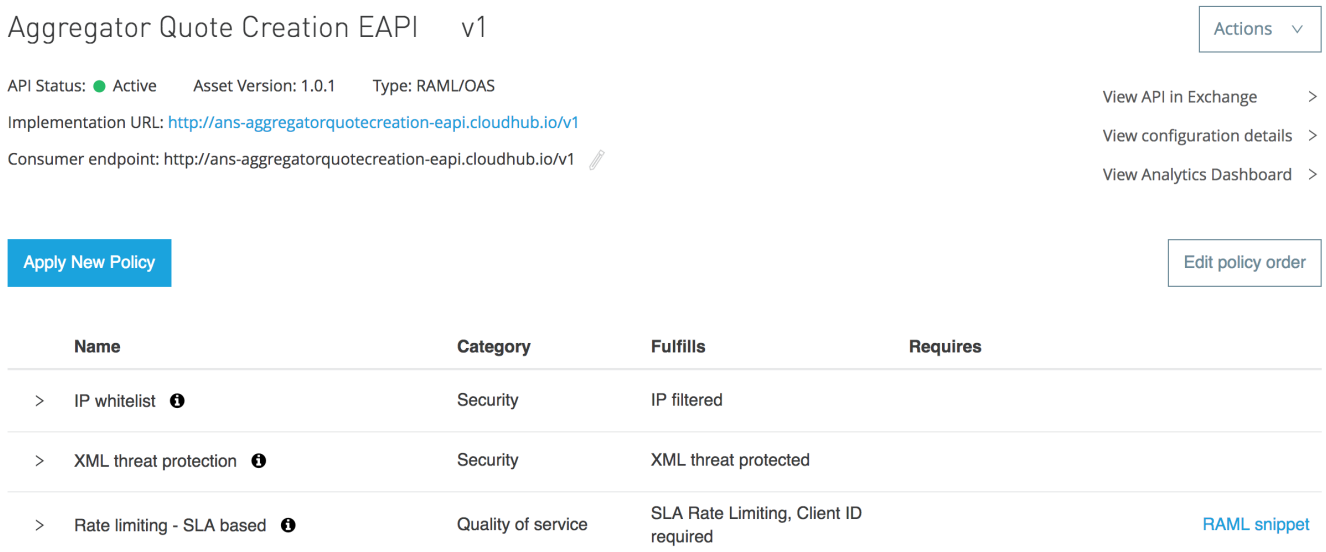

Figure 70. API policies defined for a particular API instance of the "Aggregator Quote Creation EAPI".

For the "Mobile Policy Holder Summary EAPI" and "Mobile Auto Claim Submission EAPI" consumed by Acme Insurance's own Customer Self-Service Mobile App you define:

- *JSON threat protection*
- *OAuth 2.0 access token enforcement* matching the Identity Provider configured for Client Management
- *Client ID enforcement*
- *Non-SLA-based Rate Limiting* (not Spike Control) to 100 requs/s for "Mobile Policy Holder Summary EAPI" and 10 requs/s for "Mobile Auto Claim Submission EAPI"
  - `X-RateLimit-` HTTP response headers should not be exposed to API clients from this API policy because API clients are external
- No Spike Control API policy, to limit the resource consumption incurred by queuing requests
- These API policies should be applied in this order

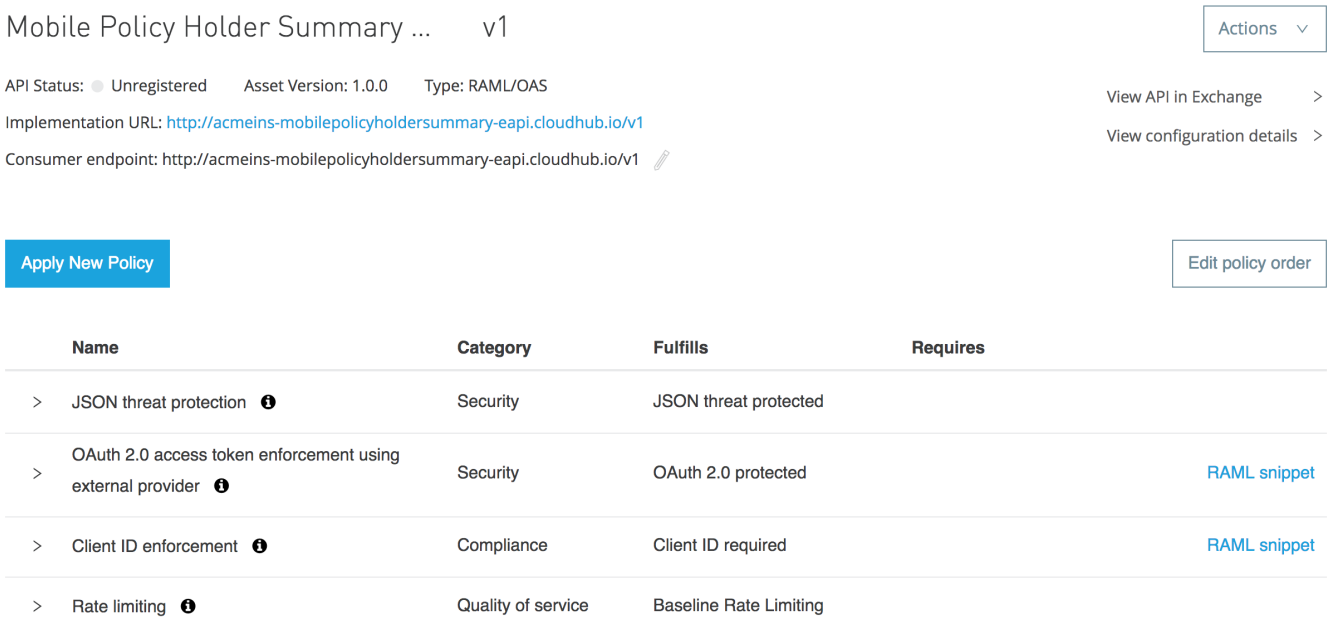

Figure 71. API policies defined for a particular API instance of the "Mobile Policy Holder Summary EAPI". The "OAuth 2.0 access token enforcement using external provider" API policy should be replaced with one that fits the Identity Provider configured for Client Management.

From this the following general guidelines for API policies on Experience APIs emerge:

- Whitelisting of IP address range of API clients, if possible
- Protection against oversized JSON/XML payloads
- Some form of enforcement of API client identity and, if applicable, user identity (via OAuth 2.0)
- *Rate Limiting instead of Spike Control* to reduce resource consumption in case of excessive number of API invocations, such as during DoS attacks. No exposure of `X-RateLimit-` HTTP response headers
- No Spike Control API policy
- These API policies should be applied in this order

5.3.25. Reviewing API policies for the APIs involved in the "Create quote for aggregators" feature

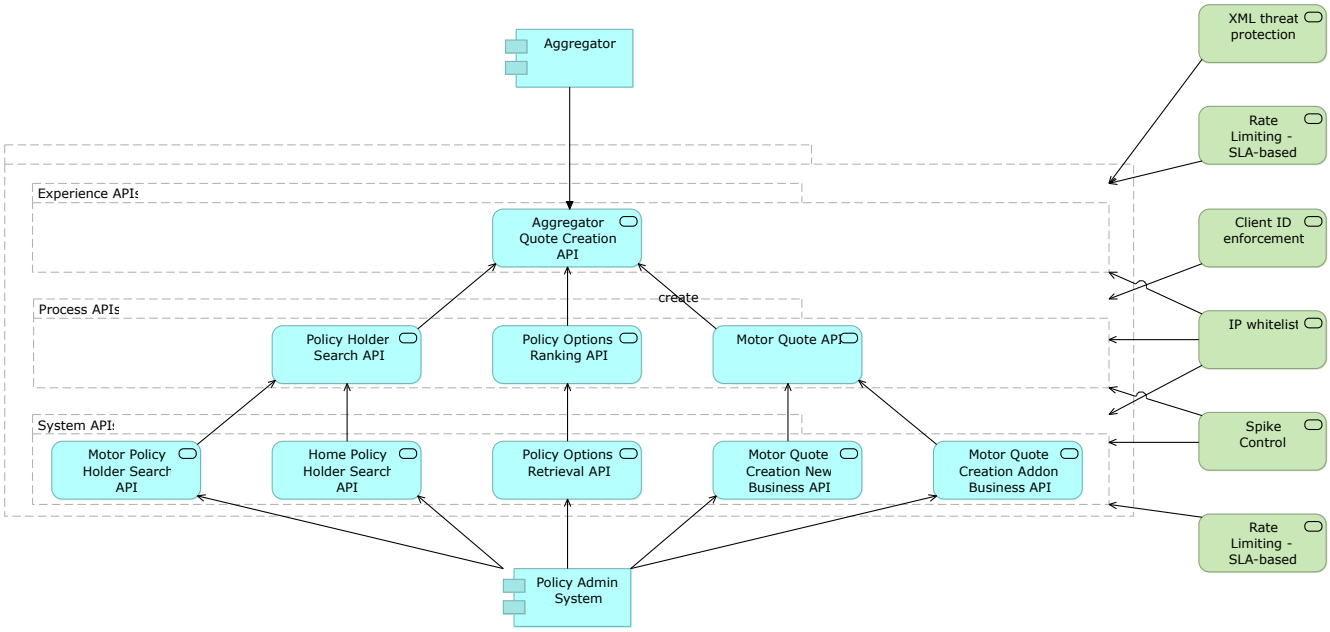

Figure 72. API policies applied to APIs in all tiers collaborating for the "Create quote for aggregators" feature (the suggested order of API policies is not visible in this diagram).

5.3.26. Reviewing API policies for the APIs involved in the "Retrieve policy holder summary" feature

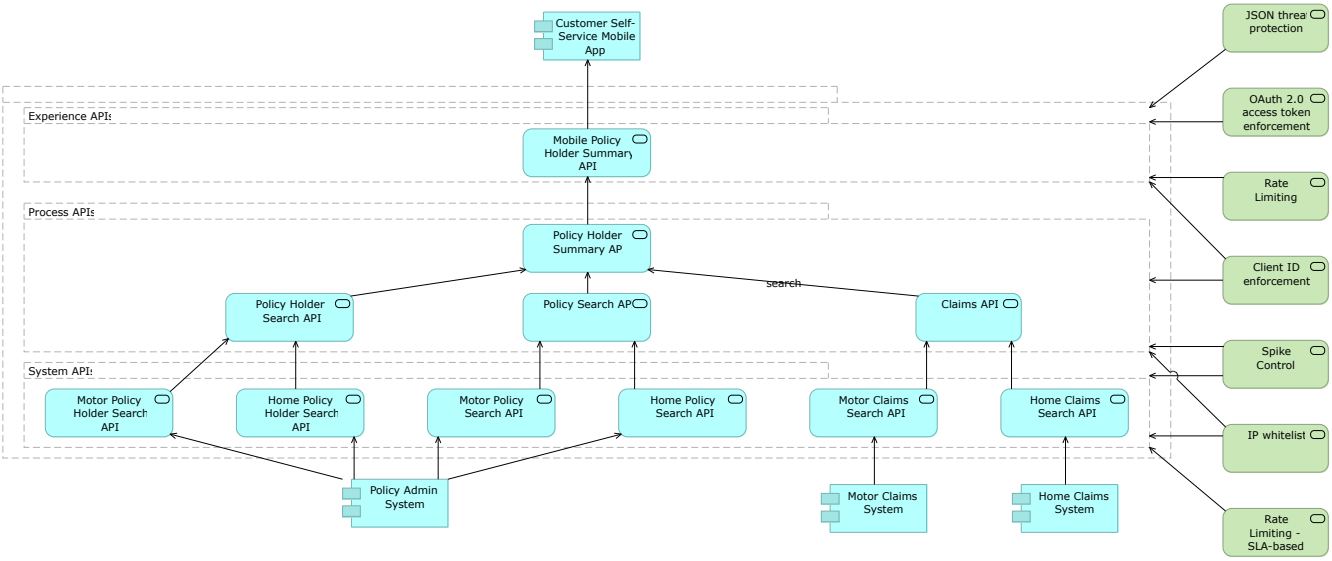

Figure 73. API policies applied to APIs in all tiers collaborating for the "Retrieve policy holder summary" feature (the suggested order of API policies is not visible in this diagram).

5.3.27. Reviewing API policies for the APIs involved in the "Submit auto claim" feature

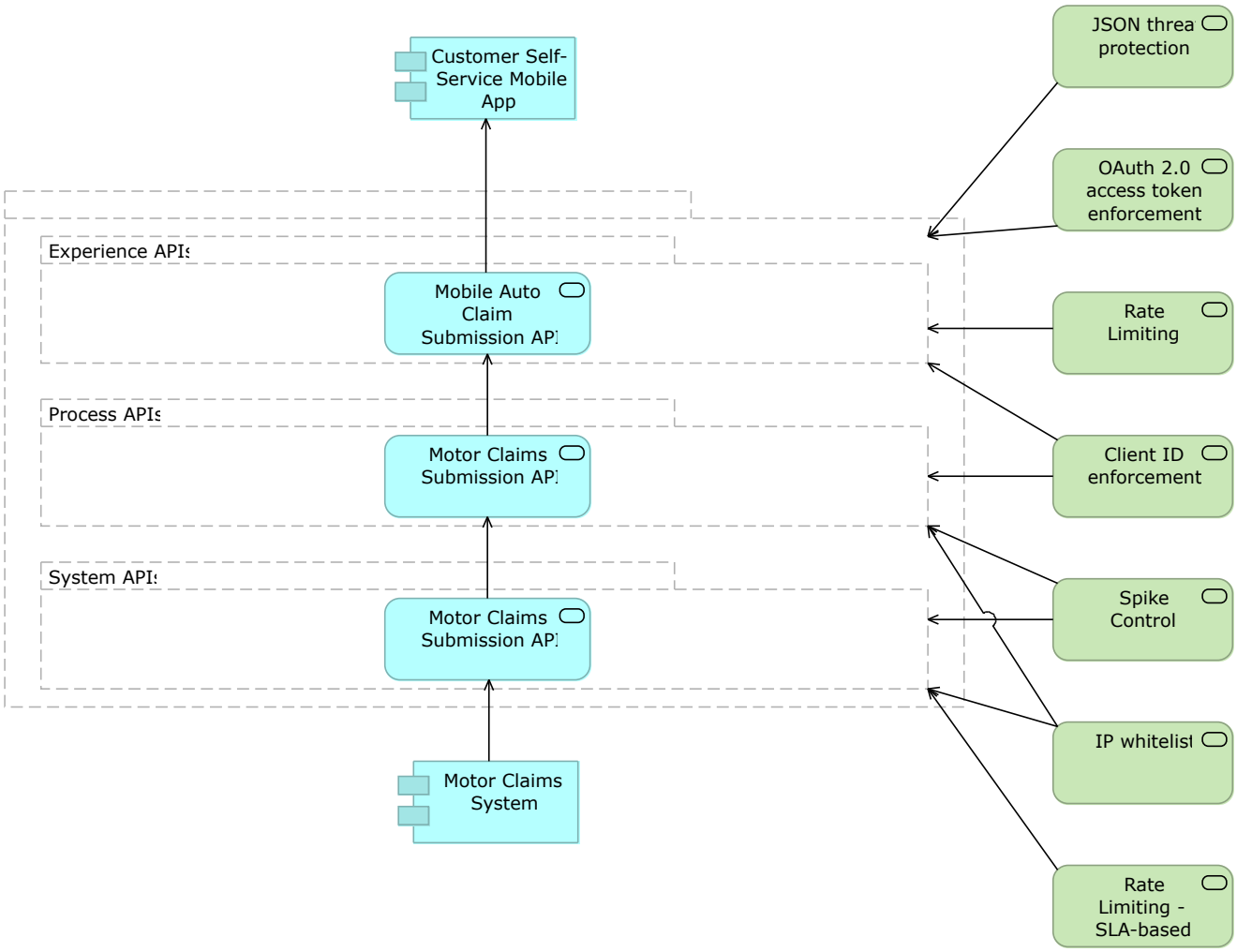

Figure 74. API policies applied to APIs in all tiers collaborating for the "Submit auto claim" feature (the suggested order of API policies is not visible in this diagram).

5.3.28. Reflecting the application of API policies in the RAML definition of an API

Many API policies change the HTTP request and/or HTTP response of API invocations subtly, for instance by

- requiring certain HTTP request headers, e.g., `Authorization`
- requiring certain query parameters, e.g., `client_id`
- adding HTTP response headers, e.g., `X-RateLimit-Limit`

or in other similar ways.

These changes to the contract between API client and API implementation must be reflected in the RAML definition of the API. In other words, *applying API policies often requires the RAML definition to be changed* to reflect the applied API policies.

In the case of security-related API policies, RAML has specific support through `securitySchemes`, e.g. of type `OAuth 2.0` or `Basic Authentication`. In other cases, RAML `traits` are a perfect mechanism for expressing the changes to the API specification introduced by the application of an API policy.

The *C4E owns the definition of reusable RAML fragments* for all commonly used API policies in Acme Insurance. These RAML fragments are *published to Anypoint Exchange* to encourage consumption and reuse.

5.3.29. Latency overhead of applying API policies

Applying an API policy to API invocations adds processing overhead, which results in increased latency (decreased response time) as seen by API clients. Depending on the type of API policy, that latency is up to 0.38 milliseconds per HTTP request, as shown in Figure 75 [Ref15].

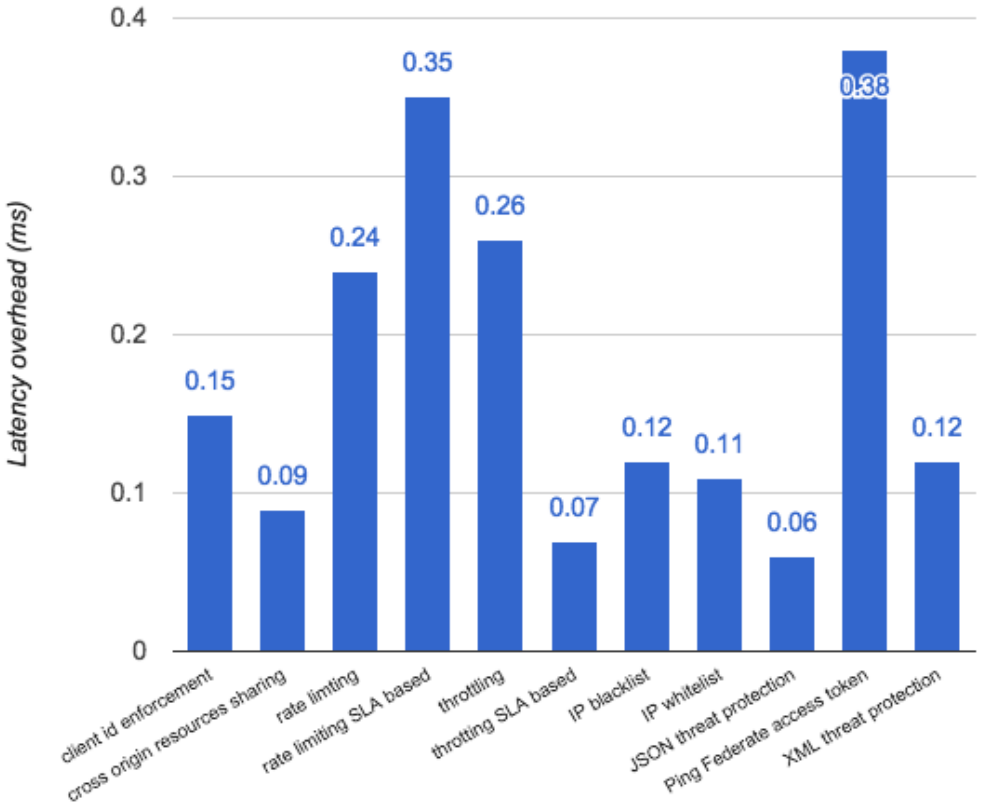

Figure 75. Increase in HTTP request-response latency through the application of various API policies, which are enforced embedded in the API implementation.

Notes:

- The timings quoted in Figure 75 for the PingFederate access token enforcement policy exclude the actual remote call to PingFederate
- Measurements for Figure 75 where performed using API Gateway 2.0, 1kB payloads, c3.xlarge (4-vCore, 7.5GB RAM, standard disks, 1Gbit network)

Summary

- The NFRs for the "Aggregator Integration" product and "Customer Self-Service App" product are a combination of constraints on throughput, response time, security and reliability
- Anypoint API Manager and API policies control APIs and API invocations and can impose NFRs on that level in various areas
- API policies can be enforced directly in an API implementation that is a Mule application or in a separately deployed API proxy
- Client ID-based API policies require API clients to be registered for access to an API

instance

  - Must pass client ID/secret with every API invocation, possibly implicitly via OAuth 2.0 access token

- The Acme Insurance C4E has defined guidelines for the API policies to apply to System APIs, Process APIs and Experience APIs
- C4E has created reusable RAML fragments for API policies and published them to Anypoint Exchange

Module 6. Designing Effective APIs

Objectives

- Appreciate the importance of contract-first API design and RAML fragments
- Opt for semantic API versioning and where to expose what elements of an API's version
- Choose between Enterprise Data Model and Bounded Context Data Models
- Consciously design System APIs to abstract from backend systems
- Apply HTTP-based asynchronous execution of API invocations and caching to meet NFRs
- Identify idempotent HTTP methods and HTTP-native support for optimistic concurrency

6.1. Understanding API design on Anypoint Platform

6.1.1. API design with Anypoint Platform and RAML

MuleSoft advocates API specification-driven, i.e., contract-first, API design:

1. Start by creating the API specification, ideally in the form of a RAML definition
2. Simulate interaction with the API based on the API specification
3. Gather feedback from potential future API consumers
4. Publish documentation and API-related assets, including the RAML definition of the API
5. Only then implement the API

This is the approach that has been followed in Module 4.

Anypoint Platform has support for API specifications in the form of

- RAML definitions
  - First-class support in all relevant components
- OpenAPI (OAS, Swagger) documents
  - Import/export in Anypoint Design Center
  - Import in Anypoint Exchange
- WSDL documents
  - Import in Anypoint Exchange

6.1.2. Identifying and publishing reusable RAML fragments

It is not only entire APIs that can be reused in an application network: if an API specification is defined in RAML then it is likely that it makes use of concepts that are reusable in other contexts. These reusable parts of a RAML definition are called RAML fragments. RAML fragments define aspects of the interface between an API client and an API implementation of the API in question - they are *partial interface definitions* that can be mixed-in to a complete RAML definition.

With the support of the Acme Insurance C4E you isolate these and other RAML fragments from the APIs identified so far:

- RAML `SecurityScheme` definitions for HTTP Basic Authentication and OAuth 2.0
- A RAML `Library` containing `resourceTypes` to support collections of items
- A RAML `Library` containing `resourceTypes` and `traits` to support asynchronous processing of API invocations with polling
- RAML `traits` for the API policies recommended at Acme Insurance, amongst them:
  - Client ID enforcement
  - SLA-based and non-SLA-based Rate Limiting, Throttling and Spike Control

These RAML fragments are represented in Anypoint Platform

- as Anypoint Design Center projects
- as Anypoint Exchange assets

This makes them discoverable and reusable within the Acme Insurance application network.

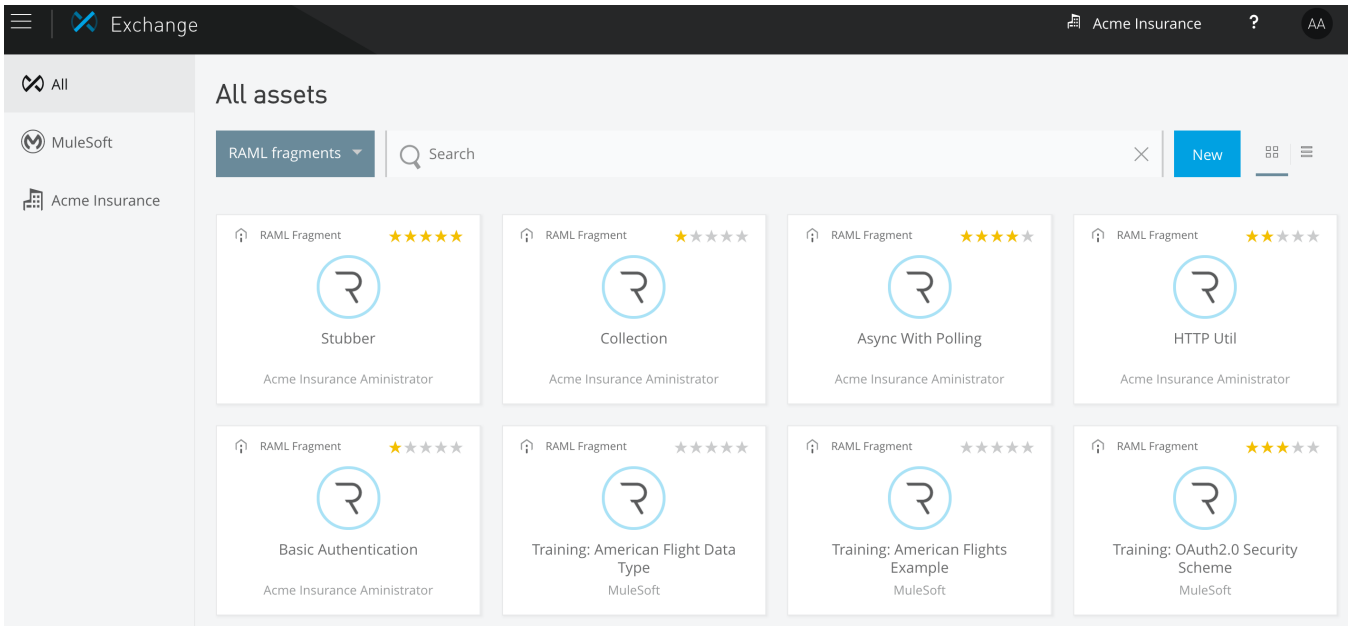

Figure 76. Some RAML fragments identified by the Acme Insurance C4E and by MuleSoft and made available in the public and Acme Insurance-private Anypoint Exchange, respectively.

6.2. Versioning APIs

6.2.1. Strategy for API versions on Anypoint Platform

Follow a *split strategy* on versioning APIs:

1. Try very hard to *make all changes to APIs backwards-compatible*
2. But assume that incompatible changes will be needed at some point and hence *version your APIs from the start*

An API versioning approach is *visible throughout the application network* and should therefore be *standardized by the C4E*.

On Anypoint Platform the version of an API is visible in these places:

- The URL of the API endpoint
- The RAML definition of the API: `version` and `baseUri`
- The Anypoint Exchange entry (asset) for the API: "API version", "Asset version", "API instances"
- The Anypoint API Manager entry for an API instance: "API version", "Asset version", "Implementation URL" and "Consumer endpoint"

6.2.2. Understanding semantic versioning of APIs on Anypoint Platform

Use semantic versioning with major, minor and patch version numbers for APIs: `major.minor.patch`:

- *Major versions* introduce backwards-*incompatible* changes in the structure of the API that require API clients to adapt
- *Minor versions* introduce backwards-*compatible* changes to the API that do not require API clients to change, unless the API client wants to take advantage of the newly introduced changes.
- *Patch versions* introduce small fully backwards-compatible fixes, such as documentation changes

If semantic versioning is followed, then version 1.2.3 of an API is a perfect stand-in for version 1.1.5, and so all API clients that have previously used version 1.1.5 can be made to use version 1.2.3 instead, without having to be made aware of the transition. For this reason, typically, *only the major version of an API is made visible to API clients*.

This means that *only the major version of the API should be visible in*

- The URL of the API endpoint
- The RAML definition of the API in both `version` and `baseUri`
- The Anypoint Exchange entry (asset) for the API in "API version"
- The Anypoint API Manager entry for an API instance in "API version", "Implementation URL" and "Consumer endpoint"

By contrast, the *full semantic version is visible in*

- The Anypoint Exchange entry (asset) for the API in "Asset version" and "API instances"
- The Anypoint API Manager entry for an API instance in "Asset version"

This is because

- the Anypoint Exchange entry of type "REST API" also stores the RAML definition itself - 4.3.6
- the Anypoint API Manager entry for an API instance may depend on that API specification for the definition of API policies that apply only to selected combinations of API resources and HTTP methods.

6.2.3. Versioning API endpoint URLs

In what part of a URL to surface the API version:

- Encode the version *just in the URL path*, e.g., http://ans-policyholdersummary-papi.cloudhub.io/v1
  - Requires a DNS lookup of `ans-policyholdersummary-papi.cloudhub.io` to resolve to an API implementation (or API proxy) that supports all version of the API, including future major versions
  - Alternatively, all API invocations must be routed on the server-side to the API implementation that supports the requested version
    - Supported through URL mapping rules in CloudHub Dedicated Load Balancers (7.1.10)
- Encode the version *just in the hostname*, e.g., http://ans-policyholdersummary-papi-v1.cloudhub.io/
  - Allows future (major) versions of the API to be backed by different API implementations (or API proxies) without having to do URL rewriting
- Encode the version *in the hostname and the URL path*, e.g., http://ans-policyholdersummary-papi-v1.cloudhub.io/v1
  - Redundant but allows the URL path on its own to identify the requested API version, without URL rewriting
  - Allows the same API implementation to expose endpoints for more than one major version

6.2.4. API versioning guidelines

The Acme Insurance C4E defines the following API versioning guidelines, which are fully supported by Anypoint Platform:

- Only expose major API versions as `v1`, `v2`, etc. in RAML definition, API endpoint URL and Anypoint API Manager entries in "API version", "Implementation URL" and "Consumer endpoint"
- In the API endpoint URL expose the major API version only in the URL path
  - E.g., http://ans-policyholdersummary-papi.cloudhub.io/v1
  - Requires future major versions to either be implemented in same API implementation or to route API invocations with URL mapping rules
  - To be augmented as and when the need arises with CloudHub Dedicated Load Balancer-supported URL mapping (7.1.10 and 7.1.12)

- Publish to Anypoint Exchange using the full semantic version in "Asset version", and refer to that from Anypoint API Manager "Asset version"

6.2.5. Deprecation of API instances in Anypoint API Manager

See 9.4.

6.3. Deciding granularity, separation and abstraction of APIs

6.3.1. Where we are along Acme Insurance's application network journey

At this point you have

- identified APIs to be developed and reused
- assigned functional responsibilities to them
- discussed the implications of fine-grained APIs (4.2.5)
- ensured alignment of API functionalities to business processes
- documented the current understanding of these APIs in RAML definitions and Anypoint Exchange entries
- made documentation and API-related assets discoverable in the Acme Insurance Anypoint Exchange and, potentially, its Public (Developer) Portal (Exchange Portal)
- designed a high-level Application Architecture that identifies API interactions
- sketched a first draft of a Technology Architecture, using components from Anypoint Platform, that is expected to support all NFRs

The above characterizes *Acme Insurance's nascent application network*.

You will now look in more detail at the APIs (this module) and their API implementations (next module) and address the most important design questions that arise in doing so. You will restrict this investigation to *architecturally significant design topics*, i.e., you will ignore design questions that have no implication for the effectiveness of the resulting application network.

6.3.2. Defining API data model

The APIs you have identified and started defining in RAML definitions exchange data representations of business concepts, mostly in JSON format. Examples are:

- The JSON representation of the Policy Holder of a Motor Policy returned by the "Motor Policy Holder Search SAPI"
- The XML representation of a Quote returned by the "Aggregator Quote Creation EAPI" to the Aggregator
- The JSON representation of a Motor Quote to be created for a given Policy Holder passed to the "Motor Quote PAPI"
- The JSON representation of any kind of Policy returned by the "Policy Search PAPI"

All data types that appear in an API (i.e., the *interface*) form the *API data model* of that API. The API data model should be specified in the RAML definition of the API. *API data models are clearly visible across the application network* because they form an important part of the interface contract for each API.

The API data model is conceptually clearly separate from similar models that may be used inside the API implementation, such as an object-oriented or functional domain model, and/or the persistent data model (database schema) used by the API implementation. *Only the API data model is visible to API clients* in particular and *to the application network* in general - all other forms of models are not. Consequently, *only the API data model is the subject of this discussion*.

6.3.3. Enterprise Data Model versus Bounded Context Data Models

The data types in the API data models of different APIs can be more or less coordinated:

- In an *Enterprise Data Model* - often called *Canonical Data Model*, but the discussion here uses the term Enterprise Data Model throughout - there is *exactly one canonical definition of each data type*, which is *reused in in all APIs* that require that data type, within all of Acme Insurance
  - E.g., one definition of `Policy` that is used in APIs related to Motor Claims, Home Claims, Motor Underwriting, Home Underwriting, etc.
- In a *Bounded Context Data Model* several *Bounded Contexts* are identified within Acme Insurance by their usage of common terminology and concepts. Each Bounded Context then has its own, distinct set of data type definitions - the Bounded Context Data Model. The Bounded Context Data Models of separate Bounded Contexts are formally unrelated, although they may share some names. *All APIs in a Bounded Context reuse the Bounded Context Data Model* of that Bounded Context
  - E.g., the Motor Claims Bounded Context has a distinct definition of `Policy` that is formally unrelated to the definition of `Policy` in the Home Underwriting Bounded Context

- In the extreme case, every API defines its own API data model. Put differently, every API is in a separate Bounded Context with its own Bounded Context Data Model.

Also see [Ref13].

6.3.4. Selecting between Enterprise Data Model and Bounded Context Data Models

- The coordination of API data models between APIs *adds overhead*, which can become significant if APIs are owned by separate groups. Coordination effort applies to initial data modelling, to all changes to the API data model and to the rollout of these changes to all APIs that share that API data model
- This is one reason why Enterprise Data Models, although a seemingly attractive idea, are often not successful
- If there is no successful Enterprise Data Model, it is *most pragmatic to use Bounded Context Data Models*
- If there is a successful Enterprise Data Model, then all Process APIs and System APIs should reuse that Enterprise Data Model as much as possible
- The *API data model of Experience APIs*, on the other hand, is determined by the needs of the top-level API clients (such as user-visible apps) and thus is *very unlikely to be served by an Enterprise Data Model*
  - E.g., Aggregator or Customer Self-Service Mobile App are unlikely to be served well be an Enterprise Data Model
- The Enterprise Data Model, even if it exists, typically does not define all data types needed by all APIs. Hence the decision for or against and Enterprise Data Model must be made on a per-data type basis
  - E.g., if the Enterprise Data Model defines `Policy` then that data type should be used in all Process APIs and System APIs that deal with policies in their APIs, while those same APIs need to define `Customer` by some other approach if `Customer` is not part of the Enterprise Data Model

6.3.5. Identifying Bounded Contexts and Bounded Context Data Models

Because Acme Insurance does not have a well-defined, successful Enterprise Data Model, it uses Bounded Context Data Models. To do so:

- *Identify* Bounded Contexts
- *Assign* each API to exactly one Bounded Context, based on the defining (dominant) data types for that API

  - E.g., the definining data type of the "Motor Policy Holder Search SAPI" is `Motor Policy Holder`
  - If an API has no clear set of definining data types, or if those data types are used in significantly different variations in different operations/resources of that API, then the API is likely too coarse-grained and should be broken up
- *Define a Bounded Context Data Model* for each Bounded Context based pragmatically on the needs of the APIs in that Bounded Context
- *Reuse* the Bounded Context Data Model in the APIs of that Bounded Context

To identify Bounded Contexts:

- Start with the *organizational structure*, aiming for structural units where important *business concepts* are used in a coherent and homogenous way
  - E.g., Motor Claims, Home Claims, Motor Underwriting, Home Underwriting, Customer Relationship Management, …
- If in doubt *prefer smaller Bounded Contexts*
- If still in doubt put *each API in its own Bounded Context*
  - I.e., do not coordinate API data models between APIs

A Bounded Context Data Model should be published as RAML fragments (RAML `types`, possibly in a RAML `Library`) in Anypoint Design Center and Anypoint Exchange, so that it can be easily re-used in all APIs in a Bounded Context.

The Acme Insurance C4E owns this activity and the harvesting of API data types from existing APIs.

6.3.6. Exercise 8: Identify Bounded Contexts in Acme Insurance's application network

You have already identified a large number of APIs (Figure 57):

1. Delineate the boundaries of meaningful Bounded Contexts so that
   a. every API belongs to exactly one Bounded Context
   b. there is more than one Bounded Context overall (i.e., no Enterprise Data Model)
2. Identify the defining API data types for each Bounded Context and verify that they apply to all APIs in that Bounded Context
3. Discuss the implications of the identified Bounded Context Data Models for
   a. coordination between the teams responsible for the APIs in each Bounded Context

b. data transformation when APIs in different Bounded Contexts invoke each other

Solution

For instance:

- Motor and home policy administration, although both implemented by the same Mainframe-based Policy Admin System, are distinct Bounded Contexts, not only because they serve different teams within Acme Insurance but also because they are implemented by different data schemata and database tables in the Mainframe
- Similarly, motor and home claims are supported by significantly different concepts and teams and fall into different Bounded Contexts
- *Policy Holder* search, on the other hand, although implemented differently in the Policy Admin System for motor and home policies, deals with a concept (Policy Holder) that should be fairly homogenous between motor and home underwriting and can be assigned to the same Bounded Context
- Similarly, *Policy Options* retrieval and ranking are two procedural operations on the same concept and can be grouped into one Bounded Context
- *Motor Quote Creation* is separated in the Policy Admin System into new and addon business, but this is not reflected in how the business thinks of Motor Quotes, which groups deal with them and how they are represented as API data types. The involved APIs therefore belong to the same Bounded Context
- The distinction between "Motor Claims Submission PAPI" and "Motor Claims Submission SAPI" seems to be a purely technical one, that has no business meaning and no organizational reflection, hence they belong to the same Bounded Context

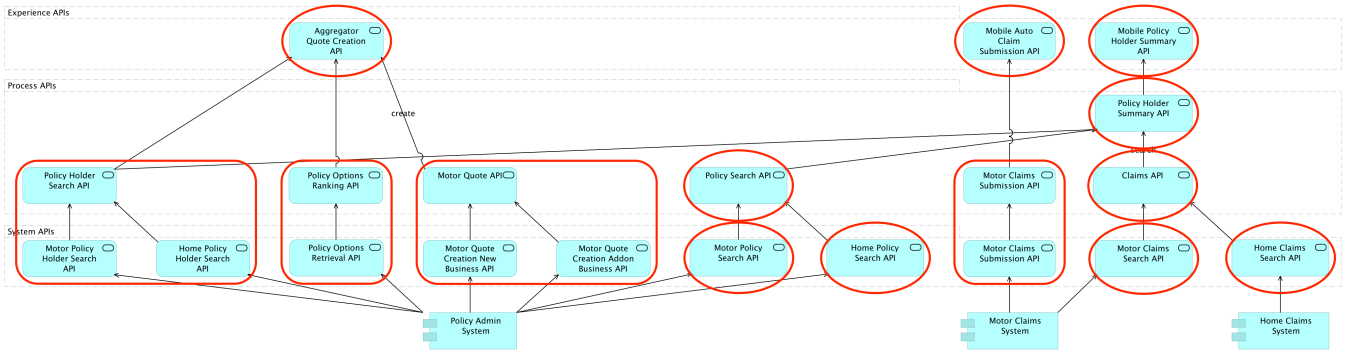

Figure 77. Bounded Contexts for the APIs in Acme Insurance's application network, where each Bounded Context defines its own API data model of semantically homogenous and well-defined API data types.

6.3.7. Mapping between Bounded Context Data Models

Distinct Bounded Context Data Models necessitate *data transformation* whenever APIs from different Bounded Contexts need to cooperate. More precisely:

- An API implementation (the *caller*) belonging to an API in a given Bounded Context
- that must invoke an API (the *called*) in a different Bounded Context
- must transform the Bounded Context Data Model of the called API
- to its own Bounded Context Data Model

This approach to mapping between Bounded Context Data Models is called *anticorruption layer* [Ref13]. There are other variants of mapping between Bounded Context Data Models, depending on where transformation occurs, but this falls into the domain of detailed design and is not visible on the Enterprise Architecture level and therefore out-of-scope for this discussion.

API implementations implemented as Mule applications can draw on the advanced data mapping capabilities of Anypoint Studio and the Mule runtime for implementing data transformations of this kind.

6.3.8. Understanding relationships when integrating between Bounded Contexts

More generally, there are important power relationships at play whenever an API implementation from one Bounded Context invokes an API from another Bounded Context. For instance, using DDD terminology [Ref13]:

- *Partnership*: Coordination of caller and called in terms of features and timeline
- *Customer/Supplier*: Caller requests features from the called, who may have to coordinate many callers' feature requests
- *Conformist*: Caller must work with whatever called provides

6.3.9. Exercise 9: Identify Bounded Context power relationships

Based on the previous identification of Bounded Contexts within Acme Insurance’s application network (Figure 77), or drawing on your own experience,

- Identify at least one example for each type of relationship, i.e.,
  - Partnership

- Customer/Supplier
- Conformist

Solution

- Partnership: *Aggregator Quote Creation* Bounded Context and *Motor Quote Creation* Bounded Context because located in same LoB and cooperate on "Customer Self-Service App" product
- Customer/Supplier: *Claims* Bounded Context towards *Home Claims Search* Bounded Context, because the Home Claims System is externally developed and outsourced and the "Home Claims Search SAPI" may be a widely used API
- Conformist: *Aggregator Quote Creation* Bounded Context towards Aggregator, because Aggregator determines interface
- Conformist: all invocations of Google APIs, …

6.3.10. Abstracting from backend systems with System APIs

System APIs mediate between backend systems and Process APIs by unlocking data in these backend systems:

- Should there be one System API per backend system or many?
- How much of the intricacies of the backend system should be exposed in the System APIs in front of that backend system? In other words, how much to abstract from the backend system data model in the API data model of the System APIs in front of that backend system?

General guidance:

- System APIs, like all APIs, should be defined at a granularity that makes business sense and adheres to the Single Responsibility Principle (4.2.5)
  - It is therefore very likely that any non-trivial backend system must be fronted by more than one System API (see for instance Figure 57)
- If an Enterprise Data Model is in use then
  - the API data model of System APIs should make use of data types from that Enterprise Data Model
  - the corresponding API implementation should translate between these data types from the Enterprise Data Model and the native data model of the backend system
- If no Enterprise Data Model is in use then
  - each System API should be assigned to a Bounded Context, the API data model of

System APIs should make use of data types from the corresponding Bounded Context Data Model
    - the corresponding API implementation should translate between these data types from the Bounded Context Data Model and the native data model of the backend system
    - In this scenario, the data types in the Bounded Context Data Model are defined purely in terms of their business characteristics and are typically not related to the native data model of the backend system. In other words, the *translation effort may be significant*
- If no Enterprise Data Model is in use, and the definition of a clean Bounded Context Data Model is considered too much effort, then
    - the API data model of System APIs should make use of data types that approximately mirror those from the backend system
    - same semantics and naming as backend system
    - but for only those data types that fit the functionality of the System API in question - backend system often are Big Balls of Mud that cover many distinct Bounded Contexts
    - lightly sanitized
        - e.g., using idiomatic JSON data types and naming, correcting misspellings, …
    - expose all fields needed for the given System API's functionality, but not significantly more
    - making good use of REST conventions

The latter approach, i.e., exposing in System APIs an API data model that basically mirrors that of the backend system, does not provide satisfactory isolation from backend systems through the System API tier on its own. In particular, it will typically not be possible to "swap out" a backend system without significantly changing all System APIs in front of that backend system - and therefore the API implementations of all Process APIs that depend on those System APIs! This is so because it is not desirable to prolong the life of a previous backend system's data model in the form of the API data model of System APIs that now front a new backend system. The API data models of System APIs following this approach must therefore change when the backend system is replaced. On the other hand:

- It is a very *pragmatic* approach that adds comparatively little overhead over accessing the backend system directly
- *Isolates* API clients from intricacies of the backend system outside the data model (protocol, authentication, connection pooling, network address, …)
- Allows the usual *API policies* to be applied to System APIs
- Makes the *API data model for interacting with the backend system* explicit and visible, by exposing it in the RAML definitions of the System APIs
- Further isolation from the backend system data model does occur in the API

implementations of the Process API tier

6.3.11. Designing Acme Insurance's System APIs

- Many System APIs unlock data and functionality of the Policy Admin System
  - Defined by Bounded Contexts (motor, home) as well as by use cases (policy holder search, option retrieval, etc.)
- The same principle applies to System APIs infront of the Motor Claims System and Home Claims System
- All System APIs are JSON REST APIs and therefore define JSON data types in their RAML definitions/libraries
  - RAML actually allows the definition of data types that are representable in both XML and JSON, so the data types in the RAML definitions of the System APIs need/should not be JSON-specific
- No Enterprise Data Model is in use and Acme Insurance goes through the effort of defining a few clean, business-oriented, pragmatic Bounded Context Data Models
- For instance, the representation of `Motor Policy` returned by the "Motor Policy Search SAPI" is defined in terms of Acme Insurance's understanding of `Motor Policy` in the *Motor Policy Search* Bounded Context and is therefore not directly related to the native data definition in the Policy Admin System. In particular, it
  - uses property names that have business meaning rather than mirroring field names in the Policy Admin System
  - uses JSON data types and JSON-compatible naming
  - omits fields not relevant for Motor Underwriting, which may well be contained in the Mainframe-based database tables

See Figure 57.

6.4. Selecting API invocation patterns that address NFRs

6.4.1. Asynchronously executing API invocations with polling

The "Submit auto claim" feature requires that the typically involved and lengthy execution of submitting a claim is executed asynchronously, so that the Customer Self-Service Mobile App is not blocked while claim submission occurs.

HTTP has native support for asynchronous processing of the work triggered by HTTP requests. This feature is therefore immediately available to REST APIs (but not non-REST APIs like SOAP

APIs).

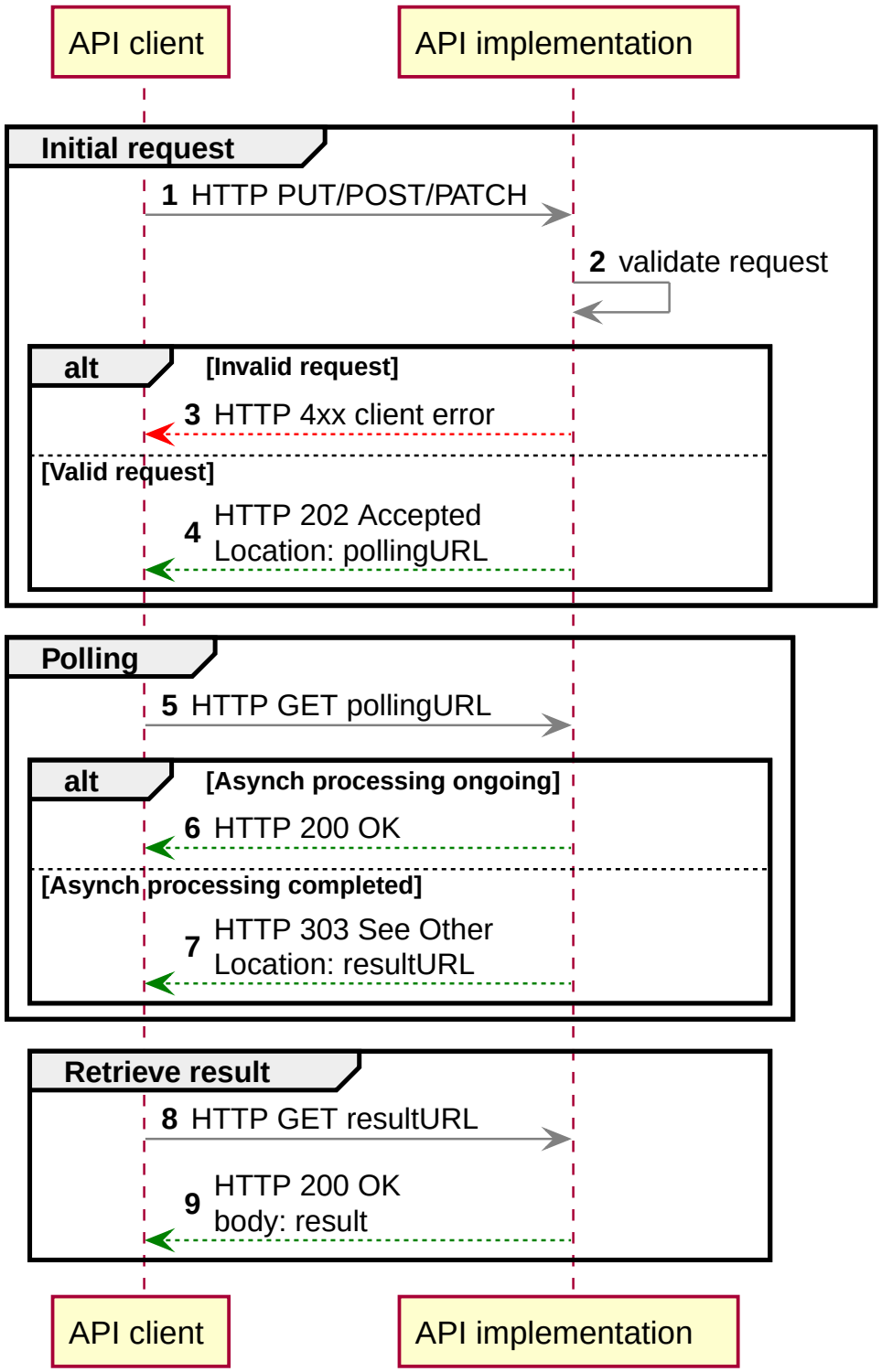

Figure 78. Asynchronous processing triggered by an API invocation from an API client to an API implementation, with polling by the API client to retrieve the final result from the API implementation.

1

The API client sends a HTTP request that will trigger asynchronous processing

2

The API implementation accepts the request and validates it

3

If the HTTP request is not valid then, as usual, a HTTP response with a *HTTP 4xx client error response code* is returned

4

If HTTP request validation succeeds then the API implementation triggers asynchronous processing (in the background) and returns a HTTP response with the *HTTP 202 Accepted response status code* to the API client with a `Location` response header containing the URL of a resource to poll for progress

5

The API client then regularly sends HTTP GET requests to the polling resource

6

The API implementation returns *HTTP 200 OK* if asynchronous process is still ongoing

7

but returns a HTTP response with *HTTP 303 See Other redirect status response code* and a `Location` response header with the URL of a resource that gives access to the final result of the asynchronous processing

8

The API client then sends a HTTP GET request to that last URL to retrieve the final result of the now-completed asynchronous processing

The fact that HTTP-based asynchronous execution of API invocations is used should be documented in the RAML definition of the respective APIs. In fact, the Acme Insurance C4E has already published a resuable RAML library for this purpose: Figure 76.

6.4.2. Asynchronously executing API invocations with callbacks

An alternative to the HTTP-supported polling-based async execution of API invocations is the use of HTTP callbacks, aka *webhooks*. This *requires the original API client to be reachable by HTTP requests from the API implementation*, and is therefore typically only available within intranets.

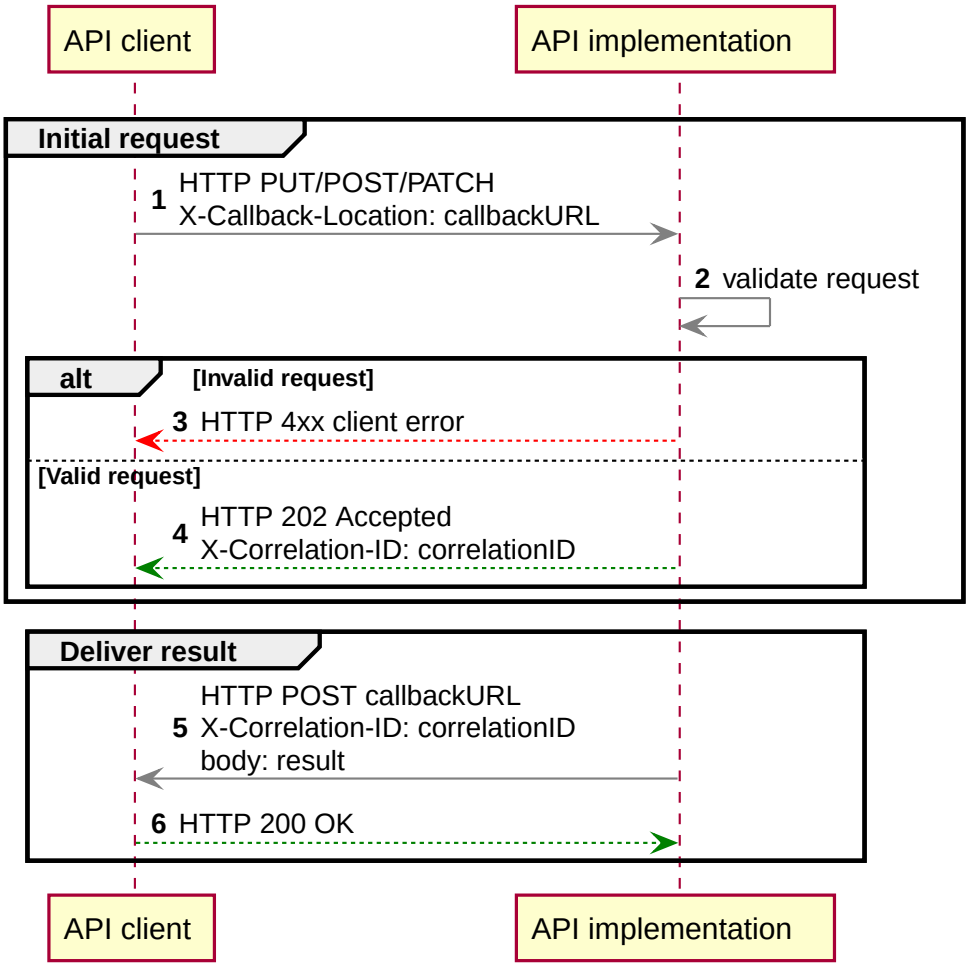

Figure 79. Asynchronous processing triggered by an API invocation from an API client to an API implementation, with a callback from the API implementation to the API client to deliver the final result. The callback URL is sent as a custom HTTP request header and the correlation ID is also exchanged in a custom HTTP header.

1

The API client sends a HTTP request that will trigger asynchronous processing

- With that HTTP request it sends the URL of a resource of the API client that will receive callbacks
- The callback URL can be sent as a URL query parameter or custom HTTP request header

2

The API implementation accepts the request and validates it

3

If the HTTP request is not valid then, as usual, a HTTP response with a *HTTP 4xx client*

error response code is returned

4

If HTTP request validation succeeds then the API implementation triggers asynchronous processing (in the background) and returns a HTTP response with the *HTTP 202 Accepted response status code* to the API client

- The HTTP response must also contain the *correlation ID* for the request, e.g., in a custom response header

5

Once asynchronous processing is completed, the API implementation sends a HTTP POST request to the callback URL containing the final result of the now-completed asynchronous processing, sending the correlation ID (for instance) as a request header

6

The API client acknowledges receipt of the callback by returning a HTTP 200 OK from the callback

The fact that HTTP-based asynchronous execution of API invocations is used should be documented in the RAML definition of the respective APIs. The Acme Insurance C4E should publish a resuable RAML library for this purpose.

6.4.3. Asynchronous API invocations for the "Submit auto claim" feature

For the "Submit auto claim" feature, asynchronous API invocations are essential:

- Polling is used for the "Mobile Auto Claim Submission EAPI" because callbacks to the Customer Self-Service Mobile App are impossible due to obvious networking restrictions.
- The "Motor Claims Submission PAPI", which is invoked by the "Mobile Auto Claim Submission EAPI", can implement asynchronous processing via a callback, which is more efficient than polling.

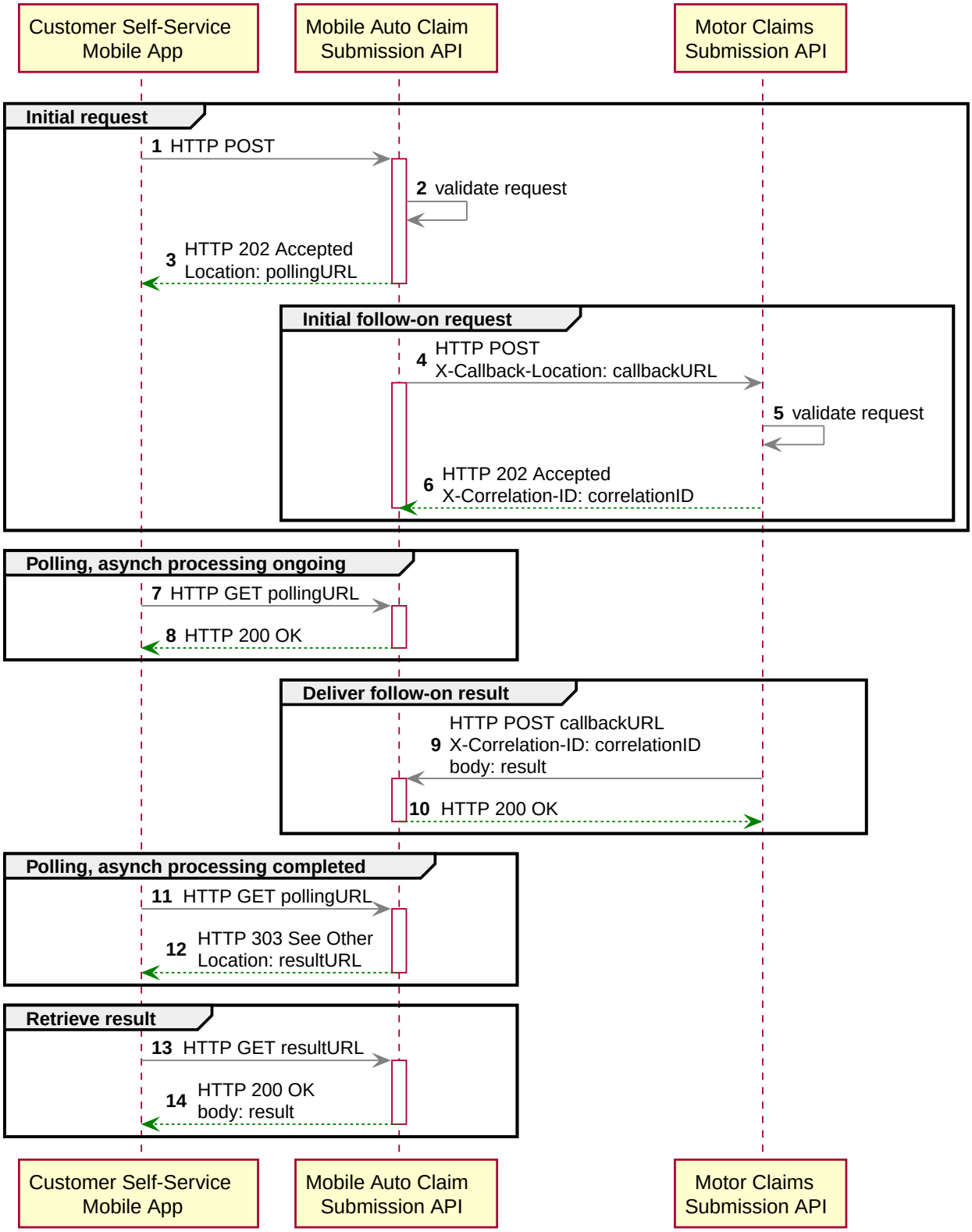

Figure 80. Asynchronous execution of the "Mobile Auto Claim Submission EAPI", with polling from the Customer Self-Service Mobile App, feeding into asynchronous execution of the "Motor Claims Submission PAPI", with callback to the "Mobile Auto Claim Submission EAPI".

6.4.4. State handling for asynchronous execution of API invocations

Whichever approach to asynchronously executing API invocations is used, it requires the API implementation to maintain the state of each async invocation. It therefore *makes API implementations stateful*.

Options for keeping state in API implementations implemented as Mule applications and executing in a Mule runtime:

- an Object Store, a feature of the Mule runtime
- an external database

6.4.5. Exercise 10: Meaning and implications of statefulness

Asynchronous execution of claim submissions is the first example in our Enterprise Architecture of stateful API implementations:

1. Discuss the *exact meaning of "stateful API implementation"*
2. Is statefulness of the API implementations of the "Mobile Auto Claim Submission EAPI" and the "Motor Claims Submission PAPI" avoidable?
3. List scenarios where statefulness makes a difference

Solution

- Statefulness of an API implementation is typically interpreted to mean that the nodes (machines, CloudHub workers) on which the API implementation executes store data either *in RAM or locally on disk*
- If data is stored on *other nodes* than those of the API implementation, such as an external database or message broker, then the API implementation is not typically considered stateful
- If an API implementation is deployed to several nodes, as is typically the case, then a distinction must be made between state that is *local to a node* or state that is *replicated to all nodes*: the former means that all nodes are not equal, whereas the latter means that the API implementation is still a homogenous, replicated service
- The *state handling* for the API implementations to the "Mobile Auto Claim Submission EAPI" and the "Motor Claims Submission PAPI" is unavoidable - but where state is stored, and whether these API implementations become stateful as a result, is an *architectural choice*
- Both the CloudHub Object Store (7.1.14) as well as an external database keep state *outside the nodes (CloudHub workers) to which the API implementation is deployed* and therefore

do not make the API implementation stateful in the above sense

- If state were kept in local memory or disk on a node, or in-memory replicated amongst nodes (these are also options with Object Stores, where the latter is not available in CloudHub) then the API implementation were stateful
- Statefulness and whether state is local to a node or replicated across nodes makes a difference for:
  - *Load-balancing*: if round-robin as in CloudHub then all nodes must be homogenous
  - *Updates/restarts*: does state survive?
  - *Data purging*: are separate data cleanup processes needed?
  - *Scalability*: does increasing the number of nodes for an API implementation incur increased communication between nodes to keep replicated state in sync, thereby limiting scalability?
  - *Latency*: if state is updated or read on a node, is additional latency incurred due to replicating or retrieving state to/from remote nodes?
  - *Operations*: keeping state in external services like a database or message broker incurse management and operations overhead

6.4.6. Caching and safe HTTP methods

APIs use HTTP-based protocols: cached HTTP responses from previous HTTP requests may potentially be returned if the same HTTP request is seen again.

Safe HTTP methods are ones that do not alter the state of the underlying resource. That is, the *HTTP responses to requests using safe HTTP methods may be cached*.

The HTTP standard requires the following HTTP methods on any resource to be safe:

- GET
- HEAD
- OPTIONS

Safety must be honored by REST APIs (but not by non-REST APIs like SOAP APIs): It is the *responsibility of every API implementation* to implement GET, HEAD or OPTIONS methods such that they never change the state of a resource.

6.4.7. Caching API invocations using HTTP facilities

Only API functionality exposed via safe HTTP methods may return a cached HTTP response.

HTTP natively defines rigorous caching semantics using these (and more) HTTP headers. This

feature is therefore immediately available to REST APIs (but not non-REST APIs like SOAP APIs):

- `Cache-Control`
- `Last-Modified`
- `ETag`
- `If-Match`, `If-None-Match`, `If-Modified-Since`
- `Age`

Caching requires

- storage management
- the manipulation of HTTP request and response headers in accordance with the HTTP specification

Since HTTP-based caching is communicated by HTTP headers, it should be *documented in the RAML definition* of each API. In particular, the C4E should define or reuse a RAML fragment to be applied to cacheable API resources.

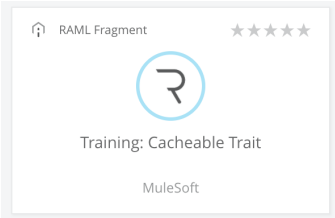

Figure 81. A cacheable RAML fragment published to the public Anypoint Exchange.

Caching may occur

- in the API client
- in the API implementation
- in any network component, such as a caching proxy, between the API client and API implementation

Mule applications acting as API clients or API implementations may make use of a `caching` scope, which may be used, for instance, in a custom API policy, but Anypoint Platform as such contains no ready-made facilities for caching. Custom API policies for various caching scenarios are available in the public Anypoint Exchange.

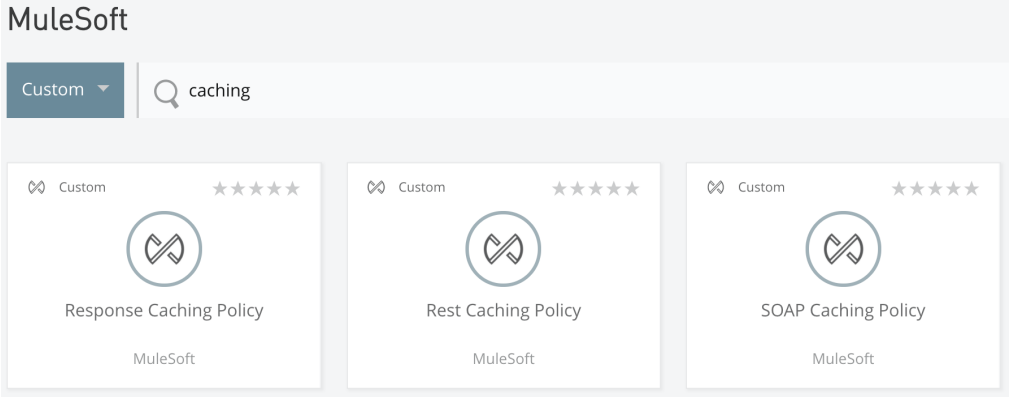

Figure 82. Custom API policies performing caching, published in the public Anypoint Exchange.

6.4.8. Exercise 11: Apply caching to API invocations

Investigate Acme Insurance's application network (Figure 57) and

1. Identify API invocations that are or are not cacheable (safe)
2. Specify whether caching is likely to benefit performance of the cacheable API invocation
3. Decide where caching is best performed: in the API client, the API implementation or in between
4. What implications does the site of caching have for storage management (for the cache) and deployment architecture?
5. Identify cache parameters, i.e., criteria that determine whether a cached HTTP response may be returned

Solution

- The *benefits of caching*, as a first approximation, correlate positively with the *throughput* of the cached API invocations, the *cache hit rate* achievable for these API invocations, and the *processing time* for an uncached API invocation
- All API invocations related to the "Create quote for aggregators" feature would therefore particularly benefit from caching - but not all of them are safe (Figure 83)
- *Client-side caching* in RAM (or local disk) avoids the network roundtrip to the caching instance. But if there are many API clients performing the same API invocation, including many instances of the same API invocation, then each API client keeps its own separate cache, which reduces the hit rate significantly
- Cache storage is best kept *in RAM* for performance reasons:
  - Increased memory requirements for the nodes keeping caches. Hence the deployment coniguration must be aware of the caching parameters

- Non-replicated cache state results in a reduction of the cache hit rate with the increase of the number of nodes keeping cache state
- Replicated cache state (e.g., via CloudHub Object Store) does not have this limitation but has communication overhead

- Cache lookup must be by all *relevant inputs to the API invocation*, not the entire HTTP request. This should clearly be deducible from the *API specification*
  - E.g., API client's IP address may be relevant or a geo-sensitive API invocation but is otherwise not
- The choice of cache parameters such as TTL and size requires business knowledge, i.e., insight into how the API invocation is used and what parameters are acceptable for that usage

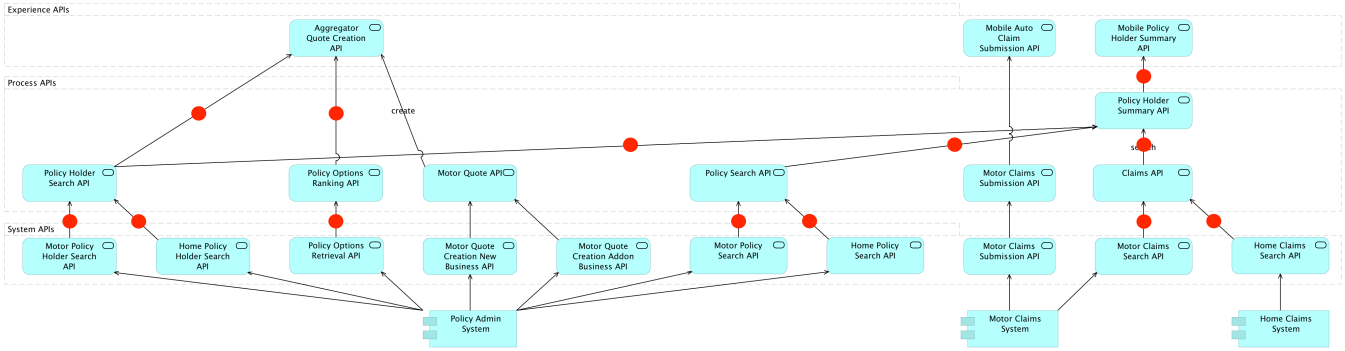

Figure 83. Safe (cacheable) API invocations in Acme Insurance's application network.

6.4.9. Retrying and idempotent HTTP methods

APIs use HTTP-based protocols: HTTP requests and responses may be lost due to a variety of issues:

- A lost HTTP request should be retried
- A lost HTTP response may cause duplicate processing if the HTTP request is retried

Idempotent HTTP methods are ones where a HTTP request may be re-sent without causing duplicate processing. That is, *idempotent HTTP methods may be retried*.

The HTTP standard requires the following HTTP methods on any resource to be idempotent.

- GET
- HEAD
- OPTIONS
- PUT

- DELETE

Of these methods, only PUT and DELETE may change the state of a resource (i.e., are not safe). On the other hand, POST and PATCH are not idempotent (and not safe): if the HTTP response to a POST or PATCH request does not reach the API client then the API client can, in general, not re-send the HTTP request without causing duplicate processing.

Idempotency must be honored by REST APIs (but not by non-REST APIs like SOAP APIs): It is the *responsibility of every API implementation* to implement PUT and DELETE methods such that they do not alter the underlying resource if the PUT or DELETE was already successfully processed.

Mule applications acting as API implementations may make use of an `idempotent-message-filter` to aid the implementation of this requirement.

How does an API implementation decide if a HTTP request was already received and must therefore be disregarded (if it is a PUT or DELETE)?

- One approach is to treat all requests with identical "content" (HTTP request body and relevant request headers) as identical. This is similar to the default implementation of the `idempotent-message-filter` (which makes that decision based on a hash of the Mule message body). However, this approach prohibits identical PUT or DELETE requests from being processed at all, which may be problematic.
- A common refinement of this approach is therefore to require the API client to generate a *unique request ID* and add that to the HTTP request. If the API client re-sends a request it must use the same request ID as in the original request. It follows that the content of requests of this kind can only be identical if the request ID is also identical, so that identical requests can be determined by the API implementation simply by comparing request IDs. The `idempotent-message-filter` can easily be configured to do that.

6.4.10. HTTP-based optimistic concurrency control

In some cases the updates to resources must be serialized, so that concurrent updates to the resource do not result in previous changes being overwritten without warning.

HTTP supports *optimistic concurrency control* of this kind natively with the combination of the following facilities. This feature is therefore immediately available to REST APIs (but not non-REST APIs like SOAP APIs):

- `ETag` HTTP response header to send a resource version ID in the HTTP response from the API implementation to the API client
- `If-Match` HTTP request header to send the resource version ID on which an update is

based in an HTTP PUT/POST/PATCH request from API client to API implementation

- *HTTP 412 Precondition Failed client error response code* to inform the API client that the resource version ID it sent was stale and hence the requested change not performed
- *HTTP 428 Precondition Required client error response code* to inform the API client that the resource in question is protected against concurrent modification and hence requires `If-Match` HTTP request headers, which were however missing from the HTTP request

Because usage of these headers changes the technology interface of the API, this should be fully *documented in the RAML definition* of each API that uses optimistic concurrency. A RAML fragment in the form of a `trait` should be used to capture this element of the contract between API client and API. This RAML fragment is reusable and should be made available as a Anypoint Design Center project and published to Anypoint Exchange.

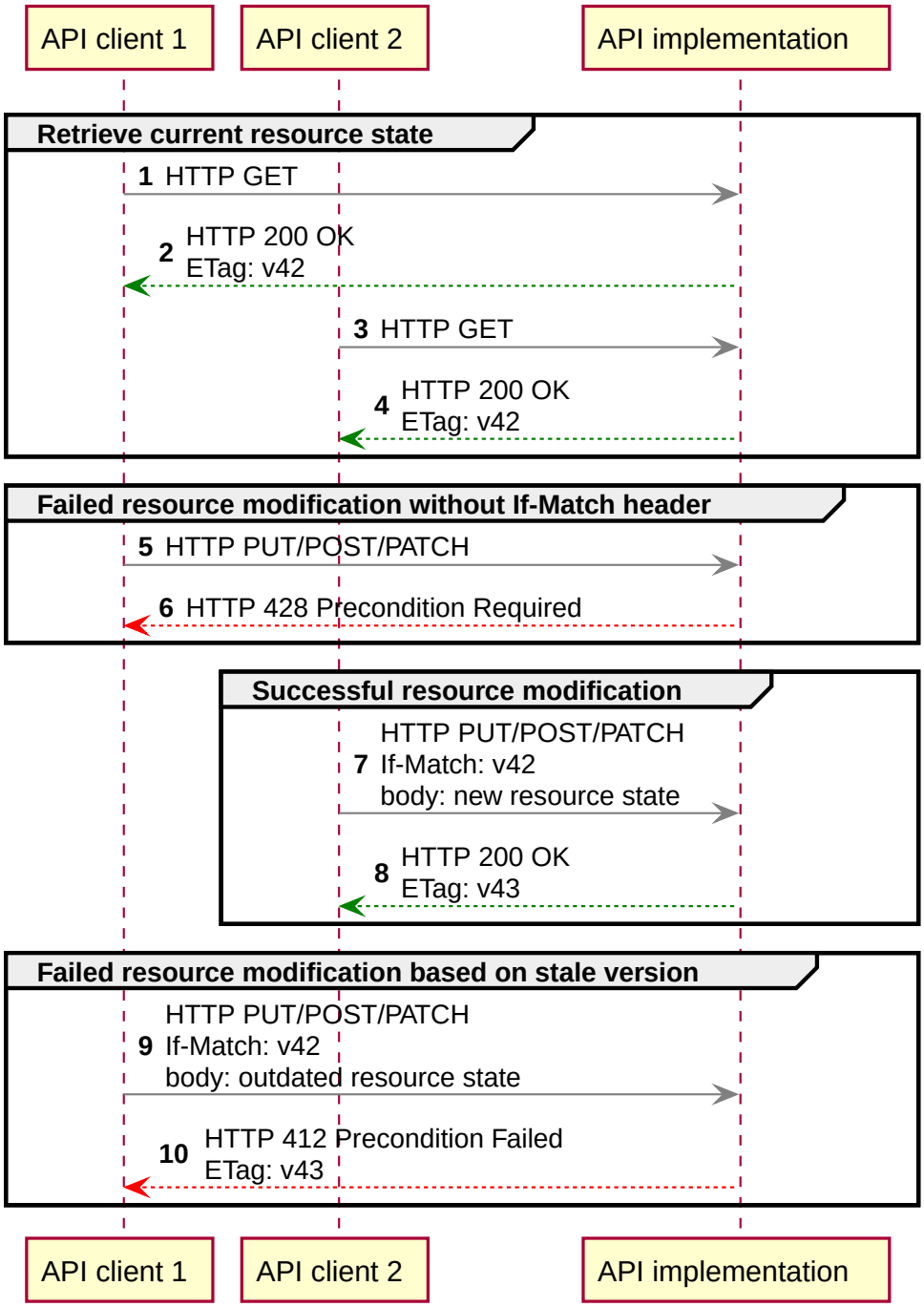

Figure 84. Optimistic concurrency preventing concurrent modification of a REST API resource.

6.4.11. Exercise 12: Identify API resources that may require protection from concurrent modification

Inspect all APIs in Acme Insurance's application network (Figure 57) and

1. Identify resources accessed by those APIs that may require protection from concurrent modification.
2. Discuss what implementing concurrent modification protection in these APIs would mean for their API clients.
3. Do you consider HTTP-based optimistic concurrency control a standard feature of APIs that is helpful in many situations?

Solution

The features discussed so far do neither require nor benefit from optimistic concurrency control:

- "Create quote for aggregators" feature is not a modification but a de-novo creation of new quotes
- "Retrieve policy holder summary" feature is a read-only operation
- "Submit auto claim" feature is not a modification but a de-novo creation of a new claim submission

A hypothetical feature that might benefit from optimistic concurrency control is the modification of policy holder details through the Customer Self-Service Mobile App:

- It is theoretically conceivable that updates to the details of the same policy holder occur concurrently, through separate session of the Customer Self-Service Mobile App or through the Customer Self-Service Mobile App and a backend system.
- But this is unlikely, and even if it happened there is no clear business need or value in protecting against this kind of situation.

This discussion shows that HTTP-based optimistic concurrency control to achieve protection of API resources from concurrent modification is a technical approach that is to be *used with caution and only when there is clear business value* in doing so. Most often, the nature of the interactions in an application network calls for embracing concurrency rather than imposing the illusion that all modifications to resources occur sequentially along one universally applicable time axis.

6.5. Taking stock

6.5.1. Reviewing the current state of Acme Insurance's application network

At this point you have

- identified all APIs
- for both products/projects currently in-scope

In collaboration with the C4E you have selected an enterprise-wide (application network-wide) approach to

- API versioning
- API data model design (no Enterprise Data Model)

For each API you have

- captured NFRs
- decided on architecturally relevant design aspects (such as caching, asynchronous execution and concurrency handling)
- selected and configured appropriate API policies

Along the way, important Technology Architecture-aspects of the Enterprise Architecture have been decided:

- Anypoint Platform and CloudHub have been chosen or event configured to support the APIs and their API policies
  - For instance, CloudHub Dedicated Load Balancers have been selected for TLS mutual authentication
- API policies are being enforced in the API implementations themselves rather than in API proxies
- An Identity Provider (PingFederate) has been chosen for OAuth 2.0 Client Management

Last but not least, a decentralized C4E has been established at Acme Insurance IT, which

- enables the LoB IT project teams to implement the strategic products "Aggregator Integration" product and "Customer Self-Service App" product
- owns the harvesting and publishing of reusable assets into Acme Insurance’s Anypoint Exchange
- helps setting and spreading the aforementioned application network-wide standards

All but one API, namely the "Aggregator Quote Creation EAPI", whose technology interface is defined by the Aggregator, are JSON REST APIs.

Almost all of the above information is directly visible in the interface contracts of APIs and has therefore been defined in RAML fragments and RAML definitions.

The Anypoint Exchange entry of each API augments the "raw" RAML definition with an API Notebook, an API Console and essential textual descriptions and context-setting. All of the above information has been published to Acme Insurance's Anypoint Exchange and is visible across the Acme Insurance application network. It may also have been published to the Acme Insurance Public (Developer) Portal (Exchange Portal) and be visible on the public internet.

Summary

- In designing APIs start with the RAML definitions, using the simulation features of API designer in Anypoint Design Center
- Extract reusable RAML fragments and publish them to Anypoint Exchange
- A semantic versioning-based API versioning strategy was chosen, that exposes only major version numbers in all places except where the exact RAML definition asset is required in Anypoint Exchange and Anypoint API Manager
- API data models were defined by the Bounded Context to which the APIs belong
- System APIs were chosen to abstract from backend systems towards the Bounded Context they belong to
- HTTP-based asynchronous execution of API invocations and caching were employed wherever needed to meet NFRs
- Most decisions change the interface contract of APIs and were captured in RAML fragments and RAML definitions and published to Anypoint Exchange

Module 7. Architecting and Deploying Effective API Implementations

Objectives

- Describe auto-discovery of API implementations implemented as Mule applications
- Appreciate how Anypoint Connectors serve System APIs in particular
- Describe CloudHub
- Choose Object Store in a CloudHub setting
- Apply strategies that help API clients guard against failures in API invocations
- Describe the role of CQRS and the separation of commands and queries in API-led connectivity
- Describe the role of Event Sourcing

7.1. Developing API implementations for Mule runtimes and deploying them to CloudHub

7.1.1. Introducing API implementations management on Anypoint Platform

API implementations may be

- Either developed *as Mule applications for the Mule runtime*
  - in *Anypoint Studio* or *Flow designer*
  - and deployed to an *Anypoint Platform runtime plane*, such as the CloudHub (3.2.2)
  - with *API Management, Analytics*, etc. provided by an *Anypoint Platform control plane* with or without separate API proxies
- Or developed for *other runtimes* (Node.js, Spring Boot, etc.)
  - and deployed to matching runtimes outside Anypoint Platform
  - with *API Management, Analytics*, etc. provided by an *Anypoint Platform control plane* via *API proxies executing in an Anypoint Platform runtime plane*

7.1.2. Introducing Auto-discovery of API implementations

API implementations developed as Mule applications and deployed to a Mule runtime (this includes API proxies) should always be configured to participate in *auto-discovery*:

- When the API implementation starts up it automatically registers with Anypoint API Manager as an implementation of a given API instance (which implies API, version and environment)
- Receives all API policies configured in Anypoint API Manager for that API instance
- The API implementation by default refuses API invocations until all API policies have been applied
  - This is called the *gatekeeper* feature of the Mule runtime

The above suggests that *auto-discovery* is a form of *auto-registration* of an API implementation with Anypoint API Manager for an API instance.

7.1.3. Anypoint Connectors to implement System APIs

Acme Insurance has extensive need to connect to backend systems (Figure 57). By definition, this mostly concerns the API implementations of System APIs:

- The Mainframe-based Policy Admin System, which needs to be accessed via WebSphere MQ
- The WebSphere-deployed Motor Claims System, which needs to be integrated with by directly accessing its DB/2 database
- The web-based Home Claims System, which exposes SOAP APIs

Connectors are either pre-built (Anypoint Connectors) or custom-developed components that help integrate Mule applications with external systems:

- Anypoint Platform has over 120 Anypoint Connectors, many of which are bundled with Anypoint Studio, while others are available in the public Anypoint Exchange (Figure 22) and the MuleSoft Nexus Maven repository
- Support categories: Premium, Select, MuleSoft Certified, Community

Acme Insurance can use the following Anypoint Connectors to implement its System APIs:

- The *JMS Connector* to connect to WebSphere MQ (https://docs.mulesoft.com/connectors/jms-connector)
- The *Database Connector* to access the DB/2 database (https://docs.mulesoft.com/connectors/db-connector-index)
- The *Web Service Consumer* to invoke SOAP APIs (https://docs.mulesoft.com/connectors/web-service-consumer)

See https://docs.mulesoft.com/connectors/ for documentation on the more popular Anypoint Connectors.

The existence of Anypoint Connectors is one more reason why Acme Insurance elects to implement API implementations for the Mule runtime, using Anypoint Studio.

7.1.4. Fundamentals of CloudHub Technology Architecture

CloudHub provides a scalable, performant and highly-available MuleSoft-hosted option for executing Mule applications - such as those representing API implementations - on Mule runtime instances.

For the purpose of this discussion, the term Mule application is used in preference to the term API implementation because CloudHub fundamentally is not concerned with the kinds of services provided by the Mule applications it deals with.

- CloudHub itself and Mule applications deployed to CloudHub execute on AWS infrastructure (3.2.10)
- CloudHub workers are AWS EC2 instances of various sizes (Table 2), running Linux and a Mule runtime within a JVM
- The maximum number of CloudHub workers for a single Mule application is currently 8
- The size and number of CloudHub workers can be *automatically scaled* to match actual load (9.3.1)
- CloudHub workers also execute CloudHub system services on the OS and Mule runtime level (Figure 85), which are required for the platform capabilities provided by CloudHub, such as
  - *Monitoring* of Mule applications in terms of CPU/memory/etc. usage, number of messages and errors, etc., which allows Anypoint Platform to provide analytics and alerts based on these metrics
  - *Auto-restarting* failed CloudHub workers (including a failed Mule runtime on an otherwise functional EC2 instance)
  - *Load-balancing* only to healthy CloudHub workers
  - *Provisioning* of new CloudHub workers to increase/reduce capacity
  - *Persistence*, using Object Stores, of message payloads or other data across the CloudHub workers of a Mule application
  - *DNS entries* for the CloudHub workers and CloudHub Load Balancer as detailed in 7.1.11

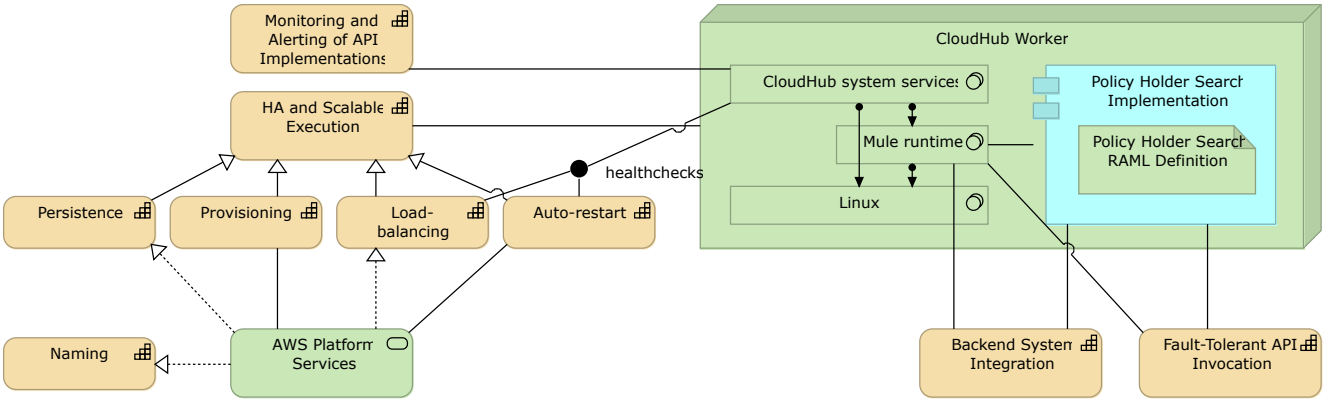

Figure 85. Anatomy of a CloudHub worker used by the API implementation of the "Policy Holder Search PAPI" and the capabilities bestowed on the API implementation by the CloudHub worker, also by making use of AWS Platform Services.

Table 2. CloudHub worker sizing and illustrative mapping to EC2 instance types, subject to change without notice.

Worker Name	Worker Memory	Worker Storage	EC2 Instance Name	EC2 Instance Memory
0.1 vCores	500 MB	8 GB	t2.micro	1 GB
0.2 vCores	1 GB	8 GB	t2.small	2 GB
1 vCore	1.5 GB	8 + 4 GB	m3.medium	3.75 GB
2 vCores	3.5 GB	8 + 32 GB	m3.large	7.5 GB
4 vCores	7.5 GB	8 + 40 + 40 GB	m3.xlarge	15 GB
8 vCores	15 GB	8 + 80 + 80 GB	m3.2xlarge	30 GB
16 vCores	32 GB	8 + 160 + 160 GB	m4.4xlarge	64 GB

- Every Mule application is assigned a DNS name that resolves to the CloudHub Load Balancer (7.1.9 and Table 4)
- Every Mule application also receives two other well-known DNS names which resolve to the public and private IP addresses of all CloudHub workers of the Mule application (Table 4)
  - Private IP addresses are only routable within the AWS VPC in which the CloudHub worker resides (Figure 92)
  - CloudHub workers of a Mule application can be assigned *static public IP addresses*, while private IP addresses are always dynamic
- DNS entries are maintained by the AWS platform service Route 53

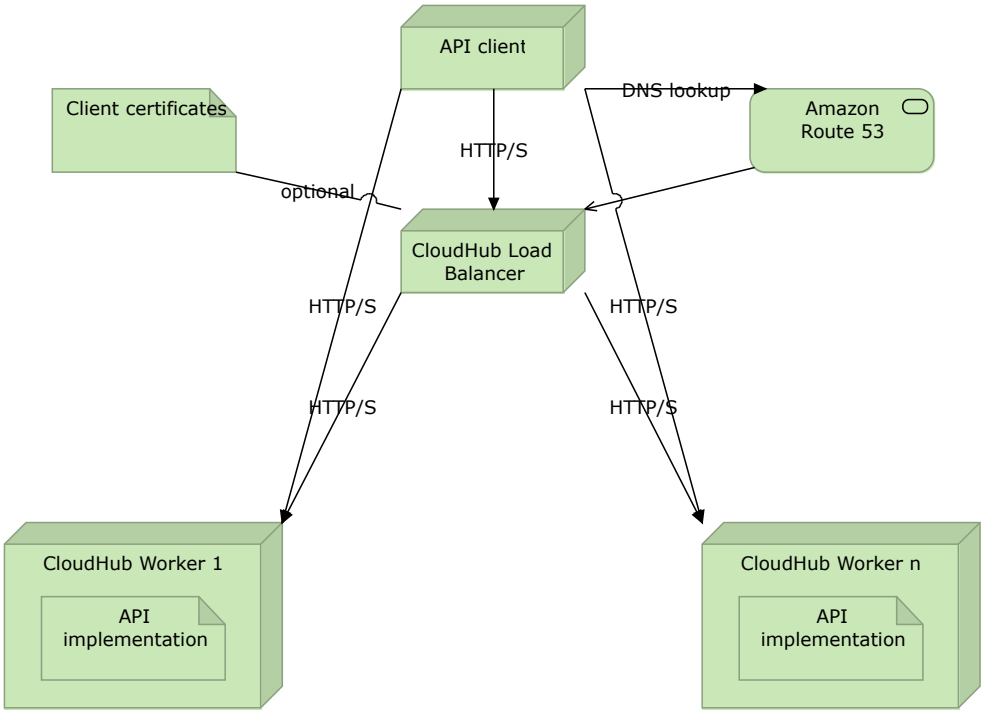

Figure 86. A very simple depiction of the general high-level CloudHub Technology Architecture as it serves API clients invoking APIs exposed by API implementations implemented as Mule applications running on CloudHub. The association of client certificates with the CloudHub Load Balancer requires a CloudHub Dedicated Load Balancer and is not supported for the CloudHub Shared Load Balancer.

7.1.5. Performance of CloudHub workers of various sizes

The number of HTTP requests per second achievable by deploying an API implementation to CloudHub workers of various sizes is shown in Figure 87 [Ref15].

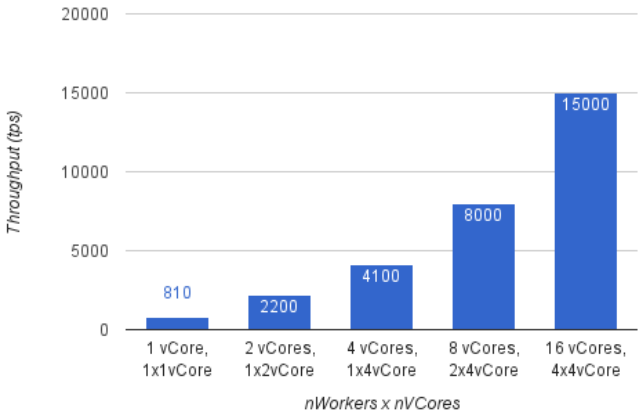

Figure 87. Throughput in terms of HTTP requests per second of a simple API implementation deployed to one or two CloudHub workers of size 1, 2 or 4 vCores (Mule 3.7).

Notes:

- The simple API implementation for which timings are quoted in Figure 87 exposes a SOAP/HTTP endpoint that performs some minor transformation and processing of the SOAP request

7.1.6. Understanding the CloudHub Shared Worker Cloud

Every Mule application deployed to CloudHub gives rise to one or more CloudHub workers in an AWS VPC in a particular AWS region. By default, this is a multi-tenant AWS VPC shared by all Mule applications, from all Anypoint Platform users, deployed to that particular AWS region. This is the so-called *CloudHub Shared Worker Cloud*:

- CloudHub distributes the CloudHub workers of a Mule application over the AWS AZs available in the AWS VPC the Mule application is being deployed to, thereby providing higher availability (Figure 89)
- Firewall rules (AWS Security Groups, a form of stateful firewall) of the CloudHub Shared Worker Cloud are fixed (Figure 88 and Figure 92):
  - TCP/IP traffic from anywhere to port 8081 (HTTP) and 8082 (HTTPS) on each CloudHub worker
  - TCP/IP traffic from within the AWS VPC to port 8091 (HTTP) and 8092 (HTTPS) on each CloudHub worker

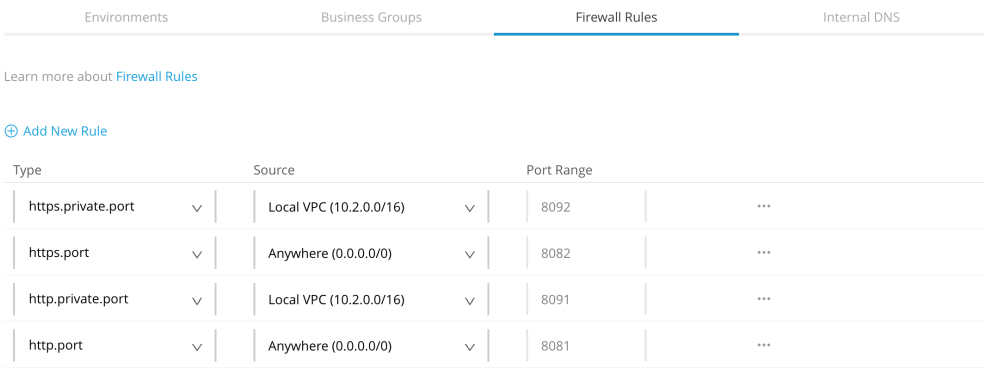

Figure 88. Firewall rules for the CloudHub Shared Worker Cloud, which are also the default rules for any Anypoint VPC.

7.1.7. Understanding Anypoint VPCs

Anypoint Platform also allows the configuration of Anypoint VPCs for a particular Anypoint Platform organization (or business group within that organization). Mule applications deployed to CloudHub can then be deployed to one of these Anypoint VPCs instead of the CloudHub Shared Worker Cloud.

An Anypoint VPC is backed by an AWS VPC, in the requested AWS region, and is assigned exclusively to its owning Anypoint Platform organization. That is, only (authorized) users from this Anypoint Platform organization may deploy Mule applications to this Anypoint VPC. The result of any such deployment is, as always, one or more CloudHub workers - but this time in this particular, private AWS VPC in that region (Figure 89).

An Anypoint VPC is created and managed by Anypoint Platform users through the Anypoint Platform web UI, Anypoint Platform APIs or Anypoint CLI. Doing so implicitly operates on the underlying AWS VPC, which is assigned exclusively to the managing Anypoint Platform organization. But AWS-level access to this AWS VPC is not provided, nor can that AWS VPC be used outside of Anypoint Platform, e.g., to launch arbitrary EC2 instances. Similarly, an existing AWS VPC cannot be re-purposed to become an Anypoint VPC.

An Anypoint VPC builds on the characteristics of the CloudHub Shared Worker Cloud as introduced in 7.1.6 but provides additional options and services:

- In general, an Anypoint VPC is similar to a *virtual network* in a given AWS region
- The *private IP address range and region* must be defined when creating the VPC and cannot be changed
- Every CloudHub worker receives a *private IP address from the address range of its VPC* - be it an Anypoint VPC or the CloudHub Shared Worker Cloud. A well-known DNS record resolves to those private IP addresses (Table 4)
- Every CloudHub worker also receives a a *public IP address* that is not under the control of the Anypoint VPC admin. Again, a well-known DNS record resolves to these public IP addresses (Table 4)
- The administrator of an Anypoint VPC has full control over the *Firewall Rules* for that VPC
  - By default these are the same as for the CloudHub Shared Worker Cloud (Figure 88)
  - The default firewall rules that provide *public access to ports 8081/8082* on any CloudHub worker can therefore be *removed* by the Anypoint VPC admin
  - Also, firewall rules can be added that *allow access to other ports than 8081/8082 and 8091/8092* on the CloudHub workers either from within the VPC or from any given source IP range
- DNS lookups of CloudHub workers in an Anypoint VPC can use *customer-supplied DNS servers* for particular DNS domains (assuming the DNS server IP addresses are reachable from the Anypoint VPC: 7.1.8)
- *Anypoint VPCs and Anypoint Platform environments* can be in a many-to-many relationship
  - Often Production environment assigned to Production VPC and all other environments assigned to "Sandbox VPC" (Figure 89)

    - The Production environment can also comprise additional Anypoint VPCs in other regions
- *CloudHub Dedicated Load Balancers* can be instantiated *within* an Anypoint VPC (7.1.10)
- An Anypoint VPC can be *connected to an on-premises network using an IPsec tunnel or AWS DirectConnect* (7.1.8)
- An Anypoint VPC can be privately *connected to a customer's AWS VPC using VPC peering* (7.1.8)

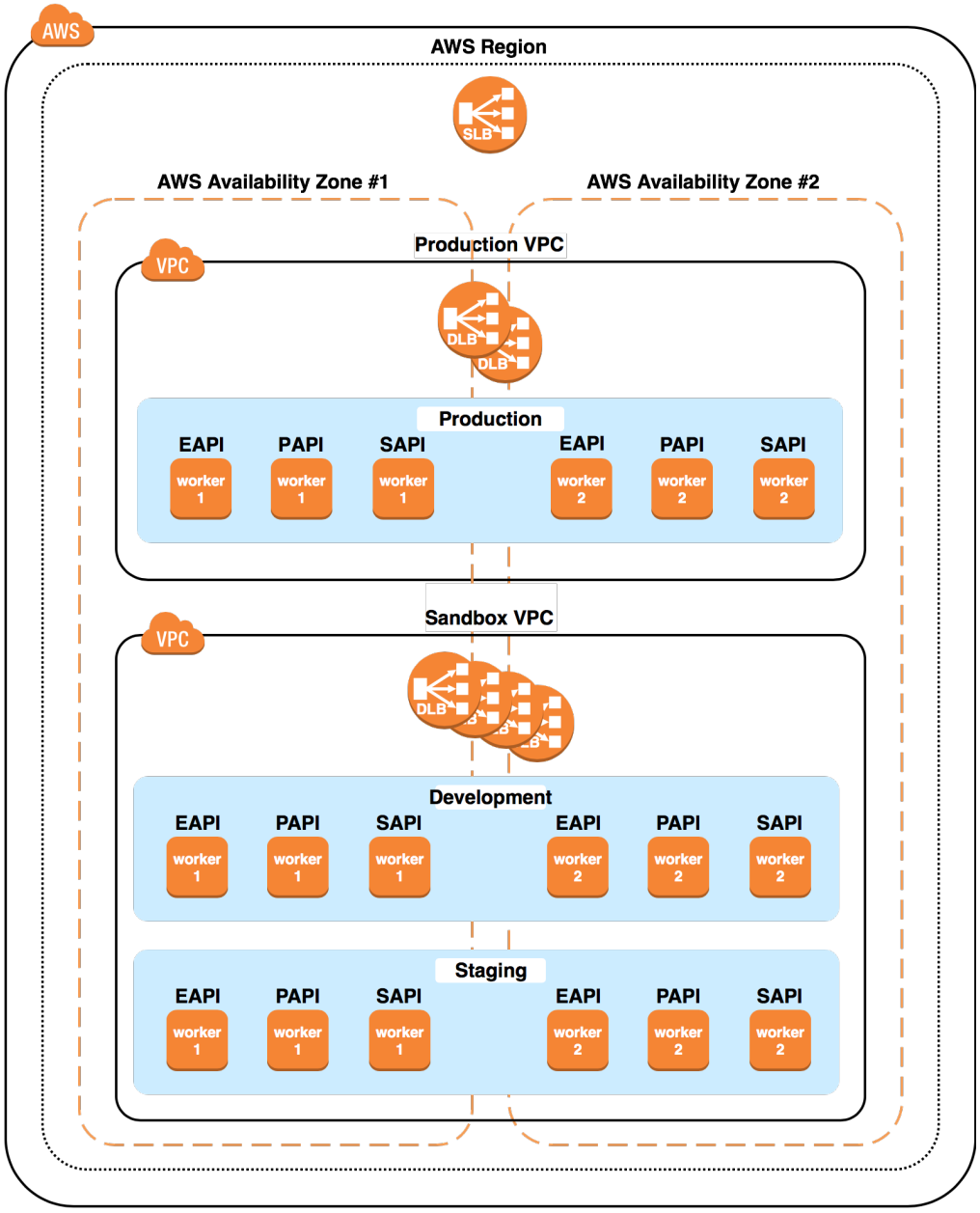

Figure 89. Two Anypoint VPCs were created in a CloudHub region: The Production VPC is associated with the Production environment and hosts two CloudHub Dedicated Load Balancers (DLB; presumably one for public APIs and the other for VPC-private APIs) while the Sandbox VPC is associated with the Development and Staging environments and hosts four CloudHub Dedicated Load Balancers (presumably separating public and VPC-private APIs for the two environments). Several API implementations from all tiers of API-led connectivity have been deployed to all environments using two CloudHub workers each. The CloudHub workers for a given Mule application are assigned to different AZs. The CloudHub Shared Load Balancer (SLB) for the region is available for public APIs (but may go unused given the configured CloudHub Dedicated Load Balancers).

7.1.8. Connecting an Anypoint VPC to other networks

Because Anypoint VPCs are for the exclusive use of the owning Anypoint Platform organization, they can be connected to other types of networks the organization controls. In particular, an Anypoint VPC can be *privately connected* to:

- An *on-premises network using an IPsec tunnel or AWS DirectConnect* (Figure 90)
  - IPsec VPN can use more than one AZ of the AWS VPC, i.e., is not necessarily a single point of failure (at the time of this writing, this feature is not yet generally available)
- Another *AWS VPC using VPC peering* (Figure 91)
  - Not possible if the other AWS VPC originates from an Anypoint VPC, i.e., Anypoint VPCs cannot be peered with each other
  - Both VPCs are typically (but not necessarily) *in the same region*
  - Avoids traffic over the public internet for communication between these VPCs: all traffic stays within AWS infrastructure, which results in *higher bandwidth and reliability, lower latency and better security*

Both types of connections are private, hence the private IP addresses of the Anypoint VPC are relevant. Please note that these private IP addresses are currently always dynamically assigned.

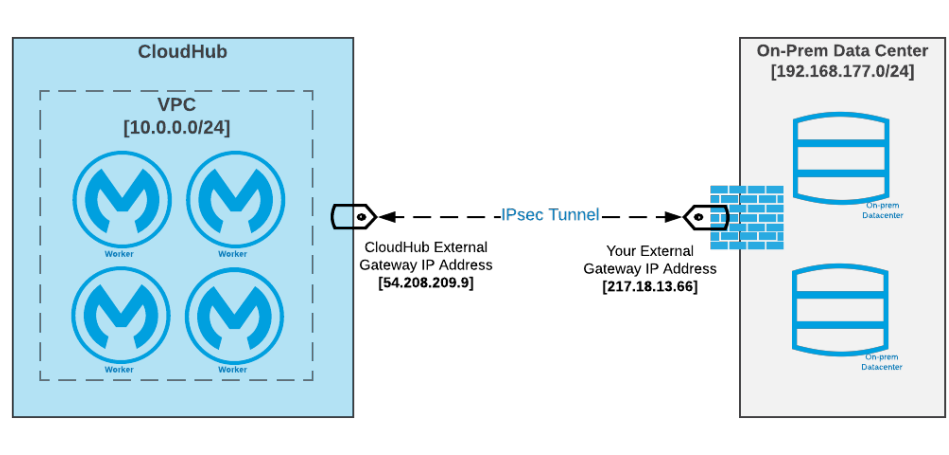

Figure 90. Private network connection via IPsec tunnel between an on-premises network (right side) and the AWS VPC underlying an Anypoint VPC (left side).

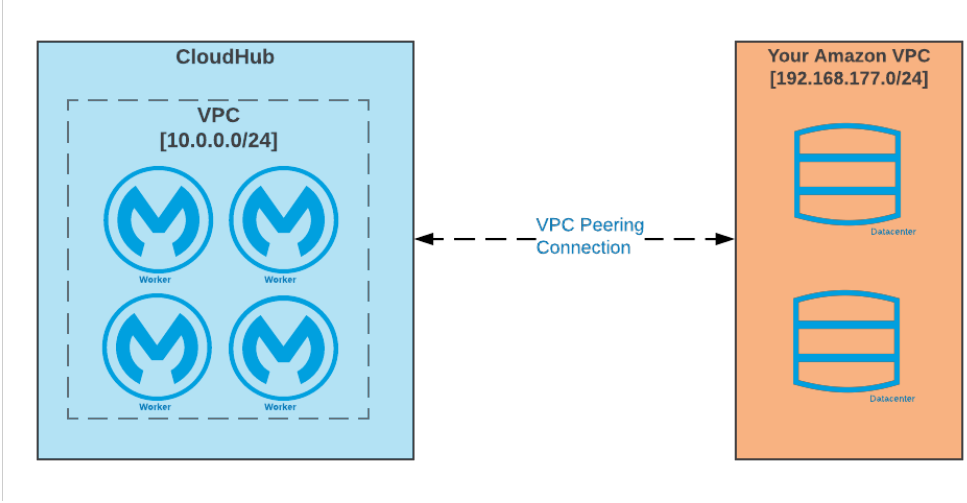

Figure 91. Private network connection via VPC peering between an AWS VPC (right side) and the AWS VPC underlying an Anypoint VPC (left side), which implies that the same organization controls both VPCs.

7.1.9. Understanding the CloudHub Shared Load Balancer

One form of the CloudHub Load Balancer service is the *CloudHub Shared Load Balancer*.

Every Mule application deployed to CloudHub receives a DNS entry pointing to the CloudHub Shared Load Balancer:

- The DNS entry is a *`CNAME` for the CloudHub Shared Load Balancer* in the region to which the Mule application is deployed (Table 4)
  - Therefore *no "vanity domain name"* can be defined to point to the CloudHub Shared Load Balancer
- The CloudHub Shared Load Balancer in a region is shared by all Mule applications in that region
- The CloudHub Shared Load Balancer builds on top of the AWS Elastic Load Balancer (ELB)
- HTTP and HTTPS requests, i.e., API invocations, sent to a CloudHub Shared Load Balancer on port 80 and 443, respectively, are forwarded by the CloudHub Shared Load Balancer to one of the Mule application's CloudHub workers (round-robin), where they reach the Mule application on *port 8081 and 8082*, respectively (Figure 92)
  - The CloudHub Shared Load Balancer hereby maintains the protocol (HTTP, HTTPS) initiated by the API client
  - By exposing only an HTTP endpoint on port 8081 or only an HTTPS endpoint on port 8082 the Mule application determines the protocol available to its API clients also via the CloudHub Shared Load Balancer

    - Only Mule applications exposing HTTP or HTTPS endpoints at port 8081 or 8082 can therefore make use of the CloudHub Shared Load Balancer
- The *CloudHub Shared Load Balancer terminates TLS connections* and uses its own server-side certificate

7.1.10. Understanding CloudHub Dedicated Load Balancers

The other form of the CloudHub Load Balancer service is the *CloudHub Dedicated Load Balancer*, which is available to Mule applications deployed to an Anypoint VPC. (This is in addition to the CloudHub Shared Load Balancer, which is also available to these Mule applications.)

One or more *CloudHub Dedicated Load Balancer* can be instantiated *within an Anypoint VPC* (Figure 89 and Figure 92):

- Each CloudHub Dedicated Load Balancer receives (more than one) private IP address from the private address range of the VPC, as well as (more than one) public IP address (Table 3)
    - The public IP addresses DNS name is an `A` record so that arbitrary `CNAME`s for "vanity domain names" can be defined to point to that record
- HTTP and HTTPS requests, i.e., API invocations, sent to a CloudHub Dedicated Load Balancer on port 80 and 443, respectively, are forward by the CloudHub Dedicated Load Balancer to one of the Mule application's CloudHub workers (round-robin), where they reach the Mule application on *port 8091 and 8092*, respectively (Figure 92)
    - Because a CloudHub Dedicated Load Balancer sits within an Anypoint VPC, the traffic between it and the CloudHub workers is internal to the VPC and can therefore use the customary internal service ports 8091/8092
    - Flexible *Mapping Rules* can be defined for manipulating request URLs and selecting the CloudHub Mule applications to the workers of which requests are forwarded
    - The *upstream protocol* for the VPC-internal communication between CloudHub Dedicated Load Balancer and CloudHub workers can be configured to be *HTTPS or HTTP*, i.e., can be different from the protocol used by the API client (unlike with the CloudHub Shared Load Balancer)
- The IP addresses of permitted API clients to a CloudHub Dedicated Load Balancer can be *whitelisted*
    - This is often used to define a CloudHub Dedicated Load Balancer for VPC-private APIs, in addition to a separate CloudHub Dedicated Load Balancer for public APIs (without whitelisting).
- CloudHub Dedicated Load Balancers perform *TLS termination* (just like the CloudHub

Shared Load Balancer)

- Each CloudHub Dedicated Load Balancer must be configured with *server-side certificates* for a public-private keypair for the HTTPS endpoints it exposes
- Optionally, client certificates can be added to a CloudHub Dedicated Load Balancer so that it performs *TLS mutual authentication*

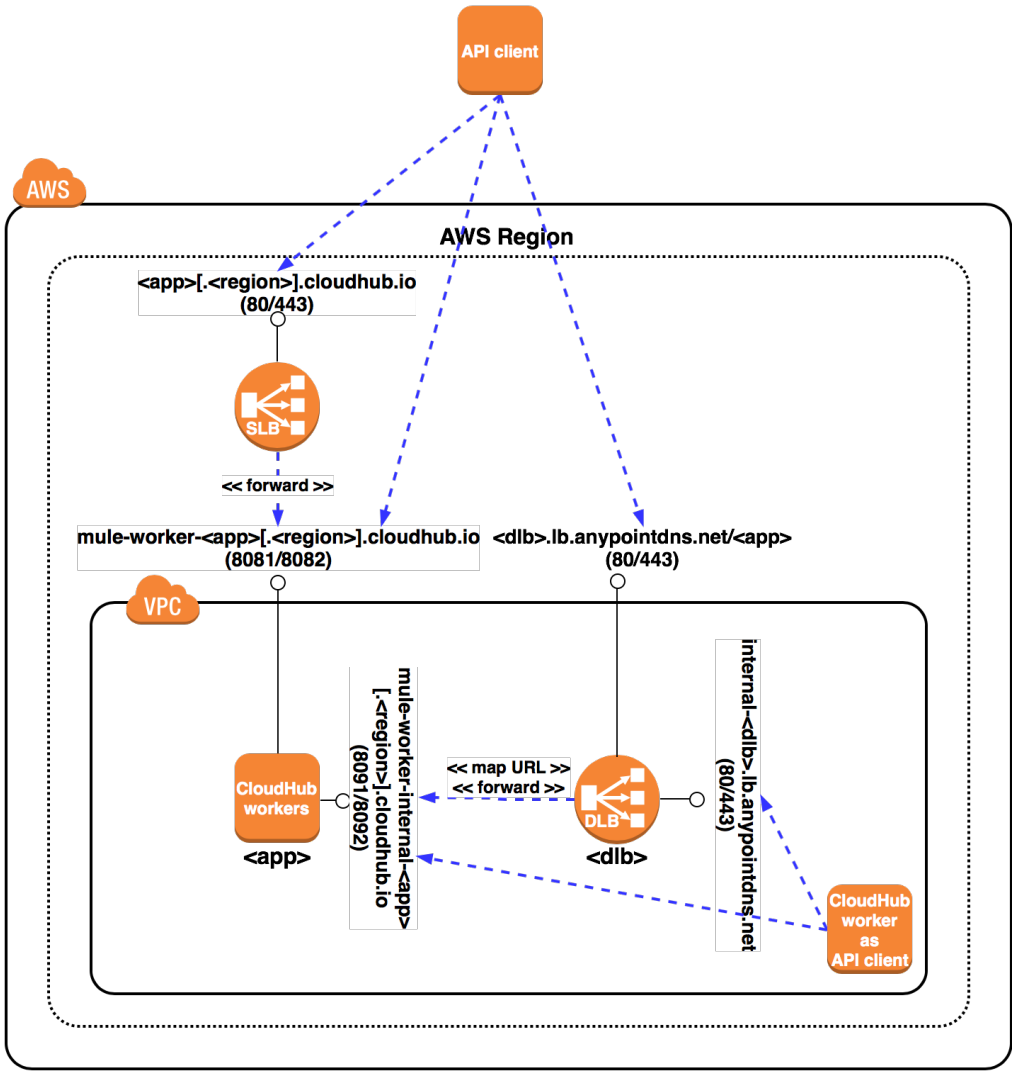

Figure 92. An API implementation/Mule application `<app>` is deployed to an Anypoint VPC, under management of the US Anypoint Platform control plane. This `<app>` exposes HTTP/HTTPS endpoints on ports 8081/8082 as well as on ports 8091/8092 (this combination in the same API implementation is unusual): depicted are the default access routes for API clients inside and outside this VPC, both directly to the CloudHub workers (unusual) as well as via the CloudHub Shared Load Balancer (SLB) and a CloudHub Dedicated Load Balancer `<dlb>`. The latter depends on `<dlb>`'s URL mapping rules, which are not shown.

7.1.11. Summarizing CloudHub DNS names

Several DNS names characterize the various aspects and features of the CloudHub implementation of the Anypoint Platform runtime plane, as already illustrated in Figure 92 and as detailed in Table 3 and Table 4.

Table 3. DNS names for CloudHub deployments under the management of Anypoint Platform control planes in different regions. Abbreviations: "DLB": CloudHub Dedicated Load Balancer, `<dlb>`: name of a particular CloudHub Dedicated Load Balancer.

	Control Plane Region	
	US East	**EU (Frankfurt)**
Web UI, Platform APIs	`anypoint.mulesoft.com`	`eu1.anypoint.mulesoft.com`
DLB Public IPs	`<dlb>.lb.anypointdns.net`	`<dlb>.lb-prod-eu-rt.anypointdns.net`
DLB Private IPs	`internal-<dlb>.lb.anypointdns.net`	`internal-<dlb>.lb-prod-eu-rt.anypointdns.net`

Table 4. DNS names for CloudHub deployments to Anypoint Platform runtime planes in various regions (rows) under the management of Anypoint Platform control planes in different regions (columns). The given DNS names are those resolving to the CloudHub Shared Load Balancer public IP. To obtain the DNS name resolving to the corresponding CloudHub workers' IPs prepend `mule-worker-` (public IPs) and `mule-worker-internal-` (private IPs), respectively. Abbreviations: `<app>`: name of a particular Mule application, "N/A": currently not available.

	Control Plane Region	
	US East	**EU (Frankfurt)**
US East (N Virginia)	`<app>.cloudhub.io`	N/A
US East (Ohio)	`<app>.cloudhub.io`	N/A
US West (N California)	`<app>.us-w1.cloudhub.io`	N/A
US West (Oregon)	`<app>.us-w2.cloudhub.io`	N/A
EU (Frankfurt)	`<app>.eu.cloudhub.io`	`<app>.de-c1.eu1.cloudhub.io`
EU (Ireland)	`<app>.eu.cloudhub.io`	`<app>.ir-e1.eu1.cloudhub.io`
EU (London)	`<app>.uk-e1.cloudhub.io`	N/A
APAC (Singapore)	`<app>.sg-s1.cloudhub.io`	N/A
APAC (Sydney)	`<app>.au-s1.cloudhub.io`	N/A
APAC (Tokyo)	`<app>.jp-e1.cloudhub.io`	N/A
Canada (Central)	`<app>.ca-c1.cloudhub.io`	N/A
S America (Sao Paulo)	`<app>.br-s1.cloudhub.io`	N/A

7.1.12. Exercise 13: CloudHub Technology Architecture for Acme Insurance

Acme Insurance has decided to deploy all their API implementations to CloudHub, under the management of the US Anypoint Platform control plane. Based on everything you already know about Acme Insurance - and making assumptions about the things you do not know - sketch the core elements of their CloudHub Technology Architecture for the Production environment. In particular, decide on, or provide the rationale for, the following:

1. Region(s) of their Anypoint Platform runtime plane(s)
2. Number of workers per API implementation and worker size
3. Need for and use of CloudHub Shared Worker Cloud, Anypoint VPCs, CloudHub Shared Load Balancers, CloudHub Dedicated Load Balancers, IPsec tunnels, VPC peering
4. API URL patterns, in particular hostnames, as seen by API clients
5. Ports on which API implementations should expose their HTTP/HTTPS endpoints

Solution

- System APIs need to connect to backend systems and should therefore be deployed to a region that is close to Acme Insurance’s data center for efficiency reasons. Assuming Acme Insurance is located in California this would be the *US West* region
- Experience APIs should be located close to consumers. With Acme Insurance being primarily regional, US West is also an appropriate choice for Experience APIs
- Acme Insurance sees no need for cross-region HA: cross-AZ HA in one region, as provided by CloudHub by default, is sufficient
- Acme Insurance has no reason to separate Experience APIs, Process APIs and System APIs into separate VPCs
- Therefore: deploy *all API implementations into CloudHub’s US West Anypoint Platform runtime plane*
- For availability reasons, every API implementation must be deployed to *at least two CloudHub workers*
- In production, each CloudHub worker should have *at least 1 vCore*
- The API implementations involved in the "Create quote for aggregators" feature, due to its demanding latency and throughput requirements, will likely require more CloudHub workers of a larger size
- There is currently not enough information to decide on more precise estimates of CloudHub worker size and number of workers
- To satisfy the *HTTPS mutual authentication* requirement for the "Aggregator Quote Creation

EAPI", *at least one CloudHub Dedicated Load Balancer* is required, which must be configured with the (client) certificate of the Aggregator

- If a CloudHub Dedicated Load Balancer is required for the "Aggregator Quote Creation EAPI", it makes sense to use it for all Experience APIs (which are all public)
- This necessiates *at least one Anypoint VPC* to host that CloudHub Dedicated Load Balancer - the Production VPC
- Connectivity between System APIs and backend systems is over Websphere MQ to the Policy Admin System, directly to the DB/2 database of the Motor Claims System and over SOAP to the Home Claims System. The latter could conceivably be over the public internet, but the first two really require dedicated private network connectivity. An *IPsec tunnel from Acme Insurance's data center to their Anypoint VPC* is therefore necessary
- The Home Claims System is hosted by an external outsourcing provider, but since VPN connectivity between the Acme Insurance data center and that provider is already set up, there is no need for a separate IPsec tunnel between that provider and the Anypoint VPCs
- The Production environment therefore requires *one Anypoint VPC in the US West region with a CloudHub Dedicated Load Balancer for the Experience APIs* and an IPsec tunnel to Acme Insurance's data center
- The hostnames of backend systems are maintained in Acme Insurance-internal DNS servers
- The Anypoint VPCs must therefore be configured to *resolve hostnames of Acme Insurance backend systems against these on-premises DNS servers*
- For security reasons, access to Process APIs and System APIs must be prohibited from outside the VPC
- The Acme Insurance security team demands that all API invocations be over HTTPS and none over HTTP
- Therefore: all API implementations must only expose *HTTPS endpoints on port 8092*
- To provide load-balanced access to the Process APIs and System APIs (only) from within the VPC, a *second CloudHub Dedicated Load Balancer, for VPC-private APIs*, is needed, with whitelisted source IP addresses only from the VPC's private IP address range
- To decouple API clients from the names chosen for the CloudHub Dedicated Load Balancer, *`CNAME`s must be defined for the two CloudHub Dedicated Load Balancers* in Acme Insurance's DNS server. For instance, `api.acmeinsurance.com` for `ans-pub.lb.anypointdns.net` (the public IPs of the CloudHub Dedicated Load Balancer for the public APIs, i.e., Experience APIs) and `private-api.acmeinsurance.com` for `internal-ans-priv.lb.anypointdns.net` (the private IPs of the CloudHub Dedicated Load Balancer for the private APIs, i.e., Process APIs and System APIs)
- Acme Insurance adopts a simple URL pattern convention: the *API name becomes the 1st part of the URL path*. E.g., `/policyoptionsranking-papi/`. As seen here, the `ans-` prefix used by all Acme Insurance API implementations should not be visible in URLs

- In addition, the URL mapping rules for public APIs should hide the `-eapi` suffix from API clients
- *External API clients*, which invoke Experience APIs, would therefore see an *URL pattern* exemplified by `https://api.acmeinsurance.com/aggregatorquotecreation/`, which would forward to API implementation `ans-aggregatorquotecreation-eapi` at port 8092
- *VPC-internal API clients*, which invoke Process APIs and System APIs, would therefore see an *URL pattern* exemplified by `https://private-api.acmeinsurance.com/policyoptionsranking-papi/`, which would forward to API implementation `ans-policyoptionsranking-papi` at port 8092
- Corresponding *URL mapping rules must be defined in both CloudHub Dedicated Load Balancers*
- Finally, the CloudHub Shared Worker Cloud and CloudHub Shared Load Balancer will not be used by Acme Insurance and there is no need for VPC peering
- Public DNS names pointing to the CloudHub Shared Load Balancer (e.g., `ans-policyholdersummary-papi.cloudhub.io`) will still exist, but the API implementations will not be reachable under these hostnames, because they do not listen on ports 8081 or 8082. (But note that these are the names used in other parts of this document.)

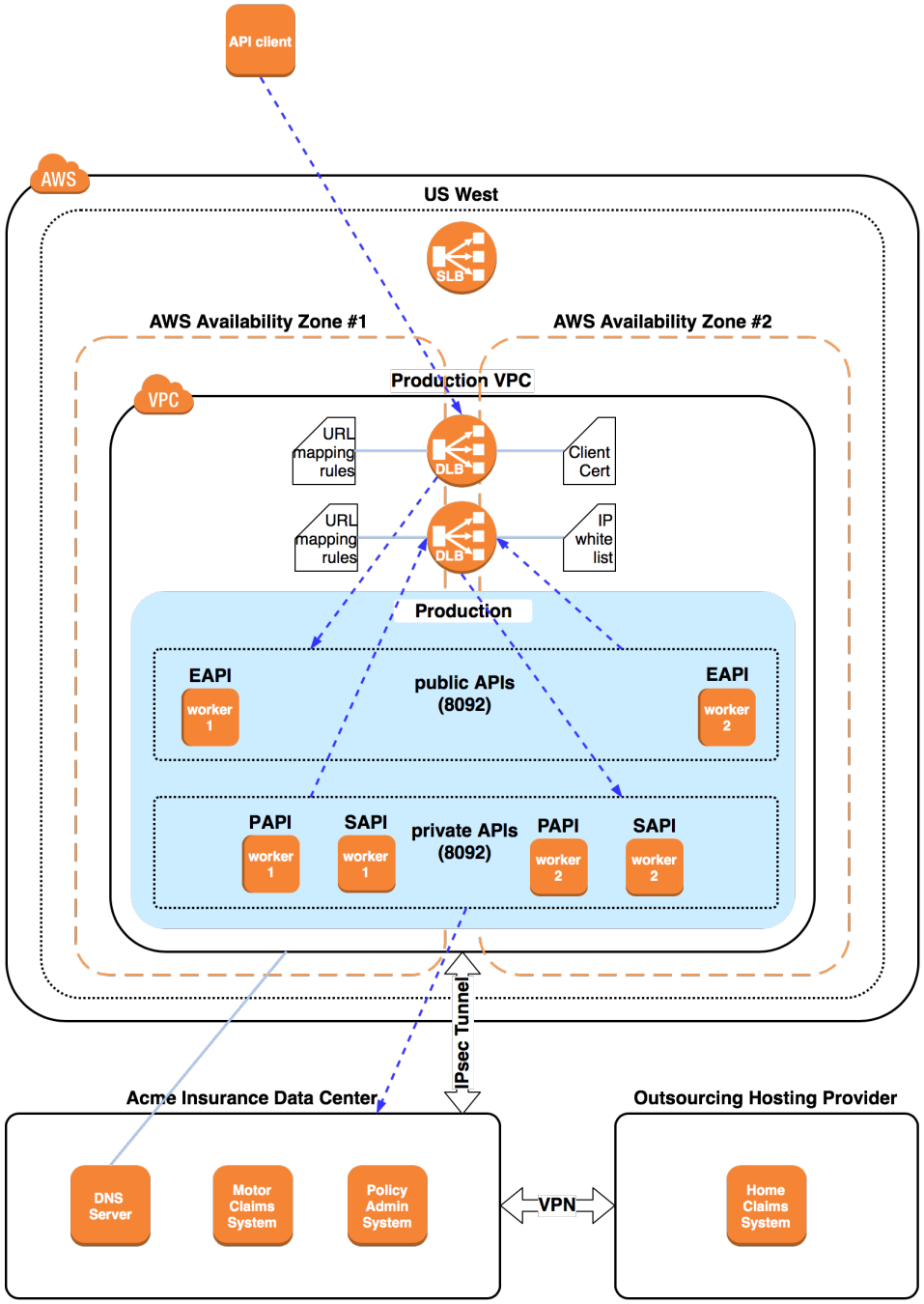

Figure 93. CloudHub Technology Architecture for Acme Insurance. Blue dashed arrows indicate typical request flows (over HTTPS unless from a System API to a backend system), which are subject to firewall restrictions in the Anypoint VPC and Acme Insurance data center (not shown) and IP whitelisting in the private CloudHub Dedicated Load Balancer (DLB). Also not shown are: individual concrete API implementations, CloudHub worker sizes, hostnames and URLs. The grouping around public APIs and private APIs is not reflected in any configuration and only serves to visualize the association of these API groups with the respective CloudHub Dedicated Load Balancers. The CloudHub Shared Load Balancer (SLB) for the region is shown but not used.

7.1.13. Understanding Object Store

Object Store is a key-value service available to all Mule applications executing in a Mule runtime, regardless of how and where that Mule runtime is deployed. Acme Insurance makes extensive use of Object Stores for several use cases, including

- storing correlation information for various forms of asynchronous processing (5.2.4, 6.4.4)
- circuit breakers (7.2.6) and caches (7.2.9)

The following is true for Object Stores general - in all Mule runtime deployment options, not just in CloudHub:

- Any Mule application can use *several Object Store instances*
- Any given Object Store instance is only available from within the Mule application that defined it - *no inter-Mule application data exchange* via Object Stores
- An Object Store is a *key-value store*, but the values can be, for instance, *JSON documents*
- No support for *transactions or queries* - Object Store is not a general-purpose database service
- An Object Store instance can be *persistent or transient*, *expires entries* after a given TTL, and may have a max *capacity*
- A suitable *default persistent Object Store* instance (named `_defaultUserObjectStore`) is always available to all Mule applications without any extra configuration
- Mule applications typically use an Object Store through the *Object Store Connector*
- Many Mule runtime components and Anypoint Connectors use Object Store for persistence: idempotent filters/validators, watermarks, OAuth 2.0 token stores, stateful API policies such as Rate Limiting, …

7.1.14. Understanding Object Store in CloudHub

For Mule runtimes and Mule applications deployed to CloudHub, Object Store has the following characteristics:

- Persistent Object Stores keep data (keys and values) in the same AWS region as the owning Mule application (and therefore its CloudHub workers and Mule runtimes; 3.2.10)
- The state of an Object Store is (only) available to all workers of its owning Mule application
  - Can be circumvented using the Object Store REST API
- Content of (the default) Object Store of a Mule application is exposed in Anypoint Runtime Manager
- Max TTL of 30 days

- Unlimited number of keys; each value limited to 10 MB
- Cross-AZ HA within an AWS region
- Encryption in transit and at rest

7.2. Designing API clients to use fault-tolerant API invocations

7.2.1. Exercise 14: Failures in API invocations

1. List every kind of failure that can occur when an API client invokes an API implementation
2. Think of ways of guarding against each of the failures you have identified

Solution

The invocation of an API implementation by an API client fails if any of the intervening components fails at the moment of the API invocation:

- The hardware on which the API client or API implementation execute
- The virtualization or container stacks on which the API client or API implementation execute
- The operating system, JVM or other runtime and libraries on which the API client or API implementation rely
- Any network component such as Ethernet cards, switches, routers, cables, WIFI transmitters, …
- This includes all Anypoint Platform components and underlying AWS services involved in the API invocation: API policies, API proxies, load balancers, CloudHub workers, etc.
- The API implementation itself, e.g., because of a bug in the application code or a failed deployment

7.2.2. Understanding the significance of failure in application networks

The impact of failed API invocations is amplified by the typically high degree of dependency of any API implementation on other APIs. This high degree of dependency of APIs is an inherent feature of application networks. It is also typically a *desired* feature because it is one direct consequence of a high degree of *reuse* of APIs, and as such is proof of the success of the application network.

Expressed in graph theoretic terms, the min/max/average *degree of a graph* of APIs is the min/max/average number of API implementations that depend on any given API. In this

sense, successful application networks are characterized by a *high (average) degree*.

However, a high degree of dependency between APIs means that failures in the invocation of an API affect many other API implementations and the services they offer to the application network - which in turn affects many other API implementations and their services - and so on. That is, if unmitigated, failures in API invocations propagate transitively through an application network.

In other words, a successful application network - one with a high degree of dependency between its APIs - is also an application network in which the failure of any API triggers downstream failures of many other APIs.

For this reason, *making API invocations fault-tolerant* is an essential aspect of successful application networks.

7.2.3. Designing API clients for fault-tolerant API invocations

A fault-tolerant API invocation is *an invocation from an API client to an API implementation* via its API where a *failure of the API invocation does not necessarily lead to a failure of the API client*. Because the API client itself is typically an API implementation, this means that the API provided by that API implementation does not in turn cause *its* API clients to experience failures.

The most important goal of making API invocations fault-tolerant is *breaking transitive failure propagation*.

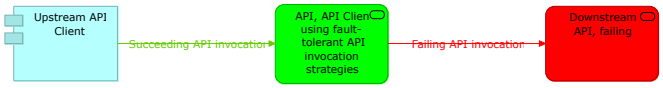

Figure 94. Visualization of an API client (green) that is itself and API implementation and exposes an API to upstream API clients (light blue) and which employs fault-tolerant API invocations to a downstream API (red).

There is a fairly well-established repertoire of *approaches for implementing fault-tolerant invocations* of any kind. Most of these strategies should be *applied by all API clients to all API invocations* whenever possible:

- Timeout
- Retry
- Circuit Breaker
- Fallback API invocation
- Opportunistic parallel API invocations

- Cached fallback result
- Static fallback result

Also see [Ref11], [Ref12].

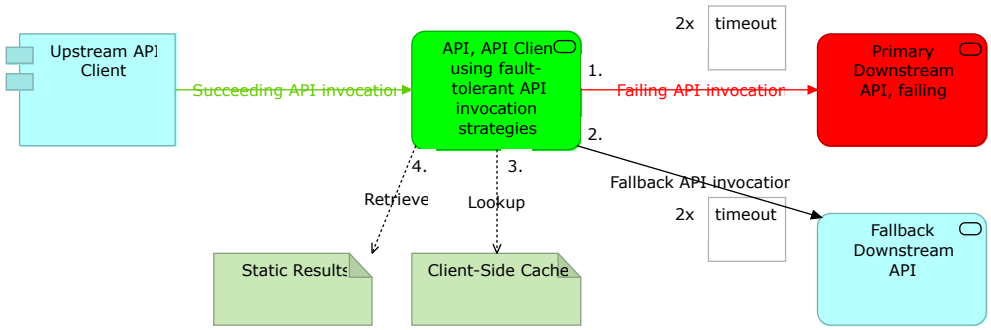

Figure 95. The most important fault-tolerant API invocation strategies and the order in which they should be employed.

7.2.4. Using timeouts to guard against slow APIs

When the invocation of a downstream API is slow to return

- the SLA of the API client is put at risk even if the API invocation ultimately succeeds
- the API invocation has a higher-than-usual probability of eventually failing anyways

For both of these reasons, *timeouts for API invocations* should be set carefully to

- be in accordance with the SLA of the API client
- be as low as possible, given the SLA-defined expected response time of the API

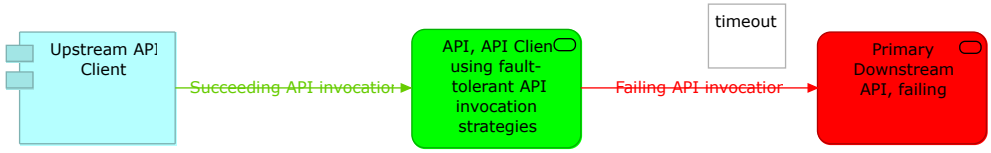

Figure 96. Short timeouts of API invocations are the most fundamental protection against failing APIs.

For instance:

- The "Aggregator Quote Creation EAPI" has an SLA (defined by the NFRs for the "Create quote for aggregators" feature) of a median/max response time of 200 ms/500 ms
- It synchronously invokes 3 Process APIs, in turn, starting with "Policy Holder Search PAPI"

- The task performed by the "Policy Holder Search PAPI" is the simplest of all 3 Process APIs and should hence be fastest to accomplish
- Hence the "Aggregator Quote Creation EAPI" should define a timeout of no more than approx. 100 ms for the invocation of the "Policy Holder Search PAPI"
- The SLA for the "Policy Holder Search PAPI" must therefore guarantee/promise that a sufficient percentage (say, 95%) of API invocations will complete within 100 ms. If "Policy Holder Search PAPI"'s SLA does not live up to this, then this API (or better, its current API implementation) is unsuitable for implementing the "Aggregator Quote Creation EAPI". For instance, if the percentage of API invocations to the "Policy Holder Search PAPI" that can be expected to complete within the timeout of 100 ms is significantly below 95% (say, 80%) then, under normal conditions (without failure) the percentage of these API invocations that will be timed-out is unreasonably high (20%) and will therefore jeopardize the effiicent operation of the "Aggregator Quote Creation EAPI". (Response times are typically log-normally distributed.)

A timed-out API invocation is equivalent to a failed API invocation, hence timing-out can only be the first in a sequence of fault-tolerance strategies.

API clients implemented as Mule applications and executing on a Mule runtime have various options for configuring timeouts of API invocations:

- At the level of the HTTP request
- For an entire transaction
- As part of higher-level control constructs like Scatter-Gather

7.2.5. Retrying failed API invocations

After an API invocation has failed it *may succeed when tried again*. This will only be the case *if the failure was transient rather than permanent*. This is, retrying an API invocation is wasteful if the API invocation failure is of a permanent nature.

It is typically difficult for an API client to decide beyond doubt that a failure which occurred during an API invocation was of a transient nature. This is way the default approach is often to retry all failed API invocations.

In general, all connection issues (which materialize in Mule runtime as Java net or IO exceptions) should be dealt with as transient failures.

In REST APIs, which are assumed to make correct use of HTTP response codes, *response codes in the 4xx range signify client errors and are therefore mostly permanent* and the API invocation that led to such a response code should hence not be retried. HTTP *response codes*

in the 5xx range, by contrast, should be expected to signify transient failures. These HTTP 4xx client error response codes are exceptions to the previous rule and should also be *treated as transient failures*:

- HTTP 408 Request Timeout
- HTTP 423 Locked
- HTTP 429 Too Many Requests

In addition, HTTP states that only idempotent HTTP methods (6.4.9) may be retried without causing potentially unwanted duplicate processing:

- GET
- HEAD
- OPTIONS
- PUT
- DELETE

Retrying an API invocation by necessity increases the overall processing time for the API client. It should therefore be

- restricted to *only a few retries*
- with *short wait times* between retries
- combined with *short timeouts* for each API invocation

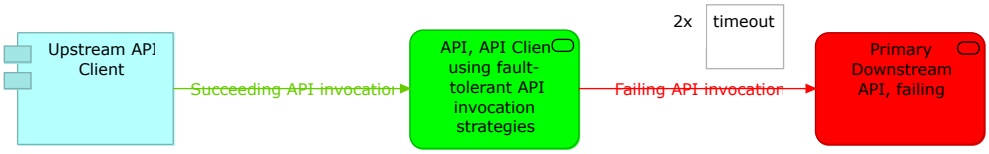

Figure 97. Retrying API invocations a few times with short delays between retries and short timeouts for each API invocation.

API clients implemented as Mule applications and executing on a Mule runtime have the HTTP Request Connector and Until Successful Scope at their disposal for configuring retries of API invocations. The HTTP Request Connector has (configurable) support for interpreting certain HTTP response status codes as failures.

7.2.6. Invoking failing APIs less often with Circuit Breakers

An API implementation that fails, maybe due to a failing downstream system, may need time to recover. This recovery process may actually be slowed-down by insistent API clients that continue invoking the API exposed by that API implementation - only to receive failures in

return.

A Circuit Breaker

1. monitors API invocations
2. after a certain number (or percentage) of failed invocations of an API "opens the circuit"
3. while in the "open" state immediately fails invocations of that API to the API client without even invoking the API itself, thereby giving the underlying API implementation time to heal
4. after a certain time gently begins passing API invocations through to the API ("half-opening the circuit")
   a. immediately transitioning back to the "open" state upon an API invocation failure, because this indicates that the API implementation is not yet ready to handle API invocations
   b. while "closing the circuit" if API invocations are sufficiently successful, because this indicates that the API implementation has recovered.

Seen from an API client, the most important contribution of a Circuit Breaker is that, when it is in the "open" state, it saves the API client the wasted time and effort of invoking a failing API. Instead, thanks to the Circuit Breaker, these invocations fail immediately and the API client can quickly move on to apply fallback strategies as discussed in the remainder of this section.

A Circuit Breaker is by definition a *stateful component*, i.e., it monitors and manages API invocations over "all" API clients. The scope of "all" may vary:

- In the easiest case, all API invocations from within a single instance of a type of API client are managed
  - E.g., all API invocations from the API client executing in a single CloudHub worker are managed together
  - This means that the different instances of the API client executing in different CloudHub workers keep distinct statistics and state for the invocations of the failing API
- Alternatively, all API invocations from all instances of a type of API client are managed together
  - E.g., all API invocations from all instances of the API client executing in all CloudHub workers are managed together
  - This means that the different instances of the API client executing in different CloudHub workers all share the same statistics and state for the invocations of the failing API
  - Requires remote communication between instances of the API client, e.g., via a Mule runtime Object Store
- In the extreme case, all API invocations from all types of API clients are managed together

- E.g., all API invocations from all instances of any API client executing in any CloudHub worker are managed together
- This means that the different instances of any API client executing in any CloudHub workers all share the same statistics and state for the invocations of the failing API
- Typically requires the Circuit Breaker to be an application network-wide shared resource, e.g., an API in its own right

API clients implemented as Mule applications and executing on a Mule runtime have open-source implementations of the Circuit Breaker pattern at their disposal for the configuration of API invocations.

7.2.7. Invoking a fallback API

When an API invocation fails, even after a reasonable number of retries, it may be possible to invoke a different API as a fallback.

For instance, when the API implementation of the "Policy Holder Search PAPI" has definitely failed invoking the "Motor Policy Holder Search SAPI", it may instead invoke a fallback API that is sufficient, if not ideal, for its purposes. (A fallback API by definition is never ideal for the purposes of the API client at hand - otherwise it would be its primary API.)

This fallback API

- could be an old, deprecated version of the same API, that may however still be available; e.g., an old but sufficiently compatible version of the "Motor Policy Holder Search SAPI"
- could be an alternative endpoint of the same API and version; e.g., an endpoint of the "Motor Policy Holder Search SAPI" from the DR site or a different (possibly distant) CloudHub region
- could be an API that does more than is required, and is therefore not as performant as the primary API; e.g., the "Motor Policy Search SAPI" may provide the option to search the Policy Admin System for Motor Policies by the name of the policy holder, similar to the input received by the "Motor Policy Holder Search SAPI", returning entire Motor Policies, from which the motor policy holders actually needed by the "Policy Holder Search PAPI" (the API client of the "Motor Policy Holder Search SAPI") can be extracted
- could be an API that does less than is required, therefore forcing the API client into offering a degraded service - which is still better than no service at all

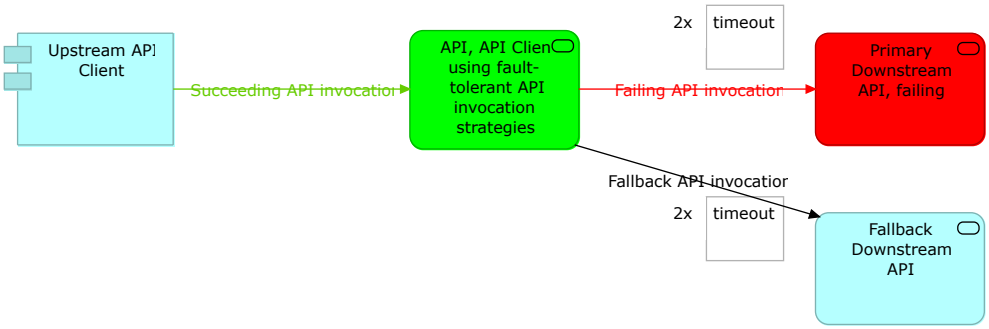

Figure 98. After retries on the primary API have been exhausted, at least one fallback API should be invoked, even if that API is not a full substitute for the primary API.

API clients implemented as Mule applications and executing on a Mule runtime have the Until Successful Scope and exception strategies at their disposal, which together allow for configuring fallback actions such as the fallback API invocations.

7.2.8. Opportunistically invoking APIs in parallel

If

- the possibility of a failure in an API invocation is considered generally high
- and/or the API client performing the API invocation is particularly important
- and a fallback API as discussed previously is available

then the invocation of the primary API and the fallback API may be performed in parallel. If the primary API does not respond in time (i.e., a short time-out should be used) and the fallback API has delivered a result, then the result from the fallback API invocation is used. Overall then, little time has been lost because the two API invocation were performed in parallel rather than serially.

This is an *opportunistic, egotistical strategy* that puts increased load on the application network by essentially doubling the number of API invocations for the cases where this strategy is used. As thus it should be used *only in exceptional cases* as indicated above.

7.2.9. Using a previously cached result as a fallback

After retries of an API invocation are exhausted and a fallback API has also failed or is not available, a result from a previous API invocation may be used instead. In other words, the API client needs to implement some form of *client-side caching*, where the results of API invocations are preserved indexed by the input to those API invocations. An API invocation result can then be looked up in the cache by the input to the API invocation that has just failed.

Client-side caching is governed by the same rules as general caching in an HTTP setting - it's just performed within the API client rather than by a API implementation or an intervening network component.

In particular, only results from safe HTTP methods (6.4.6) should be cached:

- GET
- HEAD
- OPTIONS

and HTTP caching semantics should be honored (6.4.7). Although, in practice, it would typically still be preferable to use an outdated/stale cached HTTP response, thereby effectively ignoring HTTP caching semantics and control headers, than to not recover from the failed API invocation.

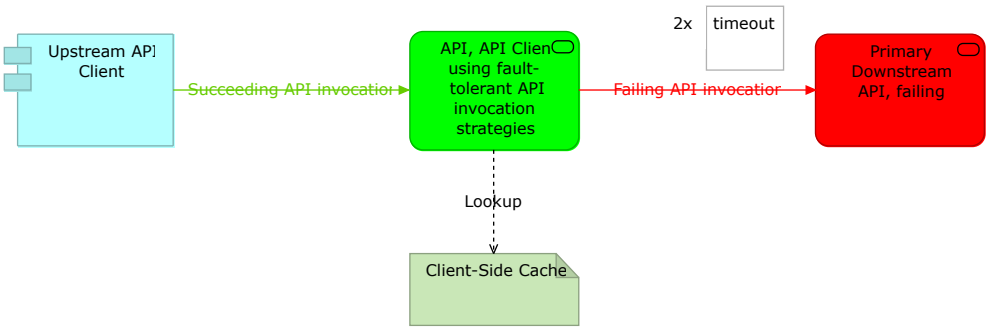

Figure 99. Using an API invocation result cached in the API client as a fallback for failing API invocations.

For instance,

- if the API implementation of the "Policy Options Ranking PAPI" performs client-side caching of its invocations of the "Policy Options Retrieval SAPI"
- and API invocations of the "Policy Options Retrieval SAPI" currently fail
- then the "Policy Options Ranking PAPI" API implementation may use the cached response from a previous, matching request to the "Policy Options Retrieval SAPI"

Caching increases the memory footprint of the API client, e.g. the API implementation of the "Policy Options Ranking PAPI". It also adds processing overhead for populating the cache after every successful API invocation. There is furthermore the question of the scope of caching, similar to the state handling discussed in 7.2.6.

API clients implemented as Mule applications and executing on a Mule runtime have the Cache Scope and Object Store Connector available, which both support client-side caching.

7.2.10. Using a static fallback result

After all previously discussed options for recovering from a failing API invocation have failed, too, a prepared static result may be used instead of the result expected from the API invocation.

Typically this works best for APIs that return results akin to reference data, e.g.,

- countries
- states
- currencies
- products

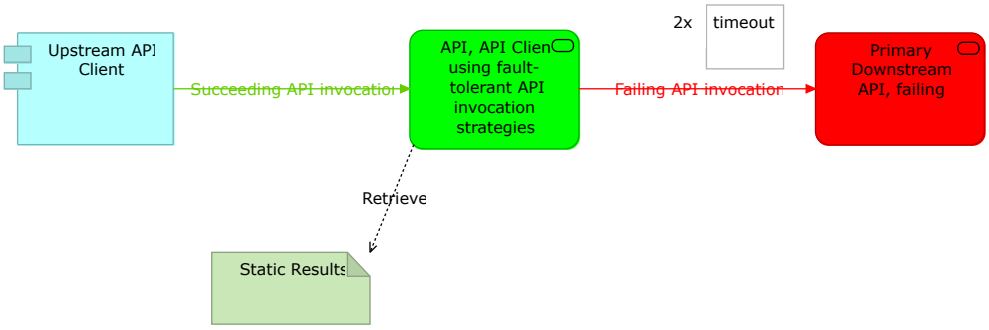

Figure 100. Using statically configured API invocation results as fallback for failing API invocations.

For instance, when the API implementation of the "Policy Options Ranking PAPI" fails to invoke the "Policy Options Retrieval SAPI", it may be able to work with a list of common policy options loaded from a configuration file. These policy options may be limited, and may not be ideal for creating the best possible quote for the customer, but it may be better than not creating a quote at all. Again, the principle is to prefer a degraded service to no service at all.

API clients implemented as Mule applications and executing on a Mule runtime have many options available for storing and loading static results. One such option is to use properties, which, when the Mule application is deployed to CloudHub, can actually be updated via the Anypoint Runtime Manager web UI.

7.3. Understanding architecturally significant persistence choices for API implementations

7.3.1. Introducing CQRS as an API implementation strategy

See the corresponding glossary entry.

Characteristics of CQRS:

- Allows *reads and writes* to be *optimized and scaled independently*
- Allows the choice of *independent persistence mechanisms* for reads and writes
- *Commands* are formulated in the domain language of the Bounded Context and *trigger writes*
- *Commands* are typically queued and executed *asynchronously*
- *Queries* are optimized for the API clients' exact needs (joining and aggregating data as needed) and execute *synchronously*
- Because *queries* may operate against a distinct data store they *may return slightly out-of-date data*
- Complicates the architecture, design and implementation of an API implementation and filters through to the API itself

CQRS is a design choice to be *made independently for each API implementation*, but it is *architecturally significant* because

- it causes *eventual consistency* between read-sides and write-sides
- it is typically *apparent in the API specification* of the API exposed by the API implementation
  - It may also cause an API to be split into one API for queries and another API for commands

The option to select persistence mechanisms independently for the read-side and write-side of an API implementation may manifest itself in choosing entirely different persistence technologies - e.g., a denormalized NoSQL database for the read-side and a fully normalized relational database schema for the write-side. But it may equally just mean storing read-side and write-side data in different table spaces in the same RDBMS or even just different tables in the same table space - potentially with less normalization and fewer database constraints on the read-side.

7.3.2. Viewing Acme Insurance's application network from a CQRS perspective

At Acme Insurance a separation between commands and queries has emerged naturally:

- The "Motor Claims Submission PAPI" and "Motor Claims Submission SAPI" accept commands in the form of claim submissions, which are executed asynchronously
- The "Claims PAPI" and "Motor Claims Search SAPI" provide synchronous queries against claims, including ones that result from previous claim submissions
- The persistence performed by the read-side and write-side, in this case, are both hidden inside the Motor Claims System, which may, but typically will not, use different persistence mechanisms for the two cases
  - But on the level of the APIs there is clear CQRS-style separation
  - This is a typical situation in integration solutions where the imperative of reusing existing systems results in technologically and stylistically more muddled architectures than would be the case in greenfield application development

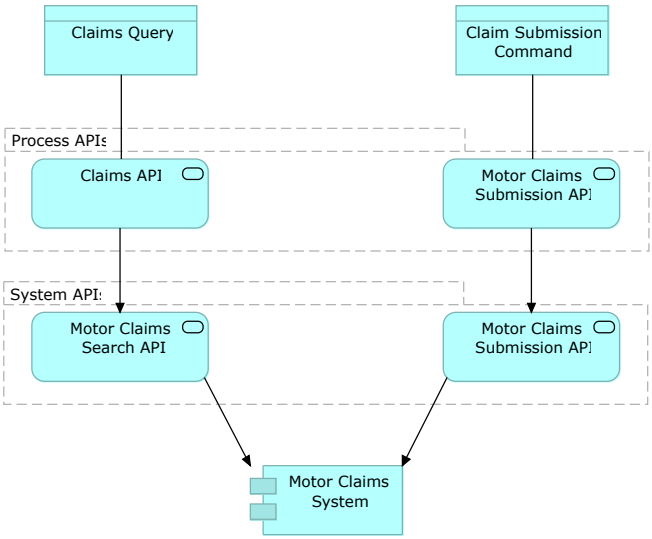

Figure 101. CQRS emerging naturally on the API-level for motor claims, through the write-side via "Motor Claims Submission PAPI" and "Motor Claims Submission SAPI" (shown on the right) accepting asynchronous commands and the read-side via "Claims PAPI" and "Motor Claims Search SAPI" (shown on the left) accepting synchronous queries - although persistence is encapsulated in the shared Motor Claims System.

7.3.3. Introducing persistence with Event Sourcing

See the corresponding glossary entry.

Event Sourcing is *very similar* in spirit *to database transaction logs*. The important difference is that Event Sourcing is an approach in the application layer, chosen by application components and API implementations, rather than hidden behind the technology service offered by a RDBMS.

Characteristics of Event Sourcing:

- Events are expressed in the domain language of the Bounded Context
- Events typically arise directly from executing CQRS-style commands
- If CQRS is used then events are typically used to propagate state changes from the write-side to the read-side
  - This often relies on messaging systems like Anypoint MQ to propagate events asynchronously
- A typical performance optimization is to accumulate events into a snapshot state at regular intervals

Unlike CQRS, Event Sourcing by itself is invisible in the API specification of the API exposed by an API implementation. It is therefore an implementation-level design decision and outside the scope of this discussion.

Summary

- API implementations can target the Mule runtime or other runtimes while being managed by the Anypoint Platform control plane
- API implementations implemented as Mule applications can be automatically discovered by Anypoint Platform
- Anypoint Platform has over 120 Anypoint Connectors, which are indispensable for implementing System APIs
- CloudHub is an AWS-based iPaaS for the scalable, performant and highly-available deployment of Mule applications
- Object Store is a key-value persistence service available in all Mule runtime deployment scenarios
- In CloudHub, Object Store allows time-limited persistence local to the AWS region of the Mule runtime
- API clients, in particular those which are in turn API implementations, must employ strategies to guard against failures in API invocations: retry, Circuit Breaker, fallbacks, etc.
- Some API implementations may benefit from using CQRS as a persistence strategy, which in turn influences the design of their API
- Separation of commands and queries often arises naturally in API design, even in the absence of true CQRS
- Event Sourcing is an implementation-level design decision of each API implementation

Module 8. Augmenting API-Led Connectivity With Elements From Event-Driven Architecture

Objectives

- Selectively choose elements of Event-Driven Architecture in addition to API-led connectivity
- Make effective use of events and message destinations
- Impose event exchange patterns in accordance with API-led connectivity
- Describe Anypoint MQ and its features
- Apply Event-Driven Architecture with Anypoint MQ to address NFRs of the "Customer Self-Service App" product

8.1. Choosing Event-Driven Architecture to meet some NFRs of the "Customer Self-Service App" product

8.1.1. Revisiting the NFRs for the "Customer Self-Service App" product

The "Customer Self-Service App" product requires that claim submissions made from the Customer Self-Service Mobile App shall be visible immediately through that app, even though the Motor Claims System does not give access to newly submitted claims until they have passed a lengthy processing phase. In other words, claim submissions made via the "Submit auto claim" feature should be reflected sooner via the "Retrieve policy holder summary" feature than the Motor Claims System gives access to them.

See 5.2.1, 5.2.3 and 5.2.4.

8.1.2. Exchanging events between API implementations

The consistency requirement for the "Customer Self-Service App" product leads to the conclusion that the command-query paths shown in Figure 101 must be "short-circuited" such that also very recent claim submissions, which are not yet exposed by the Motor Claims System, can be retrieved via the "Claims PAPI" and "Motor Claims Search SAPI".

One way of architecting this "short-circuit" is through an approach not unlike event sourcing on top of CQRS (Figure 102):

1. The "Motor Claims Submission SAPI", after transmitting a claim submission to the Motor Claims System, must also publish a "Motor Claim Submitted" event.

2. The "Motor Claims Search SAPI", in addition to retrieving claims from the Motor Claims System, must also consume "Motor Claim Submitted" events, store them and include them in the search results it returns to its API clients, specifically the "Claims PAPI".

This amounts to a non-API communication channel between the "Motor Claims Submission SAPI" and the "Motor Claims Search SAPI" that follows the general architectural principles of *Event-Driven Architecture*.

You will improve on the solution shown in Figure 102 in 8.2.6.

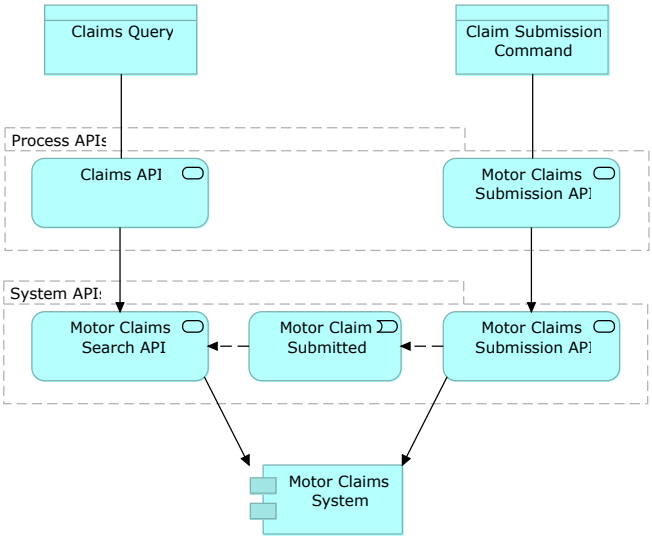

Figure 102. To make recent claim submissions, which are not yet exposed by the Motor Claims System, available to the "Motor Claims Search SAPI", the latter consumes "Motor Claim Submitted" events published by the "Motor Claims Submission SAPI". This shown solution has several drawbacks which will be corrected in the following.

8.2. Understanding the nature of Event-Driven Architecture in the context of API-led connectivity

8.2.1. Defining Event-Driven Architecture

See the corresponding glossary entry.

8.2.2. Exercise 15: Differences between API-led connectivity and Event-Driven Architecture

Drawing on your experience and the preceding discussions:

1. List differences and similarities between events and APIs
2. List differences and similarities between Event-Driven Architecture and API-led connectivity

In your comparison consider the following aspects and concepts:

- Components and nodes involved in the communication between application components
- Static and dynamic dependencies between the communicating application components
- Meaning of such a "communication"
- contracts and data types

Solution

See 8.2.3 and 8.2.4.

8.2.3. Comparing events and APIs

Events as used in Event-Driven Architecture and APIs as used in API-led connectivity can be compared and contrasted along the following dimensions:

- *Programmatic*: Both are primarily *programmatic* communication approaches in nature, i.e., intended for communication between application components.
- *Meaning*: An event is (or captures/describes) a *state change* that has occurred in the application component that acts as the event producer, whereas an API is a programmatic interface to a service provided by the application component exposing that API (the target of the API invocation).
- *Dynamic nature*: An event is therefore more similar to an API invocation than to an API, i.e., it is a *concrete instance of the communication* between application components. Even then, the API invocation, for instance for a REST API, is a combination of resource and HTTP method and therefore typically denotes an *action to be performed* upon receipt of the API invocation, whereas an event describes what has already happened.
- *Static nature*: Events denoting state changes of the same kind are said to be of the same *event type*. This is the equivalent of the API and specifically the API data model used in the API specification.
- *Granularity*: An event/event type is typically for one well-defined state change, whereas an API can expose many resources and support many HTTP methods per resource. Thus an API is more akin to a group of event types than a single event type.
- *Synchronicity*: API invocations are by definition synchronous, consisting of request and synchronous response (even though the *processing* triggered by an API invocation can be performed asynchronously, as discussed for instance in 6.4.1). Thus if API client and API implementation are not both available throughout the duration of the API invocation then

that API invocation fails. Events by contrast are exchanged asynchronously (as messages) and therefore event producer and consumer are decoupled in time.

- *Communication path*: An API invocation is from one API client to one API implementation, and the API client specifically addresses the API implementation. (API proxies are a stand-in for the API implementation, must be explicitly targeted by the API client, and therefore do not change the nature of this argument.) Events are sent by an event producer to a destination (queue, topic, message exchange; depending on messaging paradigm) and are then received by event consumers from that destination - without the producer being aware of the consumers or the consumers being aware of the producer. Furthermore, there can be more than one consumer for each event.
- *Broker*: Exchanging events requires a message broker, such as Anypoint MQ, as the owner of destinations, whereas API invocations, at a minimum, only require API client and API implementation.
- *Contract*: The contract for an API is its API specification, typically a RAML definition. The contract for an event is the combination of destination and event (data) type and is not typically captured formally.

8.2.4. Comparing Event-Driven Architecture and API-led connectivity

API-led connectivity defines the *three tiers* of Experience APIs, Process APIs and System APIs. Although application components exchanging events can be organized similarly, this is not an inherent part of Event-Driven Architecture.

API-led connectivity restricts *communication patterns* according to these three tiers (essentially top to bottom), whereas application components exchanging events do not a-priori have to adhere to the same communication pattern restrictions.

API implementations typically have *well-defined static dependencies* on other APIs and/or backend systems (for example: Figure 39). While similar relationships may materialize in Event-Driven Architecture at *runtime*, there are *no static dependencies* between the application components exchanging events. Instead, these application components only depend on the exchanged event types, the destinations and the message broker hosting those destinations. Furthermore, event consumers may change dynamically at any time, thereby dynamically *reconfiguring the relationship graph* of the application components in an Event-Driven Architecture, without the event producers becoming aware of that change.

Event-Driven Architecture requires a *message broker* as an additional component of the Technology Architecture, with all application components who want to exchange events having to agree on the same message broker (this is not a strict requirement in some broker architectures).

API-led connectivity and in particular application networks are defined by the API-centric assets published for self-service consumption. The equivalent for Event-Driven Architecture would revolve around destination and event types.

Enforcing NFRs by applying API policies in Anypoint API Manager on top of existing API implementations has no equivalent in Event-Driven Architecture on Anypoint Platform.

8.2.5. Event exchange patterns in API-led connectivity

API-led connectivity imposes the architectural constraint that Experience APIs must only invoke Process APIs, Process APIs must only invoke System APIs or other Process APIs, and System APIs must only communicate with backend systems. This constraint brings order and predictability to the communication patterns in an application network.

When Event-Driven Architecture is applied in the context of API-led connectivity then the application components exchanging events are predominantly API implementations.

Event-Driven Architecture by itself is agnostic about the three tiers of API-led connectivity and hence does not restrict event exchanges between API implementations in different tiers. But breaking the communication patterns of API-led connectivity through arbitrary, unrestricted event exchanges risks destroying the order and structure created in an application network by the application of API-led connectivity.

Importantly, API-led connectivity is an API-first approach, which does not rule-out the exchange of events, but views it as an addition to API invocations as the dominant form of communication between application components. It is therefore advisable to *require API implementations that exchange events to follow communication patterns in accordance with API-led connectivity* as follows:

- Any API implementation that publishes events should define its own destinations (queues, message exchanges) to send these events to. Often, there will be one destination per event type published by that API implementation.
- In this way destinations belong logically to the same API-led connectivity tier as the API implementation publishing events to them.
  - I.e., a System API publishes events to destinations that logically belong to the System API tier and can hence be described as "system events"
  - I.e., a Process API publishes events to destinations that logically belong to the Process API tier and can hence be described as "process events"
  - I.e., an Experience API publishes events to destinations that logically belong to the Experience API tier and can hence be described as "experience events"

- Any API implementation that consumes events must not do so from a destination that belongs to a higher tier than the consuming API implementation itself. In other words, *events must not flow downwards* across the tiers of API-led connectivity:
  - Events published by Experience APIs to their destinations ("experience events") must not be consumed from those destinations by Process APIs or System APIs.
  - Events published by Process APIs to their destinations ("process events") must not be consumed from those destinations by System APIs.
  - Put differently: *Events may only be consumed within the same tier or in higher tiers relative to the API implementation that publishes the events*.
  - The logic for this rule is the same as for the communication patterns underlying API-led connectivity: the rate of change of Experience APIs (which are relatively volatile) is higher than the rate of change of Process APIs which is higher than the rate of change of System APIs (which are comparatively stable). *And a slow-changing component must never depend on a fast-changing component.*
- In addition, in analogy with API-led connectivity, it should be prohibited that Experience APIs directly consume events published by System APIs, thereby *bypassing Process APIs*.

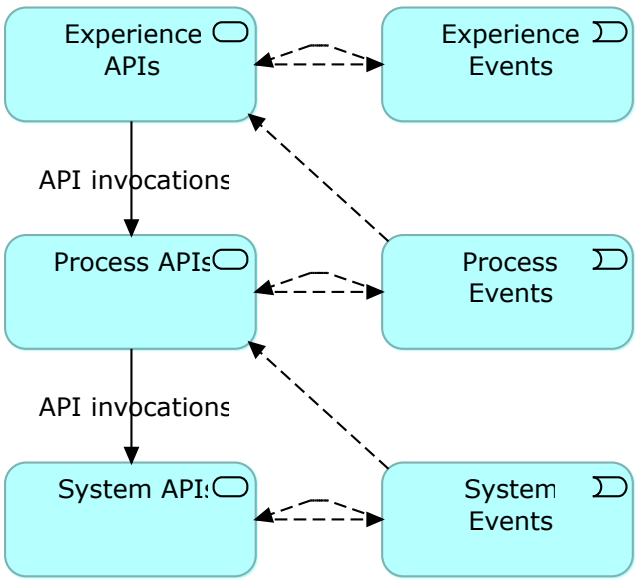

Figure 103. Flow of events and API invocations in API-led connectivity augmented with elements from Event-Driven Architecture (permitted API invocations between Process APIs are not shown).

8.2.6. Exercise 16: Event-Driven Architecture on top of API-led connectivity for "Customer Self-Service App" product

The solution shown in Figure 102 assumed that "Motor Claim Submitted" events are published

by a System API and consumed by another System API. The same consistency requirement could also have been realized by other APIs acting as event producers and consumers, for instance:

- Events are published by "Motor Claims Submission SAPI" and consumed by "Motor Claims Search SAPI" (as shown in Figure 102)
- Events are published by "Motor Claims Submission SAPI" and consumed by "Claims PAPI"
- Events are published by "Motor Claims Submission PAPI" and consumed by "Motor Claims Search SAPI"
- Events are published by "Motor Claims Submission PAPI" and consumed by "Claims PAPI"

1. Discuss and decide on the precise meaning of "Motor Claim Submitted" events
2. Decide which API should best publish an event with this meaning
3. Assess the implementation effort of publishing and consuming events in this scenario
4. Redesign the solution shown in Figure 102 to honor the Single Responsibility Principle (SRP) and event exchange patterns compatible with API-led connectivity

Solution

See 8.2.7.

8.2.7. Separating concerns when exchanging events between API implementations

- If "Motor Claim Submitted" captures the historical fact that a claim submission has been (successfully) passed to the Motor Claims System then the publishing of that event should occur as close to the Motor Claims System as possible, i.e., in the "Motor Claims Search SAPI"
- If the API implementation of a System API publishes an event then it should only be consumed by the API implementation of a System API or Process API (8.2.5)
- The publishing and consumption of events in a System API adds responsibilities to that System API that are unrelated to backend system connectivity (thereby violating the Single Responsibility Principle). This "muddling" of responsibilities is trivial in the case of event publishing but significant in the case of event consumption and storage. It should therefore be avoided that System APIs also consume and store events in addition to performing "normal" backend connectivity
- An architecturally clean solution therefore mandates a *new System API* that consumes and stores "Motor Claim Submitted" events - the "Submitted Motor Claims Search SAPI" - and requires the existing "Claims PAPI" to coordinate between the two types of System APIs that search against motor claims ("Motor Claims Search SAPI" and "Submitted Motor Claims

Search SAPI"). This is captured in Figure 104, which hence improves on and replaces Figure 102

- This is another example of the claim made in 2.2.13 that application networks "bend but do not break": the consistency requirement for the "Customer Self-Service App" product was added to an existing application network by changig the API implementations but not the API specifications of a System API ("Motor Claims Submission SAPI") and a Process API ("Claims PAPI"), and adding a System API ("Submitted Motor Claims Search SAPI")

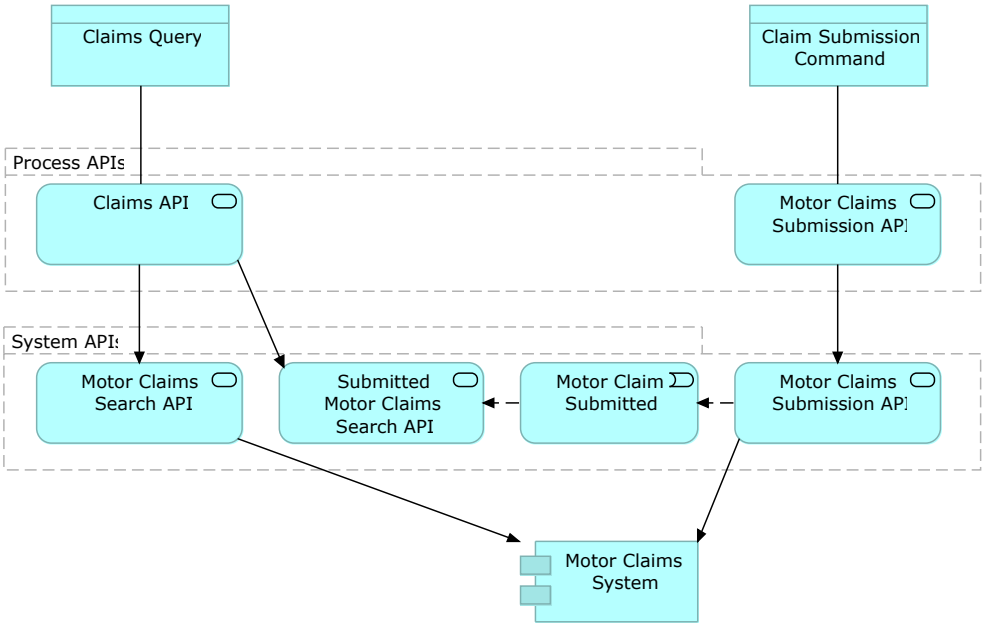

Figure 104. Architecturally clean separation of concerns between "Motor Claims Search SAPI" for accessing the Motor Claims System and the new "Submitted Motor Claims Search SAPI" for consuming "Motor Claim Submitted" events published by the "Motor Claims Submission SAPI". The event store used by "Submitted Motor Claims Search SAPI" to persist and search "Motor Claim Submitted" events is not shown.

8.3. Adding support for asynchronous messaging to the Technology Architecture

8.3.1. Introducing Anypoint MQ

Anypoint MQ

- is Anypoint Platform's *multi-tenant cloud-based* (hosted) *messaging service*
- that is only available in the MuleSoft-hosted Anypoint Platform

- offers *role-based access-control* in line with the rest of Anypoint Platform
- provides token-based *client access control*
- implements an async messaging model using *queues*, *message exchanges* and bindings between them
  - these must reside in one of the AWS regions supported for the Anypoint Platform runtime plane
- supports: *point-to-point*, *pub/sub*, *FIFO queues*, *payload encryption*, *persistent/durable messages*, *DLQs*, message *TTL*
- exposes a *REST API* and an *Anypoint Connector* for the Mule runtime on top of that
  - This means that message producers and consumers can be located anywhere, as long as they can make API invocations to the MuleSoft-hosted Anypoint MQ broker.
- has a *web-based management console* in the Anypoint Platform web UI

In Anypoint MQ, messages can be sent to queues or message exchanges and consumed only from queues. Message exchanges must therefore be (statically) configured to distribute the messages sent to them to one or more queues in order for those messages to be consumable.

8.3.2. Event-Driven Architecture with Anypoint MQ for the "Customer Self-Service App" product

Figure 105 visualizes relevant interactions for an implementation of Figure 104 using Anypoint MQ.

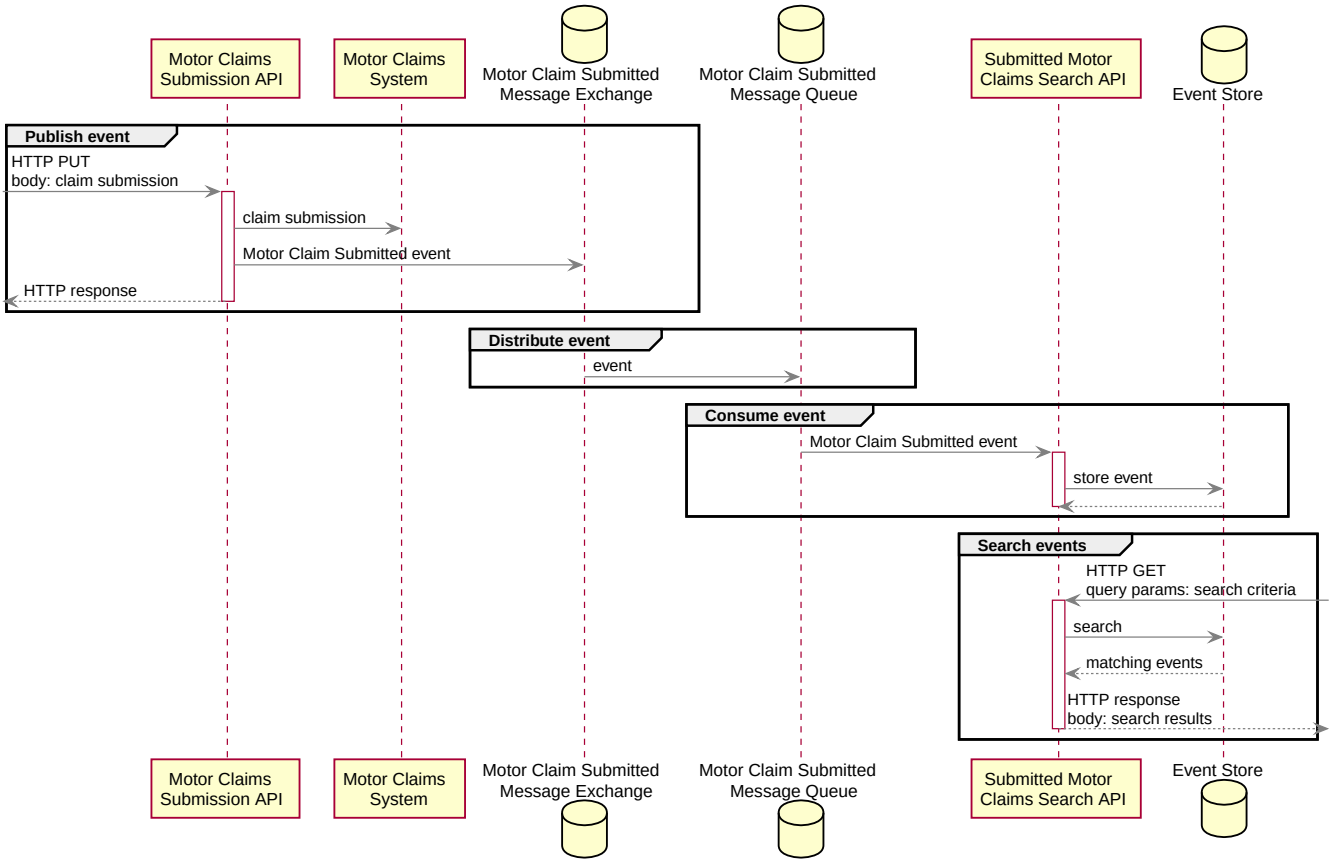

Figure 105. "Motor Claims Submission SAPI" produces "Motor Claim Submitted" events by publishing them to an Anypoint MQ message exchange, from where they are consumed by "Submitted Motor Claims Search SAPI" and stored in an event store (an external database or potentially a Mule runtime Object Store), such that when a search request arrives at "Submitted Motor Claims Search SAPI" it can respond by searching that event store for matching "Motor Claim Submitted" events.

Summary

- Some NFRs are most naturally realized by adding elements from Event-Driven Architecture to a solution based on API-led connectivity
- Events describe historical facts and are exchanged asynchronously between application components via destinations such as message exchanges
- Event exchange patterns in an application network should follow the rules established by API-led connectivity
- Anypoint MQ is a MuleSoft-hosted multi-tenant cloud-native messaging service that can be used to implement Event-Driven Architecture and other forms of message-based integration
- The consistency requirement of the "Customer Self-Service App" product can be realized by introducing a new System API that consumes events published by the "Motor Claims

Submission SAPI" without changing existing APIs

Module 9. Transitioning Into Production

Objectives

- Locate API-related activities on a development lifecycle
- Interpret DevOps using Anypoint Platform tools and features
- Design automated tests from the viewpoint of API-led connectivity and the application network
- Identify the factors involved in scaling API performance
- Use deprecation and deletion of API versions in Anypoint Platform
- Identify single points of failure

9.1. Understanding the development lifecycle and DevOps

9.1.1. Keeping the development lifecycle in perspective

MuleSoft's proposed lifecycle for developing API-led connectivity integration solutions, depicted in Figure 106, is API-first and therefore

- starts with API-centric activities that lead to the creation of an API specification
- then proceeds to building the API implementation
- before transitioning both the API as well as the API implementation into production

Each of these three phases is supported by Anypoint Platform components, as shown in Figure 106.

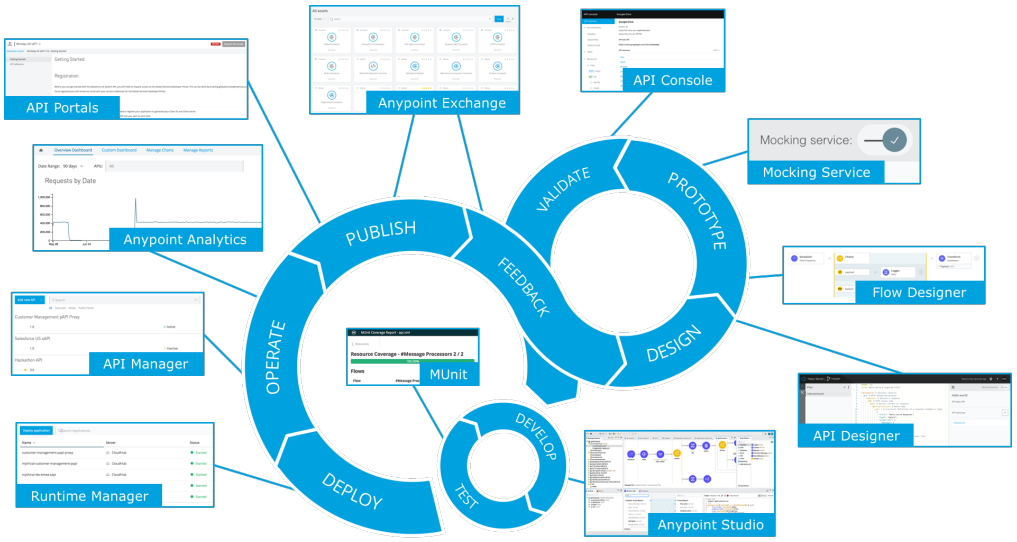

Figure 106. The API-centric development lifecycle and Anypoint Platform components supporting it.

An alternative, more process-centric depiction of the API development lifecycle is shown in Figure 107.

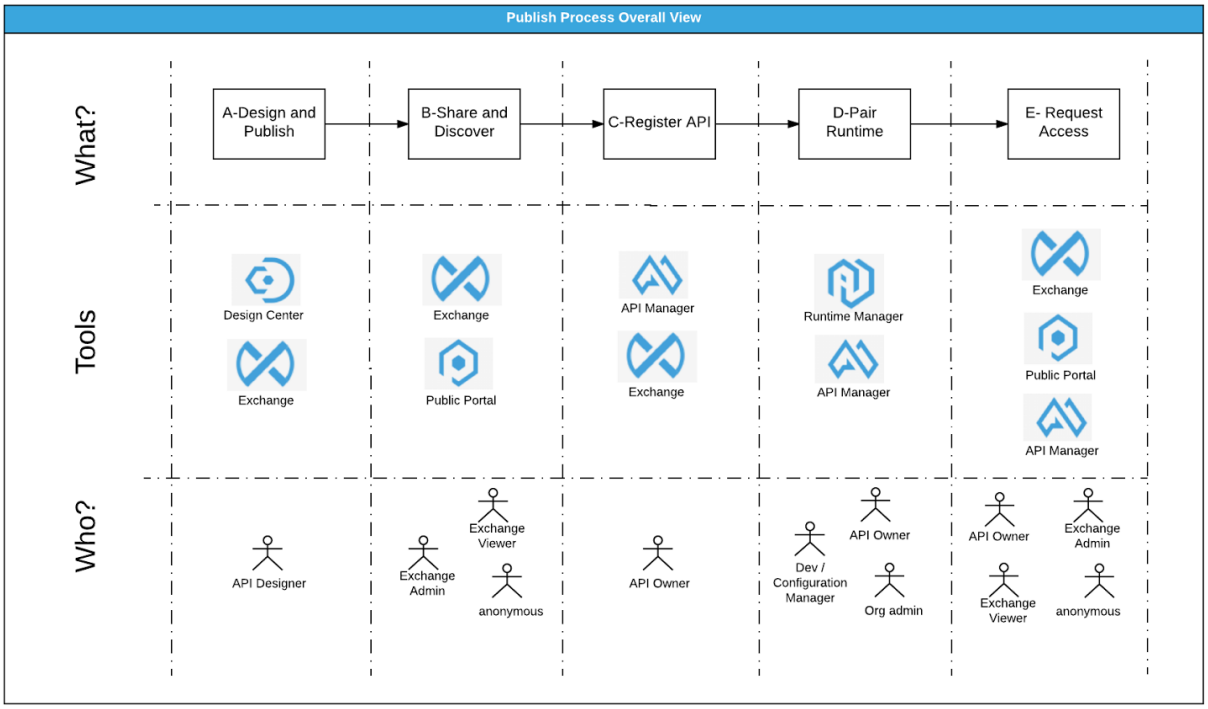

Figure 107. The API-centric development process.

9.1.2. Building on a strong DevOps foundation

For the health of the application network it is paramount that the process by which API implementations are created and deployed to production is reliable, reproducible and as much as possible automated. This is because defects or failures in business-critical API implementations have the potential of affecting the application network. Therefore, strong DevOps capabilities and a rigorous development and deployment process are an absolute necessity.

Acme Insurance, with strong guidance from the C4E, standardizes on the following *DevOps principles*:

- All RAML definitions and RAML fragments must live in an *artifact repository*, specifically in *Anypoint Exchange*
- All source code for API implementations must live in a *source code repository*
  - Acme Insurance chooses Git and *GitHub*, with every API implementation having its own GitHub repo

The chosen development and *DevOps process* is depicted at a high level in Figure 108 and described in more detail the following:

1. Developers of an API implementation implement on *short-lived feature branches* off the Git `develop` branch of that API implementation's repo
   a. This is the GitFlow branching model (Figure 109)
2. Developers *implement* the Mule application and all types of automated tests (unit, integration, performance) and make them pass
   a. Acme Insurance chooses a combination of *Anypoint Studio, JUnit, MUnit, SOAPUI and JMeter*
   b. For build automation Acme Insurance chooses *Maven* and suitable plugins such as the *Mule Maven plugin*, the *MUnit Maven plugin* and those for SOAPUI and JMeter
3. Once done, developers submit *GitHub pull requests* from their feature branch into the `develop` branch
4. A developer from the same team performs a *complete code review* of the pull request, confirms that *all tests pass*, and, if satisfied, merges the pull request into the `develop` branch of the API implementation's GitHub repo
5. This triggers the *CI pipeline*:
   a. Acme Insurance chooses Jenkins, delegating to Maven builds to implement CI/CD
   b. The Mule application is compiled, packaged and all unit tests are run against an embedded Mule runtime

   c. The Mule application is deployed to an artifact repository
      i. Acme Insurance chooses a private *Nexus* installation
6. When sufficient features have accumulated for the API implementation, the release manager for that API implementation *"cuts a release"* by tagging, creating a release branch and ultimately merging into the `master` branch of the API implementation's repo
7. This triggers the *CI/CD pipeline* in Jenkins:
   a. The CI pipeline is executed as before, leading to deployment of the Mule application into Nexus
   b. Automatically, and/or through a manual trigger, the CD pipeline is executed:
      i. A well-defined version of the Mule application from the artifact repository is deployed into a staging environment
      ii. Integration tests and performance tests are run over HTTP/S against the API exposed by the API implementation
      iii. Upon success the Mule application from the artifact repository is deployed into the production environment
      iv. The "deployment verification sub-set" of the functional integration tests is run to verify the success of the deployment
         A. Failure leads to immediate rollback via the execution of the CD pipeline with the last good version of the API implementation

Anypoint Platform has no direct support for "canary deployments", i.e. the practice of initially only directing a small portion of production traffic to the newly deployed API implementation.

The above discussion assumes that a previous version of the API implementation in question has already been developed, and that, therefore, the Anypoint API Manager and API policy configuration for the API instance exposed by the API implementation is already in place. Creation of this configuration can be automated with the Anypoint Platform APIs.

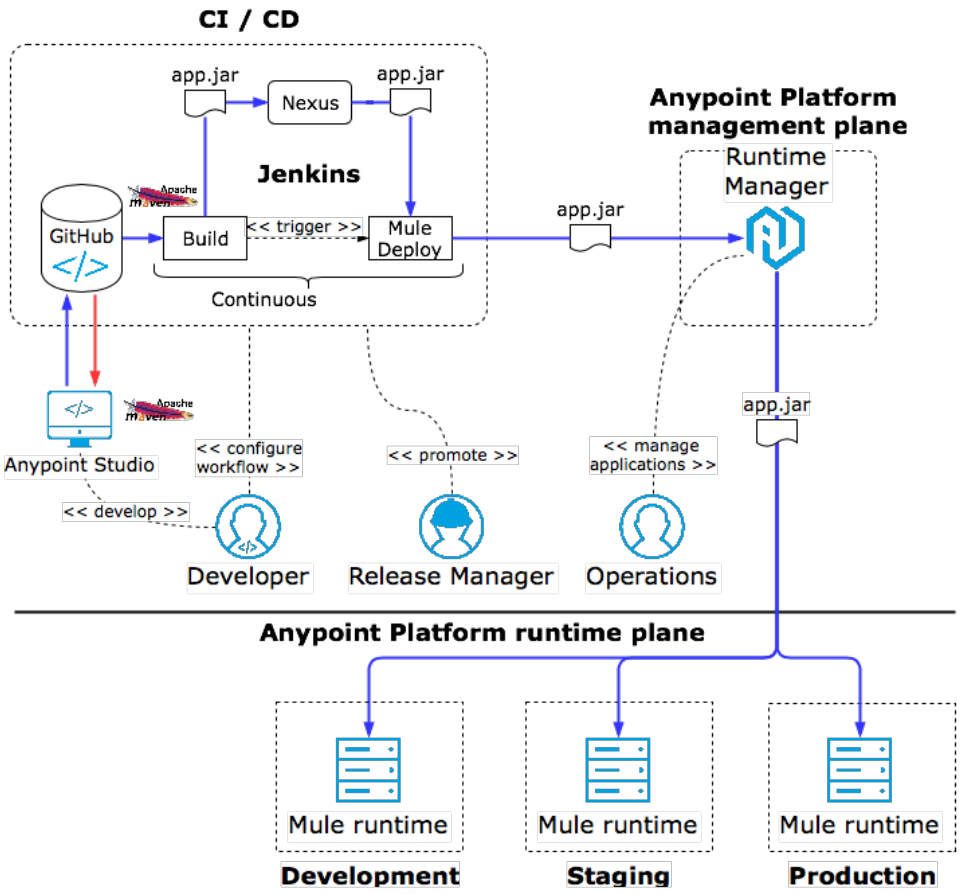

Figure 108. High-level DevOps process chosen by Acme Insurance.

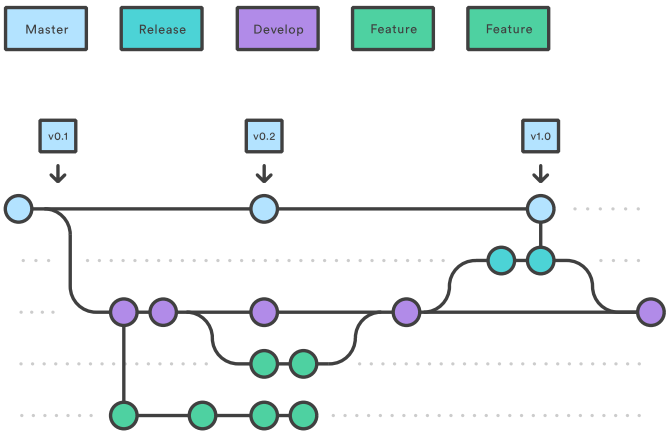

Figure 109. The GitFlow branching model showing the `master` and `develop` branch as well as release and feature branches. Source: Atlassian.

9.1.3. Promoting APIs and API implementations to higher environments

As a variant of DevOps-style deployment of API implementations, API instances, API policies, alerts, etc. to all environments (Staging, Production, etc.), Anypoint Platform supports the *promotion* of the Anypoint API Manager definition for an API instance from one environment (e.g., Staging) to another (e.g., Production), i.e., *between environments*.

Promoting all supported parts of the Anypoint API Manager entry for an API instance does *not* copy API clients (client applications) to the target environment. This means that promoted API instances start off without any registered API clients. It also means that API consumers must separately request access for their API clients to API instances in all environments - even if the exact same API and version is requested access to.

The client ID/secret a given API client uses for accessing an API is, however, independent of the environment. That is, if the same API client has been granted access to an API instance in the Sandbox and Production environments, it must make API invocations to both API endpoints with the same client ID/secret.

This is not typical usage, though. Instead, it would be typical for an API client in the Sandbox environment to access an API instance from the Sandbox environment, and a separate API client in the Production environment (even if it represents the same application component) to access an API instance from the Production environment. In this case, the two API clients would use distinct client ID/secret pairs, which is preferable for governance reasons.

After the promotion of an API instance in Anypoint API Manager, *the newly created/promoted API instance shares the "Implementation URL" and "Consumer endpoint" with the original API instance*: this is not realistic and *must typically be changed after promotion*, such that, e.g., the Production API instance uses Production URLs.

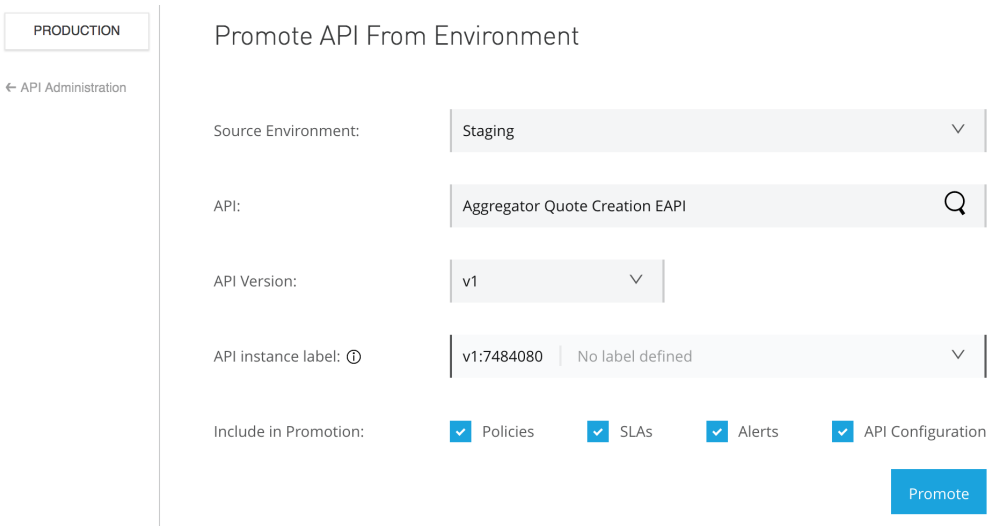

Figure 110. Promotion of the entire Anypoint API Manager configuration for a particular API instance of the "Aggregator Quote Creation EAPI" from the Staging to the Production environment.

Anypoint Runtime Manager supports a similar promotion of Mule applications, such as API implementations, between environments. As always, DNS names of Mule applications must differ between environments.

Promotion of Anypoint API Manager definitions and/or matching API implementations can of course be automated by Anypoint Platform APIs and/or the Anypoint CLI (2.4.2). It can then be integrated into a variant of the DevOps approach sketched in 9.1.2.

9.2. Designing automated tests for APIs, API implementations and the application network

9.2.1. Understanding API-centric automated testing

The practice of testing does not change fundamentally when API-led connectivity is employed and application networks are grown. All the same types of tests and activities of preparing for, executing and reporting on tests are still applicable. Therefore this discussion addresses only a few selected topics that are of special interest in the context of API-led connectivity and application networks.

Due to the interconnected nature of the application network, *resilience tests* (9.2.4) become particularly important in establishing confidence in the fail-safety of the application network.

Unsurprisingly, APIs and API specifications take center stage when testing application networks.

A distinction can be made between *unit tests* that work on the application component level of the API implementation and *integration tests*:

- *Unit tests*
  - Do not require deployment into any special environment, such as a staging environment
  - Can be run from within an embedded Mule runtime
  - Can/should be implemented using MUnit
    - For read-only interactions to any dependencies (such as other APIs): allowed to invoke production endpoints ("sociable" unit tests)
    - For write interactions: developers must implement mocks using MUnit
  - Require knowledge of the implementation details of the API implementation under test (white-box tests)
- *Integration tests*
  - Are true end-to-end tests: exercise the API just like production API invocations would, i.e., over HTTP/S with API policies applied
  - Require deployment into a staging environment with all dependencies available just as in production
    - No mocking is permissible
  - Can be implemented using SOAPUI and its Maven plugin to trigger API invocations against the deployed API implementation and assert responses
  - Do not require (best: prohibit) knowledge of the implementation details of the API implementation (black-box tests)

9.2.2. Designing automated API-centric integration tests

- You define *test scenarios for each of the APIs* in Acme Insurance's application network
- The scenarios should comprise both *functional and non-functional tests*, including performance tests
- The test scenarios for an API are *driven by the API specification* and, in particular, the *RAML definition* of the API
  - Test scenarios and test cases should be defined *on the basis of* just the discoverable documentation available from the *Anypoint Exchange* entry of the API (black-box tests)
    - This guards against pointlessly driving tests for an API by the actual API implementation
    - It also highlights deficiencies of the API's discoverable documentation
  - There should not be a single interaction in the API Notebook of an API that is not covered by a test scenario

- You automatically execute the test cases for each API by *invoking an actual API endpoint over HTTP/S*
- The most basic (and per-se insufficient) test assertions are those that test for *adherence to the RAML definition* in terms of data types, media types, etc.
- Must execute in special production-like staging environment
  - Alert dependencies before performance tests are run!
- A *safe sub-set of integration tests* can also be run in production as a *"deployment verification test suite"*

9.2.3. Unit testing API implementations

API implementations are typically characterized by numerous and complex interactions with other systems:

- Experience APIs such as the "Aggregator Quote Creation EAPI" invoke Process APIs such as the "Policy Holder Search PAPI"
- Process APIs such as the "Policy Holder Summary PAPI" invoke other Process APIs such as the "Policy Holder Search PAPI" and/or System APIs such as the "Motor Policy Holder Search SAPI"
- System APIs such as the "Motor Policy Holder Search SAPI" interact with backend systems such as the Policy Admin System over whatever protocol the backend system supports (MQ in the case of the Policy Admin System)

Unit testing such complex API implementations can be daunting due to the need for dealing with all these dependencies of an API implementation.

With *MUnit* Anypoint Platform provides a dedicated unit testing tool that

- is specifically designed for unit testing Mule applications
- can stub-out external dependencies of a Mule application
- has dedicated IDE-support in Anypoint Studio
- can be invoked from Maven builds using the MUnit Maven plugin

9.2.4. Testing the resilience of application networks

The foundation of application networks is a web of highly interconnected APIs. *Resilience testing* is the practice of *disrupting* that web and asserting that the resulting inevitable *degradation of the quality of all relevant services* offered by/on the application network is within *acceptable* limits.

Resilience testing is an *important practice* in the move to API-led connectivity and application networks.

Acme Insurance plans to implement resilience testing using the following automated approach:

- Using a software tool similar in spirit to Chaos Monkey
- which acts as an API client to the API Platform API
- and with the help of this API automatically adds, configures and removes customer API policies on all APIs in (a test sub-set of) the application network
- where the custom API policies manipulated by the tool aim to erratically throttle or interrupt invocations of the APIs to which they are applied

While this resilience testing tool runs and disrupts the application network, "normal" automated integration tests are executed.

Also see [Ref11].

9.2.5. Exercise 17: Reflect on resilience testing

Think back on your experience with resilience testing complex distributed systems in any organization you have worked with:

1. In your experience, is resilience testing an established part of organizations' testing strategy?
2. Should resilience tests be run against the production environment?
   a. What about performance tests?
3. Does focusing resilience testing on API invocations make this a more approachable practice?
   a. Does it reduce the effectiveness of resilience testing compared to more general resilience testing approaches?

Solution

None provided.

9.2.6. Integration test scenarios for Acme Insurance

A small selection of integration test scenarios and classes of test cases for Acme Insurance:

- "Aggregator Quote Creation EAPI":
  - Invoke with valid, invalid and missing policy description from Aggregator

    - Invoke for existing and new policy holder
    - Invoke for policy holder with a perfectly matching in-force policy
    - Invoke at 500, 1000 and 1500 requs/s
    - Invoke over HTTP and HTTPS
    - Invoke from API client with valid, expired and invalid client-side certificate
- "Motor Policy Holder Search SAPI":
    - Invoke with valid, invalid and missing search criteria
    - Invoke for search criteria matching 0, 1, 2 and almost-all policy holders
    - Invoke for policy holder with only home and no motor policies
    - Invoke at 500, 1000 and 1500 requs/s
    - Invoke with valid and invalid client token and without client taken
- "Motor Claims Submission PAPI":
    - …
    - Invoke polling endpoint once per original request, 1000 times per second, not at all

Test cases based on the above test scenarios should be executed

- with the application network in a (supposedly) healthy state
- while Chaos Monkey-like resilience tests are disrupting the application network (9.2.4)

9.3. Scaling the application network

9.3.1. Ways to scale the performance of an API

Given an API and its API implementation, there are two main ways of scaling (typically: increasing) the performance of that API:

- *Vertical* scaling, i.e., scaling the performance of each node on which a Mule runtime and a deployed API implementation or API proxy executes
    - In CloudHub this is supported through different *worker sizes* (7.1.4) and is a disruptive process that requires the provisioning of *new CloudHub workers* with the desired size to *replace the existing CloudHub workers*
- *Horizontal* scaling, i.e., scaling the number of nodes on which Mule runtimes and deployed API implementations or API proxies executes
    - In many Anypoint Platform runtime planes (incl. CloudHub, Anypoint Platform for Pivotal Cloud Foundry and Anypoint Runtime Fabric) this is directly supported through *scaleout and load balancing*, currently with a limit of 8 workers per API implementation

Both types of scaling can be triggered in one of two ways:

- *Explicitly* by a *user* via Anypoint Runtime Manager or a *script* via Anypoint Platform APIs or Anypoint CLI
- *Automatically* by CloudHub itself due to a significant change in the runtime characteristics of the API implementation: this is called *autoscaling* and can be configured through policies that scale up or down, horizontally or vertically, based on actual CPU or memory usage over a given period of time (Figure 111)

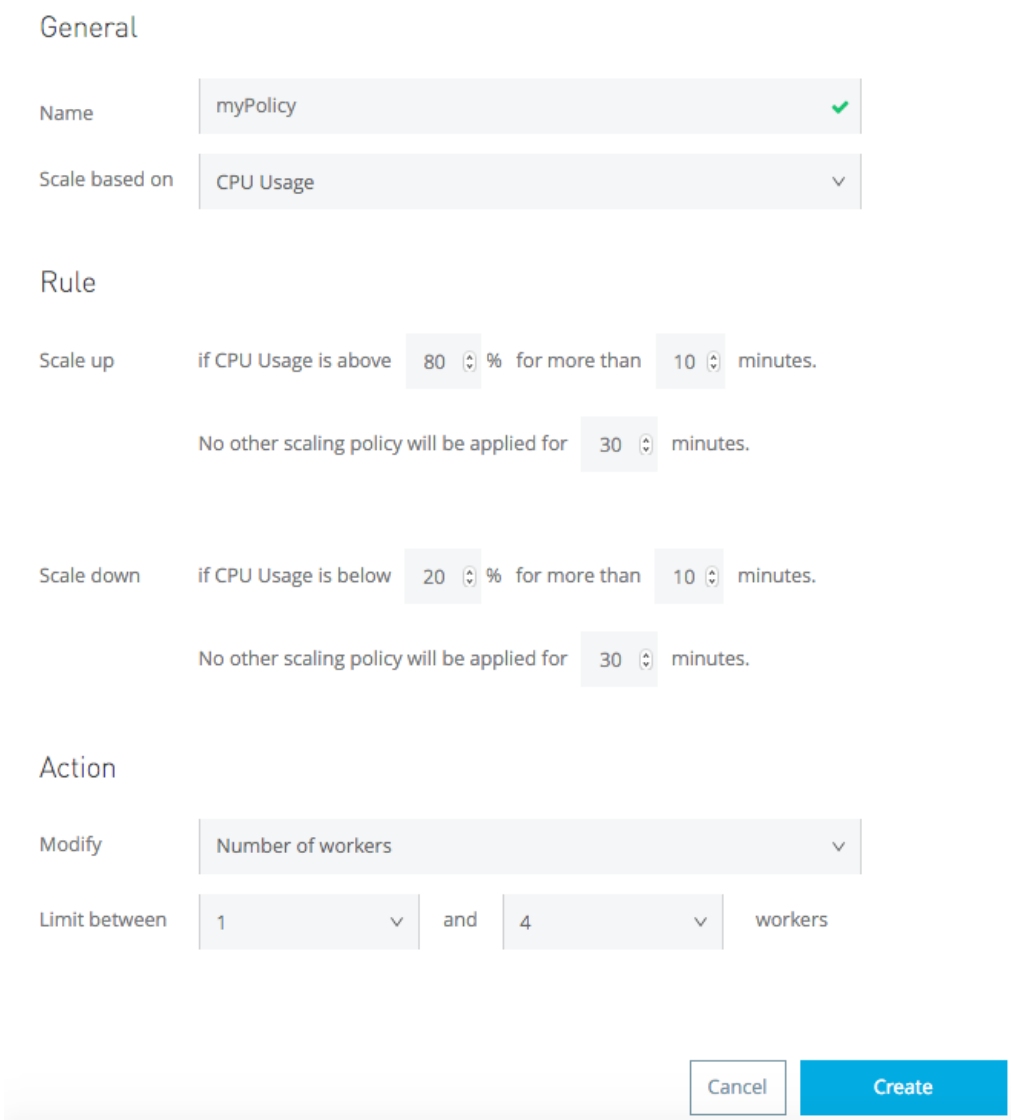

Figure 111. Configuring an autoscaling policy in CloudHub to (horizontally) scale the number of CloudHub workers of a given Mule application between 1 and 4 based on whether CPU usage over a 10-minute interval is above 80% or below 20%. After a re-scaling step was triggered, autoscaling is suspended for 30 minutes.

If separate API proxies are deployed in addition to API implementations then the two types of nodes can be scaled independently. In most real-world cases the performance-limiting node is the API implementation and not the API proxy, hence there typically need to be more/larger instances of the API implementation than the corresponding API proxy.

9.3.2. An API scales for its API clients

Because an API's raison d'être is to be consumed by API clients, the QoS and performance characteristics of an API must suit these API clients. The same is true for the *change* in these performance characteristics over time - the scaling of the API performance over time.

It is therefore essential that the team responsible for an API understands the projected performance needs of this API's API clients and is prepared to scale their API's performance to meet those needs. This may require re-evaluating QoS guarantees and SLAs with all API dependencies of their API.

9.3.3. The C4E owns systemic performance considerations

APIs do not exist in a vacuum. They do not even exist just for their API clients. A the end of the day, all APIs exist to contribute to strategic business goals (Figure 3), such as

- To create new sales channels by opening to Aggregators and thereby increase revenue,
- To enable customer self-service by providing them the required self-service capabilities and thereby reduce cost

The performance of any individual API in the Acme Insurance application network is only important insofar as it enables and supports these strategic business goals.

It is therefore not efficient if individual APIs provide performance that is unbalanced with respect to the performance of other APIs in the application network. For instance, stellar response time of the "Mobile Auto Claim Submission EAPI" used in the "Submit auto claim" feature is irrelevant if the "Retrieve policy holder summary" feature is so slow that users are put off from ever using the "Submit auto claim" feature.

What this means is that there is a systemic perspective on API performance, i.e., a viewpoint that calls for an application network-wide analysis of API performance. This systemic viewpoint must consider

- dependencies between APIs
- the business goals to which each API ultimately contributes
- and the relative importance of these business objectives, now and in the future.

Application network-wide performance analysis and governance is a responsibility best assigned to the C4E. The data for this type of analysis can be extracted from Anypoint Platform through the Anypoint Platform APIs.

9.3.4. Exercise 18: Scaling "Policy Options Retrieval SAPI"

Acme Insurance has deployed the API implementation of "Policy Options Retrieval SAPI" (Figure 39) to one 1 vCore CloudHub worker and has applied the API policies shown in Figure 68. This configuration does not deliver the required throughput (5.1.1). Using your experience to guide your assumptions, suggest an ordered sequence of steps that could be taken to improve the throughput of "Policy Options Retrieval SAPI".

Solution

- The NFRs are interpreted to mean that the goal is a throughput of 5000 requs/s for "Policy Options Retrieval SAPI"
- Figure 87 suggest that more than 4 vCores are therefore required
- It is assumed that the Policy Admin System is not nearly capable of delivering that throughput: caching is therefore essential
- Caching in memory delivers best performance but hit rate is better with fewer CloudHub workers keeping an in-memory cache
- Input to "Policy Options Retrieval SAPI" is expected to be amenable to caching (recurring input resulting in high cache hit rate)

1. Increase the number of workers for the "Policy Options Retrieval SAPI" implementation to two (for HA reasons) or more such that the Policy Admin System rather than the System API is the bottleneck. This can be done either based on past throughput measurements of the Policy Admin System, or by simply adding workers to the System API until no increase in throughput is seen. The configuration of the Spike Control API policy on "Policy Options Retrieval SAPI" must be adjusted accordingly to protect Policy Admin System. (The SLA for the Rate Limiting policy is expected to be much higher, similar to the target 5000 requs/s, and therefore needs no adjustment.) It is assumed that two 1 vCore workers suffice to saturate the Policy Admin System with requests.
2. Add an API proxy in front of the "Policy Options Retrieval SAPI" API implementation and apply a caching (custom) API policy and the previous SLA-based Rate Limting policy to the API proxy. Deploy the API proxy to two 4 vCore CloudHub workers to provide sufficent memory for caching as well as the necessary throughput
3. Measure throughput and cache hit rate of the above scenario: if it is insufficient, scale vertically to 8 vCore workers, which improves cache hit rate without sacrificing HA
4. If throughput is still insufficient, experimentally deploy to a single 16 vCore worker (with auto-restart!), which maximizes cache hit rate at the expense of HA. If cache hit rate

does not increase significantly then go back to the previous configuration (two workers). If this configuration offers significantly better throughput than the previous two-worker deployment, then the API clients of "Policy Options Retrieval SAPI", i.e., "Policy Options Ranking PAPI", must be urged to apply cient-side caching (7.2.9) and/or static fallback results (7.2.10) to mitigate for the reduced HA of "Policy Options Retrieval SAPI"

9.4. Gracefully ending the life of an API

9.4.1. End-of-live management on the level of API version instead of API implementation

The needs of an API client are fulfilled

- if it can successfully send API invocations to the published endpoint of an API
- and if it receives responses in accordance with the contract and expected QoS of the API.

This process requires an API implementation of that API to be available and to live up to the expected QoS standards. But since the API client is not aware of the API implementation itself - it is only aware of the API exposed at a certain endpoint - the API implementation can be changed, updated and replaced without alerting the API client.

On the other hand, any change to the API, its contract or promised QoS, that is not entirely backwards-compatible needs to be communicated to all API clients of that API. This is done through the introduction of a new version of the API - and the subsequent phased ending of the live of the previous API version.

9.4.2. Deprecating and deleting an API version on Anypoint Platform

Anypoint Platform supports end-of-live management of APIs as follows:

- In Anypoint API Manager an API instance (which exists in a particular environment) can be set to *deprecated*
  - This prevents API consumers from requesting access to that API instance for their API clients
- At a later stage, the API instance can then be *deleted from that environment*
- In addition, and independently, in Anypoint Exchange an individual asset version (full semantic version) of an API entry can be *deprecated*
  - This informs API consumers that that version of the API specification should no longer be used, but does not prevent them from requesting access to API instances of that API version

- If all asset versions of an Anypoint Exchange API entry have been deprecated, then the entire Anypoint Exchange entry for that API (major) version is automatically highlighted as *deprecated*

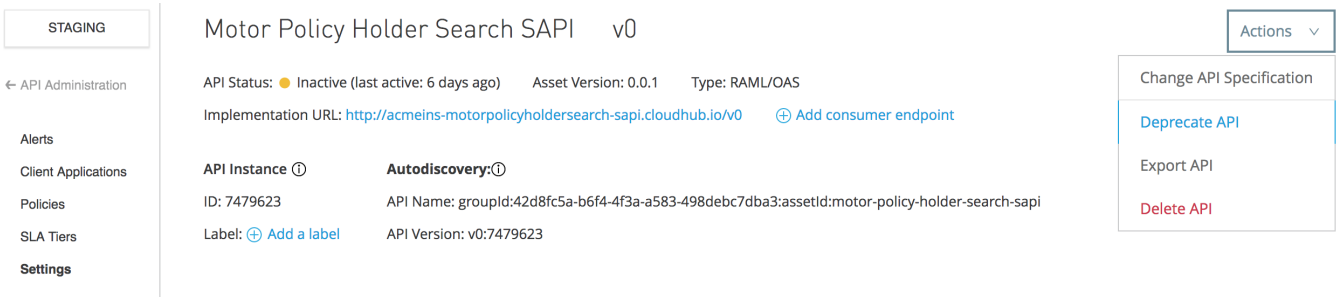

Figure 112. Option to deprecate or delete an instance of the "Motor Policy Holder Search SAPI" in Anypoint API Manager.

9.5. Identifying points of failure

9.5.1. Exercise 19: Identify points of failure in Acme Insurance's application network

Assume a deployment of all APIs in Acme Insurance's application network (Figure 57) to a MuleSoft-hosted Anypoint Platform using CloudHub:

1. Identify *points of failure* in this architecture
2. Are there any components that are not redundant, i.e., that constitute *single points of failure*?

Solution

- *Potential* points of failure are everywhere: every node and system involved in processing API invocations is a potential point of failure
- For instance, failure of the Anypoint API Manager would mean that
  - Already-applied API policies continue being in force
  - New Mule runtimes that start-up (and are configured with the gatekeeper feature) will not become functional until they can download API policies from the Anypoint API Manager
  - Continued unavailability of the Anypoint API Manager would lead to overflow of the buffers in the Mule runtime that hold undelivered API analytics events
- *Single* points of failure are much rarer
  - At first sight it seems as if there were no single points of failure

- Every API implementation that is deployed to only one CloudHub worker constitutes a single point of failure (although CloudHub is typically configured to auto-restart failed workers and hence the duration of failure is relatively short)
- The AWS region is a single point of failure for the Anypoint Platform control plane and - unless explicitly mitigated via cross-region HA - the Anypoint Platform runtime plane (CloudHub deployments) in that region
- There is little information about the Home Claims System and it may therefore potentially contain single points of failures.
- Every deployment of an API implementation, in its entirety, constitutes a single point of failure: If the deployment of an API implementation technically succeeds but deploys a deficient API implementation, then API invocations to that API will fail. Thus this API now constitutes an actually failed single point of failure for all API clients of that API (but they may have a fallback API to invoke). This highlights the importance of reliable and fully automated DevOps and testing processes, including a "deployment verification test suite".

Summary

- API definition, implementation and management can be organized along an API development lifecycle
- DevOps on Anypoint Platform builds on and supports well-known tools like Jenkins and Maven
- API-centric automated testing augments standard testing approaches with an emphasis on integration tests and resilience tests
- Scaling API performance must match the API clients' needs and requires the C4E’s application network-wide perspective
- Anypoint Platform supports gracefully decommissioning API versions using deprecation
- Anypoint Platform has no inherent single points of failure but every deployment of an API implementation can become one

Module 10. Monitoring and Analyzing the Behavior of the Application Network

Objectives

- Describe the origins of data used in monitoring, analysis and alerting on Anypoint Platform
- Describe the metrics collected by Anypoint Platform on the level of API invocations
- Describe the grouping of API metrics available in Anypoint Analytics
- Make use of options for performing API analytics within and outside of Anypoint Platform
- Define alerts for key metrics of API invocations for all tiers of API-led connectivity
- Use metrics and alerts for API implementations to augment those for API invocations
- Recognize operations teams as an important stakeholder in API-related assets and organize documentation accordingly

10.1. Understanding monitoring data flow in Anypoint Platform

10.1.1. Data flows for Anypoint Platform monitoring, analytics and alerting

Compare also with Table 1.

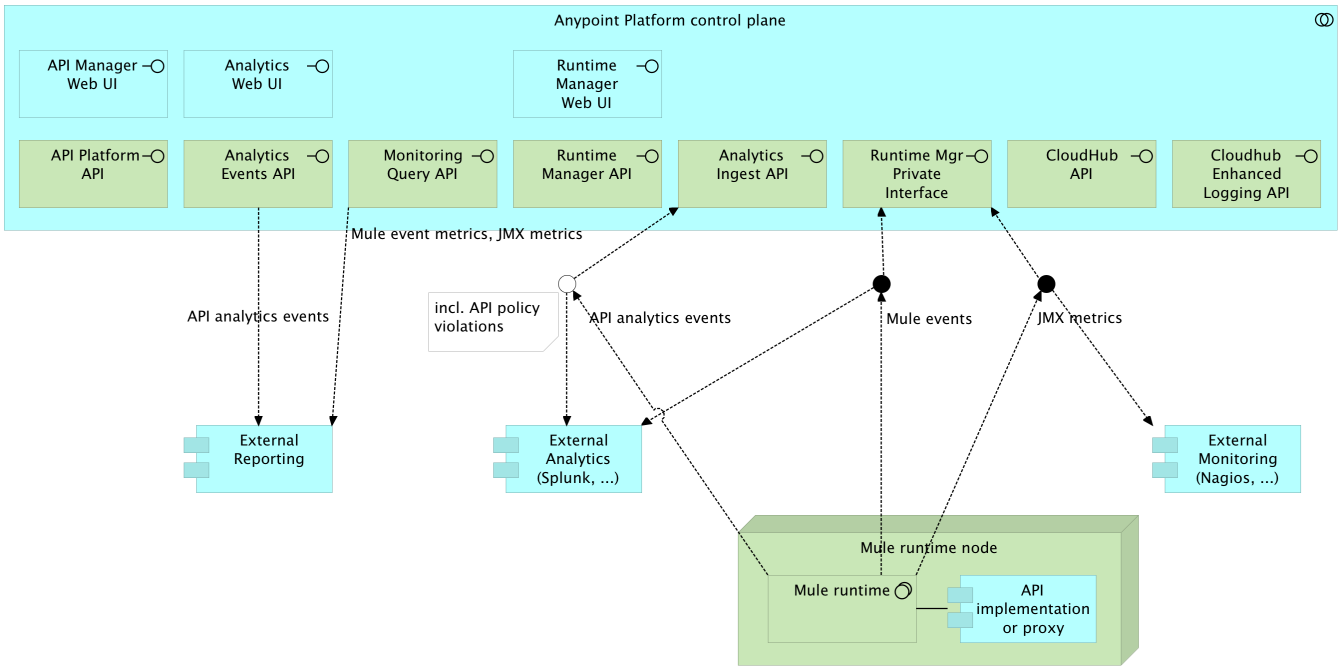

Figure 113. Different types of data used for monitoring, analytics and alerting flow from Mule runtimes to Anypoint Platform control plane components like Anypoint Runtime Manager, Anypoint API Manager and Anypoint Analytics and/or to external systems. Not all options are available in all Anypoint Platform deployment scenarios.

10.2. Using Anypoint Analytics to gain insight into API invocations

10.2.1. Metrics in Anypoint Analytics

Anypoint Analytics provides features for analyzing API invocations based on various metrics, such as:

- *Number of API invocations* (requests)
  - *Successful*, i.e., those with a HTTP response status code in range `[100, 400)`
  - Unsuccessful due to a *client error*, i.e., those with a HTTP response status code in range `[400, 500)`
  - Unsuccessful due to a *server error*, i.e., those with a HTTP response status code in range `[500, 600)`
- *Mean response time* (average latency)
- Request and response payload *size*
- Properties of the API client such as its *client ID* (if registered), geographical *location*, OS *platform*, etc.

- Properties of the API invocation itself such as *resource path*, *HTTP method*, etc.

Note that only for REST APIs are HTTP response status codes indicative of success, failure and reason for that failure.

Metrics can be grouped and displayed along various dimensions:

- per API for all API clients
- per API and API client
- per API client for all APIs
- custom

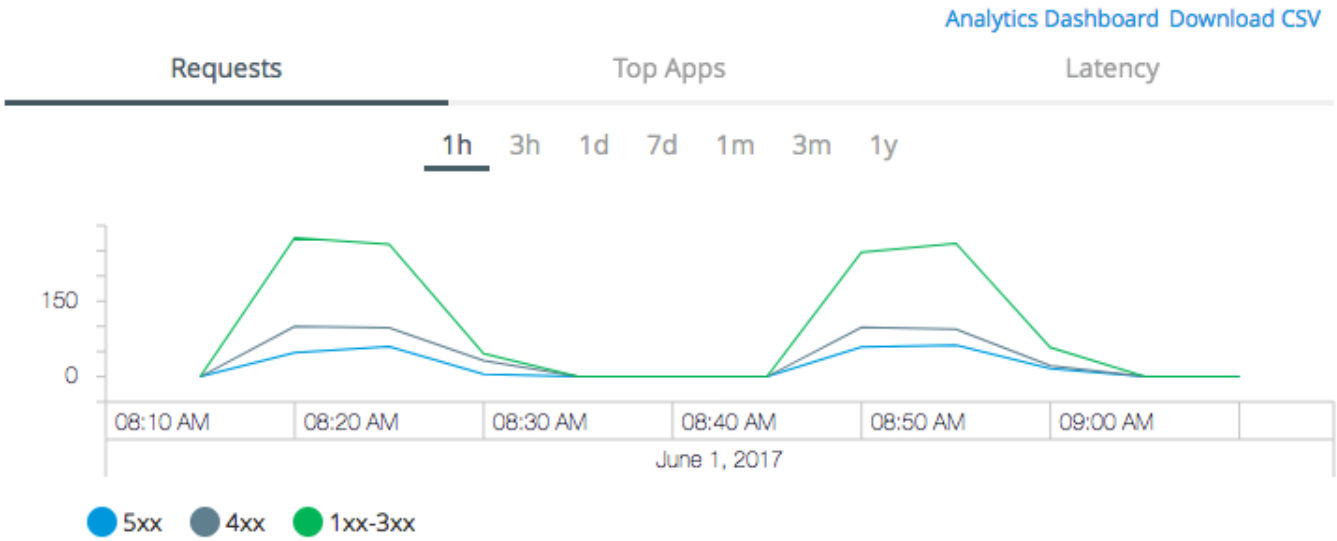

Figure 114. Number of API invocations (requests) over time for a given API and all its API clients, grouped by HTTP status code class.

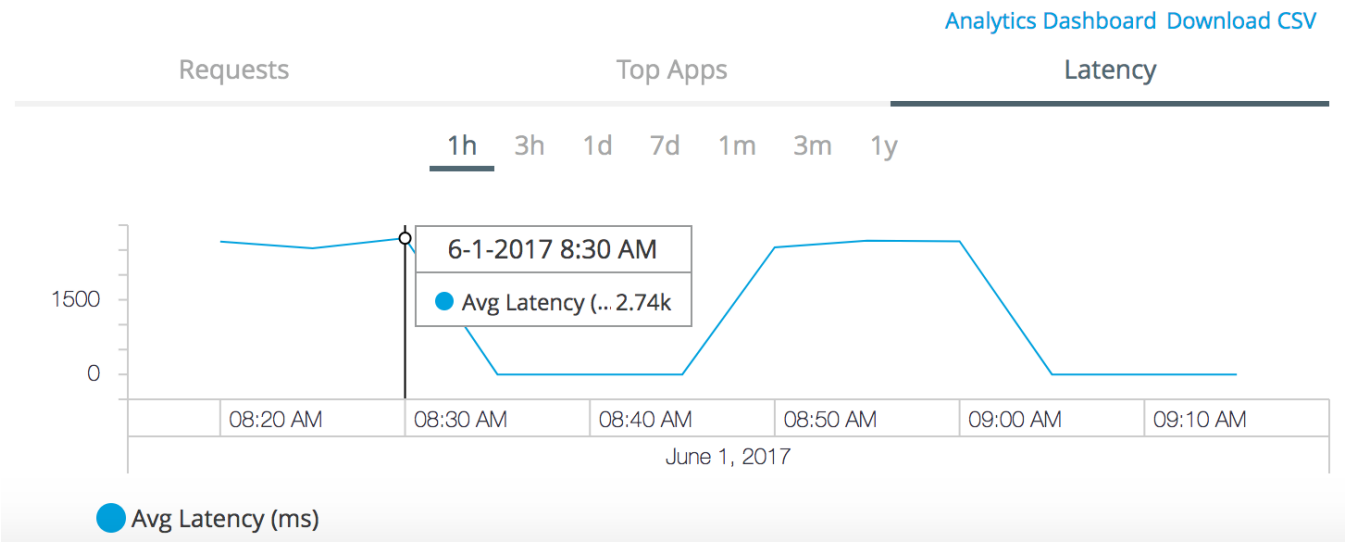

Figure 115. Mean response time (average latency) of API invocations over time for a given API and all its API clients and all HTTP status codes.

10.2.2. Analyzing API invocations to the "Mobile Auto Claim Submission EAPI"

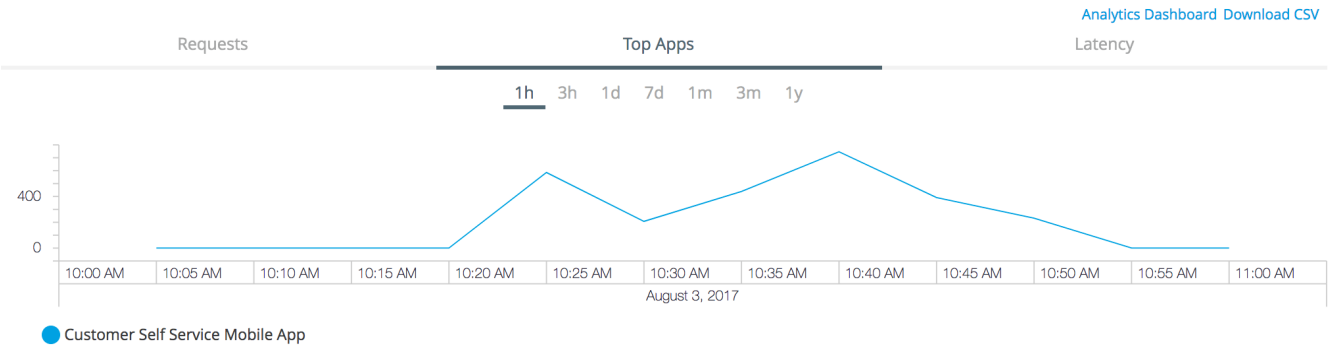

Figure 116. Number of API invocations (requests) over time to the "Mobile Auto Claim Submission EAPI", grouped by each of its top 5 API clients.

10.2.3. Analyzing API invocations to the "Policy Holder Search PAPI"

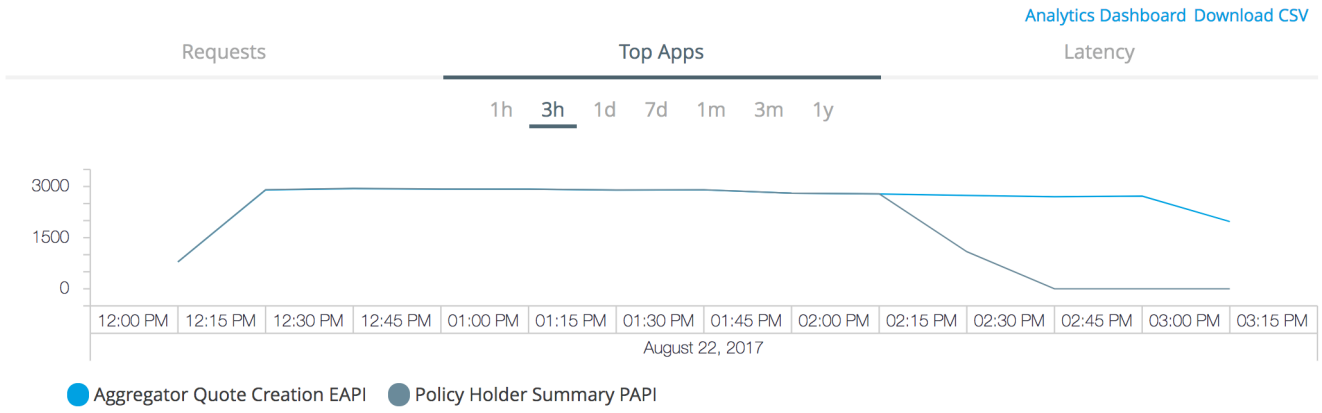

Figure 117. Number of API invocations (requests) over time to the "Policy Holder Search PAPI", grouped by each of its top 5 API clients.

10.2.4. Analyzing API invocations from the perspective of the Customer Self-Service Mobile App

An API consumer is typically interested in analytics of all API invocations performed by their API clients. An example of this is the Customer Self-Service Mobile App.

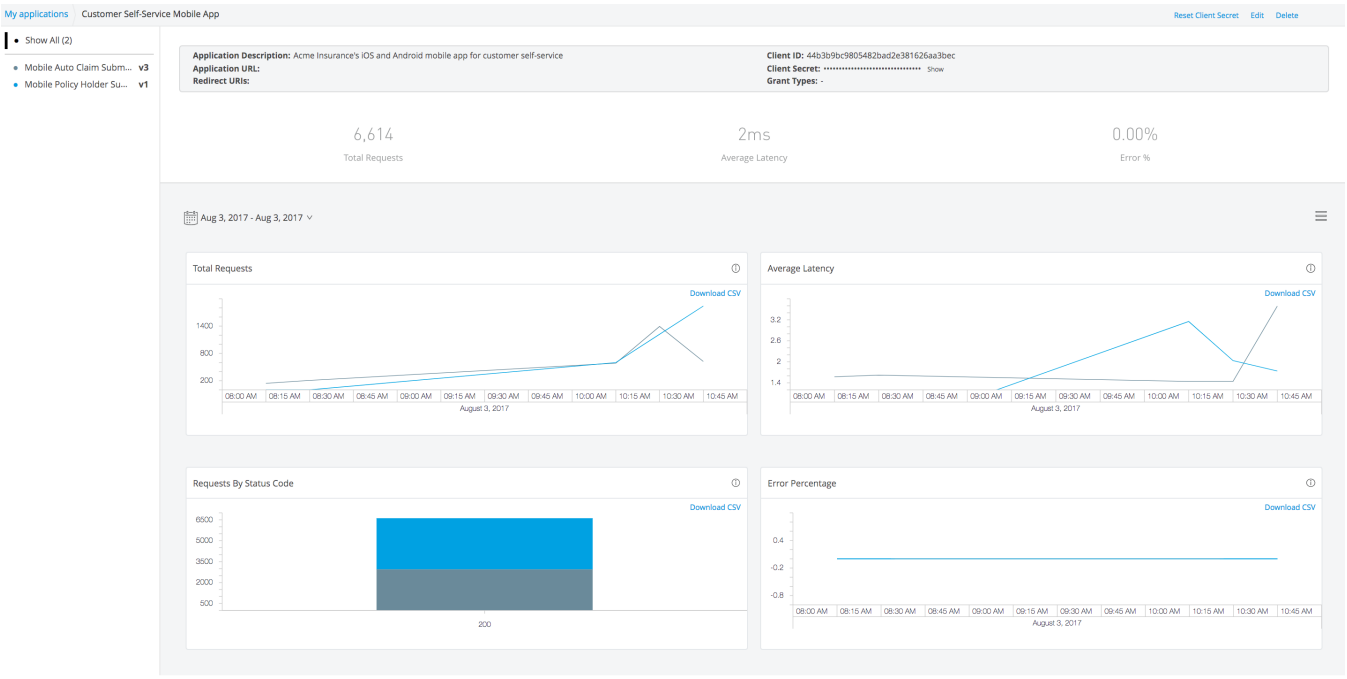

Figure 118. Overview of API invocations from the Customer Self-Service Mobile App to all APIs it invokes - which are, by definition, Experience APIs.

10.3. Analyzing API invocations across the application network

10.3.1. Introducing application network-wide API analytics

The Anypoint Analytics component of Anypoint Platform can be used to perform *standard and custom analyses* across all API invocations in an application network:

- Interactive exploration through drill-down
- Definition of custom charts and dashboards
- Definition of custom reports
- Exporting all data underlying a graph to CSV files
- Programmatic access to all data via Anypoint Platform APIs

10.3.2. API invocation analysis by geography

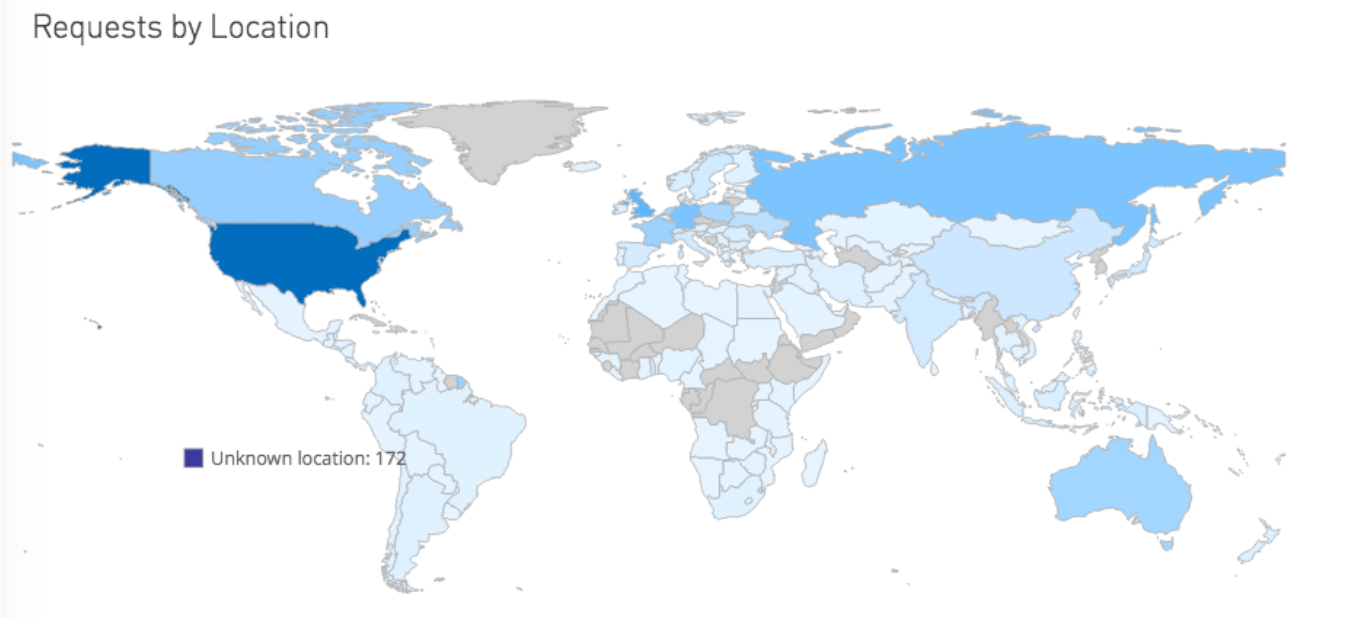

Figure 119. Number of API invocations from all API clients to all Experience APIs, grouped by geography.

10.3.3. API invocation analysis by API client

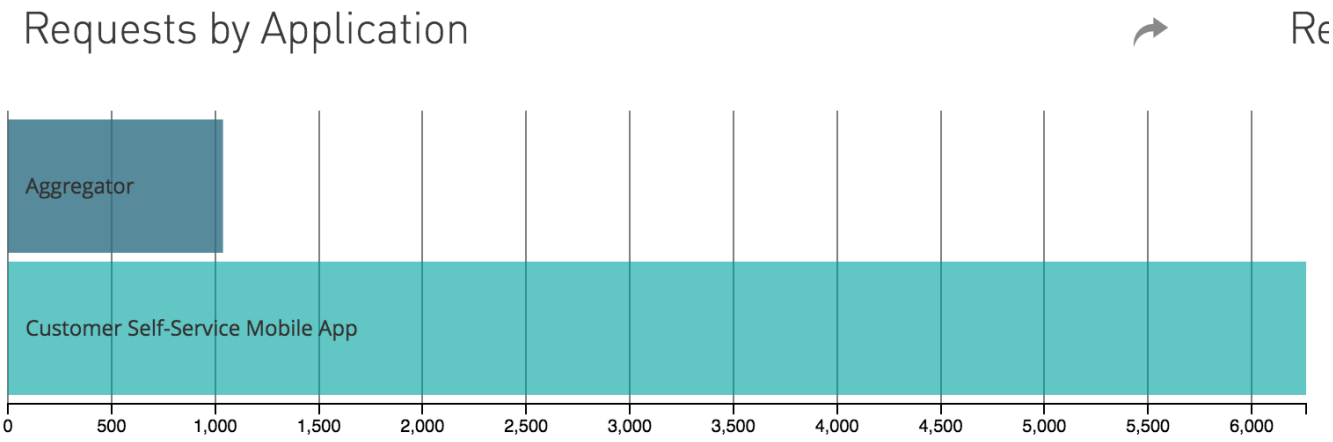

Figure 120. Number of API invocations to all Experience APIs, grouped by API clients.

10.3.4. API invocation analysis by response time

Figure 121 shows an analysis based on response time, which is expressed in milliseconds. It is important to note that response time is measured at the API implementation side, not the API client side, and hence does not include network roundtrip latency from the API client to the API implementation.

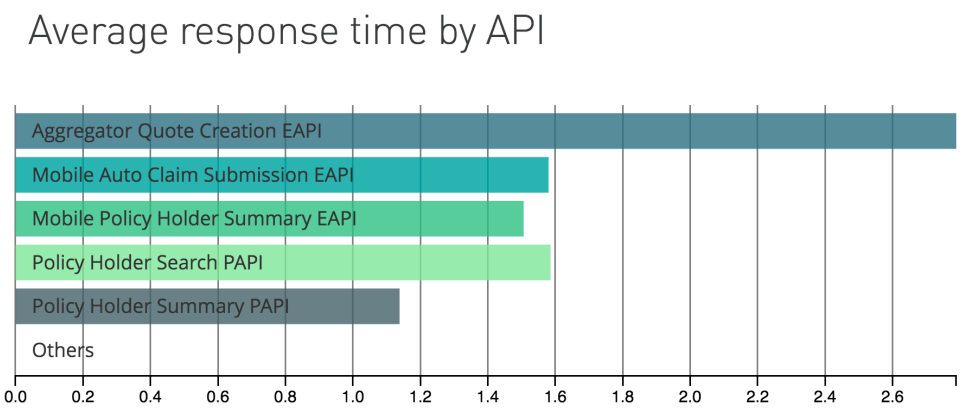

Figure 121. Custom chart showing average (mean) response time per API invocation in milliseconds for the top 5 slowest APIs, for the last 90 days. Response times shown are for no-op implementations of the respective APIs.

10.3.5. API governance analysis

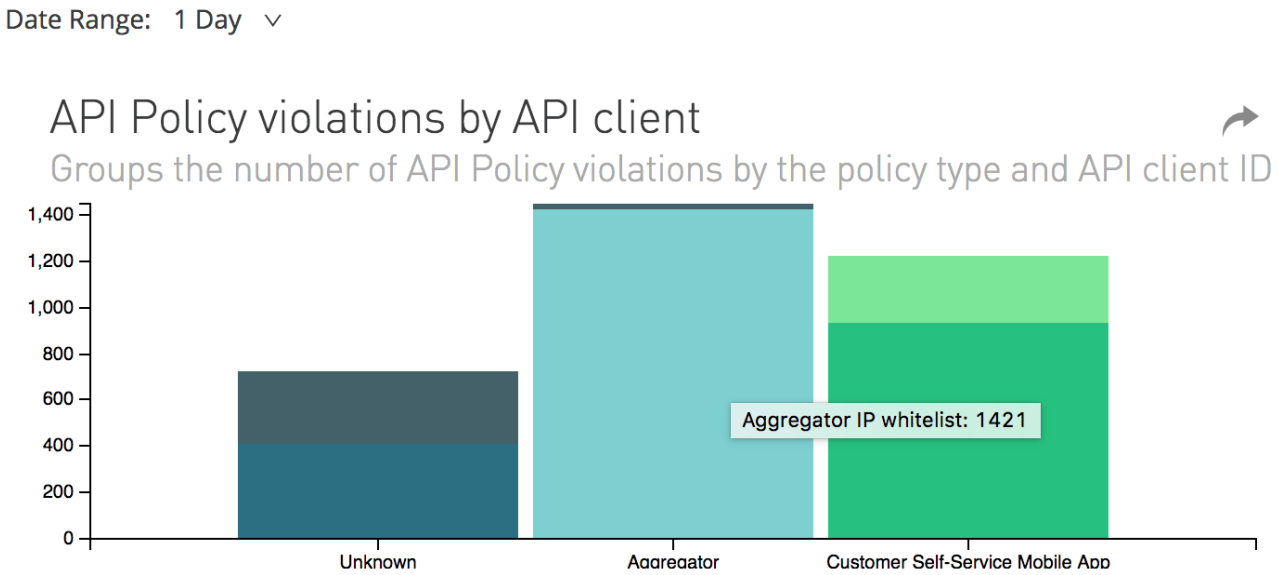

Figure 122. Custom chart showing number of policy violations, grouped by API policy and API client.

10.4. Defining alerts for exceptional occurrences in an application network

10.4.1. Introducing alerts at the level of API invocations

Anypoint Platform can raise alerts based on these *metrics* of API invocations:

- Number of violations of a given policy
- Request count (number of API invocations)
- Response code in given set of HTTP response status codes
- Response time exceeding given threshold in milliseconds

For each of these metrics Anypoint Platform typically *triggers an alert when the metric*

- falls above or below a given *threshold*
- for a given number of *time periods* of a given duration

In general, the *C4E establishes the guideline* that alerts for APIs in the Acme Insurance application network should at least cover:

- All violations of API policies
- All violations of QoS guarantees not explicitly captured in API policies

10.4.2. Defining alerts for "Policy Options Retrieval SAPI"

Applying the C4E guidelines for alerts to "Policy Options Retrieval SAPI", referring to the NFRs and API policies applicable to this API (Figure 123):

- "SLA tier exhausted for "Policy Options Retrieval SAPI"": for violation of *SLA-based Rate Limiting*, severity *Info*, more than 60 violations for at least 3 consecutive 10-minute periods
  - Alerts when approx. 10% of 1-second intervals are above SLA tier-defined rate limit
  - Also alerts on invalid client ID/secret supplied
- "Client not in Process API subnet for "Policy Options Retrieval SAPI"": for violation of *IP whitelist*, severity *Critical*, more than 1 violation for at least 3 consecutive 1-minute periods
- Could add alert for violations of *Spike Control*

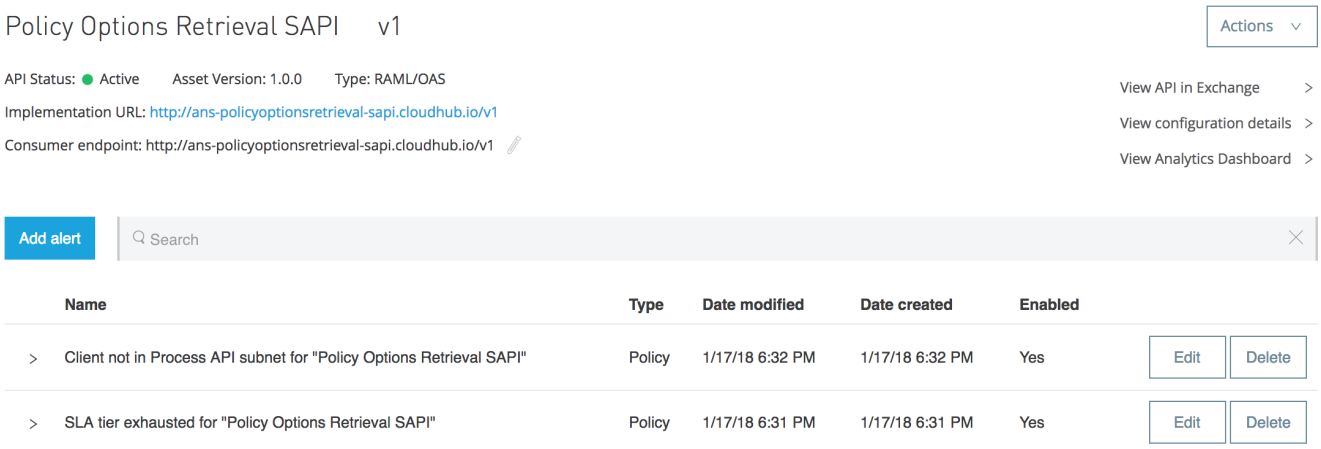

Figure 123. Alerts defined for the "Policy Options Retrieval SAPI".

10.4.3. Defining alerts for "Policy Holder Search PAPI"

Applying the C4E guidelines for alerts to the "Policy Holder Search PAPI", referring to the NFRs and API policies applicable to this API (Figure 124):

- "Throughput QoS guarantee exhausted for "Policy Holder Search PAPI"": for violation of *Spike Control*, severity *Info*, more than 60 violations for at least 3 consecutive 10-minute periods
  - Alerts when approx. 10% of 1-second intervals are above rate limit defined in the API policy
- "Client not in Experience API or Process API subnet for "Policy Holder Search PAPI"": for violation of *IP whitelist*, severity *Critical*, more than 1 violation for at least 3 consecutive 1-minute periods

- "Response time QoS guarantee violated by "Policy Holder Search PAPI"": for violation of QoS guarantee not captured in any API policy, severity *Warning*, more than 6600 requests whose response time exceeds 100 ms for at least 3 consecutive 10-minute periods
  - Alerts when approx. 1% of API invocations (1% of 1100*60*10 = 6600) take longer than 100 ms (twice the target median of 50 ms)
  - Note that exact QoS guarantee cannot be expressed in alert: median = 50 ms, maximum = 150 ms
- Should add alert for violations of *Client ID enforcement*

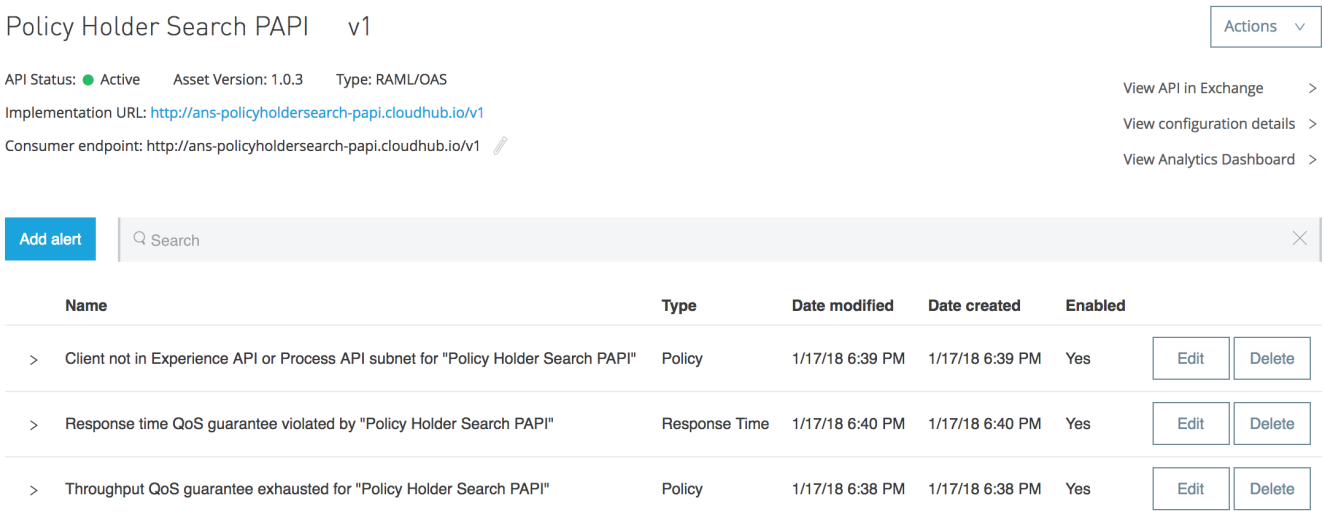

Figure 124. Alerts defined for the "Policy Holder Search PAPI".

10.4.4. Defining alerts for "Aggregator Quote Creation EAPI"

Applying the C4E guidelines for alerts to "Aggregator Quote Creation EAPI", referring to the NFRs and API policies for this API (Figure 125):

- "SLA tier exhausted for "Aggregator Quote Creation EAPI"": for violation of *SLA-based Rate Limiting*, severity *Info*, more than 60 violations for at least 3 consecutive 10-minute periods
  - Alerts when approx. 10% of 1-second intervals are above SLA tier-defined rate limit
  - Also alerts on invalid client ID/secret supplied
- "TLS mutual auth circumvented for "Aggregator Quote Creation EAPI"": for violation of *IP whitelist*, severity *Critical*, more than 1 violation for at least 3 consecutive 1-minute periods
- "XML attack on "Aggregator Quote Creation EAPI"": for violation of *XML threat protection*, severity *Warning*, more than 30000 violations for at least 3 consecutive 10-minute periods
  - Alerts when approx. 5% of requests (5% of 1000*60*10 = 30000) are identified as XML threats

- "Response time QoS guarantee violated by "Aggregator Quote Creation EAPI"": for violation of QoS guarantee not captured in any API policy, severity *Warning*, more than 6000 requests whose response time exceeds 400 ms, for at least 3 consecutive 10-minute periods
  - Alerts when approx. 1% of API invocations (1% of 1000*60*10 = 6000) take longer than 400 ms (twice the target median of 200 ms)
  - Note that exact QoS guarantee cannot be expressed in alert: median = 200 ms, maximum = 500 ms

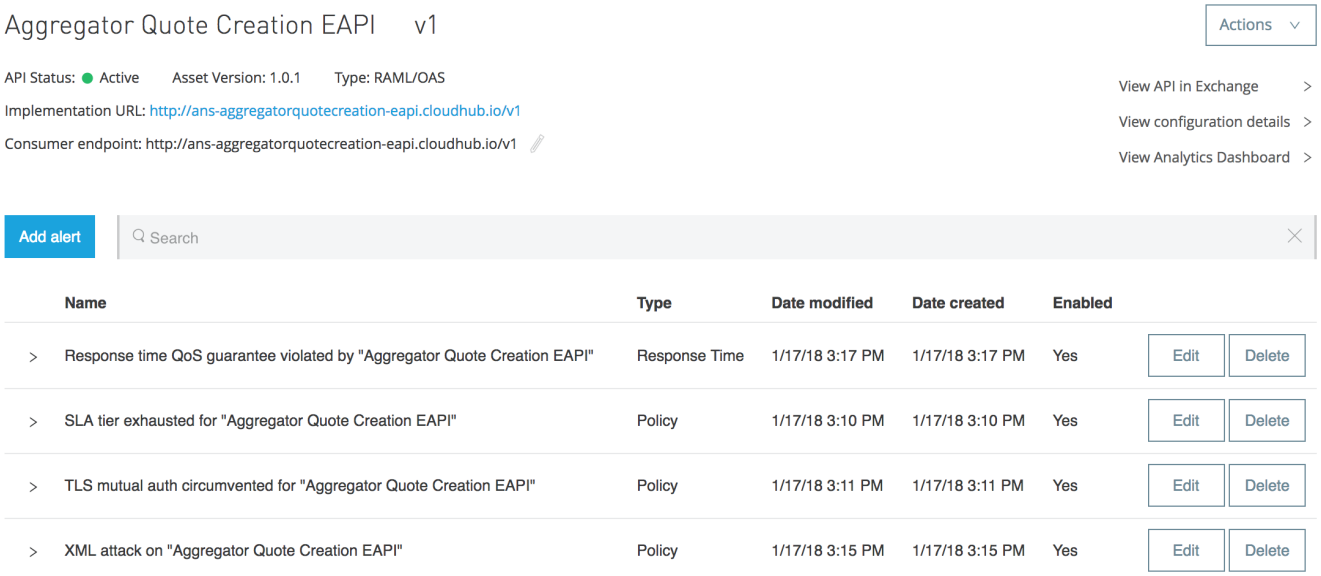

Figure 125. Alerts defined for the "Aggregator Quote Creation EAPI".

10.4.5. Alerts on API implementations augment alerts for API invocations

Alerts on the level of API invocations and on the level of API implementations (i.e., the level of individual application components) complement each other:

- If an API implementation *crashes* (and is not configured to be automatically restarted by CloudHub) and no API client invokes that API then no alert on the level of API invocations is raised - but alerts on the API implementation can be raised
  - Can trigger alert when number API invocations falls below a threshold, but this risk being triggered during periods of legitimately low activity
- Similar for consistently *high CPU usage* on the Mule runtime (CloudHub worker) to which an API implementation is deployed
  - Although this would typically also materialize as *high response times* on the level of API invocations

- *Deployment failures* of an API implementation to a Mule runtime in the production and staging environments should not occur and be covered by an alert

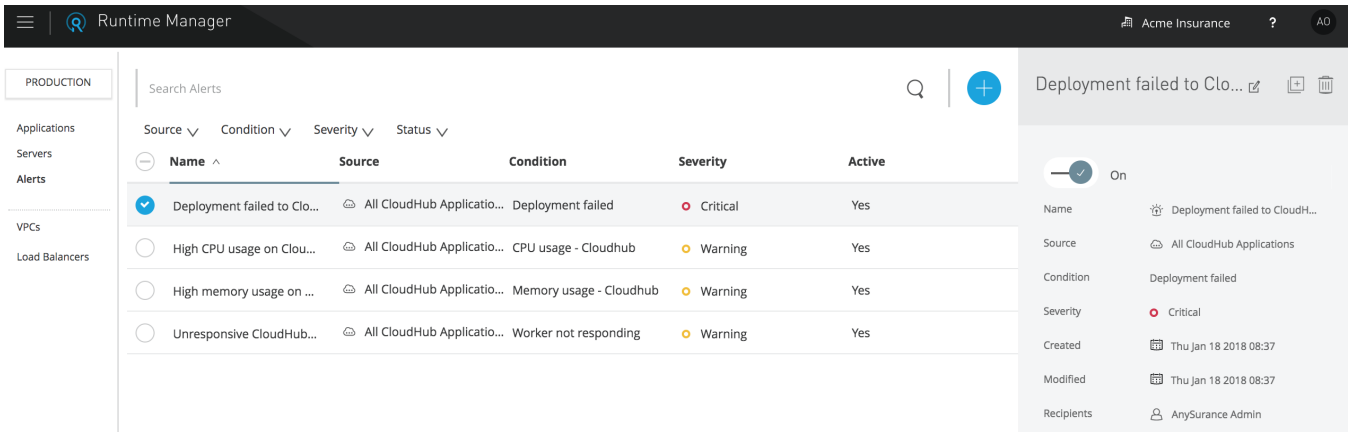

Figure 126. Alerts defined for all API implementations executing on Mule runtimes, such as CloudHub workers, complement alerts on the level of API invocations.

10.5. Organizing discoverable documentation for operations

10.5.1. Operations teams as a stakeholder in APIs

Some prominent organizations recommend that development teams should also operate the APIs and API implementations they implement. If this advice is followed, then development teams are at the same time operations teams. Regardless of whether this recommendation is followed or not, operations teams are an important stakeholder in APIs.

In application networks, some of the needs of operations teams are addressed by

- Dashboards and alerts in Anypoint Runtime Manager, Anypoint API Manager and Anypoint Analytics
- Custom-written documentation:
  - *Runbooks*, which are written for the on-call operations teams and must succinctly give guidance on how to *address alerts*
  - *On-call registers*, which identify the current on-call operations teams and are for anyone who needs to contact them about an issue with "their" API, e.g., the operations team responsible for an API client invoking their API

In an application network, custom-written documentation for operations teams should be discoverable through Anypoint Exchange.

Also see [Ref11].

10.5.2. Organizining discoverable documentation

The Acme Insurance C4E recommends that API documentation and assets be organized and cross-linked as follows to facilitate discovery and navigation, particularly for operations teams.

The Anypoint Exchange entry for a particular major version of an API is the portal to this API's documentation and assets. This Anypoint Exchange entry (Figure 127)

- links to
  - the Anypoint Exchange entry of the auto-generated *Anypoint Connector* to be used by API clients of this API implemented as Mule applications (automatically maintained by Anypoint Platform)
  - the *Anypoint API Manager* API administration entries (Figure 128) for all instances of this API in all environments
- contains
  - the *API specifications* for all asset versions (full semantic versions) of that major version (automatically maintained by Anypoint Platform)
  - the *API Console* ("API summary", automatically maintained by Anypoint Platform)
  - all known *API instances* (endpoints, automatically maintained by Anypoint Platform but can be augmented)
  - succinct *documentation* of the essential architectural aspects of this API and its API implementations
  - *API Notebooks* demonstrating how API clients interact with this API
  - the *Runbook*, which details resolution guidelines for all Anypoint API Manager and Anypoint Runtime Manager alerts for this API and its API implementations
  - *Dev onboarding* documentation for developers joining the team responsible for this API and/or its API implementations
  - *On-call register*, which identifies the current on-call operations teams
  - a section dedicated to all known *API implementations* of this API (Figure 129), which in turn, for each API implementation, links to
    - the Anypoint Exchange entries for all *APIs it depends on* (invokes)
    - the *Anypoint Runtime Manager dashboards* for its deployment (Figure 130) in all environments, e.g., to CloudHub
    - its *GitHub repository*
    - *CI/CD build pipelines*, i.e., Jenkins jobs

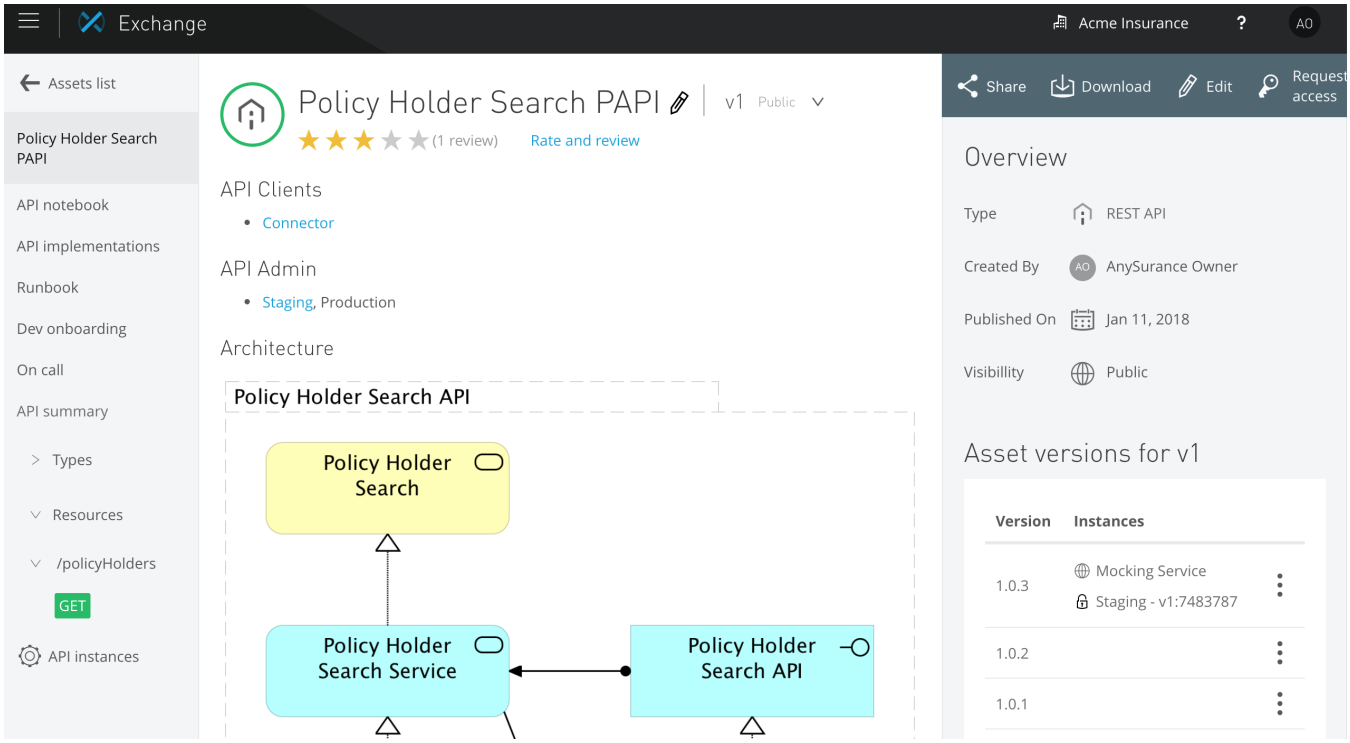

Figure 127. The Anypoint Exchange entry for a particular major version of an API is the portal to this API's documentation and assets.

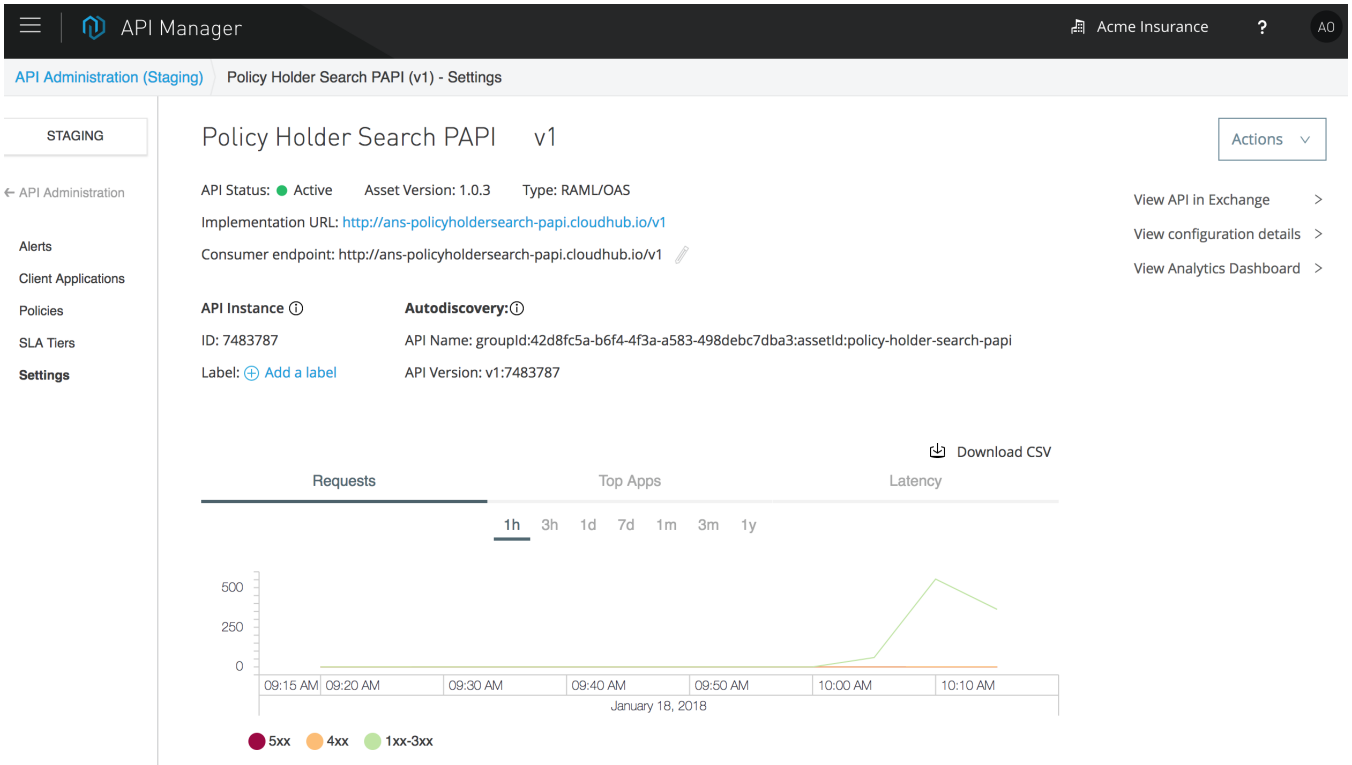

Figure 128. The Anypoint API Manager API administration entry linked-to from its Anypoint Exchange entry, showing summary API analytics.

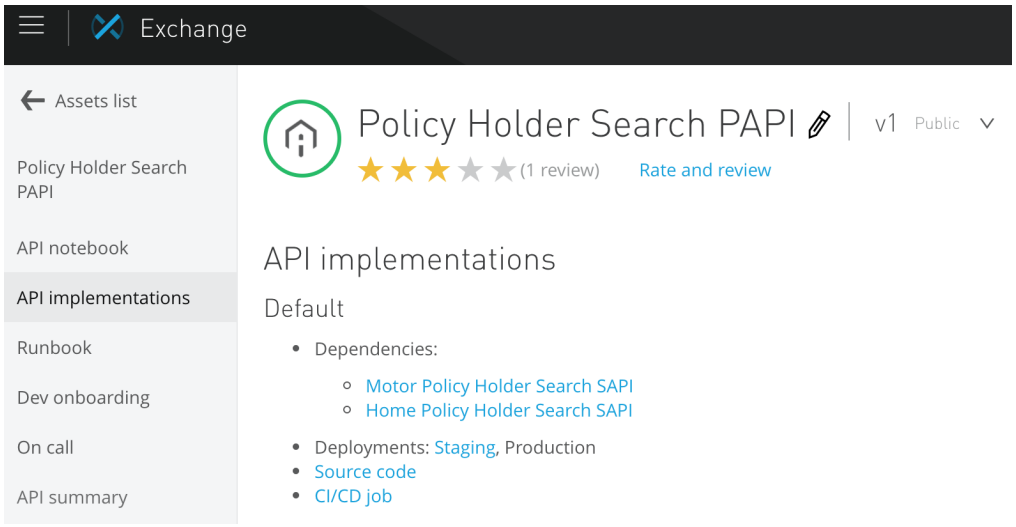

Figure 129. The Anypoint Exchange entry section dedicated to all known API implementations of a particular major version of an API.

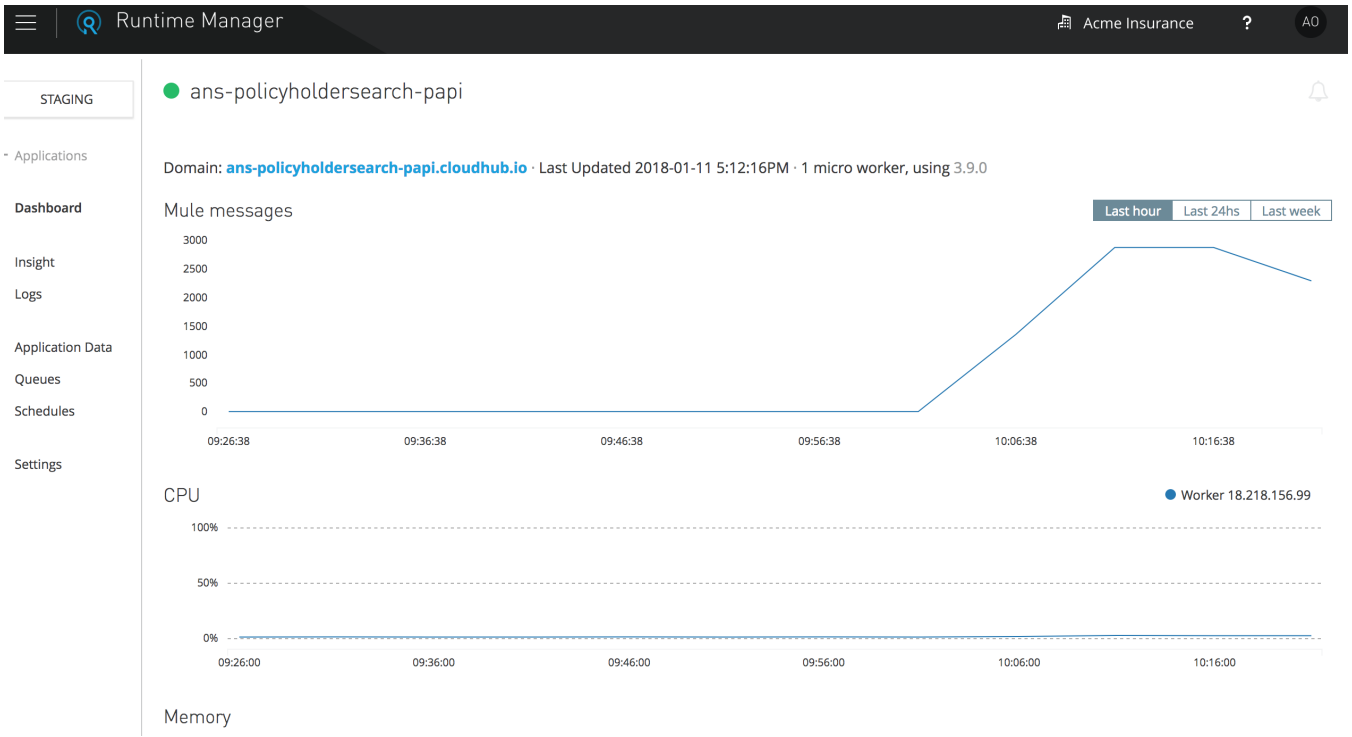

Figure 130. The Anypoint Runtime Manager dashboard for the CloudHub deployment of the API implementation linked-to from its Anypoint Exchange entry.

Please note that currently linking to Anypoint Runtime Manager dashboards for a Mule application in a specific environment does not work reliably, because the environment is not part of the Anypoint Runtime Manager dashboard URL.

Summary

- Data used in monitoring, analysis and alerting flows from Mule runtimes to external monitoring/analytics systems and/or the Anypoint Platform control plane, from where it is available via APIs for external reporting
- Anypoint Platform collects numerous metrics for API invocations, such as response time, payload size, client location, etc.
- For analysis in Anypoint Analytics metrics can be grouped by API, API client or any of the other metrics
- Anypoint Platform supports analyses targeted specifically at API consumers and their API clients
- In addition to interactive analyses, Anypoint Analytics supports custom charts and reports
- All analytics data can be downloaded in CSV files and/or retrieved through Anypoint Platform APIs
- Alerts can be defined based on these API invocation metrics: request count and time, response status code and number of violations of an API policy
- Metrics for API implementations and alerts based on these metrics must be defined in Anypoint Runtime Manager in addition to API invocations alerts
- Operations teams are an important stakeholder in API-related assets
- Structure and link Anypoint Exchange entries for APIs and API implementations, Anypoint API Manager administration screens and Anypoint Runtime Manager dashboards to support operations teams

Wrapping-Up Anypoint Platform Architecture: Application Networks

Objectives

- Review technology delivery capabilities
- Review OBD
- Review the course objectives
- Know where to go from here
- Being aware of the MuleSoft Certification program
- Take the class survey

Reviewing technology delivery capabilities

Review Figure 18 and confirm that all technology delivery capabilities needed in a context of API-led connectivity and the application network are adequately provided by Anypoint Platform.

Reviewing OBD

Review Figure 5 and briefly place all topics discussed in the course along one of the dimensions of OBD.

Reviewing course objectives

Review Course goals from the start of the course and verify that goals and objectives have been adequately addressed.

Knowing where to go from here

Take additional MuleSoft training courses

See https://training.mulesoft.com/course-catalog

- Anypoint Platform:
  - Flow Design (1 day)
  - API Design (2 days)
- Anypoint Platform Development:

    - Fundamentals (5 days)
    - Mule 4 for Mule 3 Users (2 days)
    - Advanced (3 days)
    - DataWeave (1 day)
- Anypoint Platform Operations:
    - CloudHub (1 day)
    - Customer-Hosted Runtimes (2 days)
    - API Management (1 day)
- Anypoint Platform Architecture:
    - Solution Design (4 days)

Introducing the MuleSoft Certification program

MuleSoft Certification program

- Offers multiple types of professional accreditation for developers, architects, and partners
- Three levels of certification
- With each certification, you get
    - A digital badge for your communications
    - The ability to add the certification to your LinkedIn profile
    - A free t-shirt

MuleSoft Certified Developer (MCD) and MuleSoft Certified Architect (MCA) exams

See https://training.mulesoft.com/certification

- MCD - Integration and API Associate
    - First Level: Trained in Mule
    - Recommended for students who have taken the Anypoint Platform Development: Fundamentals class
- MCD - Integration Professional
    - Second Level: Experienced in Mule
    - Must show product and project experience
    - Recommended for developers who have 6+ months work experience in Mule

- MCD - API Design Associate
- MCD - Connector Specialist
- MCA - Solution Design Specialist
- MCA - "to be announced"

Why take the exams?

- To obtain recognized industry certification
- To differentiate yourself in the marketplace
- Get a free t-shirt when you pass

Taking the class survey

Class survey

- You should have received an email with a link to the class survey
  - Your instructor can also provide the direct link
    - http://training.mulesoft.com/survey/<surveyID>.html
  - Or you can go to a general page and select your class
    - http://training.mulesoft.com/survey/survey.html
- **Please fill the survey out now!**
  - *We want your feedback!*

Appendix A: Documenting the Architecture Model

This section presents one concrete approach to documenting an architecture model by arranging information and views (diagrams) for the various aspects of the architecture within a particular framework and order. Taken together, this arrangement of information and views characterizes and documents a given architecture model - such as that of Acme Insurance's application network at the stage discussed in this course - along different dimensions and for different audiences. This is a well-established approach familiar from ArchiMate, TOGAF and the like, although the viewpoints used here do not always rigorously follow those defined by ArchiMate or TOGAF.

Following the focus of this course, this architecture documentation is mostly on the Enterprise Architecture level, with elements of Solution Architecture, and is in-line with the dimensions of OBD and the approaches of API-led connectivity and application networks.

The information and views collected here have all been produced for Acme Insurance during the course: no new information or views are introduced here.

A.1. Architecture vision

A.1.1. Business outcomes dimension

Motivation: Acme Insurance's motivation to change

A.1.2. Platform dimension

Anypoint Platform capabilities: Figure 17, Figure 18

A.2. Business architecture

A.2.1. C4E dimension

1. Organization: Figure 23
2. Operational steering: 3.1.4

A.3. Application architecture

A.3.1. Business outcomes dimension

Requirements realization: Figure 37

A.3.2. Platform and C4E dimension

All these application network-wide application architecture concerns are within the remit of the C4E and are mostly related to Anypoint Platform

Link to Anypoint Exchange and Public (Developer) Portal (Exchange Portal): Figure 54, Figure 52

APIs

This section documents the application architecture of APIs and API implementations removed from the projects they were created in and the products they are used for. This is because API-related assets are application network-scoped - despite being created and used in projects.

For each of the APIs in the application network

1. API data model design decision: 6.3.4, 6.3.11
2. Integration test scenarios: 9.2.6
3. Link to Anypoint Design Center API specification project: Figure 45
4. Link to API Anypoint Exchange entry: Figure 47, which in turn provides access to:
   a. API Console and API Notebook: Figure 53
   b. Anypoint API Manager administration screen: Figure 128
   c. API implementation documentation: Figure 129
   d. Anypoint Runtime Manager dashboards for API implementations: Figure 130
   e. GitHub repository
   f. CI/CD build pipelines

A.3.3. Projects dimension

For each project/product, e.g., "Aggregator Integration" product or "Customer Self-Service App" product

1. Functional requirements: 4.1.1, 4.2.2
2. End-to-end API cooperation: Figure 39, Figure 55, Figure 56
3. Business alignment: Figure 43
4. Asynchronous API invocations: Figure 80
5. Concurrency considerations: 6.4.11

6. Event and message exchanges: Figure 104, Figure 105

A.4. Technology architecture

A.4.1. Platform and C4E dimension

All these application network-wide technology architecture concerns are within the remit of the C4E and are mostly related to Anypoint Platform

1. Anypoint Platform deployment option: Figure 28, Figure 29, Figure 30, Figure 31, Figure 32
2. Anypoint Platform organizational setup: 3.3
3. Common API invocation dashboards and reports: Figure 122
4. Common application component alerts: Figure 126
5. DevOps guidelines: 9.1.2
6. Resilience testing guidelines: 9.2.4
7. API-related guidelines:
   a. API policy enforcement guidelines: 5.3.5
   b. API policy guidelines: 5.3.22, 5.3.23, 5.3.24
   c. API versioning guidelines: 6.2.4
   d. Alerting guidelines: 10.4.2, 10.4.3, 10.4.4

APIs

This section documents the technology architecture of APIs and API implementations removed from the projects they were created in and the products they are used for.

For each of the APIs in the application network

1. API policy enforcement choice
2. API policies: Figure 68, Figure 69, Figure 70, Figure 71
3. Deployment architecture: 7.1.4
4. Approach to HA and scaling: 9.3.1
5. Failure mode analysis: 9.5.1
6. Persistence requirements and statefulness
7. Caching of exposed API: 6.4.8

8. Fault-tolerant API invocation strategies for each dependency: 7.2
9. Events and messages published and consumed, with destinations
10. Alerts: Figure 123, Figure 124, Figure 125

A.4.2. Projects dimension

For each project/product, e.g., "Aggregator Integration" product, "Customer Self-Service App" product

1. Non-functional requirements: 5.1.1, 5.2.1, 5.2.3
2. Approach to meeting NFRs: 5.1.2, 5.2.2, 5.2.4
3. End-to-end API policy cooperation: Figure 72, Figure 73, Figure 74

Appendix B: ArchiMate Notation Cheat Sheets

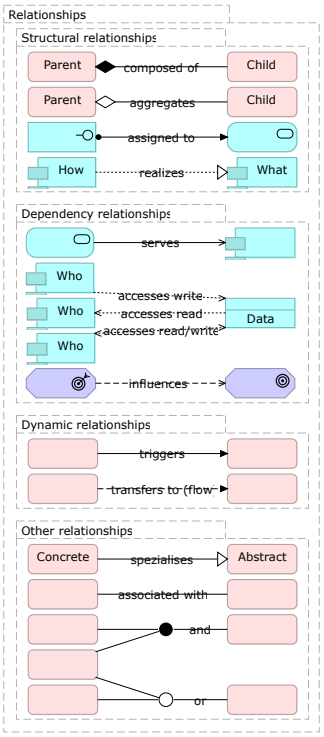

Figure 131. ArchiMate notation for relationships.

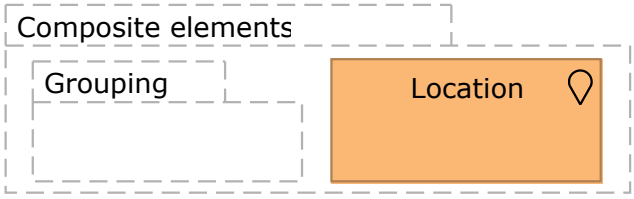

Figure 132. ArchiMate notation for composite elements.

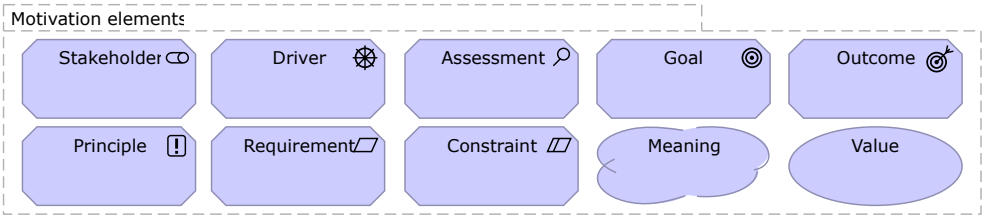

Figure 133. ArchiMate notation for motivation elements.

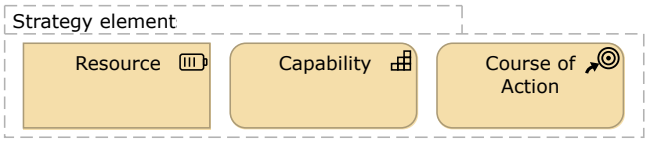

Figure 134. ArchiMate notation for strategy elements.

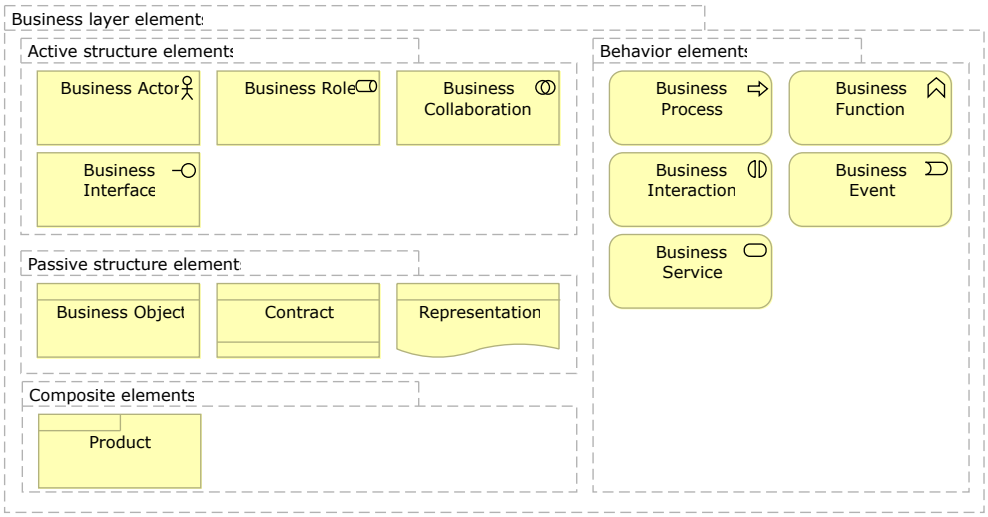

Figure 135. ArchiMate notation for business layer elements.

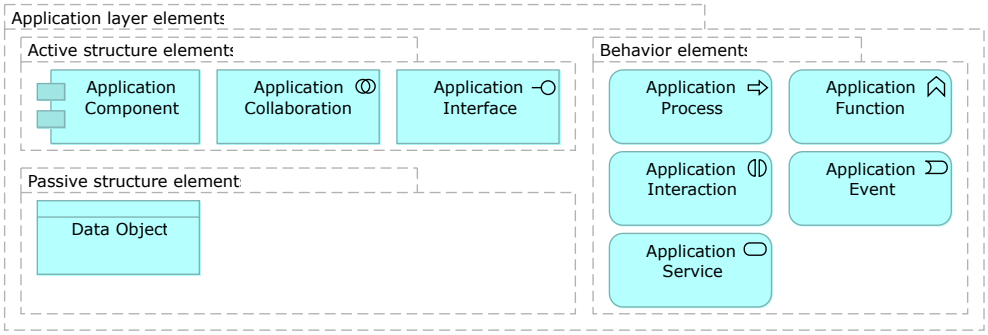

Figure 136. ArchiMate notation for application layer elements.

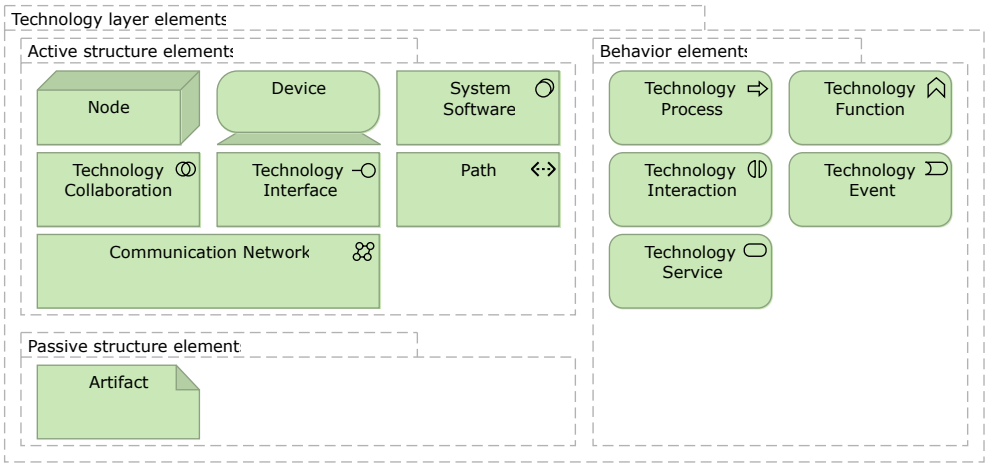

Figure 137. ArchiMate notation for technology layer elements.

Glossary

API

- Application Programming Interface
- A kind of application interface, i.e., a point of access to an application service
- to programmatic clients, i.e., API clients are typically application components
- using HTTP-based protocols, hence restricts the technology interfaces that may realize this application interface to be HTTP-based
- typically with a well-defined business purpose, hence the application service to which this application interface provides access typically realizes a business service
- See Figure 138
- Remarks:
  - The prototypical API is a REST API using JSON-encoded data
  - Non-programmatic interfaces, e.g., web UIs, are not APIs
    - HTTP interfaces using HTML microdata are a corner case, as they are usable for both human and programmatic clients
  - Non-HTTP-based programmatic interfaces are not APIs
    - E.g., Java RMI, CORBA/IIOP, raw TCP/IP interfaces not using HTTP
    - Note that WebSocket interfaces are not APIs by this definition, and are not currently supported by Anypoint Platform
  - HTTP-based programmatic interfaces are APIs even if they don’t use REST or JSON
    - E.g., REST APIs using XML-encoded data, JSON RPC, gRPC, SOAP/HTTP, XML/HTTP, serialized Java objects over HTTP POST, …
    - Note that interfaces using SSE (HTML5 Server-Sent Events) are APIs by this definition, but are not currently supported by Anypoint Platform
  - Interfaces using HTTP/2 are APIs. Also note that HTTP/2 adheres to HTTP/1.x semantics
    - E.g., gRPC
    - But HTTP/2-based APIs are not currently supported by Anypoint Platform
- For instance:
  - Auto policy rating API
    - Is a programmatic application interface
    - Is realized by these HTTP-based technology interfaces
      - Auto policy rating JSON/REST programmatic interface

    - Auto policy rating SOAP programmatic interface
- Provides access to this application service: Auto policy rating
    - which has well-defined business purpose because it
        - realizes this business service: Auto policy rating
        - serves this business service: Policy administration

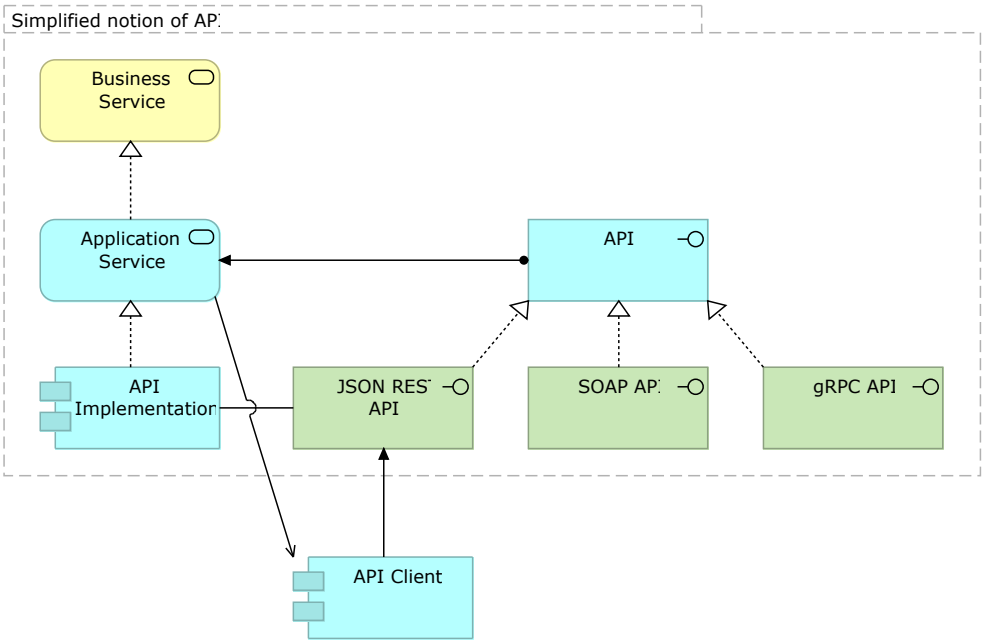

Figure 138. An API is primarily a programmatic application interface but the concept actually combines aspects of Business Architecture, Application Architecture and Technology Architecture

API client

- An application component
- that accesses a service
- by invoking an API of that service - by definition of the term API over HTTP
- See Figure 138

API consumer

- A business role, which is often assigned to an individual
- that develops API clients, i.e., performs the activities necessary for enabling an API client to invoke APIs

API cross-cutting concern

- Functionality that lends itself to implementation in an API policy

API definition

- Synonym for API specification, with API specification being the preferred term

API implementation

- An application component
- that implements the functionality
- exposed by the service
- which is made accessible via one or more APIs - by definition to API clients
- See Figure 138

API interface

- Synonym for API, with API being the preferred term
- Sometimes used in contexts where the simplified notion of API is the dominant one and the interface-aspect of API needs to be addressed in contrast to the implementation-aspect

API-led connectivity

- A style of integration architecture
- prioritizing APIs over other types of programmatic interfaces
- where each API is assigned to one of three tiers: System APIs, Process APIs and Experience APIs

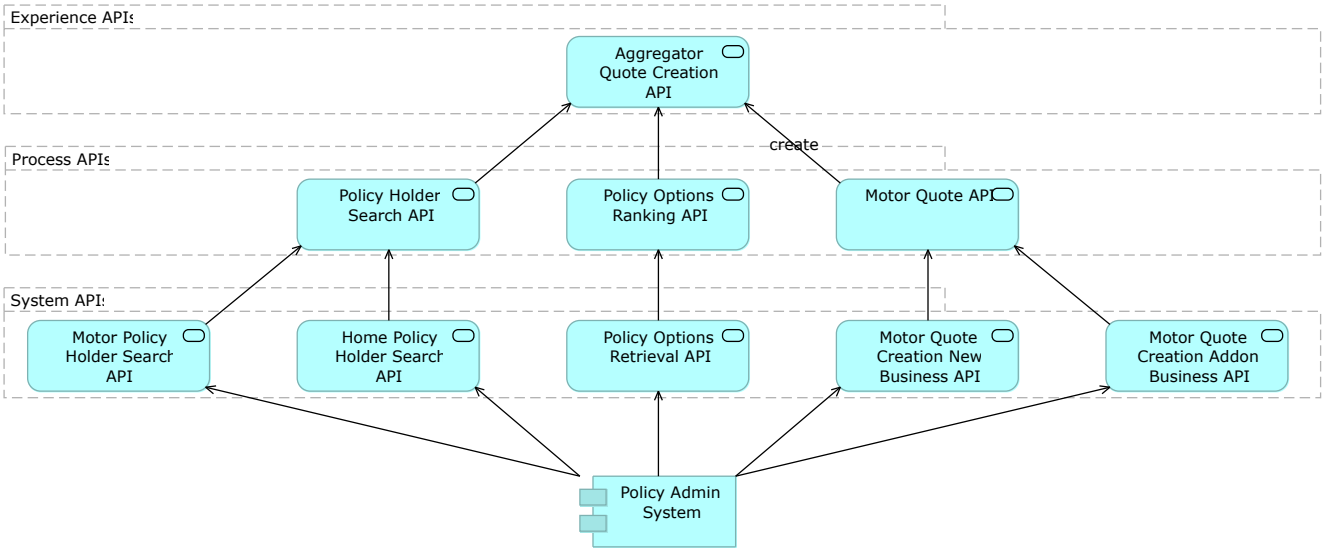

Figure 139. An example of the collaboration of APIs in the three tiers of API-led connectivity. Note the use of the simplified notion of API in this diagram, as lower-level APIs are shown serving higher-level APIs

API policy

- Defines a typically non-functional requirement
- that can be applied to an API
- by injection into the API invocation between an API client and the API endpoint
- without changing the API implementation listening on that API endpoint
- Consists of API policy template (code) and API policy definition (data)
- Technically, in Anypoint API Manager an API policy is applied to an API instance
- For instance:
  - Rate Limiting to 100 requs/s
  - HTTP Basic Authentication enforcement using a given Identity Provider

API policy definition

- A concrete parameterization of a specific API policy
- by supplying values for all (required) parameters given in the API policy template for that API policy
- that is therefore sufficient for applying that API policy to an API endpoint
- For instance:
  - Rate Limiting to 100 requests per second

API endpoint

- The URL at which a specific API implementation listens for requests
- and hence is available for access from API clients.

API policy enforcement

- The technology function that ensures an API policy definition is adhered-to
- Can either be realized by an API proxy or embedded within the API implementation itself.

API policy template

- The code and configuration parameters (but not their values) that implement a specific API policy
- Similar to a class in object-oriented programming
- For instance for Rate Limiting:
  - The Mule application flow implementing the Rate Limiting functionality
  - The declaration of the parameters for number of requests, time period and time unit

API provider

- A business role
- that develops, publishes and operates API implementations and all related assets

API proxy

- A dedicated node that enforces API policies
- by acting as an HTTP proxy between API client and the API implementation at a specific API endpoint.
- API proxies need to be accessed explicitly by the API client in place of the "normal" API implementation, via the same API.

API proxy endpoint

- The API endpoint on which a specific API proxy listens for requests.

API specification

- A formal, machine-readable definition of the technology interface of an API
- Sufficiently accurate for developing API clients and API implementations for that API
- For instance:
  - RAML definition
  - WSDL document

- OAS/Swagger specification

Application (app)

- Used for API clients that are registered with Anypoint Platform as clients to at least one API managed by Anypoint Platform
- In this context synonym for API client, with API client being the preferred term

Application interface

- Point of access to an Application Service
- exposing that Service to clients which may be humans or Application Components
- For instance:
  - Auto policy rating programmatic interface
    - Provides access to this Application Service: Auto policy rating
  - Auto claim notification self-service UI
    - Provides access to this Application Service: Auto claim notification
  - Bank reconciliation batch interface
    - Provides access to this Application Service: Bank reconciliation

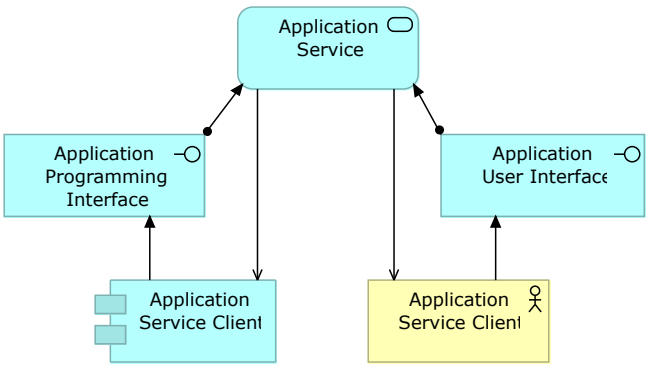

Figure 140. Business actor and application component accessing application service via different application interfaces

Application network

- The state of an Enterprise Architecture
- emerging from the application of API-led connectivity
- that fosters governance, discoverability, consumability and reuse of the involved APIs and related assets

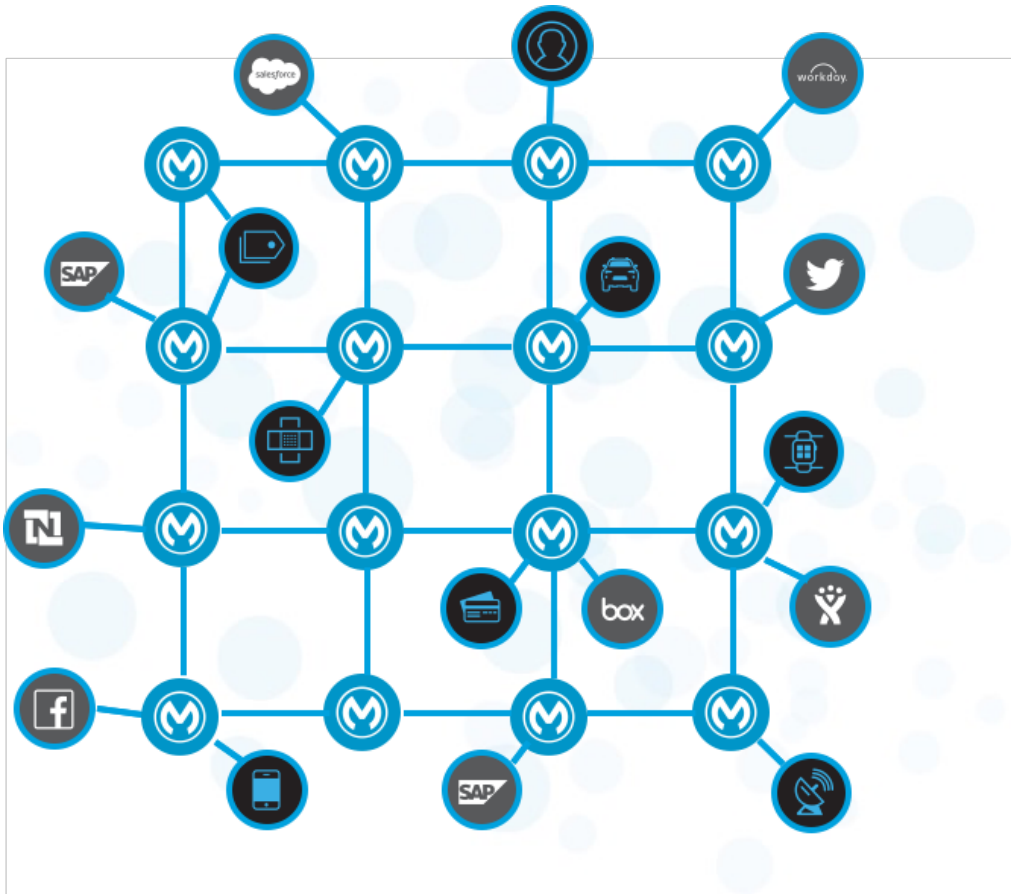

Figure 141. A visualization of the application network concept

Application service

- Exposes application functionality
- such as that performed by an Application Component
- through one or more application interfaces
- May (should) completely realize or at least serve a Business Service
- May serve other (more coarse-grained) Application Services
- For instance:
  - Auto policy rating
    - Realizes this Business Service: Auto policy rating
    - Serves this Business Service: Policy administration
  - Auto claim notification
    - Serves this Business Service: Claim management
  - Bank reconciliation
    - Realizes this Business Service: Bank reconciliation

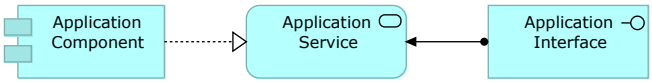

Figure 142. Application service

Business service

- Exposes business functionality
- such as that performed by a Business Actor
- through one or more business interfaces
- Has meaning and value on a business level
- May serve other (more coarse-grained) business services
- For instance:
  - Policy administration
  - Auto policy rating
    - A fine-grained Business Service that serves the coarse-grained Business Service Policy administration
  - Claim management
  - Bank reconciliation

Figure 143. Business service

Embedded API policy enforcement

- An implementation approach of API policy enforcement
- where this functionality is embedded in (co-located with) the API implementation component
- rather than being isolated in a separate API proxy
- Recent versions of the Mule runtime provide this feature under the "API Gateway" entitlement

CQRS

- Command Query Responsibility Segregation
- The usage of different models for reading from data (*queries*) and writing to data (*commands*)

Event-Driven Architecture

- An architectural style
- defined by the asynchronous exchange of events
- between application components.
- Hence a form of message-driven architecture
- where the exchanged messages are (or describe) events
- and the message exchange pattern is typically publish-subscribe (i.e., potentially many consumers per event).

Event Sourcing

- An approach to data persistence that keeps persistent state as a series of events rather than just a snapshot of the current state
- Often combined with CQRS

Interface

- Point of access to Service
- exposing that Service to Service clients
- Only if needed differentiate between business interfaces, application interface and technology interface

RAML

- REST API Modeling Language
- YAML-based language for the machine- and human-readable definition of APIs that embody most or all of the principles of REST, which are:
  - Uniform interface, stateless, cacheable, client-server, layered system, code on demand (optional)
  - Adherency to the HTTP specification in its usage of HTTP methods, HTTP response status codes, HTTP request and response headers, etc.

RAML definition

- An API specification expressed in RAML
- comprising one main RAML document
- and optional included
  - RAML fragment documents
  - XSD and JSON-Schema documents
  - examples, etc.

REST

- Representational State Transfer
- an architectural style characterized by the adherence to 6 constraints, namely
  - Uniform interface
  - Stateless
  - Cacheable
  - Client-server
  - Layered system
  - Code on demand (optional)

REST API

- An API that follows REST conventions
- and therefore adheres to the HTTP specification in its usage of HTTP methods, HTTP response status codes, HTTP request and response headers, etc.

Service

- Explicitly defined exposed behavior
- exposes functionality to Service clients
- who access it through one or more interfaces
- Only if needed differentiate between Business Service, Application Service and Technology Service
- May serve other (more coarse-grained) Services of the same kind
- For instance:
  - Policy administration (a Business Service)
  - Auto policy rating (an Application Service)
  - HTTP request throttling (a Technology Service)

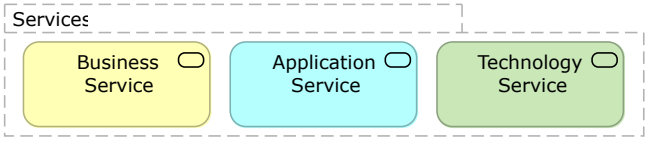

Figure 144. Services

Technology interface

- Point of access to a Technology Service
- May realize an application interface

- Specifies data formats, parameter types, protocols, etc.
- For instance:
  - Auto policy rating JSON/REST programmatic interface
    - Realizes this application interface: Auto policy rating programmatic interface
  - Auto policy rating SOAP programmatic interface
    - Realizes this application interface: Auto policy rating programmatic interface
  - Auto claim notification self-service web UI
    - Realizes this application interface: Auto claim notification self-service UI
  - Auto claim notification self-service mobile app UI
    - Realizes this application interface: Auto claim notification self-service UI
  - Bank reconciliation transaction file import
    - Realizes this application interface: Bank reconciliation batch interface
  - HTTP request throttling proxy endpoint
    - Provides access to this Technology Service: HTTP request throttling

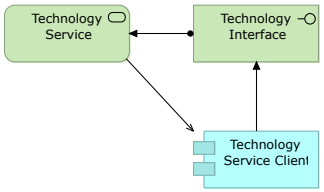

Figure 145. Application component accessing technology service via technology interface

Technology service

- Exposes technology functionality
- such as that performed by a Node or Device or System Software
- through one or more technology interfaces
- May serve Application Components
- May serve other (more coarse-grained) Technology Services
- For instance:
  - Automatic restart
    - Serves Application Components that must immediately resume operation after a failure
  - Persistent message exchange
    - Serves Application Components that require guaranteed message delivery
  - HTTP request throttling

- Serves Application Components that expose services that must be protected from request overload (DoS attacks)

Figure 146. Technology service

Web Service

- Synonym for API, with API being the preferred term

Bibliography

- [Ref1] MuleSoft, "Mule Runtime", https://docs.mulesoft.com/mule-user-guide. 2018.
- [Ref2] MuleSoft, "Design Center", https://docs.mulesoft.com/design-center. 2018.
- [Ref3] MuleSoft, "API Manager", https://docs.mulesoft.com/api-manager. 2018.
- [Ref4] MuleSoft, "Anypoint Exchange", https://docs.mulesoft.com/anypoint-exchange. 2018.
- [Ref5] MuleSoft, "Runtime Manager", https://docs.mulesoft.com/runtime-manager. 2018.
- [Ref6] MuleSoft, "Access Management", https://docs.mulesoft.com/access-management. 2018.
- [Ref7] MuleSoft, "Anypoint Analytics", https://docs.mulesoft.com/api-manager/analytics-concept. 2018.
- [Ref8] MuleSoft, "Anypoint MQ", https://docs.mulesoft.com/anypoint-mq. 2018.
- [Ref9] MuleSoft, "About Anypoint Platform Private Cloud Edition", https://docs.mulesoft.com/anypoint-private-cloud. 2018.
- [Ref10] MuleSoft, "About Anypoint Platform for Pivotal Cloud Foundry", https://docs.mulesoft.com/anypoint-platform-pcf/v/1.6. 2018.
- [Ref11] S.J. Fowler, *Production-Ready Microservices*. Sebastopol, CA: O'Reilly Media, 2016.
- [Ref12] M.T. Nygard, *Release It! Second Edition*. Raleigh, NC: Pragmatic Bookshelf, 2018.
- [Ref13] V. Vernon, *Domain-Driven Design Distilled*. Boston, MA: Addison-Wesley, 2016.
- [Ref14] ThoughtWorks, "Inverse Conway Maneuver", https://www.thoughtworks.com/radar/techniques/inverse-conway-maneuver. 2018.
- [Ref15] MuleSoft, "Anypoint Platform Performance", https://www.mulesoft.com/lp/whitepaper/api/anypoint-platform-performance

Version History

2018-05-02	v1.4	added 3.2.1 and EU vs US control plane; added 3.2.10, 7.1.13, 7.1.14; improved and updated CloudHub Technology Architecture discussion in 7.1.4, 7.1.6, 7.1.7, 7.1.8, 7.1.9, 7.1.10, 7.1.11, 7.1.12; added [Ref15], 5.3.29, 7.1.5; added roles to 3.1.2; added Anypoint Runtime Fabric to 3.2.2 and removed OpenShift; added SAML tokens to 3.3.3; refined 4.2.5, 5.3.5, 5.3.6; added naming conventions to 4.3.1; added 5.3.9; refined 5.3.8; improved Figure 63; added OIDC DCR to 5.3.12; refined 6.3.5, added solution to 6.3.6, refined solution of 6.3.9; added solution to 6.4.5, 6.4.8; moved and improved 8.2.6; added Figure 108; consolidated categories in 9.2; added autoscaling to 7.1.4 and 9.3.1; added 9.3.4; reordered and refined 10.4; shortened *Wrapping-Up Anypoint Platform Architecture: Application Networks*; added API descriptions to 4.2.4, 4.4.2, 4.4.3; refined 5.3.6
2018-02-16	v1.3	added Spike Control API policy to 5.3; client secret optional in 5.3.19; refined API policy guidelines in 5.3.22, 5.3.23, 5.3.24 and 5.3.25; added 4.2.5; updated Bibliography; refined 6.3
2018-01-23	v1.2	renamed course to "Application Networks"; updated Table 1; OpenID Connect DCR now supported; updated to Crowd2/Nov-release of Anypoint Platform; shortened API names to EAPI, PAPI and SAPI; fixed wording on IP whitelisting between API tiers
2017-11-01	v1.1	refinements to align with slides; improved 7.2; updated 9.1.1; shortened course objectives; updated 7.1.4; added Bibliography; glossary now self-contained; added Appendix A
2017-09-29	beta2	added missing queue to Figure 105; tier instead of layer for API-led connectivity; added missing images; numerous little refinements; improved layout
2017-09-06	beta	for internal review
2017-08-27	alpha (v1.0)	basis of first public delivery