IBM Cloud – Cloud-Native Programming Model – Java MicroProfile Focus

March 24th, 2019 – If you are reading this in pdf form, this is the **1.05 version** of this document, for which we are taking active feedback. Provide feedback directly to Eric Herness (herness@us.ibm.com). If you did not just download the PDF, look for an updated one here: http://ibm.biz/ProgModel

Note: This is the Java MicroProfile version. We are actively assessing whether or not to augment this material for the Spring framework and for other languages that are used for cloud native development.

Table of Contents

Acknowledgements

Thanks to all of the authors, editors and reviewers that have contributed to this effort.

Forward

January 31, 2019, on the day of publishing the 1.0 version

The speed of technology evolution has never been more rapid. The need to adopt and harness new technology to deliver business value in almost any industry is a fixture in most strategies. Early adopters enjoy a first mover's advantage, but the life span of that advantage seems to shrink with each passing decade.

While all this change can be overwhelming, it is also the case that many of the foundational concepts brought to use by past technology breakthroughs remain important. That is the case in our current era as it relates to software engineering. Distributed computing is still important and in fact underpins all cloud native solutions. Java remains an important language for capturing business logic, but in the current era is joined by other programming languages. Containers are not a new concept, but they are new to their central role in architecting software systems. We also still think about network, compute and storage but I would assert that today's higher bandwidth and more reliable networks do influence how systems are built. Similar statements can be made about storage and compute.

Harnessing the raw materials available from technology shifts or new concepts are often very challenging for the early adopters. Early adopters however, seem to thrive on the challenge and somehow get the funding and support to grind through the details and deliver new and improved value. What they learn as pioneers also shapes and evolves the technologies themselves and paves a clearer more navigable path for those that follow.

For those about to embark on a cloud native journey, it would be easy for us to point to the vast array of capabilities available and call it a day. We could suggest that unique and expensive projects to stand up a PaaS that supports cloud native development be initiated immediately. But instead of just that, we have created this programming model definition, in hopes that it can be a starting point that allows adoption to begin now, rather than waiting a few months or more for a project that will setup and define a technology approach.

It is fair to ask, "Why now?" Now is a good time to provide this programming model because we have programming languages providing frameworks for microservices that are ready to use. We have Kubernetes that has emerged as the underpinnings for container orchestration. We have Istio as a service mesh which is ready to blend together with language frameworks and the underlying Kubernetes platform. Finally, we have underlying platforms, tools and middleware that are ready to support commercial enterprise scale use of the cloud native approach. Adopters no longer have to construct their own path through the technologies and their own Platform as a Service (PaaS) to support the programming model.

Eric Herness
Hybrid Cloud CTO
IBM Fellow

About this Book

This book defines a programming model for creating new Cloud Native applications using microservices and event-driven architectures. Recommended best practices and code examples are included to show the typical capabilities found in cloud native applications.

Who should read this book?

This book is primarily intended for application developers who are using the IBM Cloud offered capabilities and targeting IBM Cloud platforms including IBM Cloud Private (ICP) and IBM Cloud Kubernetes Service (IKS). After reading this book, developers should have the background needed to start constructing cloud native solutions that run on and leverage services that are part of our broader IBM Cloud platform.

This book is intended for those that have some level of programming skill, some awareness of Docker and some awareness of Kubernetes. Docker and Kubernetes are fundamental building blocks of the programming model but are not covered in detail. References for additional background on these topics will be included throughout the book.

This book will also be of interest to solution architects and enterprise architects looking to get an appreciation for the various infrastructure, platform and middleware features that can be and should be leveraged by developers creating microservices based solutions.

While this book is primarily for developers, there are also some tips and suggestions for the operations team that will deploy and manage such applications in production. For a book more focused on this role, we recommend *Kubernetes in the Enterprise: Deploying and Operating Production Applications on Kubernetes in Hybrid Cloud Environments*, available at http://ibm.biz/BdYA4i. Note this book also uses our IBM Stock Trader as its example.

Conventions used in this book

- Code snippets and snippets from configuration files are in `Courier New`, like this:

```
FROM websphere-liberty:microProfile2
```

Currency and Applicability of Information

This book covers:

- Open Liberty and WebSphere Liberty 18.0.0.4 and later, including Java EE 8 (Jakarta EE 8), MicroProfile 2.0 and Spring Boot 2.0.
    - This book does not explicitly state which specific features apply to Open Liberty and which do not. We will make every effort to note those that do not. For the most part, most of what is described here will work on Open Liberty, as well.
- "Future" tags or boxes will be used in front of any sections or paragraphs containing information about upcoming, unreleased, or experimental capabilities. Future items could be draft specifications, or issues that have been resolved but are not yet available in an official release.

This book should be available at: http://ibm.biz/ProgModel. Look for updates here or to download your own copy.

Scope: Items not Covered

This book does not:

- specifically cover the installation, configuration and management of a Kubernetes platform such as IBM Cloud Private.
- specifically cover all of the details of the DevOps approach that supports microservices based projects. Certain elements are described as they relate to and impact the artifacts of the programming model.
- specifically cover all of the details on every approach possible for creating cloud native solutions that run on the IBM Cloud platform.
- offer thorough and complete guidance on application modernization. A brief overview is provided, however, so as to demonstrate the relationship between cloud native applications and modernized applications as those modernized applications evolve.
- cover all possible frameworks and approaches that could be used to build cloud native applications.

Introduction

Overview

The world of programming distributed systems has reached a new point. Technologies like Java have evolved. New languages have appeared on the scene to offer choices not available before. Technologies like Kubernetes and Helm are taking on important roles in how new applications are programmed and managed. The developer must bring this new set of technologies together to meet the requirements of the day.

One of the watchwords of this new era is microservices. Microservices is a concept and an approach, but there is no specific language or environment into which microservices must be written and run. Mostly what we get from the world of microservices is a clearer and more precise set of guidelines for building distributed systems than would have existed in the past generations of distributed computing. Many good things are coming from this evolution. This all needs to inspire a true programming model.

The “IBM Cloud – Cloud-Native Programming Model” is a specific programming model that can be used to create net new solutions where the components of that solution are written using microservices. A programming model for this purpose calls out specific technologies at the programming language level and at the platform level. This book shows how to blend those together to get the best resultant solution. Tradeoffs and options will be provided along the way, supported by details that point out advantages and disadvantages of various approaches. In most cases, this will remain a very technical explanation of a programming model. Tradeoffs will be elaborated upon looking at dimensions such as DevOps velocity, testability, resilience and maintainability.

What is a programming model? This is a question that often comes up. For purposes of this book the programming model is a specific and documented approach to creating solutions. A programming model is described by the precise collection of artifacts that bring the programming model to life. Relationships and linkages between the artifacts and the content inside of these artifacts are essential to fully understanding and leveraging the programming model. This programming model covers specific approaches that should be leveraged to do basic things like exposing REST APIs, managing units of work, leveraging data services and leveraging messaging services.

Creating microservices involves applying a programming model. This book suggests that this programming model, given the context of targeting IBM Cloud Private and IBM Cloud Kubernetes Service, is a combination of both polyglot and platform capabilities combined with language specific implementation approaches. Whenever appropriate, and where there are alternatives in place, this book will prefer to use a polyglot feature in favor of a language specific feature. However, in cases where the language specific feature is richer or is an accelerator to implementing a polyglot definition, then the language specific feature will be preferred or at a minimum explained in this book.

To complement the programming model basics, explanations of our tools that support the programming model will also be outlined as appropriate. While a programming model generally is described at a level that does not require any specialized tooling, commercial scale use of a programming model depends on a development platform that supports the programming model in a first-class way. The tools section will also describe how a devOps pipeline should work, so that when you commit updates to one of your microservices, those changes will get deployed into your cloud environment.

The rest of this book is organized in way that a reader can go from end to end, or can selectively look at concepts, specific programming language approaches or just reference sections that might be of interest for any given interest in programming model that might arise.

The platform for which this programming model is intended has some basic capabilities that are good to understand. These influence later sections in many ways. This comes first. Then there is an overall view of what it takes to really build and maintain systems built from these programming models. This development flow and related considerations come after the basics of the target execution environment.

Then the featured programming models are introduced, each in succession. This sets additional context around the programming language and relates frameworks that make up a particular approach.

From that set of basics, the primary programming model concepts are introduced one at a time, immediately followed by the framework and language specifics necessary for a developer to get started embracing those concepts using the selected style. Configuration, logging, tracing,

metrics, creating APIs, microservice to microservice communications, security, fault tolerance, readiness checks and managing service interaction are all tackled in succession.

Language and framework specific approaches might actually not be so language specific after all, if a more polyglot approach is recommended. Therefore, these prior sections specific to language and framework are followed up with a section on Istio. Istio features are polyglot in that they can be used with any programming language. Using Istio features as the preferred approach means that information in the Istio section is often pointed to directly from the language and framework specific sections. A reader can go get them all at once, or jump forward while navigating through a specific language and framework approach. They are certainly very important and a featured item in this overall programming model book. Therefore, they are in the heart of the book itself.

After this, additional features that may be of interest are elaborated upon, continuing to follow the approach of describing the concept and any language and framework specific guidance that might exist. Look for this topic list to fill out the basic programming model. These include:

1. Application Modernization and Cloud Native – This provides specific cloud native programming model techniques and approaches that might be necessary to employ for those involved in evolving existing Java applications.
2. Advanced Asynchronous capabilities – In these sections, patterns such as event driven and reactive are covered.
3. Higher level application patterns – This set of topics builds upon the basics put in place earlier suggesting compositions of these basics to form consistent application architectures.
4. Additional Features – These are features that are not required, but that may be of interest in certain situations. Some are language specific where there is a favored approach in the polyglot section.
5. Tooling the Cloud Native Programming model

As appropriate, there will be a brief introduction to each topic, followed by an explanation of the programming model elements involved. That element is then demonstrated in the context of a solution using an example. Relationships and linkages to other parts of the programming model are provided.

Stock Trader Example

This book will often use a sample application called "Stock Trader" to highlight various elements of the programming model. This is a simple microservices based solution which provides a stock trading application. It runs on both IBM Cloud Private (ICP) and IBM Kubernetes Service (IKS). An introductory article is available at https://developer.ibm.com/code/2018/07/23/introducing-stocktrader/.

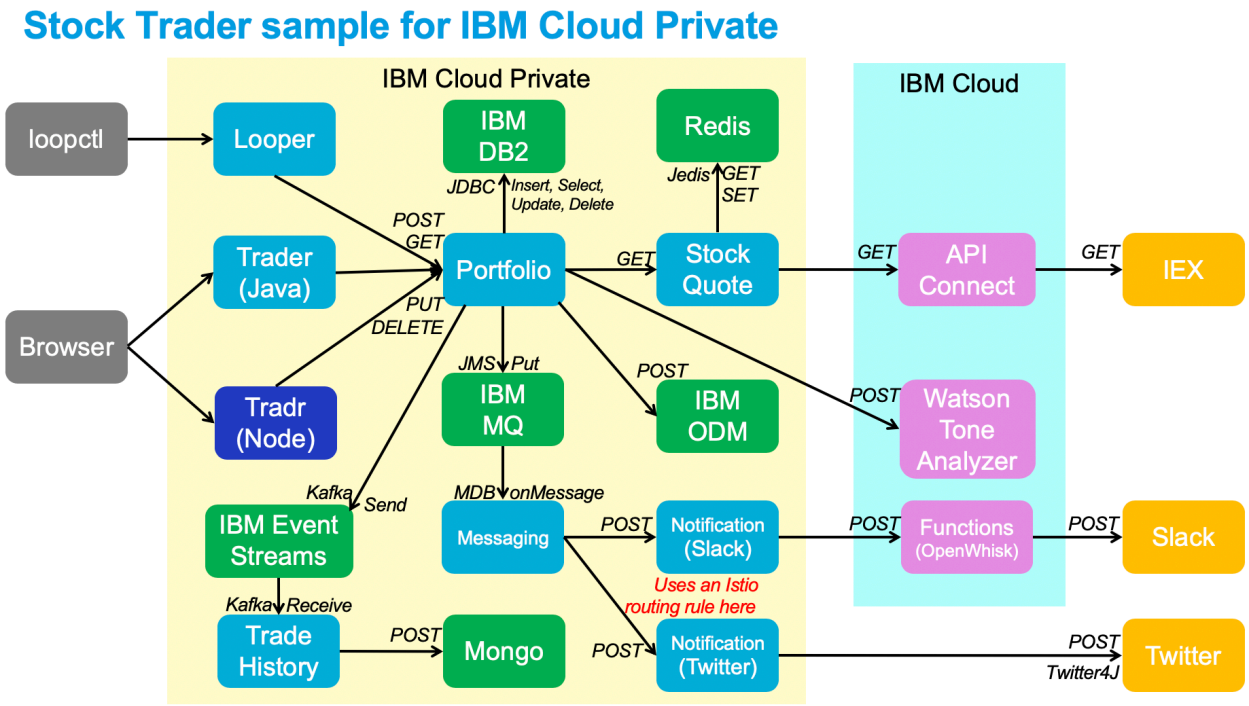


Figure 1 - Stock Trader Example Overview

Stock Trader started life as a demonstration that mixed together new microservices, existing middleware, and new cloud capabilities from both on-premises and public cloud platforms. The core codebase exists at https://github.com/IBMStockTrader, with variations published in other places, as well. As an evolving work, this book and the public repository may not always be in sync. The examples cited here should be useful nonetheless.

Stock Trader uses a lot of Liberty-based Java services, namely `Looper`, `Trader (Java)`, `Portfolio`, `Messaging`, `Notification (Slack)`, `Notification (Twitter)`, `Stock Quote` and `Trade History`. Node-based services are also present, via `Tradr (Node)`, which is an alternate UI, plus the API implementation in **API Connect**, and an action sequence in **IBM Cloud Functions**.

Introduction to Portfolio Microservice

At the core of the Stock Trader solution is the Portfolio microservice:

1. It is a simple Java class using JAX-RS: `public class PortfolioService extends javax.ws.rs.core.Application`
2. It exposes a REST API so that the user interfaces can use the service.
3. It calls REST services to get stock quotes, loyalty level information and sentiment insights.
4. It also calls non-REST services. Specifically, it uses JDBC with IBM DB2, JMS with IBM MQ, and Kafka with IBM Event Streams.

These characteristics allow demonstration of several key programming model concepts. The source for the portfolio microservice is here: `https://github.com/IBMStockTrader/portfolio`.

The Portfolio microservice uses the WebSphere Liberty `microProfile2` docker image as a base to provide capabilities that are a part of the MicroProfile 2.0 platform (such as JAX-RS) as you can see from the first line of the service's `Dockerfile`:

```
FROM websphere-liberty:microProfile2
```

The `Dockerfile` describes the contents of a WebSphere Liberty docker image customized to run the Portfolio microservice. In addition to the binaries for the server itself, the custom image contains the server configuration (in `server.xml`) and business logic (the compiled application). `Dockerfile`s are used to build docker images. When a docker image is run, it creates a container instance. In this case, the running container will be an instance of the Portfolio microservice.

We will visit deployment artifacts in more detail later, but it is worth pointing out the `deploy.yaml` file in the `manifests` directory. This is one example of how a microservice will be deployed into and live in the Kubernetes world. There are key elements in this file, such as:

- `kind: Deployment` – This describes how docker images are deployed into Kubernetes pods
- `kind: Service` – This describes the exposed capability as a service
- `kind: Ingress` – This ensures that external clients can access the Portfolio microservice

We might use `kubectl` to apply this configuration directly to the cluster, or we might include this information in helm chart templates.

More details on application runtimes, REST APIs, Docker files, and Kubernetes .yaml files will be provided in upcoming sections. For now, it is sufficient to understand that creating a microservice will involve creation of artifacts that describe:

- the actual business logic
- the runtime that the business logic executes in
- the container execution environment that surrounds and encapsulates that execution environment
- the container orchestration guidelines necessary for the microservice to live in an ecosystem with lots of containers that are all working together
- the required information necessary for the microservice to be observed, probed, measured and managed.

Note that there is also an alternative version of Portfolio, implemented in Java Spring rather than Java MicroProfile, at `https://github.com/IBMStockTrader/portfolio-spring`.

Programming Model Basic Underpinnings

This section will introduce the programming language independent concepts and provide some context as to the contributions that these basic underpinnings bring to the overall cloud native programming model. The container execution platform which includes the role of Kubernetes, Docker and Istio provides the key underpinnings for cloud native development and microservice execution. JSON is another foundation to the programming model that is covered in this section.

Specific underpinnings related to specific programming languages and frameworks will come with each major section that focuses on a particular language and approach or will be laid out in proper detail on the polyglot capabilities section that comes later. These are important concepts put here for those that may not be familiar with Docker, Kubernetes, Helm and Istio. If you are familiar with] all of these concepts, feel free to skip this chapter.

Docker and Containers

A container, as introduced by Docker, is a virtualized operating system instance dedicated to running an individual application process or set of related child processes.

The state of an individual container can be captured as a container image, and later reconstituted in another execution environment. Such container images are self-encompassed, the image containing all application specific binaries other than the shared operating kernel which is provided by the host.

Docker provides an engine to execute container images. The Open Container Initiative (OCI) promotes a community that ensures consistency across all container execution engines. IBM container runtimes are running docker engines today and are moving towards those that are aligned with containerd. Containerd is an OCI compliant core container runtime. It should also be noted that this approach and strategy applies to leveraging IBM Cloud Kubernetes Service as well as IBM Cloud Private. On IBM Cloud Private this approach is consistent for the Intel, Power and IBM Z architectures.

Based upon the idea that a container's state can be captured in a container image which is later instantiated and perhaps customized further before being captured into another independent image, Docker provides a build system and build descriptor known as a `Dockerfile`. A `Dockerfile` describes a base image along with a set of image customization commands which are run sequentially to assemble a docker image. Images are built using `docker build`. Docker images layer atop one another: a Docker image is built `FROM` another.

This is an example of the `Dockerfile` used to build the Portfolio microservice in the IBM StockTrader application. The first line is important, as it specifies the base image:

```
FROM websphere-liberty:microProfile2
COPY src/main/liberty/config /config/
COPY target/portfolio-1.0-SNAPSHOT.war
/config/apps/Portfolio.war
```

```
RUN installUtility install --acceptLicense defaultServer
```

Docker also provides *registries* to store Docker images. Depending on how your CI/CD pipeline works, you might use images from the public Docker Hub registry. Some organizations might require developers to use images from a custom registry containing a curated collection of Docker images approved for use by their IT department.

Kubernetes

IBM's cloud platforms leverage Kubernetes for container orchestration. Therefore, in addition knowing Docker basics, it is important for developers to be familiar with Kubernetes fundamentals, including basic commands and deployment artifacts. Some important Kubernetes concepts found throughout this guide include:

Kubernetes pod – a localized group of containers that are deployed together as a single unit. Pods are immutable, requiring the original pod be replaced in order to make modifications to various attributes of the pod. In a typical application, there will be one container with core business logic and optionally additional pods that provide platform capabilities at the granularity level of the pod.

Kubernetes deployments – Kubernetes deployments describe a repeatable template for a stateless pod, adding a dimension of scale to the pod concept. Additionally, the templated definition can be updated and the underlying pod instances replaced. A Kubernetes Deployment configuration is watched by a Kubernetes Deployment Controller that ensures that the declared number of pods for a given Deployment are maintained. These will show up as `kind: Deployment` in .yaml files.

Kubernetes Service – services abstract a set of relatively volatile pod addresses with a single well known address. Services can exist on the cluster private network only, or be exposed externally in which case they typically deploy a cloud provider specific load balancer. These will show up as `kind: Service` in .yaml files.

Kubernetes Ingress – while Kubernetes Services expose an individual network address on either the private or public network and directly routes TCP traffic directly to the backend pods, ingress provides the ability to share a single network address with multiple services via virtual hosting or context based routing, and can also perform network connection management activities like TLS termination. These will show up as `kind: Ingress` in .yaml files.

Kubernetes Secrets – store sensitive information for pod runtime use and separate this deployment specific information from the container image or orchestration. Secrets can be exposed to pods at runtime through either environment variables or virtual file system mounts. Without secrets, sensitive data would need to be stored in either the container image or the orchestration, both of which create more opportunities for accidental exposure or unintended access.

Kubernetes ConfigMaps -- play a similar role to Secrets in that they separate deployment specific information from the container orchestration. However, ConfigMaps are general purpose configuration structures. They are used to bind information such as command-line arguments, environment variables and other configuration artifacts to your Pod's containers and system components at runtime.

Secrets and Config Maps are covered in more detail later in the document.

All resources defined within the Kubernetes resource model which can be configured either through the RESTful API or through configuration files submitted through the `kubectl` command line. For further details about the resource model, see: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/. For more information about the `kubectl` command line tool, visit https://kubernetes.io/docs/reference/kubectl/overview/
For general details regarding Kubernetes and IBM's Kubernetes capabilities, see https://kubernetes.io/docs/tutorials/kubernetes-basics/. More details will come later as we dig into each of the main topics of the programming model.

Helm

Helm is a package manager that provides an easy way to find, share, and use software built for Kubernetes. It removes complexity from configuration and deployment, and enables greater developer productivity.

Helm addresses a common user need of deploying applications to Kubernetes by making their configurations reusable. Helm's packaging format, called charts, is a collection of files that describe a related set of Kubernetes resources.

From a programming model perspective, there are files and their contents to be considered. Charts are created as .yaml files laid out in a particular directory tree. While not strictly a mandatory part of the programming model, they are pragmatically valuable and important when considering packaging and to streamline automation around provisioning.

Istio Service Mesh

Istio (https://istio.io/) is an open source extension to orchestrators like Kubernetes for managing and securing microservices. Istio provides a declarative way to manage and control communication between services.

Istio operates using a sidecar model: A sidecar (an Envoy proxy) is a separate process that sits alongside your application. The sidecar manages all communication to and from your service, which allows Istio to apply a common level of capability to all services independent of the programming language or framework the particular service was implemented. In effect, Istio provides a mechanism to centrally configure routing and security policies, while having those policies applied (via sidecars) in a decentralized way.

We recommend using capabilities provided by Istio instead of the similar capabilities provided by individual programming languages or frameworks. Load balancing and other routing policies are more consistently defined, managed, and enforced by the infrastructure, as an example.

In some cases, as with distributed tracing, Istio and application-level libraries are complementary. Using complementary features together can improve operations. In the case of distributed tracing, Istio can only ensure that trace headers are present; application libraries provide the important context about the relationships between requests. Your understanding of the system as a whole improves when both Istio and supporting libraries or framework libraries work in concert.

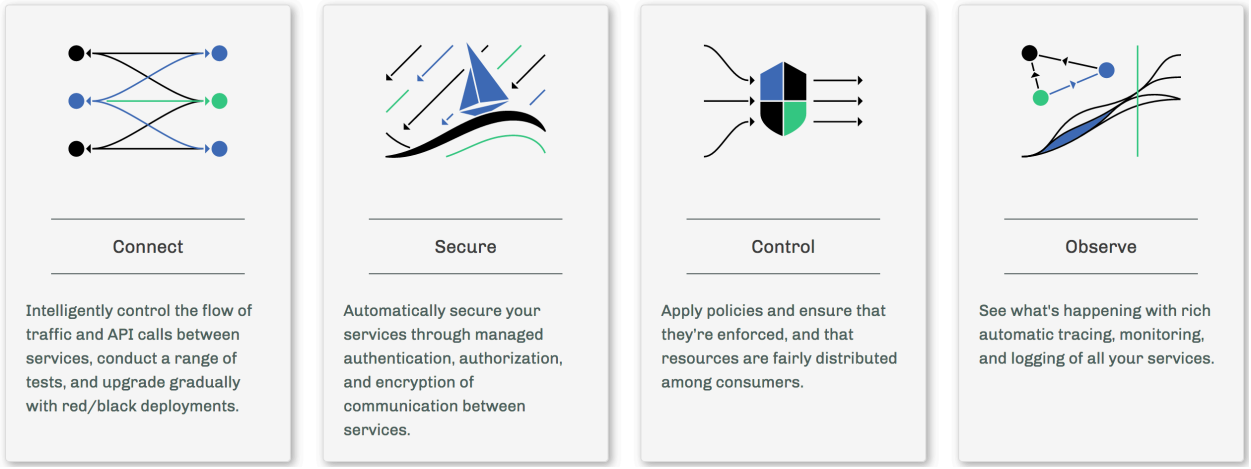
Care will be taken in the upcoming sections to describe the mismatches and overlaps between the capabilities provided by Istio, and those provided by libraries or frameworks. Further, in cases where there are complementary capabilities, these will also be described in detail, including tips and techniques to ensure policies are not enforced more than once and that any precedence rules are explained.

At the highest level, Istio extends the Kubernetes platform, providing additional management concepts, visibility, and security. As they say at https://istio.io/docs/concepts/what-is-istio/:

> *"At a high level, Istio helps reduce the complexity of deployments, and eases the strain on your development teams. It is a completely open source service mesh that layers transparently onto existing distributed applications. It is also a platform, including APIs that let it integrate into any logging platform, or telemetry or policy system. Istio's diverse feature set lets you successfully, and efficiently, run a distributed microservice architecture, and provides a uniform way to secure, connect, and monitor microservices."*

They go on to define a "service mesh" as follows:

> *"The term service mesh is used to describe the network of microservices that make up applications and the interactions between them. As a service mesh grows in size and complexity, it can become harder to understand and manage. Its requirements can include discovery, load balancing, failure recovery, metrics, and monitoring. A service mesh also often has more complex operational requirements, like A/B testing, canary releases, rate limiting, access control, and end-to-end authentication. Istio provides behavioral insights and operational control over the service mesh as a whole, offering a complete solution to satisfy the diverse requirements of microservice applications."*



Docker, Kubernetes and Istio are implemented once in the platform; other concepts have language specific implementations. REST and JSON are examples and are introduced next.

REST and JSON

Polyglot applications are only possible with language-agnostic protocols. Representational State Transfer (REST) architecture patterns define guidelines for creating uniform interfaces that separate the on-the-wire data representation from the implementation of the service.

JavaScript Object Notation (JSON) has emerged in microservices architectures as the wire format of choice for text-based data, displacing Extensible Markup Language (XML) with its comparative simplicity and conciseness. Binary serialization frameworks do exist, such as Apache Avro, Apache Thrift, or Google Protocol Buffers. However, working with these protocols is more complicated, and should be deferred until the interactions between services are well understood.

JSON uses attribute-value pairs to represent data objects. Let's do a quick comparison of how XML and JSON would serialize an Employee document containing the fields "name", "serial", "title", and a nested Address object with fields "office", "street", "city", "state", and "zip":

In XML, you'd have the following serialization for me, with my work address:

```
<employee>
  <name>John Alcorn</name>
  <serial>228264</serial>
  <title>Senior Software Engineer</title>
  <address>
    <office>501-B101</office>
    <street>4205 S Miami Blvd</street>
    <city>Durham</city>
    <state>NC</state>
    <zip>27703</zip>
  </address>
</employee>
```

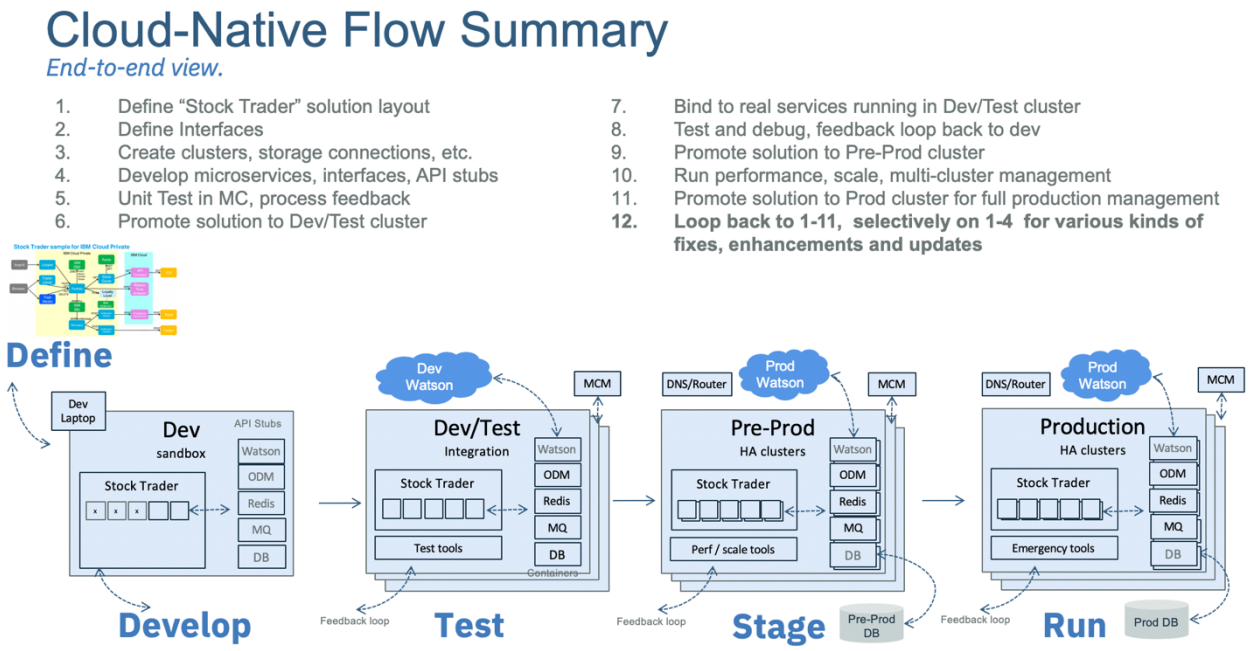
In JSON, this would be:

```
{
  "name": "John Alcorn",
  "serial": 228264,
  "title": "Senior Software Engineer",
  "address": {
    "office": "501-B101",
    "street": "4205 S Miami Blvd",
    "city": "Durham",
    "state": "NC",
    "zip": 27703
  }
}
```

While a JSON document itself is language-independent, languages typically have libraries that implement the API. Java, for example, has JSON-B (JSR 367) and JSON-P (JSR 374) as standards as well as specialized libraries like Jackson.

End to End Development Flow

There are many ways to visualize and explain the job of building, testing, deploying, running and managing a microservice or a solution constructed from microservices. In fact, you will see many cloud-native approaches where there are common environments for many of these stages. However, some clients will start their cloud-native journey with their existing stages of delivery that are similar to what we describe below. This flow illustrates how developers, DevOps engineers, and cloud admins can work together to craft elegant cloud-native solutions now, and along the journey enhance and improve the end-to-end pipeline along the way:



Some clients are far into the cloud-native journey and have adopted cloud-native pipeline approaches where these multiple physical environments are not needed any longer (Kubernetes clusters with multiple namespaces, for example) to mimic these delivery stages. However, most are in the early stages of this journey. In either case, what we can all agree on is that in any end-to-end flow there need to be guardrails put in place (whether separate physical environments or software-defined rules, circuit-breakers, or isolation policies) to keep each stage flowing without conflicts: defining and developing valuable applications, testing, preparing and staging for production, and finally running in production where even a few seconds of down-time impact revenue and customer loyalty.

From the perspective of the programming model and particularly the artifacts that make up the programming model, it is the continual iteration between the 'Define' and 'Develop' stages where most of the artifacts are created and modified.

For example, it is during 'Define' where the high-level sketch of the solution components is initially created as a collaboration between development, DevOps engineers, and operations and infrastructure teams. While certainly through iteration the details are refined, this initial "Define" work helps all involved in the end to end flow prepare for the success of the project. For example, dev leads may start with a coarse list of microservices, and dependent services and that helps the infrastructure team provide initial dev and test environments: Kubernetes platform is created, security settings assigned, 'dev' middleware services are deployed, and API stubs are created to reduce costs during early dev/test. As the DevOps process is streamlined, even as teams merge into a true DevOps model, this continuous feedback helps not only the application run well, but the operations teams prepare for the scope of what it needs to run and manage the app in production.

It is during the 'Define' and the initial stages of 'Develop' is where artifact creation likely occurs. Cloud development environments like Microclimate (learn more) provide templates that include many of the artifacts the developer will need to deploy and run the microservice, including

identifying the runtime (Liberty, for example) and configuration surrounding building the docker container, deploying into Kubernetes, and configuration to optimize the runtime environment. It is, obviously, during 'Develop' that the developer would create the .java files and put code into them. Some of the artifacts created are required and some are optional. Each supports some element of the overall programming model. A quick enumeration of those artifacts can be found in **Appendix A: Java – Summary of Files**.

Of course, roles play an important part in developing not just the code, but the configuration files surrounding the application, as well as automation files like auto-scale policies, Istio rules, etc. While the developer role is familiar, there is a DevOps/SRE role, and they are responsible for the deployment, configuration, and automation files. For example, to more easily manage a microservices based solution composed of many different components, labeling and custom resource definitions (CRDs) may be used to identify each microservice as a part of the bigger solution.

With the knowledge of the files and the basics of the content in place, which apply to all kind of microservices, we can now proceed to the business of creating a microservice using a specific language and framework. Additional details on the contents of the files described above will be provided in the relevant sections to follow or in reference sections provided in the appendices.

"Define" - Initial Planning and Architecture (App + Platform)

The 'Define' delivery stage is where initial solution architecture takes place. Back in Figure 1, the stock trader solution was defined with microservices, middleware, interfaces, public cloud services, and so on.

Some tips:

- The number of microservices is a business decision based on features needing different life cycles, or different scaling time frames, as well as how development teams are structured. You don't want dependencies across teams for a single microservice.
- Fewer microservices is better…to start: It's easy to get carried away with defining many microservices before you start coding. It's better to start with a handful, learn while you code, then iterate.
- What middleware you need: Define not only what capabilities you need, but how you want to interface to it (you'll need the Kubernetes secrets defined soon enough).
- Work with your cloud admin: The sooner you can get a cluster available for you to develop and test in the better. Remember, there are "developer" versions of many middleware services where you can quickly get them running and delete when you are done.
- Think about tagging taxonomy for your solutions. Kubernetes can help manage your apps and services that work together using tagging.

The 'Define' stage is also where labeling is defined for applications based on many microservices. For example, when a solution is defined, you can define your solution labels to inject into your deployment.yaml files so that when you deploy your microservices, they will appear as a unified "application" in IBM's Application Navigator. Later on in the "managing your application" section, we will dive deep into these capabilities. For now, you can Learn more about Application Navigator here.

"Develop" - Code, Unit test, Repeat...the Inner Loop

The 'Develop' delivery stage goes hand-in-hand with the 'Define' stage and many times will loop continuously until an initial version of the solution is ready to try out. This is where most development teams will live and breathe...coding and building and debugging. Remember, in a test-driven development world, your job is to first create the unit test, and then code to make the test pass. Since a solution will most likely have many microservices each developer will only have visibility into a small portion of the solution. Therefore, well-defined interfaces, API stubs, and even API-Connect interfaces are critical.

Tips:

- Follow this programming guide: Optimizing your solution for Kubernetes is critical for your successful dev and debug (and your future ops team will thank you)
- Follow test-driven-development practices by creating unit tests first. Additionally, create functional and performance tests for your APIs early once you have the API definition.
- Clearly define end-to-end pipeline to execute these tests.
- Use API stubs: This makes development faster, but also cheaper. Even the most economical public cloud API has costs (time and $$$). Using an API stub for dev and some test will save you both. For data intensive APIs, it is important to generate the stubs dynamically. What this means is that you will have a 'test' implementation of the APII that does not make the actual API call, but provides structurally accurate return data.
- Where possible, use a shared Kubernetes dev/test cluster where you can run 'dev-level' middleware (MQ, Db2, etc). This will harden your code.
- Instrument Jenkins to automate your build and deployment.
- Instrument your feedback loop so wherever your app is running, you can get logs and monitoring data easily (including Kibana defaults and Grafana dashboards optimized for your solution)
- Tag each microservice with your "application" tag so it can be easily identified as a participant of that application. For more details, Learn more about Application Navigator.

"Test" - Deploy via Pipeline to Integrate and Test with Others

The 'Test' delivery stage can actually be a variety of test stages ranging from testing specific subsets of microservices, platform upgrades, and more. Integration test is critical as this is where multiple teams content comes together. Here is where you'll start aligning middleware, what namespace to use, access and identity behaviors, and other access controls, and, of course, test. Some teams want a public "dev/test" cloud where the Kubernetes cluster is managed by a cloud provider, and of those some even want managed middleware ("I don't want to be a database admin... I'm a product tester"). This is also where your test tools can provide essential feedback to each developer...provided the code was instrumented.

It's important to note here that having a "test" stage does not conflict with "test-driven-development". Rather they are quite complimentary. Test-driven-development ensures the quality of code inside a single microservice. A Test stage, with integrated test suites, automated test, etc, ensures the quality across the full multi-microservice application.

Some tips:

- This is where your API stubs will at least need to toggle to real APIs for testing.
- Plan for "Emergency tooling" that can help with root cause analysis when your code is running in production
- Run chaos-monkey or equivalent to start killing resources at various levels (VMs, pods, connections to public APIs, etc)
- Orchestrate the artifacts your solution needs with your production deployment team to ensure all resources are available and deployed consistently and add these artifacts to your code repo (GitHub).

"Stage" - Prepare for Production Utilizing Production-Ready Services and Platform

In this 'prepare for production' delivery stage, we are doing final testing and validation for running not only the application, but APIs and middleware in production. As a result, there are several things to consider:

- Run middleware with HA so that the environment mimics production as much as possible. This is when IBM Cloud Paks could be considered since they are certified to run in production.
- Add performance/scalability tools to test load on the system
- Deploy your microservices with HA in mind. This means having your scalability content in place (replica sets with multiple instances, auto-scale policies, Istio policies), and run them across a multi-zone Kubernetes cluster). A multi-zone cluster simply means a Kubernetes cluster with worker nodes spanning availably zones. If you are running IBM Cloud Privagte in your data center, it may mean initially that worker nodes are running across multiple subnets, in other cases you may have multiple buildings with low latency connections).
- Continue to run chaos-monkey or equivalent to start killing resources at various levels (VMs, pods, connections to public APIs, etc)
- Maintain the orchestration artifacts your solution needs with your production deployment team to ensure all resources are available and deployed consistently.

This is also the most controversial stage of delivery. Why? Because it is a myth that you can accurately emulate a production environment outside of production. There are just too many variables and unplanned customer behaviors to properly automate. For this reason, some DevOps teams use routing rules as guardrails and run their final validation/staging testing in a true production environment. That said, data privacy and regulations still require most middleware and data to stay solely in a dedicated production environment. As a result, many teams are still separating the environments and using tools like UrbanCode Deploy to deploy into those environments for further control and governance.

"Run" - Run and Operate in Production

In the 'Run' delivery stage, the solution is running fully in production with HA clusters, microservices, middleware, data, and networking (DNS, global load balancers, etc.). In a modern DevOps environment, the developer will still need to take care that their solution (or their portion of the solution) is healthy. The same feedback loop used initially in 'Develop' will be used here. Also, if required, the emergency tooling created during Test is available for root cause analysis.

As we look forward, we want this feedback loop to constantly stream into Microclimate so while a developer may be coding an enhancement, they will instantly get notified if their code is misbehaving “in the wild”, and have the tooling available to resolve the issue quickly (code, test, deploy a patch).

As a result, this end-to-end flow turns into a continuous loop where the stages of delivery can become logical stages and the environments required to run enterprise solutions can collapse into a smaller set of large clusters that are much easier to manage, yet still contain the guardrails needed to define, develop, test, stage, and run software that exceed the strictest enterprise client demands.

Introduction to Open Liberty and WebSphere Liberty

Our programming model is built out of platform features that have been described previously. These have been described in a way that allows for them to be leveraged and used in a polyglot way. The programming model also has a core set of runtimes that are specific to a programming language. For Java, this means WebSphere and specifically WebSphere Liberty and Open Liberty.

Open Liberty is a lightweight open source Java runtime built around modular *features*. It has a minimal kernel that provides a responsive dynamic configuration system. The footprint of a running server otherwise varies by enabled features.

WebSphere Liberty is a commercial version of Open Liberty. Some features are only available in the commercial version.

Liberty has features at varying granularity providing application framework support:

- MicroProfile, an open source project defining new standards and APIs to accelerate and simplify the creation of microservices.
- Jakarta EE and Java EE, including features for individual specifications, like JNDI or JAX-RS.
- Spring Framework and Spring Boot, including mechanisms to make compact containers from Spring Boot's fat jars.

When automated systems like Kubernetes are pushing container images around, image size starts to matter. The layers in Docker images are cached to help with this. Liberty provides an efficient packaging pipeline for Docker containers, making it a great operational fit for cloud environments. One base image can be used to support many workloads, while allowing the resource footprint of running containers to vary based on the requirements of the service.

Liberty provides tools and optimized support for converting Spring Boot fat jars into compact, optimized Docker containers that take advantage of cached Docker image layers to improve cycle and publish times. See more at https://openliberty.io/blog/2018/07/02/creating-dual-layer-docker-images-for-spring-boot-apps.html.

There are wide set of practical developer guides for creating Microservice applications at https://openliberty.io/guides/

Liberty Docker Images

Both Open Liberty and WebSphere Liberty provide a variety of base images that you can use as starting points for your Java-based microservices. There are images that use small footprint or OpenJ9 VMs with different features pre-selected. For example:

1. open-liberty:microProfile2 or websphere-liberty:microProfile2
2. open-liberty:javaee8 or websphere-liberty:javaee8
3. open-liberty:springBoot2 or websphere-liberty:springBoot2

Check https://hub.docker.com/_/websphere-liberty/ or https://hub.docker.com/_/open-liberty/ for the most up to date list of base images.

Open Liberty and Docker

Choose the most appropriate image for your application from the list of images https://hub.docker.com/_/open-liberty/ (e.g. open-liberty:microProfile2) and create a `Dockerfile` for your service that extends `FROM` it. Add the commands to copy your app and server configuration into the new image. As an example:

```
FROM open-liberty:microProfile2
COPY server.xml /config/
ADD Sample1.war /config/dropins/
```

For more examples and working code, see the following Open Liberty guides:

- https://openliberty.io/guides/docker.html
- https://openliberty.io/guides/getting-started.html#running-the-application-in-a-docker-container

WebSphere Liberty and Docker

There are some differences between Open Liberty and the commercial version, WebSphere Liberty. One of the most significant for creating Docker images is that the `installUtility` command is not available in Open Liberty.

WebSphere Liberty supports the same basic customization patterns that OpenLiberty does for docker images, but the inherently modular design of Liberty makes it simple (and typical) to create a custom image that contains an application and the specific set of features it requires. WebSphere Liberty has an image for a feature-less kernel, websphere-liberty:kernel, which makes an solid base for a truly customized image.

The websphere-liberty image documentation (https://hub.docker.com/_/websphere-liberty/) describes the following simple 3-line dockerfile required to create a custom image:

```
FROM websphere-liberty:kernel
COPY server.xml /config/
RUN installUtility install --acceptLicense defaultServer
```

The `installUtility` tool looks for any features you require in your `server.xml` that aren't already available in your Liberty image. It will then pull those down from IBM and install them into your Docker image. Note that `installUtility` is only available in the commercial version of WebSphere Liberty.

This approach will create a minimal image, but the implicit feature list in `server.xml` does not work well with cached Docker layers. Any change to `server.xml` invalidates the layer with installed features, requiring the missing features to be installed again the next time the image is built. Given there is every likelihood that the server configuration will change more often than the selected feature set will, this is not the best answer.

A reasonable compromise is to explicitly install the features that you want in the Dockerfile. This still builds your minimal image, but will be much better behaved from a build/deploy perspective, as the custom feature installation layer can be cached.

```
FROM websphere-liberty:kernel
RUN /opt/ibm/wlp/bin/installUtility install  --acceptLicense \
```

```
    cdi-1.2 \
    concurrent-1.0 \
    jaxrs-2.0 \
    jndi-1.0 \
    ssl-1.0 \
    websocket-1.1
COPY server.xml /config/
```

There are multiple frameworks upon which WebSphere Liberty is leveraged. Upcoming sections will build on this basic knowledge of WebSphere Liberty in framework specific ways.

Introduction to Java Programming Model – MicroProfile

In this section, will use the commercial version of WebSphere Liberty as the runtime and will use MicroProfile and Java EE as the programming platform which sits on top of the Java language. With that assertion, a number of these terms and their relationships need to be introduced and contextualized with a programming model.

Basics of Microservices Programming Model with Liberty

Our recommended started point for building microservices in Java is to use MicroProfile/Java EE capabilities in your application with Liberty as the application container running in a Kubernetes environment.

There are wide set of practical developer guides for creating Microservice applications under https://openliberty.io/guides/

These cover topics, with working code in github projects, including;

- Getting started with the Open Liberty application server - https://openliberty.io/guides/getting-started.html
- Creating a RESTful web service with JAX-RS, JSON-P - https://openliberty.io/guides/rest-intro.html
- Consuming a RESTful web service using JSON-B and JSON-P - https://openliberty.io/guides/rest-client-java.html
- Using Contexts and Dependency Injection (CDI) to manage and inject dependencies into microservices - https://openliberty.io/guides/cdi-intro.html
- Consuming RESTful services using MicroProfile Rest Client in a type-safe way - https://openliberty.io/guides/microprofile-rest-client.html
- Using Docker containers to develop microservices - https://openliberty.io/guides/docker.html
- Securing microservices with JSON Web Tokens using MicroProfile JWT - https://openliberty.io/guides/microprofile-jwt.html
- Configuring microservices through external configuration sources using MicroProfile Config - https://openliberty.io/guides/microprofile-config.html
- Create fault-tolerant microservice using MicroProfile CircuitBreaker and Fallback policies - https://openliberty.io/guides/circuit-breaker.html
- Adding health checks and reports to microservices using MicroProfile Health - https://openliberty.io/guides/microprofile-health.html
- Providing system and application metrics from a microservice MicroProfile Metrics - https://openliberty.io/guides/microprofile-metrics.html
- Enabling and customizing distributed tracing in microservices using MicroProfile OpenTracing - https://openliberty.io/guides/microprofile-opentracing.html

Further Open Liberty guides are being added all the time independent of the environment in which Liberty runs. This section of the document extends those contexts and illustrates configurations targeted specifically at Kubernetes environments like ICP.

Basics – Overview of the Programming Model

To really start building out microservices using Java, there are a set of very basic tasks that need to be accounted for by the artifacts that a developer is being created. These basic topics and the current recommendations for new projects are summarized in the table below:

Programming Model for Java – Summary POV 01/03/2019

Developer Programming Task`	Primary Approach	Alternatives and Other Comments
Configuration Data	mpConfig	Raw System.getenv()
JSON Handling	JSON-B	Use JSONP if you don't know the schema ahead of time; Do not use JSON4J anymore
Logging	JSR 47 (java.util.logging)	Visualize in Kibana
Tracing	Istio (automatic for calls through sidecar) with mpOpenTracing feature enabled for correlation	mpOpenTracing annotations only when developer injected trace points are needed. Use OpenTracing API at your own peril
Metrics (application level)	mpMetrics	Visualize in Prometheus and Grafana
Creating Microservices with REST APIs	JAX-RS (top down from Open API definition)	mpOpenAPI to expose your metadata
Microservice to Microservice – Synchronous	mpRestClient	
Microservice to Microservice - Asynchronous	Kafka (Event Streams on ICP)	JMS/MQ
Exposing Manageable REST APIs	API Connect and Open API	
Consuming external APIs	mpRestClient or SDK if provided and preferred	SDKs often take care of security things
Security – Authentication	Outside of code; specified via server.xml. OIDC for initial login, mpJWT for SSO	Use @RolesAllowed annotation for RBAC
Fault Tolerance	Istio declarative approach	mpFaultTolerance for fallback
Kube Readiness and Liveness Check	Manually authored endpoints (in java)	mpHealth not recommended

Figure 2- Java Programming Model Basics - Summary

The reddish purple color represents things pulled from Micoprofile. The yellowish bash color represents Istio. The lighter blue shaded areas suggest things from java or Java EE. Combinations occur as well and that has been represented by the figure out available.

The following sections will go into details on these basic tasks, enabling a developer to leverage these features as part of creating a microservice. There will be two approaches taken in the sections to follow:

1. **Java/MicroProfile specific approach** – In this case, accomplishing the task will be outlined inline as part of this chapter. Relevant pointers to alternative approaches are also provided inline in this chapter.
2. **Polyglot approach** – If a language independent, or polyglot approach is recommended, then a pointer to the section with details is provided. Relevant pointers to alternative approaches are also provided inline in this chapter.

The table above not only highlights the preferred approach, but also provides additional insights and linkages to past approaches as well as possible alternatives that could be leveraged. Look for additional links and comments in the upcoming sections on the "Alternatives and Other Comments" columns from the table above.

The following sections will elaborate in more details on MicroProfile programming model. The programming model is built on Contexts and Dependency Injection (CDI).

Contexts and Dependency Injection (CDI, JSR-365)

CDI is a central element in both Jakarta-EE and MicroProfile, as it is used to wire various components together. The most fundamental capabilities provided by CDI are:

- Contexts, which bind the lifecycle of a component to well-defined lifecyle contexts.
- Dependency injection, which allows components declare dependencies, and have them injected in a typesafe way.

CDI also provides interceptors and decorators to extend the behavior of components, a loosely-coupled event model, and a powerful and flexible portable extension mechanism for other frameworks to define their own CDI beans or update existing components.

CDI 1.2 is part of MicroProfile 1.0 release and thereafter (MicroProfile 2.0 moves up to CDI 2.0). MicroProfile specifications (e.g. MicroProfile Config, MicroProfile JWT etc.) take CDI-first approach, relying on the CDI extension mechanism to enhance existing standards (like JAX-RS) with new capabilities.

CDI encourages loose-coupling with strong typing, which enables to design an interface-based, structured microservice. When creating microservices, CDI should be used to handle the interactions of the internal components.

CDI's core concept is Bean. What is a bean (CDI bean)? With very few exceptions, almost every concrete Java class that has a constructor with no arguments or a constructor with the annotation of `@Inject` is a bean. The instances of a bean are contextual. The container creates and destroys these instances and associates them with the appropriate context. The contextual instance may be injected into other objects.

Beans are created using Bean-defining annotations. CDI scope type annotations control the lifecycle of the associated bean:

- Use the `@ApplicationScoped` class-level annotation if an instance should live as long as the microservice. Any injection of this bean will result into the same instance.
- Use the `@RequestScoped` class-level annotation if a new instance should be created for every request.
- Use the `@Dependent` class-level annotation if the new instance should belong to another object. An instance of a dependent bean is never shared between different clients or different injection points. It is instantiated when the object it belongs to is created, and then destroyed when the object it belongs is destroyed.
- If no class-level annotation is defined, `@Dependent` is the default.

Enable CDI using a `beans.xml` file under `META-INF` folder for a jar archive or `WEB-INF` folder for a war archive. This file can be empty, or it can contain something like the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
    version="1.1" bean-discovery-mode="all">
 // enable interceptors; decorators; alternatives
</beans>
```

The presence of an empty `beans.xml` or a `beans.xml` with `bean-discovery-mode="all"` makes all potential POJO classes CDI beans.

When a microservice is CDI enabled with beans defined as per above, CDI can inject those beans into other components via `@Inject`.

For instance, the following POJO MyBean is a CDI bean. In the microservice, there will be just one instance.

```
@ApplicationScoped
public class MyBean {
      int i=0;
      public String sayHello() {
            return "MyBean hello " + i++;
      }
}
```

The same instance of MyBean will be injected to each JAX-RS request MyRestEndPoint. CDI will manage the lifecycle of MyBean for you. There will be no more directly calling constructors.

```
@RequestScoped
@Path("/hello")
public class MyRestEndPoint {
      @Inject MyBean bean;

      @GET
      @Produces (MediaType.TEXT_PLAIN)
      public String sayHello() {
            return bean.sayHello();
      }

}
```

Programming Model Concept – Configuration

Cloud native systems are, by design, highly automated. Kubernetes is an orchestrator: it automates deployment, scaling, and management of containerized applications. DevOps practices augment this environment, automating the creation and publication of artifacts (docker

images, Kubernetes yaml, or helm charts, as examples) into target environments. This automation is critical to the "speed" attributed to cloud native applications and microservices.

The popular twelve factors approach (`https://12factor.net/`) defines a methodology for creating applications that behave well in cloud native environments. Three of these factors directly relate to creating services that support automated build and deployment.

- The first factor recommends a 1-to-1 correlation between a running service and a versioned codebase. In practice, this means creating an immutable deployment artifact (like a Docker image) from a versioned codebase that can be deployed, unchanged, to multiple environments.
- The third factor recommends a separation between application-specific configuration, which should be part of the immutable artifact, and environment-specific configuration, which should be provided to (or injected into) the service at runtime.
- The tenth factor recommends keeping all environments as similar as possible. Environment-specific code paths are difficult to test, and increase the risk of failures as you deploy into different environments. This also applies to backing services: if you develop and test with an in-memory database, unexpected failures might occur in test, staging, or production environments because they use a database that has different behavior.

The net of these factors is a firm requirement for cloud native applications to be portable. You should be able to use the same immutable artifact to deploy to multiple environments (local development, and cloud-based test and production environments, for example), without changing code or exercising otherwise untested code paths.

Application-specific configuration should be part of the immutable artifact. For example, applications that run on WebSphere Liberty define a list of installed features, which control the binaries and services that are active in the runtime. This configuration is specific to the application, and should be included in the Docker image. In the case of Docker images, the listening (or exposed) port is also specified, as the target environment (e.g. Kubernetes) will handle the port mapping when starting the container.

Configuration injection

Environment-specific configuration, like the host and port used to communicate with other services, database users, or resource utilization constraints, are provided to the container by the deployment environment. Management of service configuration and credentials (service bindings) can vary significantly:

- Kubernetes stores configuration values (stringified JSON or flat attributes) in either ConfigMaps or Secrets. These can be passed to the containerized application as environment variables or virtual file system mounts. The mechansim used by a service is specified in the deployment metadata (either in kube yaml or the helm chart).
- Local development environments are often simplified variants that use simple key/value environment variables.
- Cloud Foundry stores configuration attributes and service binding details in stringified JSON objects that are passed to the application as an environment variable (`VCAP_APPLICATION` and `VCAP_SERVICES`).

- Using a backing service like etcd, hashicorp Vault, Netflix Archaius, or Spring Cloud config, to store and retrieve environment-specific configuration attributes is also an option in any environment.

Environment variables

You don't want to have to update your source code and rebuild and deploy a new version of your microservice every time something like an endpoint URL or a password changes. In most cases, an application will process environment-specific configuration at start time. The value of environment variables, for example, cannot be changed after a process has started. Kubernetes and backing configuration services, however, provide mechanisms for applications to dynamically respond to configuration updates. This is an optional capability. In the case of stateless, transient processes, restarting the service is often sufficient.

Kubernetes lets you define environment variables that will get passed to the container in which your microservice runs. You do this in the yaml used to deploy your container. For example, consider the following snippet, for specifying the URL used to contact the *Watson Tone Analyzer* (used in the *Portfolio* microservice in Stock Trader):

```
    spec:
      containers:
      - name: portfolio
        image: ibmstocktrader/portfolio:latest # Docker Hub
        env:
          - name: WATSON_URL
            value: https://gateway.watsonplatform.net/tone-
analyzer/api/v3/tone?version=2017-09-21&sentences=false
```

That would cause an environment variable named `WATSON_URL` to be made available to your container, with the value specified. This way, you wouldn't have to update your code and rebuild your container if this endpoint URL changed for some reason.

However, even though you have made it so you don't have to update and rebuild your container now to change this value, you still would have to update this yaml file each time. So, let's look at a few other options next.

Helm chart values

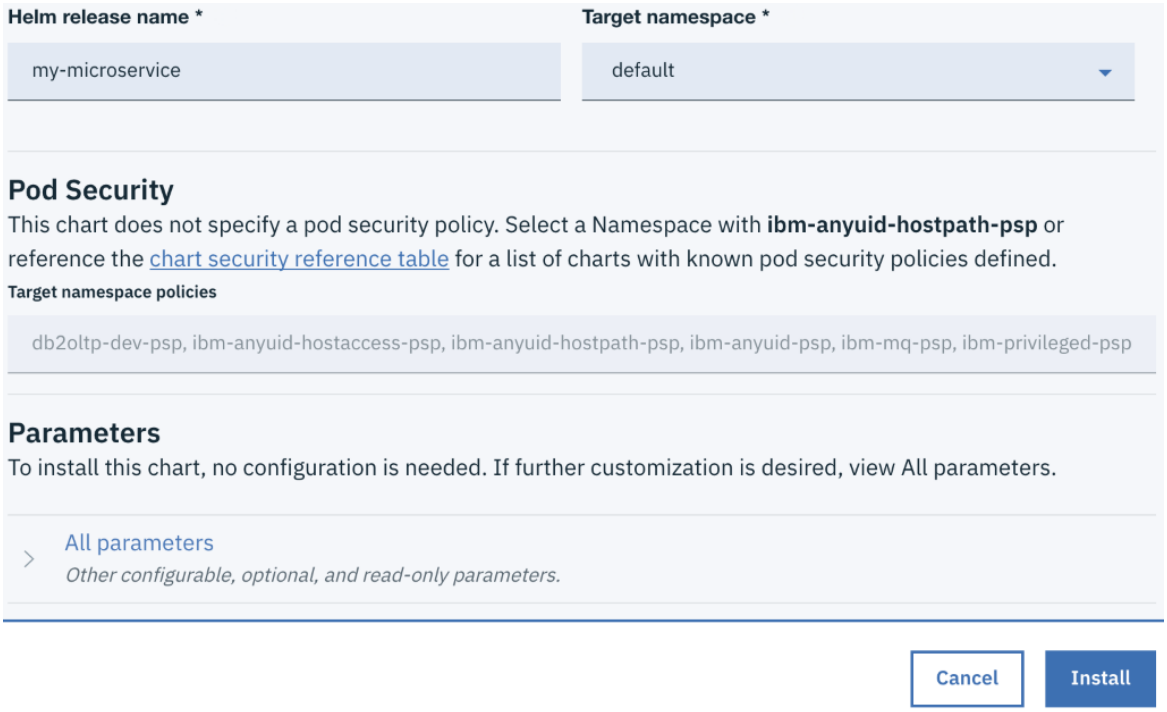
When you place your yaml inside of a helm chart, you can specify that it should use the value of an entry from your values.yaml, rather than hard-coding the value directly in the yaml. For example, consider this updated version of the previous snippet:

```
    spec:
      containers:
      - name: portfolio
        image: ibmstocktrader/portfolio:latest # Docker Hub
        env:
          - name: WATSON_URL
            value: {{ .Values.watson.url }}
```

In this case, when the helm chart is deployed, you would have the opportunity to pass in the value of this field. For example, if using the `helm` CLI, you'd do something like the following:

```
helm install my-chart --name my-microservice --set
watson.url=https://gateway.watsonplatform.net/tone-
analyzer/api/v3/tone?version=2017-09-21&sentences=false
```

Or alternatively, if you deploy your helm chart via the helm UI (like in the ICP catalog), this would be a field you could fill in, if you expanded the "All parameters" twisty, before clicking on the blue Install button:



See the *Tools* chapter for a more detailed discussion of using Helm charts for deployment.

ConfigMaps and Secrets

The above approach was better than just using a string literal. But we'd still have to redeploy to make a change. What if we want to be able to update such a value dynamically? Kubernetes helps us out here, with its concepts of **config maps** and **secrets** (secrets are basically more secure versions of config maps). Both provide a way to edit configuration values post-deployment. For example, here's the secret we actually use for working with the Watson Tone Analyzer:

```
kubectl create secret generic watson "--from-
literal=url=https://gateway.watsonplatform.net/tone-
analyzer/api/v3/tone?version=2017-09-21&sentences=false" --from-
literal=id=08af66f3-74fd-43bf-b300-5702a22a27bb --from-
literal=pwd=<myPassword> -n stock-trader
```

As you can see, we've separated out the values for the `url`, `id`, and `pwd` so that we can adjust these independently of the life cycle of versions of the application. Generally, you would just update your secret, then delete your pod, which will cause Kubernetes to create a fresh one that will inherit the new configuration values. Here's the `env` stanza for those Watson settings:

```
spec:
  containers:
  - name: portfolio
    image: ibmstocktrader/portfolio:latest # Docker Hub
    env:
      - name: WATSON_URL
        valueFrom:
          secretKeyRef:
            name: watson
            key: url
      - name: WATSON_ID
        valueFrom:
          secretKeyRef:
            name: watson
            key: id
            optional: true
      - name: WATSON_PASSWORD
        valueFrom:
          secretKeyRef:
            name: watson
            key: pwd
```

For example, that last one says that the field named `pwd` in the secret named `watson` will be exposed as the environment variable named `WATSON_PASSWORD`. You can then just use the value of that environment variable in your source code. Here is where *mpConfig* comes in, from MicroProfile, as described earlier.

Note also that the `WATSON_ID` field is marked as optional; this is because Watson also has the option to authenticate via an API key rather than an id/pwd - if using that, you'd leave out the `id`, and put the API key in the `pwd` field. Note that for required fields (that is, those not marked as optional), the deployment will fail if such a field in such a secret does not exist (in such a case, you'll need to do a `kubectl describe` on your deployment to see what missing secret or field of a secret led to the failure).

Mounting files via secrets

So far, we've looked at using secrets to pass environment variables to the container for your microservice. But let's look at another option for how to do "configuration" via secrets. We can actually do "late binding" of a file into the container too, by putting it into a secret, and then mounting that into the container's file system. One common use case for this would be for a keystore, which you'd need to update whenever an SSL certificate gets updated for a site with which you need to communicate (be that within the Kubernetes cluster, or out on the internet).

For example, we hit this situation when working with *IBM Event Streams*, which secures its Kafka topics via SSL (and an API key, that you'd put in an environment variable in a secret, as described above). The certificate needed to talk to that topic will be different in each Kubernetes environment you set up (at least, when using the default of letting it generate a self-signed certificate; you can also choose to get a certificate from a trusted certificate authority and pass that in to the *IBM Event Streams* helm chart when installing it). But you'd still like to be able to build your Docker container once and publish that to DockerHub (or whatever private Docker image registry you use), and just reference that same Docker container from whatever Kubernetes environments you choose to install.

The solution is to put the keystore file containing that certificate into a secret. For example, consider the following:

```
kubectl create secret generic kafka-keystore --from-
file=certs.jks -n stock-trader
```

As you can see, we used the `--from-file` parameter, rather than `--from-literal`. We also need to define a `volume` and `volumeMount` in our deployment yaml:

```
spec:
  containers:
  - name: portfolio
    image: ibmstocktrader/portfolio:latest # Docker Hub
    env:
      - name: KAFKA_KEYSTORE_LOCATION
        valueFrom:
          secretKeyRef:
            name: kafka
            key: keystore
            optional: true
    ports:
      - containerPort: 9080
      - containerPort: 9443
    imagePullPolicy: Always
    volumeMounts:
    - mountPath: /keystores
      name: kafka-keystore-volume
      readOnly: true
  volumes:
  - name: kafka-keystore-volume
    secret:
      secretName: kafka-keystore
```

That's a bit more complicated than past scenarios, but basically it says to mount the contents of the `kafka-keystore` secret into the `/keystores` directory in the container's file system. Note there's also a `KAFKA_KEYSTORE_LOCATION` environment variable defined, containing the fully-qualified path to this keystore file from within the Docker container, which would be

`/keystores/certs.jks` in this case (we could then reference that environment variable from our server.xml or from within our Java code, like before).

We've now seen several approaches available in Kubernetes for how to separate configuration from container. Which one you choose to use will depend upon your needs, of course. The net is, it's easy to do "late binding" of configuration data and files to your Docker container with Kubernetes!

For more information about working with Kubernetes secrets, see https://kubernetes.io/docs/concepts/configuration/secret/.

Programming Model for Java – Configuration

The MicroProfile Config API enables application configuration properties stored in multiple sources. Given that the polyglot capabilities related to secrets and config maps in Kubernetes are featured in this programming model, it is strictly an option to leverage the MicroProfile config API. For more information about MicroProfile Config, refer to this article (https://www.eclipse.org/community/eclipse_newsletter/2017/september/article3.php).

Kubernetes configuration makes configuration available via an environment variable which would then be leveraged as shown below:

```
private String watsonService = System.getenv("WATSON_URL");
```

This will give you the value you need to call Watson. However, the mpConfig feature provides an extra layer of indirection, making it so that you could feed your configuration values via other approaches, like storing them in a Cloudant database, without having to change your code. To get the value via mpConfig, you would just do the following:

```
private @Inject @ConfigProperty(name = "WATSON_URL") String watsonService;
```

This uses CDI to inject the value into the `watsonService` variable. You can also specify a **default value** to be used, if no config property is available. For example, you could do the following:

```
private @Inject @ConfigProperty(name = "WATSON_URL", defaultValue =
"https://gateway.watsonplatform.net/tone-analyzer/api/v3/tone?version=2017-09-21")
String watsonService;
```

And then, if no `WATSON_URL` is available from the environment (such as would be the case if this was omitted from the `watson` secret in Kubernetes), it would fall back to the specified default value. You can use this pattern when there are reasonable defaults for a variable, but you want to provide the flexibility to allow someone to override that default when necessary. Note that the upcoming mpConfig 1.4 (in the upcoming MP 2.2) will add the ability to specify null as a default (today, you can only default to a non-null value).

Note that you need the following import statements for this to work (as described in the *mpRestClient* section):

```
//CDI 1.2
import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;

//mpConfig 1.2
import org.eclipse.microprofile.config.inject.ConfigProperty;
```

And you need a CDI bean-defining annotation, such as `@RequestScoped`, on your class, like is done in the Summary servlet:

```
@WebServlet(description = "Portfolio summary servlet",
urlPatterns = { "/summary" })
```

```
@RequestScoped
public class Summary extends HttpServlet {
```

To get the code to compile, you'll need the following stanza in your Maven pom.xml:

```
    <dependency>
        <groupId>org.eclipse.microprofile</groupId>
        <artifactId>microprofile</artifactId>
        <version>2.0.1</version>
        <type>pom</type>
        <scope>provided</scope>
    </dependency>
```

As explained in previous CDI section, you also need the following beans.xml in the WEB-INF directory of your war file, for CDI to kick in:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       version="1.1" bean-discovery-mode="all"/>
```

With that in place, you'll now have the freedom to replace how configuration values are made available to your application, without needing to change anything in your application.

Programming Model for Java – JSON Handling with JSON-P and JSON-B

JSON-B (JSON-Binding, JSR 367) and JSON-P (JSON-Parsing, JSR 374). JSON-P provides a Java API for processing JSON-formatted data. JSON-B provides a binding layer on top of JSON-P, making it easier to convert objects to and from JSON. In most cases, JSON-B should be used in preference to the lower-level JSON-P.

JSON-B

With **JSON-B** (JSON-Binding), you define a Plain Old Java Object (POJO), with a getter/setter method for each field of the JSON. For example:

```
public class Employee {
   private String fName;
   private int fSerial;
   private String fTitle;
   private Address fAddress;

   public Employee() {}
   public Employee(String name, int serial, String title, Address address) {
fName = name; fSerial = serial; fTitle = title; fAddress = address; }

   public String getName() { return fName; }
   public void setName(String name) { fName = name; }
   public int getSerial() { return fSerial; }
   public void setSerial(int serial) { fSerial = serial; }
   public String getTitle() { return fTitle; }
   public void setTitle(String title) { fTitle = title; }
   public Address getAddress() { return fAddress; }
   public void setAddress(Address address) { fAddress = address; }
}

public class Address {
   private String fOffice;
   private String fStreet;
   private String fCity;
   private String fState;
   private int fZip;

   public Address() {}
   public Address(String office, String street, String city, int zip) {
fOffice = office; fStreet = street; fCity = city; fState = state; fZip = zip)
}

   public String getOffice() { return fOffice; }
   public void setOffice(String office) { fOffice = office; }
   public String getStreet() { return fStreet; }
   public void setStreet(String street) { fStreet = street; }
   public String getCity() { return fCity; }
   public void setCity(String city) { fCity = city; }
   public String getState() { return fState; }
   public void setState(String state) { fState = state; }
   public int getZip() { return fZip; }
   public void setZip(int zip) { fZip = zip; }
}
```

Note that it is recommended that this class be placed in the same package as the microservice that will be using this class, or a sub-package thereof.

Then in your JAX-RS (microservice) class, you'd just use the following to create an `Employee` object with the fields shown earlier:

```
Address myAddress = new Address("501-B101", "4205 S Miami Blvd", "Durham",
"NC", 27703);
Employee me = new Employee("John Alcorn", 228264, "Senior Software Engineer",
myAddress);
```

Or if you have been passed such a JSON-B object in the body of a call made to your microservice, or have received such a JSON-B object as a return value of an mpRestClient call your microservice made to another microservice, you just use the getters defined in the POJO to access the contents of the JSON. For example:

```
// mpRestClient: outbound client request returning a POJO
// JSON-B converts the JSON response into an Employee object
Employee employee = myClient.get("John Alcorn");
int serial = employee.getSerial();
Address address = employee.getAddress();
String city = address.getCity();
```

There is also a parser that will take a String representation of a JSON object, and will give you back the JSON-B object corresponding to that. To use this, you need to add the following import statements to your microservice class:

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
```

Then you do the following (like I do in `stock-quote`, with the JSON string I get back from Redis for my Quote), like if you have just received a JSON string that you stored to a `myJSON` variable:

```
Jsonb jsonb = JsonbBuilder.create();
Employee employee = jsonb.fromJson(myJSON, Employee.class);
```

Note that you need the `jsonb-1.0` feature enabled in your server.xml. Or you can use `microProfile-2.0`, which gives you all of the MP features, including JSON-B and JSON-P.

Everything is strongly typed with JSON-B. If you get a field name wrong, you would see a compile failure, for example, since there wouldn't be such a method. Creating POJOs can be tedious, but most IDEs support generating getters and setters once the fields are defined, which helps.

One thing to be careful of is over-exposure of internal details because JSON-B is serializing all fields of your object into a JSON wire format automatically. Annotations like `@JsonbTransient` or `@JsonbNillable` should be used for robustness when parsing

JSON. `JsonbAdapter` implementations can also be defined to provide additional isolation between the wire format and the internal implementation.

JSON-P

Before JSON-B existed (which arrived as part of Java EE 8), **JSON-P** (JSON-Parsing) was the standardized way to interact with JSON within Java code. Prior to this, there were various proprietary JSON parsing libraries, such as IBM's JSON4J. JSON-P can be used to parse JSON you've received from a REST call, or to construct JSON you can return from your own JAX-RS methods.

Using JSON-P requires the `jsonp-1.0` feature be enabled in your server.xml (or the `microProfile-2.0` convenience feature, which enables all of the MP technologies). Here are the import statements for working with JSON objects and arrays in JSON-P:

```
import javax.json.Json;
import javax.json.JsonArray;
import javax.json.JsonArrayBuilder;
import javax.json.JsonObject;
import javax.json.JsonObjectBuilder;
```

To work with JSON received from a REST API call, you simply call the `get` method on a `JsonObject`, passing in the key for the field you want. To continue our Employee example above, if you received it in a `JsonObject` that you called `employee`, you would call `employee.get("name")` to find out the value of the employee's name, or `employee.get("title")` to get their title. JSON-P is a lot like dealing with a `Map` in Java, as a comparison.

Now, say you want to build such a JSON object in your Java code. JSON-P uses a builder pattern: use a `JsonObjectBuilder` to add each value, then call `build()` to produce the `JsonObject`, like such:

```
JsonObjectBuilder addressBuilder = Json.createObjectBuilder();

addressBuilder.add("office", "501-B101");
addressBuilder.add("street", "4205 S Miami Blvd");
addressBuilder.add("city", "Durham");
addressBuilder.add("state", "NC");
addressBuilder.add("zip", "27703");

JsonObjectBuilder employeeBuilder = Json.createObjectBuilder();

employeeBuilder.add("name", "John Alcorn");
employeeBuilder.add("serial", "228264");
employeeBuilder.add("title", "Senior Software Engineer");
employeeBuilder.add("address", addressBuilder);

JsonObject employee = employeeBuilder.build();
```

Note that a `JsonObject` in JSON-P is immutable, unlike with JSON-B. If you want to update fields, you have to create a new `JsonObject`, duplicate the fields, and then make changes. Making your own POJOs with copy constructors can help here.

JSON-B and JSON-P are both supported by the MicroProfile Rest Client for proper serialization and deserialization, but type-safety and compile-time feedback make JSON-B the preferred choice.

Observability, Telemetry, and Monitoring

There is a culture change around monitoring that comes with a shift to Cloud Native. While applications in both traditional and cloud native environments are expected to be highly available and resilient to failure, the methods used to achieve those goals are very different. As a result, the purpose of monitoring shifts: instead of monitoring to avoid failure, we monitor to manage failure.

In traditional environments there was often this notion of having an infrastructure and operations team that was separate from the applications team. Too much separation in that model lead to incongruent expectations. Developers often felt that the infrastructure had to handle their applications properly and would be quick to point to infrastructure in times of stability challenges. Infrastructure teams seemed to think they could build infrastructures that could handle anything thrown at it. However, they would also be quick to suggest an application issue was taking down or degrading the infrastructure.

Guidelines and best practices were applied to mitigate the risk of these differing expectations. Additional checkpoints and hurdles were added to deployment processes, often slowing down delivery velocity.

The cloud-native world requires a much more unified approach organizationally and further offers technical features and approaches that do not add time and manual effort to the process. Leveraging Kubernetes as the container-orchestration mechanism, including the declarative model creates just this unified notion we are looking for. Developers will need to actively contribute in order for this to work. All of the upcoming material is focused on achieving this unification and the quality improvements and the velocity improvements promised by a properly executed cloud-native development model. If you are serious about cloud-native, this is part of the price paid to get the promised benefits.

In traditional, non-cloud environments, long-lived infrastructure and middleware is provisioned based on planned capacity and high availability patterns (e.g. active-active or active-passive). Unexpected failures can be complex in this environment, requiring significant effort for problem determination and recovery. External monitoring is performed by agents that examine resource utilization to avoid known classes of failures. As an example, consider the tuning of heap size, timeouts, and garbage collection policies for Java applications.

A cloud native application is a composition of independent microservices and their required backing services. While a cloud native application as whole must remain available and continue to function, individual service instances are started and stopped as necessary to preserve system function.

Monitoring this fluid system requires each participant to be *observable*. Each should produce appropriate data to support automated problem detection and alerting, manual debugging when necessary, and analysis of system health (historical trends and analytics).

What kinds of data should a service produce to be observable?

- **Health checks** (often custom HTTP endpoints) help orchestrators like Kubernetes or Cloud Foundry perform automated actions (changing how requests are routed or restarting processes) to maintain overall system health.

- **Metrics** are a numeric representation of data collected at intervals into a time series. Numerical time series data is easy to store and query, which helps when looking for historical trends. Over a longer time horizon, numerical data can be compressed into less granular aggregates (daily, weekly, etc).
- **Log entries** represent discrete events that have happened over time. They are essential for debugging, as they often include stack traces and other contextual information that can help identify the root cause of observed failures.
- **Distributed, Request, or End to End Tracing** captures the end to end flow of a request through the system, essentially capturing both relationships between services (the services the request touched), and the structure of work flowing through the system (synchronous or asynchronous processing, child-of or follows-from relationships).

Cloud native applications should rely on the environment for *telemetry*, automatic collection and transmission of data to centralized locations for subsequent analysis. This is emphasized by one of the twelve factors (treat logs as event streams) and extends to all data a microservice produces to ensure it can be observed.

Kubernetes has some built-in telemetry capabilities, like Heapster, but it is more often the case that telemetry will be provided by other systems that integrate with the Kubernetes control plane. Istio and Prometheus are two of the best known and we'll be going over those in detail in later sections.

It may seem like notions of observability and telemetry are just new labels on old concepts, and to some extent that is true, but the concepts help highlight some significant differences in how cloud native applications in large-scale distributed systems are monitored. It bears repeating that processes in cloud native environments are transient. Three of the twelve factors (processes, concurrency, and disposability) emphasize this point. Pre-allocated, long running, monolithic processes are replaced (or surrounded) by many more short-lived processes that will be started and stopped in response to load (for horizontal scaling) or if they aren't functioning properly. Telemetry is critical here: data must be collected and persisted somewhere else to prevent it from being lost as processes (containers) are created and destroyed (sometimes this is required for compliance reasons as well).

Further, failures are no longer rare, disruptive occurrences. Breaking a monolithic application into microservices pushes more of the mainline path onto the network, increasing the impact of latency and other network issues. Requests will also reach processes that are not ready for work for any number of reasons. Services will be automatically restarted if they run out of resources and fault tolerance strategies allow the system as a whole to keep functioning. Intervening and taking actions based on individual failures is not particularly useful or feasible in this kind of environment.

For all of the reasons above, monitoring changes focus: instead of monitoring the behavior and health of *resources* (individual processes or individual machines), we monitor the state of the system as a whole. Each individual service produces data that feeds into this aggregated view.

There are two rules-of-thumb for service-level observability:

- The Google Site Reliability Engineering (SRE) team have defined the following as essential metrics for service-level observability when monitoring a distributed system, which they call the four golden signals:
  - **Latency**: How long does it take to respond to a request? Treat the latency of successful requests as separate from the latency of failed requests.
  - **Traffic**: How much demand is there on a service? This is usually a system-specific measurement, like HTTP requests per second for a web application or HTTP REST API.
  - **Errors**: What is the failure rate? Treat explicit failures (HTTP 500), implicit failures (response body indicates an error), and policy failures (rate-limiting) separately.
  - **Saturation**: How "full" is your service? Here you are doing some predicting: how close is this service to hitting thresholds or constraints on available resources that will cause problems?
- The RED Method, described by Tom Wilkie, is a mnemonic is derived from the four golden signals that defines three key metrics you should measure for every microservice in your architecture. Those metrics are:
  - (Request) **R**ate - the number of requests, per second, your services are serving.
  - (Request) **E**rrors - the number of failed requests per second.
  - (Request) **D**uration - distributions of the amount of time each request takes.

Observability as described here, supported by telemetry and enabled by various monitoring capabilities affects the programming model in a number of ways. In the upcoming sections, more complete descriptions are provided for:

- Logging
- Tracing
- Metrics
- Fault Tolerance
- Liveness and Readiness checks
- Auto-Scaling

How these elements are addressed technically in the source code and configuration files is part of the deeper discussions and represents the "how" part of dealing with these topics. These sections, however, will also be conscious of the "what" part given the context of the cloud native observability philosophy described here.

Programming Model Concept – Logging

Introduction

Log messages are strings containing contextual information that pertains to the state and activity of a microservice when the log entry is created. Logs are required to diagnose how and why services fail. They play a supporting role to metrics in monitoring application health. Heuristics about how much logging should be done and what should be logged are choices made based on the kind of system being constructed and enterprise specific guidelines.

Log entries should be written directly to standard output and error streams. This makes log entries viewable via command line tools, and allows log forwarding services configured at the infrastructure level, like *LogStash*, *FileBeat*, or *Fluentd*, to manage log collection.

JSON Logging

As your application evolves over time, the nature of what you log can change. By using a JSON log format, you gain the following benefits:

- Logs are indexable, which makes searching an aggregated body of logs much easier.
- Logs are resilient to change, as parsing isn't reliant on the position of elements in a string.

While JSON formatted logs are easier for machines to parse, they are harder for humans to read. We'll show detailed approaches to dealing with these kinds of logs in subsequent sections. In the meanwhile, there are two general recommendations:

- Consider using environment variables to toggle the log format between plain text for local development and debugging, and JSON-formatted logs for longer term storage and aggregation.Use command-line JSON parsers, like the *JSON Query* tool (`jq`), to create a human-readable view of JSON-formatted logs.
  `kubectl logs trader-54b4d579f7-4zvzk -n stock-trader -c trader | jq .message -M`

Viewing logs in Kubernetes

Logs sent to standard output and error streams can be viewed in Kubernetes through the console or via `kubectl` commands that are of the form: `kubectl logs <podname>`.

If you use a custom namespace, e.g. `stock-trader`, you need to include that in the command as well: `kubectl logs -n stock-trader <podname>`.

If there are multiple containers per pod, as there are when using istio sidecars, you also need to specify the container. In the following example, we will work with the `stock-trader` namespace to view logs from the `portfolio` container in the `portfolio-54b4d579f7-4zvzk` pod:

```
kubectl logs -n stock-trader portfolio-54b4d579f7-4zvzk -c portfolio
```

Viewing logs in Kibana

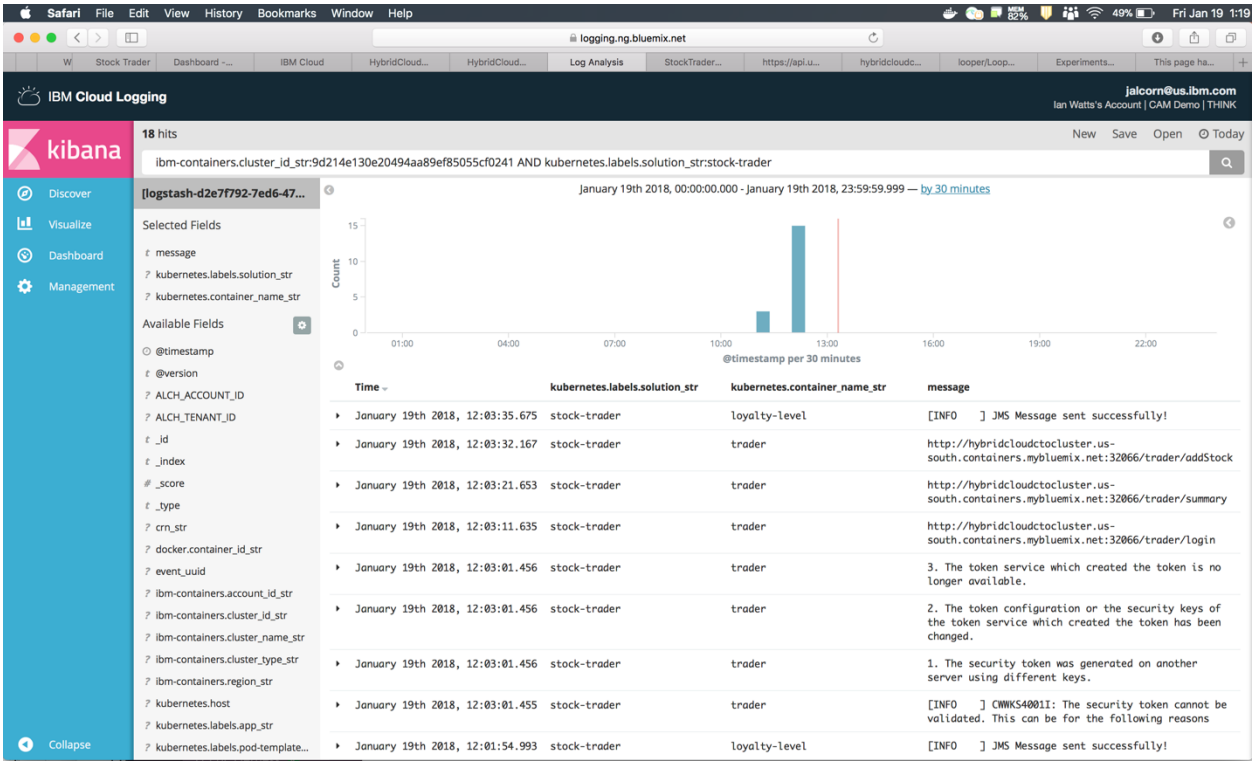


Figure 3: A filtered view of logs in Kibana

We can also access logs via *Kibana*, which shows a federated view of logs across all pods in your Kubernetes environment. Figure 2 shows a filtered view of logs in *Kibana*, which restricts visible log messages to those from pods with a label of `solution=stock-trader`.

Programming Model for Java – Logging

The recommended approach for logging in MicroProfile applications is to use JSR 47, which requires the following imports:

```
import java.util.logging.Level;
import java.util.logging.Logger;
```

And the following line of code to get an instance of such a logger:

```
private static Logger logger =
Logger.getLogger(PortfolioService.class.getName());
```

This provides access to key interfaces for logging. The `Logger` interface has methods such as `info`, `fine` and `warning` that all are going to write information to the logs.

Examples

Here are some sample usages of the API:

```
logger.info("Creating portfolio for "+owner);
logger.warning("Unable to send message to JMS provider.
Continuing without notification of change in loyalty level.");
```

And here are the corresponding entries in the pod's log:

```
    [INFO    ] Creating portfolio for John
    [WARNING ] Unable to send message to JMS provider.
Continuing without notification of change in loyalty level.
```

If you wanted to only do some expensive work to build log output if a certain trace level is currently set, you'd check the level, as in this example (from `PortfolioService.java`) that only builds and outputs a stack trace if the trace level is set to `FINE` (or higher, like `FINER` or `FINEST`):

```
if (logger.isLoggable(Level.FINE)) {
    StringWriter writer = new StringWriter();
    exception.printStackTrace(new PrintWriter(writer));
    logger.fine(writer.toString());
}
```

JSON Logging

Liberty supports **JSON Logging**. When enabled, messages going to the console will be in JSON format. Note that this is easy for a computer to parse, but tends to make life more difficult for a human to read.
With JSON Logging enabled, you can just configure that whole log to go to ELK, without needing to use any special features. This also works for Open Liberty. You enable this via the `logging` stanza in your server.xml:

```
    <logging consoleLogLevel="INFO" consoleFormat="json"
consoleSource="message,trace,accessLog,ffdc" />
```

Note the `accessLog` in the stanza above. To actually get the access log to flow to the console, you also need to add the `accessLogging` sub-tag to your `httpEndpoint` tag in your server.xml:

```
<httpEndpoint id="defaultHttpEndpoint" host="*" httpPort="9080" httpsPort="9443">
    <accessLogging
        filepath="${server.output.dir}/logs/http_defaultEndpoint_access.log"
        logFormat='%h %u %t "%r" %s %b %D %{User-agent}i'>
    </accessLogging>
</httpEndpoint>
```

JSON Logging produces messages like this in the log (such as you'd see via `kubectl logs <pod>`):

```
{"type":"liberty_message","host":"trader-54b4d579f7-
4zvzk","ibm_userDir":"\/opt\/ol\/wlp\/usr\/","ibm_serverName":"d
efaultServer","ibm_datetime":"2018-06-
21T19:23:21.356+0000","ibm_messageId":"CWWKF0011I","ibm_threadId
":"00000028","module":"com.ibm.ws.kernel.feature.internal.Featur
eManager","loglevel":"AUDIT","ibm_sequence":"1529609001356_00000
0000003E","message":"CWWKF0011I: The server defaultServer is
ready to run a smarter planet."}
```

Note that the message field of the JSON is what has the actual string that the code decided to send to the log (via `logger.audit`, in this case).

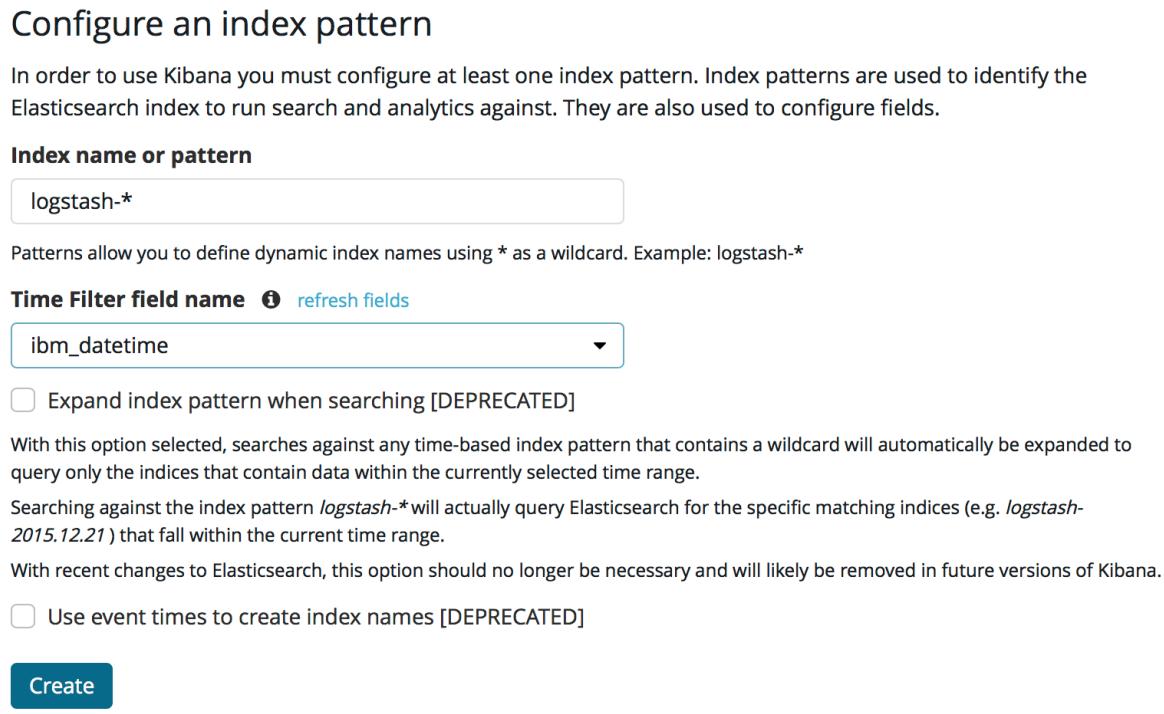
As mentioned in the logging overview, if you want to look at only the message field for your log entries, use something like the JSON Query tool, `jq` (which you can install to your Mac via `brew install jq`) to filter what is displayed. As an example:

```
kubectl logs trader-54b4d579f7-4zvzk -n stock-trader | jq .message -M
```

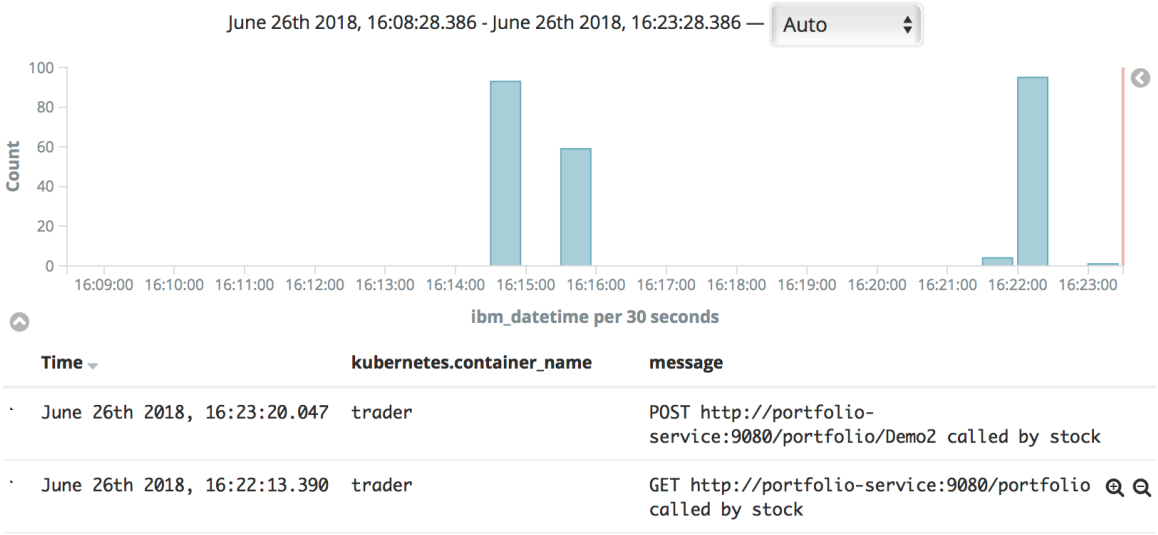
Log Visualization in Kibana

With this JSON Logging feature enabled, that `message` field is the column you want to configure Kibana to show (rather than the usual `log` field, which contains the JSON shown above).

You will need to regenerate your Kibana index pattern once some Liberty JSON Logging logs have been sent (and choose the `ibm_datetime` field, rather than the default `@timestamp` field, as the "Time filter field name"), in order to get things working with Liberty's JSON Logging and Kibana:

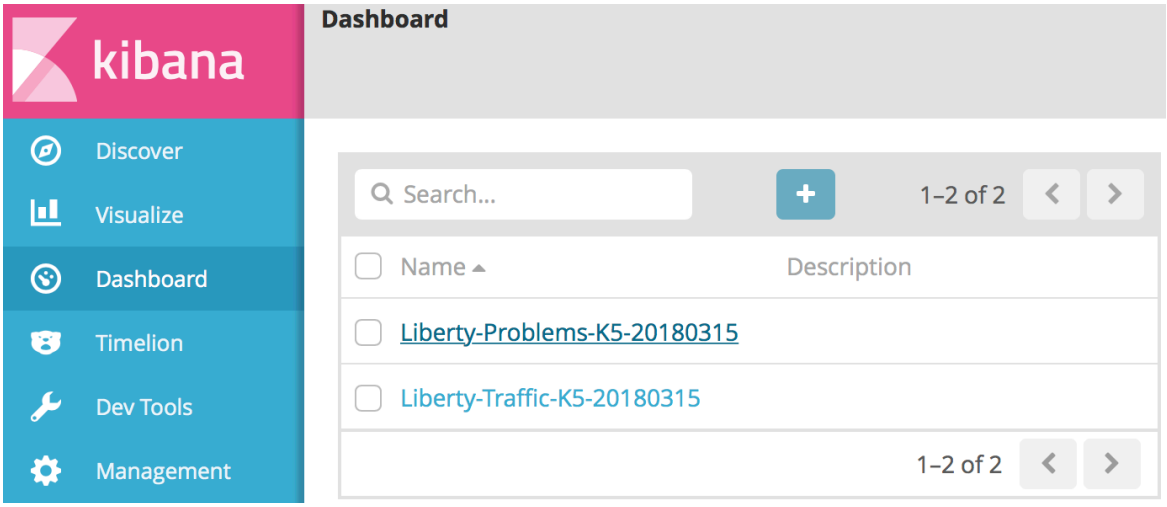


With that in place, you then will see your logs in Kibana like you'd expect:

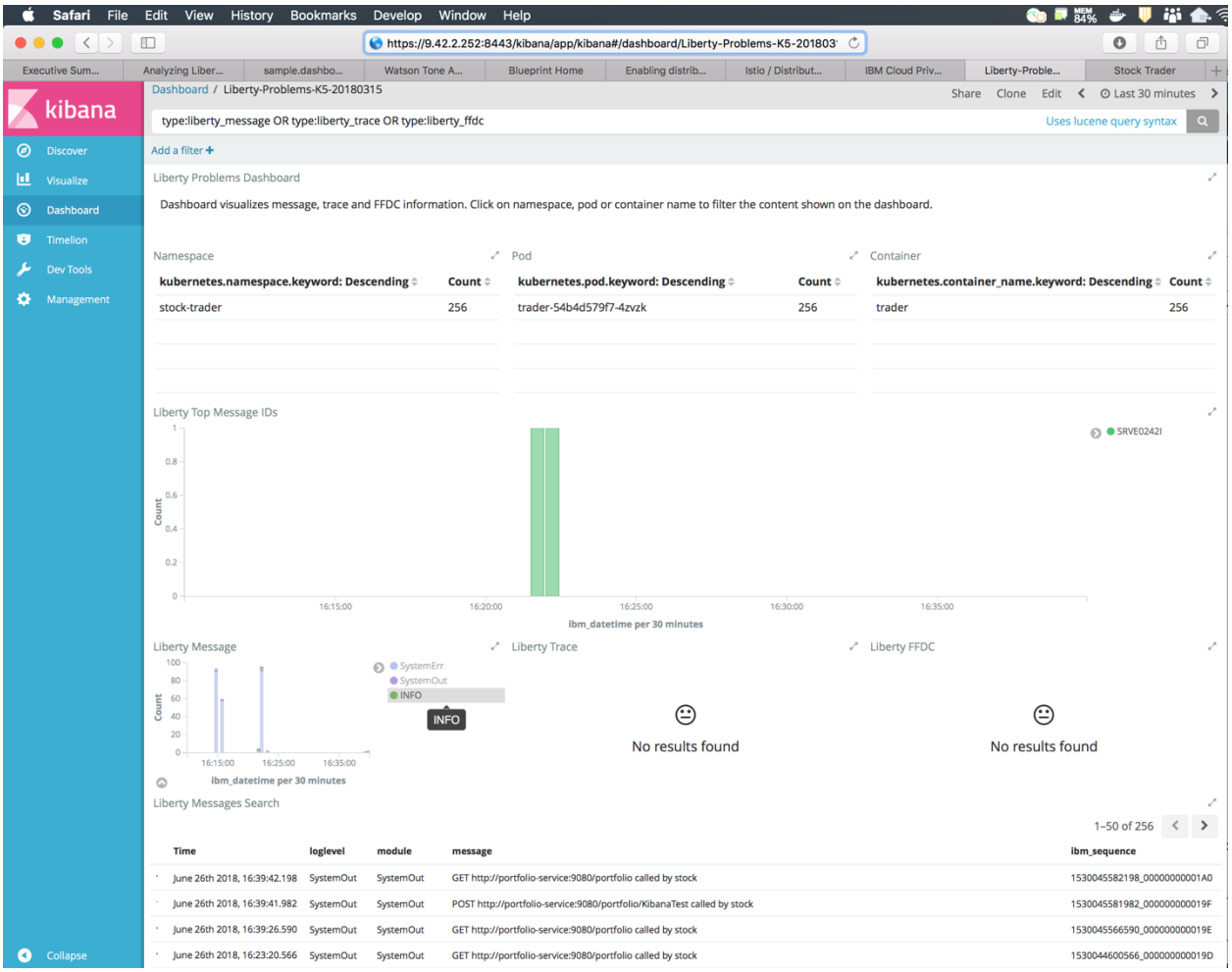


Note that the screenshots shown so far are from the **Discover** tab of Kibana. There is also a **Dashboard** tab, and Liberty now provides such dashboards (based on this new JSON Logging feature) that you can download from GitHub, as explained here: `https://www.ibm.com/support/knowledgecenter/en/SSEQTP_liberty/com.ibm.websphere.wlp.doc/ae/twlp_icp_json_logging.html`.

Basically, you just download https://github.com/WASdev/sample.dashboards/blob/master/IBMCloudPrivate_2.1.0.2/Kibana_5.x/dashboard.json and then click **Management->Saved Objects** and click the **Import** button, and navigate to where you saved that JSON file. This will make two new dashboards available:



If we then look at that first dashboard (their problem determination one), we see this:



The dashboard is interactive. For example, if you click on **INFO** in the legend for the **Liberty Message** widget, the **Liberty Messages Search** widget below will filter itself to just the loglevel=INFO messages. The nice thing about this dashboard is that it shows federated log data

from across all of your Liberty-based microservices, without all of the other noise in the system getting in the way. Note, however, that you could **NOT** use this to see a federated view of polyglot microservices; they all have to be atop Liberty, using the new JSON Logging feature, to show up here

There are also some Kibana and Grafana dashboards associated with the Liberty helm chart, that you can access from `https://github.com/IBM/charts/tree/master/stable/ibm-websphere-liberty/ibm_cloud_pak/pak_extensions/dashboards`.

Best Practices and Approaches

The developer writing the Java code should not need to do anything special to get logs to show up in a Kibana dashboard. The developer just uses the standard logging feature built in to the Java language, and everything else should happen via configuration. That is indeed the case here with Liberty-based microservices, as shown above.

The exact usage of the interfaces – like when to use `logger.info` versus when to use `logger.fine` – is something that each organization or project has to make decisions on. However, we expect that these interfaces are necessary and useful in almost any project.
One best practice is to use environment variables (fed to the pod via Kube config maps or secrets) in each relevant field in the server.xml. This allows you to change logging configuration without needing to rebuild and redeploy your Docker image.

For example, to use environment variables to set fine grained logging attributes, you would change the stanza from the previous example:

```
    <logging consoleLogLevel="INFO" consoleFormat="json"
consoleSource="message,trace,accessLog,ffdc" />
```

To something more like this:

```
    <logging consoleLogLevel="${env.LOG_LEVEL}"
consoleFormat="${env.LOG_FORMAT}"
consoleSource="${env.LOG_SOURCE}" />
```

You could also use environment variables to selectively include server configuration snippets. For example, if you added a stanza like this to your `server.xml` file:

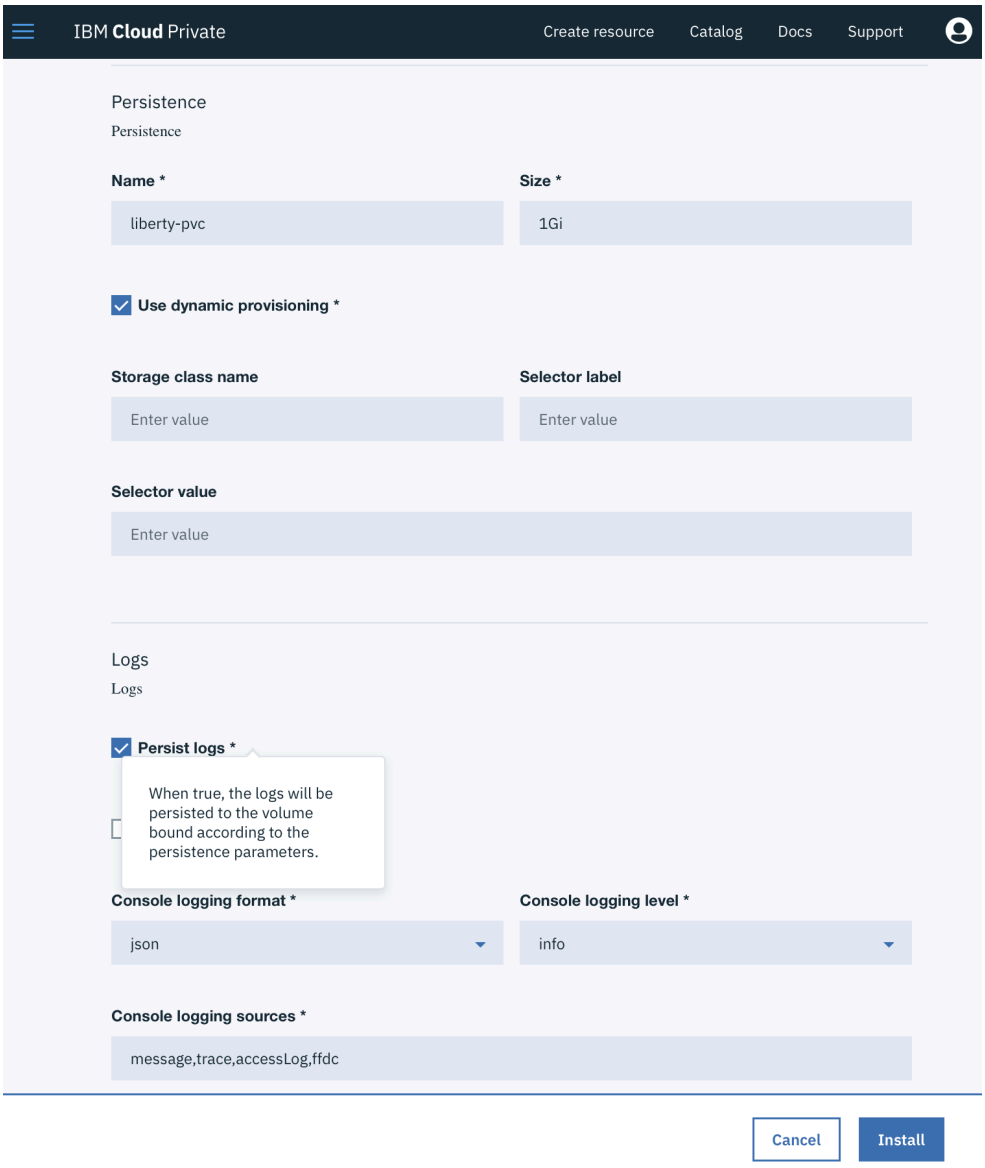
```
<include location="${env.LOG_FORMAT}-logging.xml" optional="true" />
```

You could then use a `LOG_FORMAT` environment variable to select which logging configuration to load: a value of `plaintext` would include a `plaintext-logging.xml` file containing configuration options for plain text logs, while a value of `json` would include a `json-logging.xml` file containing some of the snippets above to configure JSON logging. As the include is optional, no error will occur if the specified file can't be found (e.g. if the environment variable isn't set at all). In this case, configuration from `server.xml` or system defaults will be used instead.

Another alternative is to use the `WLP_LOGGING_CONSOLE_FORMAT` environment variable, as described here: https://www.ibm.com/support/knowledgecenter/SSEQTP_liberty/com.ibm.websphere.wlp.doc/ae/rwlp_logging.html. This is similar to the example above, in some ways: you can set the `WLP_LOGGING_CONSOLE_FORMAT` variable to either `basic` (the default) or `json`.

Finally, we should point out that it is an anti-pattern to allow your logs to be written into the filesystem of your Docker container (which unfortunately is the default). Each log entry would be a "mutation" of the docker image (tracked by ICP's *Mutation Advisor*, unless rules are defined for it to ignore stuff under /logs in the container), and would incur significant performance penalties, since Docker is optimized to be a read-only filesystem.

Instead, you should create a Persistent Volume (PV), and configure a mount point in your container to a pod-name-specific location on that PV, and configure Liberty to log there. Note that if you install your microservice via the Liberty helm chart, and choose Persist Logs, this will happen automatically:


IBM Cloud Private
Create resource
Catalog
Docs
Support
Persistence
Persistence
Name *
liberty-pvc
Size *
1Gi
Use dynamic provisioning *
Storage class name
Enter value
Selector label
Enter value
Selector value
Enter value
Logs
Logs
Persist logs *
When true, the logs will be persisted to the volume bound according to the persistence parameters.
Console logging format *
json
Console logging level *
info
Console logging sources *
message,trace,accessLog,ffdc
Cancel
Install

Programming Model Concept – Tracing

Finding and tracking errors in distributed systems is hard. Understanding system behavior requires tracking related activities across processes/programs/machines/hosts/etc. The key is being able to understand the *causal relationships* between activities in distinct processes.

An understanding of the relationships between work flowing through the systems helps with:

- user-facing latency optimization,
- root-cause analysis of backend errors,
- communication about distinct pieces of a now-distributed system, etc.

For end to end correlative trace to be useful, data must be gathered and aggregated consistently. Istio can collect trace data as traffic flows through the service mesh, but it cannot infer relationships between requests without help from the application.

OpenTracing (http://opentracing.io) is a distributed tracing instrumentation standard defining APIs that can be used by application and shared library developers to add instrumentation for distributed, cross-process tracing in a vendor neutral way.

Key Concepts for Tracing

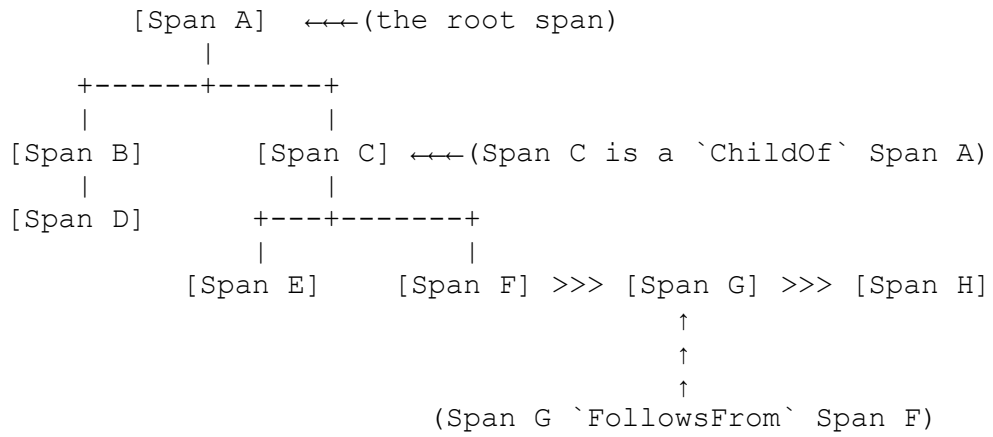
The foundational concepts for distributed tracing are *Traces* and *Spans*. *Traces* are the "story" of a transaction or workflow as it makes its way through a system. They are represented as directed acyclic graphs. A *Trace* is made up of *Spans*, which each represent one component of the story.

- **Trace** – a directed acyclic graph (DAG) of Spans
- **Span** – a representation of a timestamped operation (vertices in DAG)
  - operation name
  - start/finish timestamps
  - Span Tags (key:value)
  - Span Logs (key:value map + timestamp)
  - **SpanContext**: *Tracer-specific ids and dependent state*, and other cross-process state
  - References to causally-related Spans
- Reference – edges between Spans, which model **direct causal relationships**
  - `ChildOf` – used when the parent depends on the child span
  - `FollowsFrom` – used when one span follows from another
- **Tracer** – the actor responsible for creating, collecting, and propagating Spans.

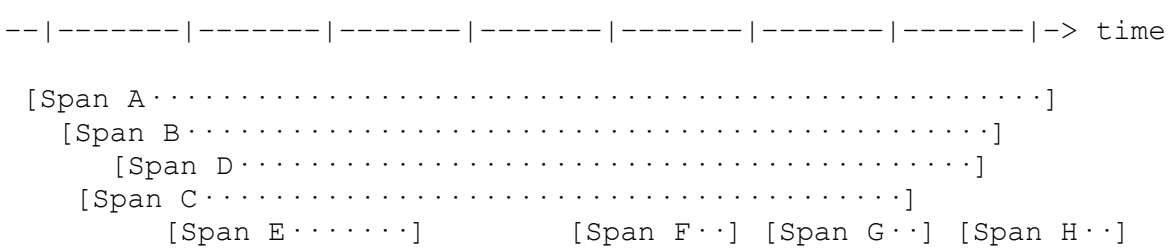
See more: https://github.com/opentracing/specification/blob/master/specification.md

Each trace starts with a span. Spans create new spans with two types of relationships that express both the semantics in the system (`FollowsFrom`), and the critical path for latency-sensitive (distributed) operations (`ChildOf`).

Causal relationships between Spans in a single Trace



Temporal relationships between Spans in a single Trace



The OpenTracing API

Istio should be leveraged to ensure tracing data is gathered consistently as requests flow through the system.

The application should provide information about the relationship between requests. That could be done either via the OpenTracing API, or through other language libraries that automatically propagate tracing headers between requests.

See more: http://opentracing.io/documentation/pages/api/cross-process-tracing.html

Programming Model Approach - Tracing

The recommended approach for tracing is to leverage Istio tracing. Look for details about this feature in the “”Creating Microservices – Polyglot Capabilities” chapter under “Istio – Tracing”.

The alternative to Istio tracing which can be leveraged for java is mpOpenTracing. Additional details about this capability can be found in the “Additional Java Features of Note” chapter under “MicroProfile Tracing”.

Programming Model Concept – Metrics

Kubernetes uses Prometheus for tracking relevant metrics for microservices, and uses Grafana to visualize the results. These infrastructure pieces are automatically configured in both IBM Cloud Private and the IBM Kubernetes Service. Certain metrics are automatically captured and displayed, like how many pods are healthy. You can contribute your own custom metrics as well.

Application metrics are captured as time series data. Aggregating and visualizing captured metrics can help to identify common performance problems such as:

1. Slow HTTP response times on some or all routes
2. Poor throughput in the application
3. Spikes in demand that cause slowdown
4. Higher than expected CPU usage
5. High or growing memory usage (potential memory leak)

Dimensional-metrics
Naming conventions

Time series: throughput, total time, maximum latency, percentiles, histograms, etc.
From Observability overview: RED Method. Rate, Errors (count), Duration

Programming Model for Java – Metrics

MicroProfile provides mpMetrics, which let you annotate your source code saying what you'd like to track in your application. You just add the `mpMetrics-1.1` feature to your server.xml to enable this. Note you can optionally also add the `monitor-1.0` feature, if you'd like additional app server specific metrics (like about JDBC connection pools, for example).

For example, in our *Portfolio* microservice, we are asserting that we would like for it to track a count of how many portfolios have been created. We do this first by importing the `Counted` annotation:

```
import org.eclipse.microprofile.metrics.annotation.Counted;
```

And then we add the `@Counted` annotation to the *createPortfolio* method:

```
    @POST
    @Path("/{owner}")
    @Produces("application/json")
    @Counted(monotonic=true, name="portfolios",
displayName="Stock Trader portfolios", description="Number of
portfolios created in the Stock Trader application")
    @RolesAllowed({"StockTrader"})
    public Portfolio createPortfolio(@PathParam("owner") String
owner) throws SQLException {
```

To build this code, we need to add the following stanza to our Maven pom.xml (note that `<version>2.0</version>` has a bug that will result in a build error; you must use `2.0.1`):

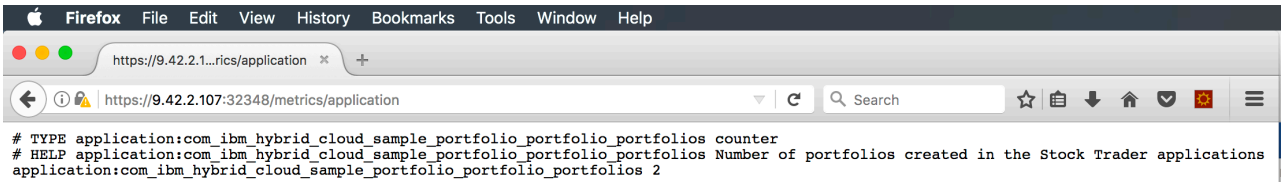
```
    <dependency>
        <groupId>org.eclipse.microprofile</groupId>
        <artifactId>microprofile</artifactId>
        <version>2.0.1</version>
        <type>pom</type>
        <scope>provided</scope>
    </dependency>
```

With that in place, every time the `createPortfolio` JAX-RS method gets called, the counter will be incremented. We can view the value of the counter by hitting the `GET /metrics` URI, to see all of the metrics (like counts of number of classes loaded, heap space used, when garbage collection last ran, etc. - plus any application-defined metrics), or you can hit `GET /metrics/application` to see just the application-defined metrics. We can hit this REST GET API via the `curl` CLI (we're specifying the node port for the http (9080) port, which is 32388 in this case):

```
Johns-MacBook-Pro-8:StockTrader jalcorn$ curl
http://9.42.2.107:32388/metrics/application
# TYPE application:com_ibm_hybrid_cloud_sample_portfolio_portfolio_portfolios counter
# HELP application:com_ibm_hybrid_cloud_sample_portfolio_portfolio_portfolios Number
of portfolios created in the Stock Trader applications
application:com_ibm_hybrid_cloud_sample_portfolio_portfolio_portfolios 2
Johns-MacBook-Pro-8:StockTrader jalcorn$
```

As you can see, it has counted that we have created 2 portfolios. Note that this is an in-memory counter, so if the pod is bounced, it will be reset to zero. Note also that if you have scaled up the portfolio microservice to multiple pods, this would be counting how many each pod has created only; there's no consolidated count across pods (unless you have Prometheus configured to scrape the `/metrics` endpoint of your pods – then you would see consolidated results from within the Prometheus or Grafana UI).

Note also that the "`# HELP`" text is what we specified as the `description` in the `@Counted` annotation. Of course, you can view the output of this REST GET endpoint in your web browser as well (in this case, we chose the node port for the https port instead):



Note that by default, the `/metrics` endpoint requires https and login credentials be passed. But Liberty 18.0.0.3 introduced the following stanza you can put in your server.xml to say you want this endpoint to allow http and to be unauthenticated:

```
<mpMetrics authentication="false"/>
```

This makes configuring the Prometheus scraper to hit this endpoint much easier. Note that if you deploy your microservice via the Liberty helm chart, checking the "Enable monitoring" checkbox will automatically configure the Prometheus scraper for you, to periodically "scrape" the `/metrics` endpoint.

If you need to manually configure the Prometheus scraper, follow the guidance in this article (despite mpFT being in the URL, it does talk about wiring up Prometheus and Grafana to `/metrics` - it also discusses additional metrics that mpFT will expose): https://openliberty.io/blog/2018/10/05/microprofile-faulttolerance-1.1-metrics.html

Tracing Versus Logging Versus Metrics

Logging is used when the developer wants to explicitly output some message for someone to see. It is coded directly into the Java class, including passing along values of relevant variables. When problems occur, the logs are useful for debugging purposes, showing where a failure occurred, such as a stack trace for an exception that got thrown. As discussed above, you use *Kibana* to see a federated view of such logs across pods/microservices.

Tracing, on the other hand, simply happens automatically, without the developer needing to take any action. For example, you can configure Liberty to send trace records to an Open Tracing

compliant trace server whenever any JAX-RS annotated method is invoked. This way you have an audit record of what got called when, by whom, and how long it took. You can also augment that trace, like to include info about what private methods have been called in your code, by adding Open Tracing annotations to such methods to say that they should be traced as well. As we discussed, you can use a tool like *Zipkin* or *Jaeger* to show a federated view of traces across pods/microservices. A *service mesh* can also provide automatic tracing (of calls passing through a side car for your container) – see the *Istio* chapter for details.

Metrics are used to track aggregate values. Like instead of wading through a trace or a log to see how often someone is creating portfolios, you can make that a custom counter metric (as we discuss in the *mpMetrics* section). You can label your deployment such that it gets "scraped" by *Prometheus*, such as periodically hitting the `/metrics` URI on your pod(s). You can then use a tool like *Grafana* to get a federated view of metrics across pods/microservices.

You can look at tracing to find out "this method got called". But you'd look at logging to find out "this is what happened inside that method when it got called". And you'd look at metrics to see "here's how many times it's been called". Generally tracing is implicit, happening automatically without any effort on the part of the developer. Whereas logging is explicit, requiring the developer to code up sending info that might be relevant for post-mortem analysis of a problem. Metrics are also explicit, in so far as you need to add the annotation to the appropriate method of your code (though there often low-level default metrics available without effort on the part of the programmer, like for memory usage, CPU usage, thread counts, etc.).

Generally, metrics are useful for analytics, whereas logging is more useful for problem determination purposes, and tracing is more useful for understanding control flow from microservice to microservice.

Programming Model Concept – Fault Tolerance

Creating a resilient system places requirements on all of the services in it. As mentioned earlier, the dynamic nature of cloud environments demands that services be written to expect and respond gracefully to the unexpected. This could mean receiving bad data, being unable to reach a required backing service, or dealing with conflicts due to concurrent updates in a distributed system.

Microservices must also prevent failures from cascading through the system. Fault tolerance mechanisms like circuit breakers and bulkheads aim to address this problem.

Programming Model Approach for Java – Fault Tolerance

Our recommended approach for these fault tolerance topics is to mainly leverage the features of Istio. See "Istio – Fault Tolerance" in the "Creating Microservices – Polyglot Capabilities" chapter of this book. This covers things like retries, timeouts, circuit breakers, bulkheads and rate limits.

MicroProfile also offers an approach, which is outlined in in the "Additional Java Features of Note" chapter under the heading "MicroProfile Fault Tolerance". That section shows how the fallback capability can be used in conjunction with Istio, as well as some more details on those interested in an approach that leverages mpFaultTolerance instead of Istio.

Istio is not capable of offering fallback capabilities, because fallback requires business knowledge. A more advanced approach is to use Istio fault tolerance together with MicroProfile fallback capabilities to achieve the maximum of resilience. For instance, you can specify a fallback backup when calling a backend service. If Istio cannot manage to get a successful return, the fallback method will be invoked.

```
@GET
@Fallback(fallbackMethod="myFallback")
public String callService() {
      //call servicer b
      return bclient.hello();
}

public String myFallback() {
      return "Greetings... This is a fallback!";
}
```

If you use Istio's fault tolerance capability, you can turn them off MicroProfile Fault Toelrance by setting the config property MP_Fault_Tolerance_NonFallback_Enabled to false.

```
apiVersion: v1
kind: ConfigMap
```

```
metadata:
   name: service-config
data:
     MP_Fault_Tolerance_NonFallback_Enabled: "false"
```

Programming Model Concept – Observability (Health Checks)

Health checks provide a simple mechanism within automated systems to introspect an individual instance for health status. The system then responds to these health inspection events by taking an action, such as replacing a failed instance, updating routing tables, or communicating the resulting health states to the user through an observable property.

Kubernetes defines two integral mechanisms for checking the health of a container:

- A ***readiness*** probe is used to indicate whether a container can handle requests (is routable).

  Kubernetes doesn't route work to a container with a failing readiness probe. A readiness probe should fail if a service hasn't finished initializing or is otherwise busy, overloaded, or unable to process requests.
- A ***liveness*** probe is used to indicate whether the process should be restarted.

  Kubernetes terminates and restarts a container with a failing liveness probe to ensure that containers in a defunct state are terminated and replaced. A liveness probe should fail if the current process state is irrecoverable, like out of memory conditions or deadlocks. Simple liveness checks that always return an OK response can identify containers in an inconsistent state, for example, where the process serving requests has crashed but the container is still running.

Readiness and liveness probes are both defined using a similar structure which includes time delay and retry intervals, failure tolerance periods, timeouts as well as the definition of the probe implementation. The probe can be implemented by executing a command, checking a TCP endpoint for connectivity or performing an HTTP invocation. Often the same probe implementation can be used for both readiness or liveness purposes, but the time delay and retry intervals will likely need to be adjusted for the particular purpose.

Understanding and applying readiness and liveness probes in Kubernetes

Fundamentally, cloud native application development is founded upon the principal that container processes do in fact fail, but these processes are readily replaced by a new container. This happens both in response to unexpected events like container or machine failure, but also due to operational events like horizontal scaling and new application image rollouts. Due to the fact that containers tend to come and go, readiness checks are important because they assure that new container instances are in fact ready to receive work prior to routing traffic, and the same

checks prevent traffic from being routed to instances which have exited or are being destroyed. When readiness checks have not been defined, Kubernetes has little insight into whether a container instance is ready to handle traffic and will begin routing traffic immediately after the container process is started. Without readiness checks, client applications are more likely to experience connectivity timeouts and connection rejection responses when work is routed to an instance which is not ready to serve the request. Readiness checks reduce but will not completely eliminate such client connectivity errors.

While changes to instance routing targets are quite normal within the lifecycle of a container enabled application, the process states which liveness is intended to identify are less frequent and represent an exception rather than the norm. Sometimes a process will enter a state from which no recovery is possible, the process is effectively defunct. Some examples of why this might happen include out of memory conditions or a semaphore which has not been released due to a programming error. The counteraction to such events is termination of the container, an action which is invasive since any processing currently happening in the container is also terminated. This also creates the possibility of terminate/restart loops in the application, where containers are unable to completely come online before being terminated and replaced, only for the replacement container to undergo the same fate.

Readiness and liveness probes therefore impact the system in different ways. This can be thought of in terms of state transition, where the positive state of a readiness check is where it is routable, while the negative state is unrouteable. Likewise, the positive state of a liveness check represents a container which is running normally while the negative state is a defunct process. When a container starts, the readiness state will initially be negative, and only changes to positive once the container is healthy. In opposition, a liveness check starts in a positive state, and only enters a negative state when the process goes defunct. Because of this dichotomy, configuring a readiness check very aggressively, such as with a low initial delay will have little effect since running the probe too soon will not cause the readiness check to change state. On the other hand, an aggressive liveness check, where the probe fires too soon, will cause a state transition change causing the system to terminate the container sooner than intended.

Configuring readiness and liveness probes in Kubernetes

Declare liveness and readiness probes alongside your Kubernetes deployment in the container element. Both probes use the same configuration parameters:

- The kubelet waits for `initialDelaySeconds` after the container is created prior to the first probe.
- The kubelet probes the service every `periodSeconds` seconds. The default is 1.
- The probe times out after `timeoutSeconds` seconds. The default and minimum value is 1.
- The probe is successful if it succeeds `successThreshold` times after a failure. The default and minimum value is 1. The value must be 1 for liveness probes.
- When a pod starts and the probe fails, Kubernetes tries `failureThreshold` times to restart the pod before giving up. The minimum value is 1 and the default value is 3.
  - For a liveness probe, “giving up” means restarting the pod.

- For a readiness probe, “giving up” means marking the pod `Unready`.

To avoid restart cycles, set `livenessProbe.initialDelaySeconds` to be safely longer than it takes your service to initialize. You can then use a shorter value for `readinessProbe.initialDelaySeconds` to route requests to the service as soon as it’s ready.

An example configuration might look something like this (note the `path` and `port` values):

```
spec:
  containers:
  - name: ...
    image: ...
    readinessProbe:
      httpGet:
        path: /health
        port: 8080
      initialDelaySeconds: 120
      timeoutSeconds: 5
    livenessProbe:
      httpGet:
        path: /liveness
        port: 8080
      initialDelaySeconds: 130
      timeoutSeconds: 10
      failureThreshold: 10
```

For more information, see how to Configure Liveness and Readiness Probes in Kubernetes.

When implementing a health probe using HTTP, consider the following HTTP status codes for readiness, liveness and health.

State	Readiness	Liveness
	Non-OK causes no load	Non-OK causes restart
Starting	503 - Unavailable	200 - OK
Up	200 - OK	200 - OK
Stopping	503 - Unavailable	200 - OK
Down	503 - Unavailable	503 - Unavailable
Errored	500 - Server Error	500 - Server Error

Due to their role in automated environments, health check endpoints must not require authorization or authentication. Since these protections are not put into place on health probe endpoints, any HTTP probe implementations should be restricted to GET requests which do not modify any data, and should only return simplified responses like “okay”. Never return data which identifies specifics about the environment, like the operating system, implementation language or software versions as these can be utilized by would be malicious actors in order to establish an attack vector.

Recommendations for readiness and liveness probes

Liveness probes should include the viability of connections to downstream services like databases or other microservices in their result only when there isn't an acceptable fallback. That is, unless the status of a downstream service would cause the local container to enter an unrecoverable state for which the only remedy is a termination/restart, downstream status should not be factored into the probe implementation. In general, since health checks occur at frequent intervals, checking downstream services introduces additional overhead as well as unintended consequences such as cascading failures. This only makes detecting the scope of an individual failure more difficult to identify.

When configuring the initial time delay (initialDelaySeconds attribute), a readiness probe should use the lowest likely value, while a liveness check uses the largest likely time value. For instance, if an application server tends to start in 30 seconds, a typical readiness delay would be 10 seconds, while the liveness check would use a 60 second value in order to assure that server start always completes before checking for terminatable states.

The periodSeconds attribute for routing decisions is typically configured to a single digit value, provided that the probe implementation is relatively lightweight. For instance, an http probe which returns a 200 okay status without substantial server side processing has a minimal processor load and can be readily repeated ever 1-5 seconds.

A liveness probe should be very deliberate about what it checks, as a failure results in immediate termination of the process. For example, it is best to avoid ambiguous metrics which only sometimes indicate a failing process. Instead, a simple http endpoint that always returns `{"status": "UP"}` with status code `200` is a reasonable choice since most processes that have entered a zombie state will fail this check, triggering a restart.

The following sections provide examples for how to write readiness and liveness checks.

Liveness and Readiness Checks with JAX-RS

As discussed in the prior section, Kubernetes provides multiple methods for integrating health probes, including execution of a command, as well as network checking through TCP or HTTP endpoints. Although it is possible to implement any of these probes in the Java language, a JAX-RS implementation of HTTP probes will be discussed in detail here.

Defining a readiness endpoint

A minimal readiness JAX-RS probe implementation looks something like the following:

```
@Path("ready")
public class ReadinessEndpoint {
  @GET
  @Produces(MediaType.APPLICATION_JSON)
  public Response testReadiness() {
        return Response.ok("{\"status\":\"UP\"}").build();
  }
}
```

An endpoint like this in an application deployed in WebSphere Liberty achieves a certain level of readiness behavior without additional effort. Liberty will fail requests to the health endpoint with an appropriate 503 response until the application has started. This basic implementation should be extended to include required capabilities. For example, consider evaluating `@ApplicationScoped` CDI resources to ensure that dependency injection processing has completed. Once the probe has been implemented, it can be enabled by configuring the HTTP probe type as noted in the prior section.

Always remember the role and purpose of fault tolerance: microservices are expected to handle failure and disruption in communications with downstream services. Only include checks for capabilities that have no feasible fallback inside a readiness check.

Defining a liveness endpoint

A liveness probe should be very deliberate about what it checks, as a failure will result in immediate termination of the process. As with the readiness example above, a simple liveness endpoint begins with a returning an 'ok' response:

```
@Path("liveness")
public class LivenessEndpoint {
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response testLiveness() {
      return Response.ok("{\"status\":\"UP\"}").build();
    }
}
```

The liveness probe implementation can be extended to consult process local conditions which indicate an unrecoverable process state. However, the challenge with such checks often lies in the fact that if the such processing is able to be performed, the server is likely sufficiently healthy to perform other requests. For this reason, a "keep it simple" mantra should be readily applied to the liveness check implementation. Once the liveness sources have been coded to enable the probe endpoint, the http liveness endpoint can be enabled in the container orchestration sources as explained in the last chapter.

To avoid restart cycles, the initialDelaySeconds attribute for the liveness check should be longer than the longest expected server start time. For a Java application server that commonly takes 30 seconds to start, choose a larger value, such as 60 seconds.

Using MicroProfile health checks

In addition to implementing application specific JAX-RS endpoints, MicroProfile also provides integrated support for health checks. More details about this mechanism are detailed in the section titled "Microprofile Health API".

Programming Model Concept – Auto-Scaling

The promise of Kubernetes with the ability to automatically schedule and scale your application deployments is enticing. However, to make your microservices-based application automatically scale requires some planning by developers and cloud operations.

The concepts of auto-scaling are not unique to microservices, but what you need to know and understand is different. Developers do have a contribution to make in this area. Some of that contribution comes by providing liveness and readiness probes in the code itself, and the rest comes from knowing about the behaviors of each microservice and interactions between microservices.

This section will briefly introduce the concept of auto-scaling in Kubernetes. This will be followed by more concrete insights and supporting examples for specific languages and frameworks.

Ingredients for Auto-Scaling

The following concepts need to be considered when you want to have your application scale in a Kubernetes environment:

Readiness and Liveness probes

As mentioned in an earlier section, these probes are essential because they let your microservice communicate to Kubernetes. The Readiness probe tells Kubernetes that the pod is up and ready to receive traffic. The liveness probe tells Kubernetes that the pod is still alive and working as expected.

The trick for the developer is to communicate meaningful status. In our example, if our "portfolio" microservice cannot connect to its data source, then the pod is not ready to receive traffic. The microservice may be up and kicking and in its own world quite healthy. However, since the portfolio microservice depends on the data source to set/get information, it will not want to receive traffic until it has an active data source connection. Once it is "ready", it then checks for consecutive errors. If portfolio detects three consecutive errors for any communication, then the developer assumes there's a problem and the pod should be killed and a new one started. Of course each microservice may require a unique definition of "am I ready" and "am I living".

Finally, while readiness and liveness probes are not required for auto-scale policies, they are essential so that the pods that are running are productive. Otherwise, the auto-scaler may detect 5 pods running under 50% utilization, but none of them are doing actual work. Timing is important, too. If the "readiness" probe says the pod is "ready" too soon, then work will be routed before the app instance is ready and the transaction will fail. If the "readiness" probe takes too long to say the pod is "ready", then the existing pods will become overworked and could start failing due to capacity limits.

Readiness and liveness probes are mentioned again because your autoscaling needs to take into account how long it takes for additional pods to become available, and that a liveness probe that

is too sensitive can cause your autoscaling to thrash needlessly due to pods being destroyed when they are not truly dead.

Resource requests and limits
Each microservice should specify how much resource (memory and CPU) is required to run. While not required to run in Kubernetes, these resource requests and limits are required for autoscalers to kick in. In Kubernetes, you will want to specify resource requests (how much CPU and Memory to allocate initially when bringing up the pod), and resource limits (the maximum CPU and Memory to allocate over time).

Here's a sample that a deploy.yaml file would have to specify resource requests and limits:

```
resources:
  limits:
    cpu: 500m
    memory: 500Mi
  requests:
    cpu: 100m
    memory: 128Mi
```

Horizontal Pod Autoscaler
The Horizontal Pod Autoscaler (HPA) is created by attaching itself to a deployment or other Kube object that controls pod scheduling. In its most basic form, the HPA will look at all pods in a deployment, average the current CPU utilization across all the pods, and if the percent utilized is more than what the HPA is set to, then the HPA will provision additional instances. Conversely, if the percent utilized is less than what the HPA is set to, then the HPA will remove instances until the utilization is approaching what was desired.

Example:
Here I'm creating an HPA that attaches itself to the deployment called "portfolio", and will scale the number of pods between 2 and 10, depending on if the average CPU utilization is above or below 50%.

```
kubectl autoscale deployment portfolio --cpu-percent=50  --min=2   --max=10
```

Learn more here.

Vertical Pod Autoscaler
An emerging capability in Kubernetes (code is at 0.4.0) is vertical pod autoscaling (VPA)…which means it can scale the resource requests / limits in real-time so an individual pod can take up more resource.

VPAs can be quite useful for applications that are not written to horizontally auto-scale. If it needs more resource then it can vertically scale. While the best practice is to architect your application for auto-scale, there are some cases where it is not possible and VPA is a good option.

Further, VPAs can be powerful companions to HPAs. For example: If I have an HPA provisioning new pods, it may come up against capacity limits for available worker nodes. In that case, a new pod will not be provisioned. This can happen when the Kubernetes scheduler fails due to affinity rules, resource availability for full pods, readiness probe indicating delay, etc). If you set a VPA to scale the pods vertically in that situation (give each pod 10-20% more CPU and/or Memory), then the existing pods could have some "breathing room" and grow with the small amount of resource still available in the worker node they're running in.

Learn more here.

Node Affinity, Taints and Tolerations
Node affinity, taints and tolerations allow users to constrain which worker nodes an application can be deployed to.

An affinity rule tells Kubernetes what kind of worker node the pod requires or prefers (you can think of it as "hard" or "soft" compliance). The affinity matching is based on node labels. This means that you can prefer that your pod runs on any of the 5 of the 50 worker nodes in the cluster that are labeled as "GPU-enabled", and also require that the nodes run on any of the 20 of the 50 worker nodes labeled as "Dept42".

To view the labels on a node, run `kubectl get nodes -o wide --show-labels`

Taints and Tolerations tell Kubernetes what kind of pods a worker node will accept. This means that a set of worker nodes can be tainted with "old-hardware" and only pods that have a toleration for "old-hardware" will be scheduled.

Think of it this way:
affinity is pod-focused: "I am a pod…there are so many nodes to choose from let me filter so I get one I will run well on". Affinity lets the pod choose what nodes it will deploy into based on how the nodes are labeled. "I prefer new hardware nodes" "I require GPU-equipped nodes" and then Kubernetes will schedule appropriately.

Taints/Tolerations is node-focused: "I am a worker node…there are so many pods out there, but only certain ones will run well on me due to my unique traits…I will add a taint to require pods to declare that it will tolerate this unique trait". Taints let the node declare "I am old hardware" or "I am only for production" and as a result, the Kubernetes scheduler will only deploy pods onto those nodes if the pod has a toleration matching that taint: "I am production pod" or, "I will run on old hardware".

This is one reason why clear communication between development and cloud ops is essential.

Learn more here and here

Resource Quota
Each namespace in Kubernetes can have a resource quota. This is set up by the cloud admin to limit how much resource can be consumed within a namespace. As this is a hard limit (actual

number, not percentage), once an application scales up to the quota limit, it will get a failure if trying to auto-scale beyond the quota.

Further, if a quota is set on a namespace, all pods must have resource requests/limits specified or the scheduling of that pod will fail.

This is yet another reason why clear communication between development and cloud ops is essential.

Learn more here.

Other Isolation Policies
IBM Cloud Private has support for many levels of multi-tenancy. This includes the ability to control workload deployments to specific resources like worker nodes, VLANs, network firewalls, and more based on teams and namespaces.

Have I mentioned that your development team needs to have clear communication with your cloud ops team? I think so.

Learn more here.

Best Practice for Setting up Apps that Auto-scale

Here is a summary of what you should consider when auto-scaling your application. The actual numbers may vary based on your application, so use this as guide.

- As you initially create your microservice, give your resource requests/limits a sensible default. This could vary greatly depending on your base image, runtime, etc. For example, for Liberty runtimes, set it between 200m and 1000m for CPU, and 128Mi and 512Mi to Memory. If you set the limits too low, the autoscaler may trigger an additional instance during initial startup because it's taking too much CPU.
- Run initial unit tests and use Grafana monitoring to see actual CPU and Memory usage during your tests.
- Refine your resource requests/limits as needed so your resource limits are 20% more than monitored maximum usage.
- Set your initial auto-scale policy to scale your application at 50% of CPU usage.
- If you use Istio, increase all resource limits by 100m CPU and 128Mi Memory to handle the additional side-car.
- Work with your cloud ops team if any affinity rules or resource quotas will be in place in the clusters across your stages of delivery: development, test, stage, production.

Note: For classic WebSphere applications that you migrated into containers using Transformation Advisor, you may not be able to auto-scale. That said you should still create good liveness and readiness probes and set your resource request/limits.

Next, let us step through how that best practice is set up using Liberty-based microservices.

Auto-Scaling WebSphere Liberty based Microservices

This section will step through what changes were made to Stock Trader in order to have it automatically scale based on CPU utilization along with liveness and readiness probes.

Step 1: Add Readiness and Liveness Probes

The readiness and liveness probes we added started with a very basic URL that said "this liberty runtime is up".

```
readinessProbe:
  httpGet:
    path: /health
    port: 9080
  initialDelaySeconds: 30
  periodSeconds: 5
livenessProbe:
  httpGet:
    path: /health
    port: 9080
  initialDelaySeconds: 60
  periodSeconds: 5
```

Over time, we added custom probes that were unique for each microservice. As mentioned above, the "Portfolio" microservice was "ready" only when it could connect to its data source. The "Stock-Quote" microservice, however, was only ready when it could connect to the stock quote API provided by API-Connect.

NOTE: as you develop your liveness probe, make sure you have resource limits and a low auto-scale maximum set. In our initial testing, the liveness probe would always fail, which resulted in an immediate restart. However, the HPA declared "I need at least 2 running at average 50% CPU utilization", and since the startup of a pod took more than 50% CPU, in a very short period of time after we deployed portfolio, we ramped up from 2 pods to 10. It could have easily scaled to 100's if we had set the maximum that high. Therefore during these initial stages, we suggest you declare your resource limits, and auto-scale to a small maximum (10, for example).

Step 2: Add Resource Requests/Limits

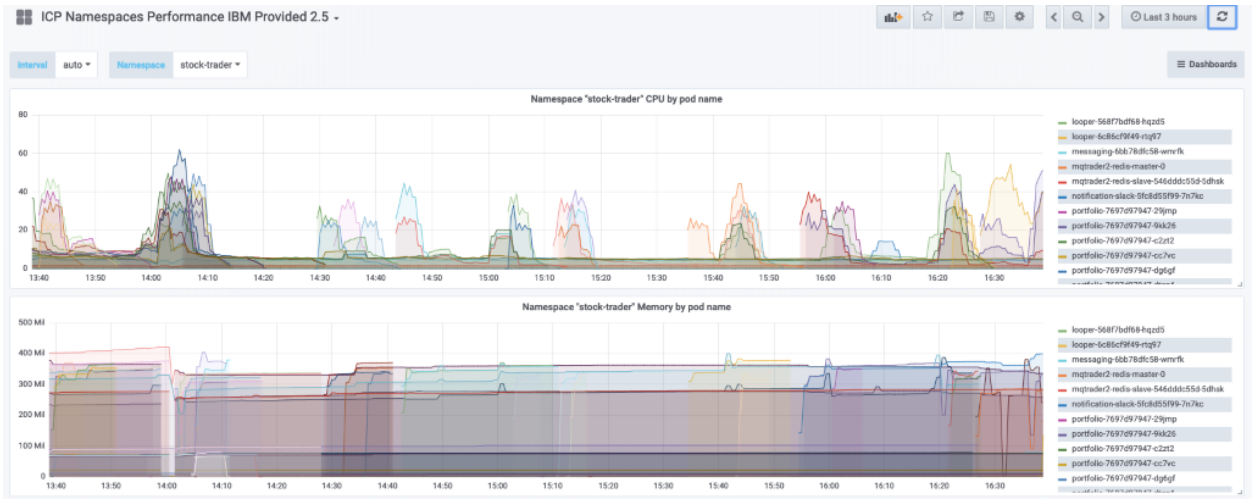
As suggested in the best-practice, if you have a Liberty-based microservice, use the following resources as a starting point:

```
resources:
  limits:
    cpu: 500m
    memory: 512Mi
  requests:
    cpu: 100m
    memory: 128Mi
```

However, as soon as you can, run your microservice with a decent load (simulated if needed) so that you can start graphing your microservice. We used the "ICP Namespace Performance IBM Provided 2.5" Grafana dashboard. This gave us the cleanest example of how our pods ran. If you

are using Microclimate, you could also open its “App Monitoring” view and add load onto your Java application.

In this image, we are showing the Stock Trader microservices and we were able to monitor CPU and memory usage in our initial tests:



Once you verify the range of your microservice CPU and memory usage, refine the resource requests and limits.

A couple last notes on resource request/limits:

- These resource values are not only critical for auto-scale, they’re also used in basic Kubernetes scheduling. This means that if you specify values that are too high, Kubernetes may not find a worker node with enough resource. We found many times cases where ICP catalog content was sized too high and we could run in development just fine with much less CPU and memory.
- Early in development, you may not specify resource requests/limits. For us, this was so that we could have one pod to debug and view logs through. Kubernetes dynamically altered resources for that single pod as long as there was resource available in the compute node. However, we found quickly that as soon as we added resource limits, the pod, during initial testing, could hit resource limits quickly. In summary, as soon as you add resource limits, you will want to create your auto-scale policy to handle additional resource needs.

Step 3: Create Auto-scale policies

When first starting out, we recommend creating your first autoscale policy through the CLI. The following example is creating an HPA called “portfolio” to attach to the deployment called portfolio, and manage the number of pods between 2 and 10, based on average CPU % utilization across all “ready” pods so that the average stays around 50%.

```
kubectl autoscale deployment portfolio -n stock-trader --cpu-percent=50  --min=2   --max=10
```

To monitor the hpa, run the following to get the table below:

```
kubectl get hpa -n stock-trader
```

```
NAME        REFERENCE              TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
portfolio   Deployment/portfolio   8%/50%    2         10        2          2m
```

Notice in the above output that the “portfolio” HPA requires at least 2 pods to be running and will alter so that the real average CPU % is roughly equal to the target average CPU %. In this case notice that since we do not have any load on the pods, CPU % is only 8%. But since the minimum number of pods is 2, it cannot scale down any further.

Once further testing is done, then we added the HPA into the yaml file. Now, every time the “portfolio” microservice is deployed, the associated HPA will also be created:

```
---
        #Deploy the autoscaler
        apiVersion: autoscaling/v1
        kind: HorizontalPodAutoscaler
        metadata:
          name: trader
        spec:
          scaleTargetRef:
            apiVersion: apps/v1
            kind: Deployment
            name: trader
          maxReplicas: 10
          minReplicas: 2
          targetCPUUtilizationPercentage: 50
```

Step 4: Edit Auto-scale policies

Initially, when we created the HPAs through `kubectl`, editing was quite messy. As a result, even the docs suggested to delete and recreate the policies. This was simple `kubectl delete hpa portfolio`. However, we found that after we created the autoscales policies through the deploy.yaml file, editing the HPA through `kubectl` was easy. In the edit file below, the “maxReplicas” and “minReplicas” and “targetCPUUtilizationPercentrage” is simple to edit.

```
kubectl edit hpa portfolio

# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this file will be
# reopened with the relevant failures.
#
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  annotations:
    autoscaling.alpha.kubernetes.io/conditions:
'[{"type":"AbleToScale","status":"True","lastTransitionTime":"2019-02-
27T23:05:00Z","reason":"ReadyForNewScale","message":"the
      last scale time was sufficiently old as to warrant a new
scale"},{"type":"ScalingActive","status":"True","lastTransitionTime":"2019-02-
27T21:33:54Z","reason":"ValidMetricFound","message":"the
```

```
      HPA was able to successfully calculate a replica count from cpu resource utilization
      (percentage of request)"},{"type":"ScalingLimited","status":"True","lastTransitionTime":"2019-02-
27T22:55:54Z","reason":"TooFewReplicas","message":"the
      desired replica count is increasing faster than the maximum scale rate"}]'
    autoscaling.alpha.kubernetes.io/current-metrics:
'[{"type":"Resource","resource":{"name":"cpu","currentAverageUtilization":5,"currentAverageValue":"5m"}}]'
  creationTimestamp: 2019-02-27T21:32:19Z
  name: portfolio
  namespace: stock-trader
  resourceVersion: "18440745"
  selfLink: /apis/autoscaling/v1/namespaces/stock-trader/horizontalpodautoscalers/portfolio
  uid: 29127ded-3ad7-11e9-8759-005056a0cb5a
spec:
  maxReplicas: 10
  minReplicas: 2
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: portfolio
  targetCPUUtilizationPercentage: 50
status:
  currentCPUUtilizationPercentage: 5
  currentReplicas: 2
```

Step 5: Add Load and Measure Usage/Scaling

For our Stock Trader microservices we ran a tool we created called “Looper” (view here). This ran our pods (except the UI) with repeated load.

Then, to exercise the UI, we ran a test container:

```
kubectl run -i --tty load-generator --image=busybox /bin/sh
```

Then run:

```
while true; do wget -q -O- https://9.42.19.205/trader; done
```

With this load across all microservices, we can now look at the results:

Graphical view:

Viewing realtime HPA details here shows that utilization is too high so the HPA is actively provisioning additional pods:

```
kubectl get hpa

NAME          REFERENCE                 TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
looper        Deployment/looper         176%/50%   2         10        6          3d
portfolio     Deployment/portfolio      288%/50%   2         10        8          3d
stock-quote   Deployment/stock-quote    261%/50%   2         40        4          4d
trader        Deployment/trader         100%/50%   2         10        7          3d
```

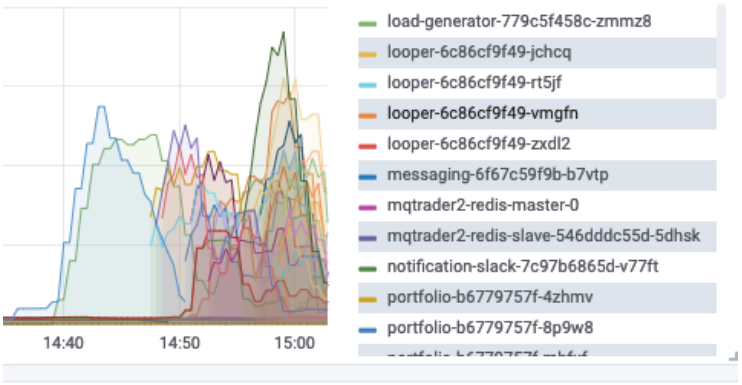
Later on, we see they are more balanced:

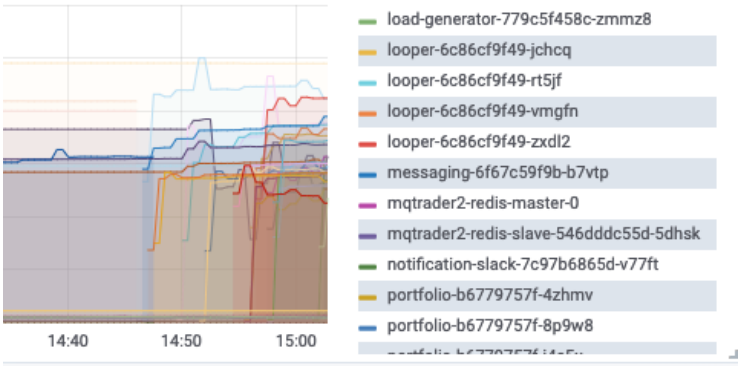
```
kubectl get hpa
NAME          REFERENCE                 TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
looper        Deployment/looper         68%/50%    2         10        4          3d
portfolio     Deployment/portfolio      110%/50%   2         10        8          3d
stock-quote   Deployment/stock-quote    75%/50%    2         40        13         4d
```

```
trader        Deployment/trader        46%/50%    2         10        10         3d
```

Finally, after we stop the load across all pods, the cool-down period of 5-7 minutes results in scaling back to the minimum:

```
kubectl get hpa
NAME          REFERENCE                 TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
looper        Deployment/looper         12%/50%   2         10        2          3d
portfolio     Deployment/portfolio      9%/50%    2         10        2          3d
stock-quote   Deployment/stock-quote    7%/50%    2         10        2          4d
trader        Deployment/trader         8%/50%    2         10        2          3d
```





Notice in this Grafana dashboard it shows not only the Stock Trader pods but the looper and load generator pods.

Step 6: Validate Stability

Once your microservices-based app is autoscaling properly, now is the time to run 'creative tests', chaos monkey-style tests, to see how it responds to outages and additional load. Here are some examples to try based on Stock Trader architecture:

- *Add multiple load sources to simulate "Peak usage"*
  In our case, we needed "Looper" and our UI load generator to generate normal steady-state usage. For additional "peak usage" load, set up a second "looper" running from a different source so that you can hit your application with bursts of activity in addition to the regular load.

- *Kill pods*
  Go into the UI or kubectl CLI and just start deleting pods. This will test a real-world scenario where a pod dies and needs to be rescheduled. You could also go into the deployment object and scale up/down manually to see how the HPA reacts.

- *Taint worker nodes*
  To simulate pulling the plug on a VM, add a taint to a worker node that prohibits scheduling and execution. This will remove all pods from the VM and schedule them on other worker nodes. In some cases, you won't have enough resource to auto-scale properly so you can see how your app works in distress.

- *Remove middleware (Redis)*
  To simulate a problem with middleware you depend on (Redis, for example), you can scale down to zero the redis deployment. This will then let you validate your code can handle the outage.

- *Define new data source (test liveness)*
  Test the liveness check in your pods. If you recall, our "Portfolio" pod said that if it gets three concurrent errors in calling the data source it is no longer "live". To test this, create a second data source, edit the secret for "portfolio", and scale down the existing data source (no need to delete … you will be testing this again). You will see the existing portfolio pods get removed after 3 failed attempts at the now-dead data source and when the new pods start, it will pick up the secret with the new (and running) data source and soon enough your app will be alive and kicking.

Step 7: Consider Istio Optimization

For those using Istio for your microservices mesh, the main consideration is the resource requests and limits. During your initial testing with Istio, increase the resource limits for your microservices by 100m CPU and 128Mi Memory so that when the Istio sidecar container is added, your pod will still auto-scale appropriately.

Auto-Scaling using Custom Metrics

If you have custom application-level metrics you want to use in your auto-scaling you can do that by following this documentation. In summary, you will need to install a Prometheus adapter into IBM Cloud Private and then follow the instructions to configure your yaml files.

Concept – Creating Microservices with a RESTful API

Within any microservices based architecture, there will be APIs. There will be APIs that are exposed outside of the solution. We will call these exposed APIs, **managed APIs**. There will also be APIs used to call between services. For now, we will just call those **APIs**. REST and JSON provide the language independent building blocks for building out RESTful APIs.

Creating a REST API

Creating an API and the implementation for a new microservice can be done in a couple of ways:

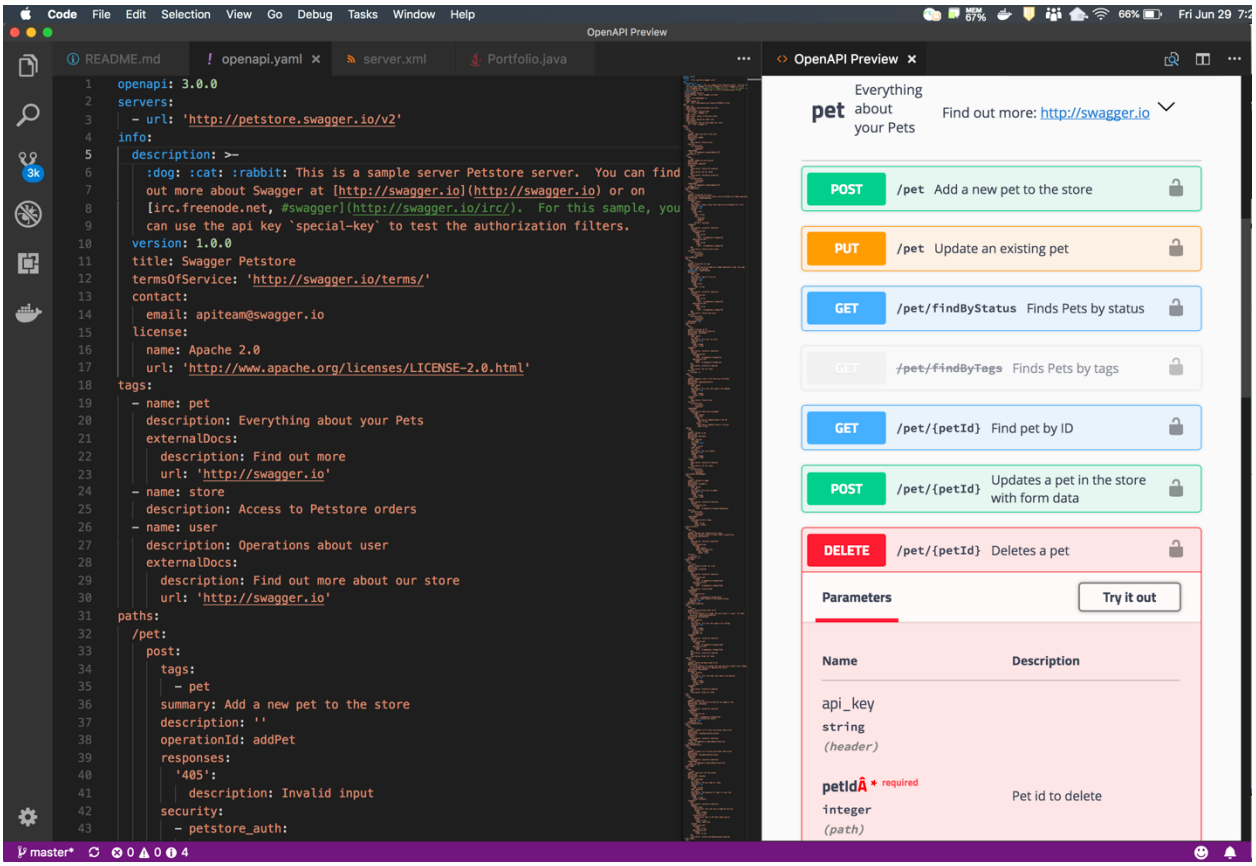
1. **Start with an OpenAPI Definition** – In this approach, a file contains the OpenAPI definition in a language independent format (usually YAML). Code generators that can create a skeleton implementation from that definition are available for most programming languages, including Java. You then work from that skeleton to build your service implementation.
2. **Start with code** – If you prefer to keep your API definition close to the code, you're in luck if you are writing in Java. OpenAPI libraries have excellent support for annotation-based REST frameworks, making generation of OpenAPI definitions from code a trivial task. There are additional OpenAPI annotations that can be used to add additional context to generated API definitions. The functionality of this path varies significantly for other languages.

In either approach there are a number of tools that make these approaches more streamlined.

Creating an API from an OpenAPI Definition

As mentioned above, creating and implementing a new API by starting with Open API involves specifying a REST interface in a language independent format specified by the Open API community. This format is an evolution of Swagger.

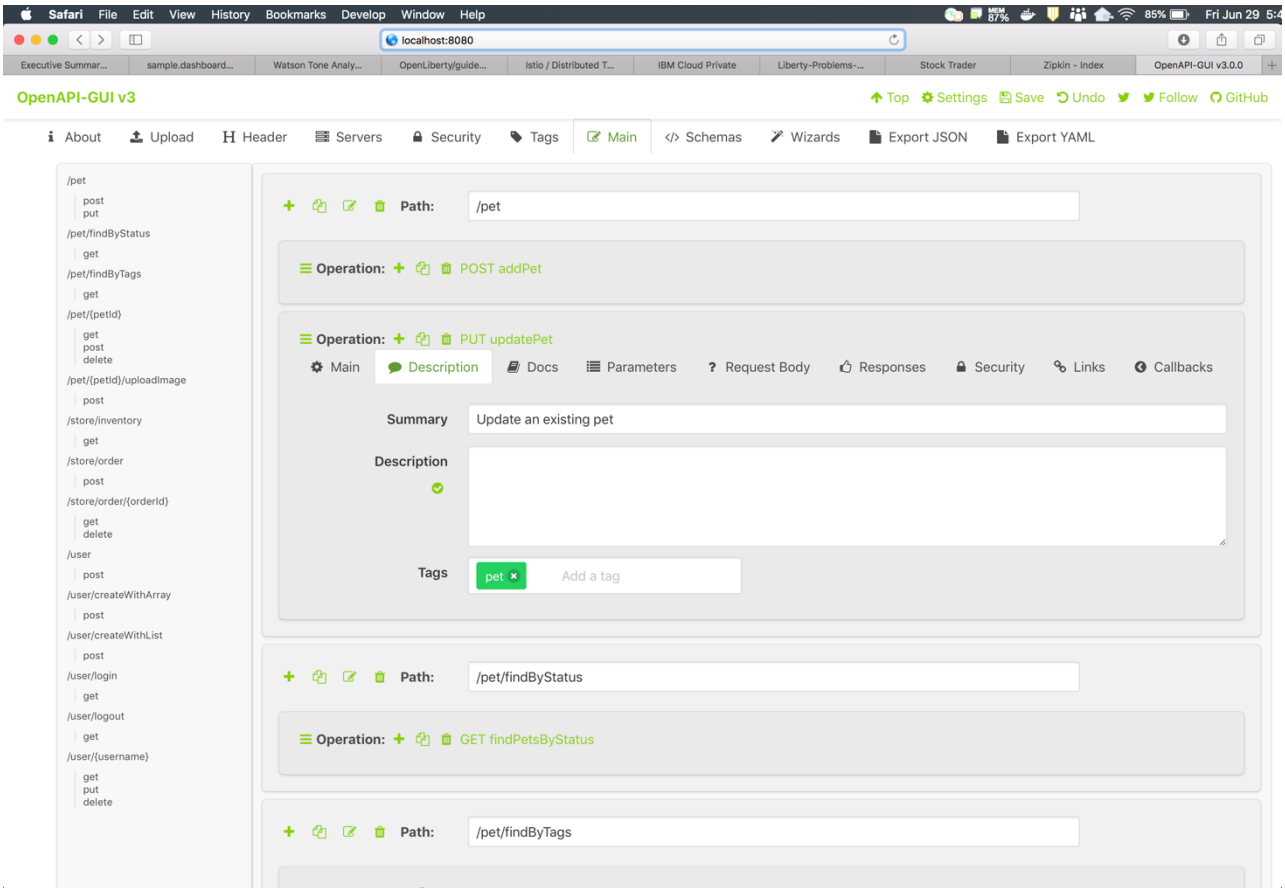
You can author your OpenAPI `.yaml` file in whatever tool you choose. If you want to use a plain text editor, you can do that, though it would be error prone. Some editors have basic support for yaml. Some may have additional extensions to support OpenAPI definitions. For example, a VS Code extension can show a HTML-representation of the OpenAPI definition in a preview pane:



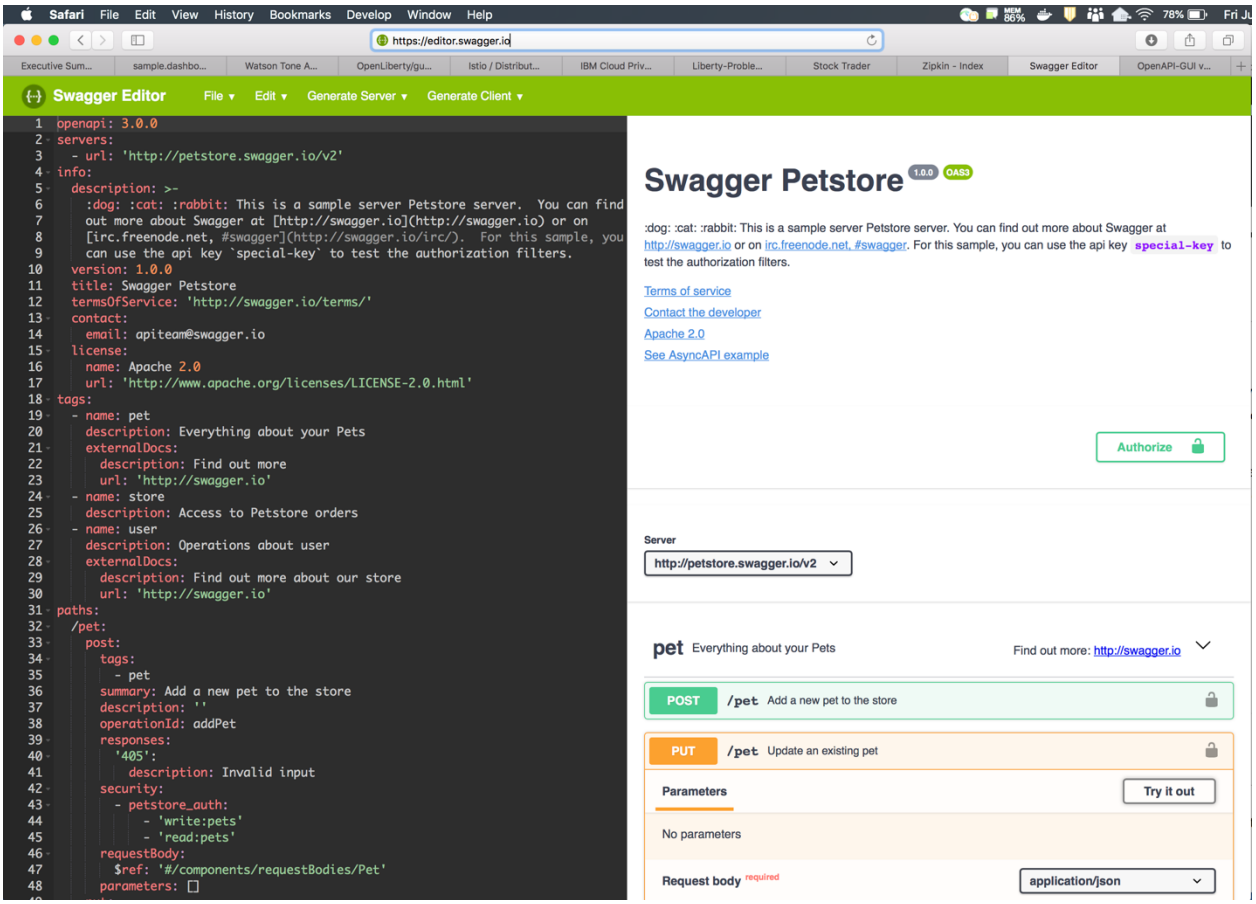
There are also a variety of browser-based, live-parsing editors you can use. For example, there's the open source OpenAPI-GUI project, at https://github.com/Mermade/openapi-gui, which has a hosted version at https://mermade.github.io/openapi-gui/, or you can host it locally from DockerHub via the following command:

```
docker run -p 8080:3000 mermade/openapi-gui
```

Then you would just access it from that `8080` port specified in the command, by pointing your browser at http://localhost:8080. By default, it comes up showing the PetStore sample's APIs, but you can of course clear that and create your own. It supports both v2 and v3 of the OpenAPI specification and can migrate a v2 doc to v3 for you.



There's also the Swagger editor from SmartBear (the original owners of Swagger, before it was donated to the Open API Initiative), which supports both v2 and v3 of OpenAPI, hosted at https://editor.swagger.io. It can also be downloaded to run locally, from https://swagger.io/tools/swagger-editor/download/. It too shows the PetStore sample by default. Besides the free version, there's also a more fully featured version called SwaggerHub (https://swagger.io/tools/swaggerhub/) that you can pay for.



IBM also has its API Connect product (https://console.bluemix.net/catalog/services/api-connect). See the API Connect later in this book below for more details.

Generating the API implementation

Once you have API defined in an OpenAPI yaml file, you can use the open-source OpenAPI generator to create a skeleton application for that API. You can then fill in the logic for each method.

As you can see at https://github.com/OpenAPITools/openapi-generator, you can run the following command (on a Mac) to install the tool:

```
brew install openapi-generator
```

The output generated by this command will vary based on the specified generator. For example, you would invoke something like the following to use the JAX-RS generator (contributed by IBMer Arthur De Magalhaes) to create a Java project with JAX-RS annotated methods for every operation in the API definition (`petstore.yaml`):

```
openapi-generator generate -g jaxrs-cxf-cdi -i ./petstore.yaml -o
petstore --api-package=com.ibm.petstore
```

Recommended practices for RESTful APIs

Note: These apply to all REST APIs, whether they are used only for microservice to microservice communications or are exposed and managed.

REST APIs should use standard HTTP verbs for Create, Retrieve, Update, and Delete (CRUD) operations, with special attention paid to whether the operation is idempotent (safe to repeat multiple times).

4. **POST** operations may be used to create or update resources. POST operations are not idempotent. For example, if a POST request is used to create resources, and it is invoked multiple times, a new, unique resource would be created as a result of each invocation.
5. **GET** operations must be both idempotent and nullipotent. In simpler terms, they must not cause side effects and should only be used to retrieve information. GET requests with query parameters should not be used to change or update information (use POST, PUT, or PATCH instead).
6. **PUT** operations can be used to update resources. PUT operations usually include a complete copy of the resource to be updated, making the operation idempotent.
7. **PATCH** operations allow partial update of resources. They might or might not be idempotent depending on how the delta is specified and then applied to the resource. For example, if a PATCH operation indicates that a value should be changed from A to B, it becomes idempotent. It has no effect if it is invoked multiple times and the value is already B. Note that support for PATCH operations is still inconsistent across languages and frameworks.
8. **DELETE** operations are idempotent, as a resource can only be deleted once. However, the return code varies, as the first operation succeeds (200 or 204), while subsequent invocations do not find the resource (401).

Create machine-friendly, descriptive results

Given that APIs are invoked by software instead of by humans, care should be taken to communicate information to the caller in the most effective and efficient way possible.

The HTTP status code should be relevant and useful. Use a 200 (OK) when everything is fine and there is data to return, but when there is no response data, use a 204 (NO CONTENT) instead. A 201 (CREATED) should be used for POST requests that result in the creation of a resource, whether there is a response body or not. Use a 409 (CONFLICT) when concurrent changes conflict, or a 400 (BAD REQUEST) when parameters are malformed. The set of HTTP status codes is robust and should be used to express the result of an operation as specifically as possible. For more information, see the RFC: https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

You should also consider what data to return in your responses to make communication efficient. For example, when a resource is created with a POST request, the response should include the location of the newly created resource in a Location header. The created resource is often included in the response body as well, to eliminate the extra GET request for the created resource. The same applies for PUT and PATCH requests.

Additional considerations for internal APIs

Internal APIs, those used between microservices, but that aren't managed or intended to be exposed to callers from outside of your Kubernetes environment, generally should not be defined as having an Ingress or a Node Port.

For images containing `curl` and `bash`, you can `exec` into a pod to run a `curl` command to try out the API that the microservice wants to call, e.g.

```
kubectl exec <pod> -it /bin/bash
```

Not all images contain either `curl` or `bash`, you may need to install it (using `apt` for ubuntu or `apk` for alpine). Or perhaps use a test/debug image that depends on the services you want to test, so you can invoke cluster-internal APIs via a pod containing that image.

Programming Model for Java – Creating Microservices with REST APIs

MicroProfile is all about helping you implement and expose REST APIs in Java. It provides Open API support to help define and expose APIs, and endorses JAX-RS for the implementation. It also provides support for generating REST clients to make it simpler to consume REST APIs (which we'll discuss in the Consuming APIs section).

JAX-RS

JSR 339 introduced JAX-RS 2.0 as the industry standard for how to implement REST services in Java, and MicroProfile has embraced this (MicroProfile 2.0 moves on up to JAX-RS 2.1). In JAX-RS, you code up a Java class like usual, and then put annotations on the class, and on the methods that you want exposed as a REST service.

At the class level, the easiest approach is to just have your class extend `javax.ws.rs.core.Application`, and then add the `@ApplicationPath` annotation, which works a lot like a context root in standard web apps. You package up your JAX-RS class(es) in a .war file and deploy that like you would any other .war file to your app server.

At the method level, you use annotations to describe the verb, the path, the MIME type of the data you'll receive or produce, and finally the path or query params. For example, here's the `createPortfolio` method signature, with its annotations:

```
@POST
@Path("/{owner}")
@Produces("application/json")
@RolesAllowed({"StockTrader"})
public JsonObject createPortfolio(@PathParam("owner") String owner)
throws SQLException {
```

Each annotation, at both the method level and the parameter level, is a JAX-RS annotation.

- `@POST` identifies the HTTP method.
- `@Path` to associates this method with a URL pattern. In this example, if someone makes a request to `POST host:port/portfolio/John`, JAX-RS will use this method because `/portfolio` is the application's context root,and `/John` matches because `{owner}` is a path parameter. In this case, when the `createPortfolio` method is invoked, `John` will be passed as the value for owner parameter due to `@PathParam`.
- The `@Produces` annotation indicates that JSON will be returned from this method, which is typical for REST APIs.
- `@RolesAllowed` is a bridge into the Java EE security model, where we are requiring the caller to be a member of the `StockTrader` role/group to call this, which is something we will revisit when discussing application security.

If all goes well, a JsonObject is retured with a HTTP 200 return code. If an exception occurred (like the SQLException defined on the method signature, or any RuntimeExceptions, like a NullPointerException), then WebSphere Liberty will catch the exception and return a HTTP 500 response to the caller. It will also emit logs containing stack trances for the failure. Define a JAX-RS `ExceptionMapper` if you need finer control over exception handling .

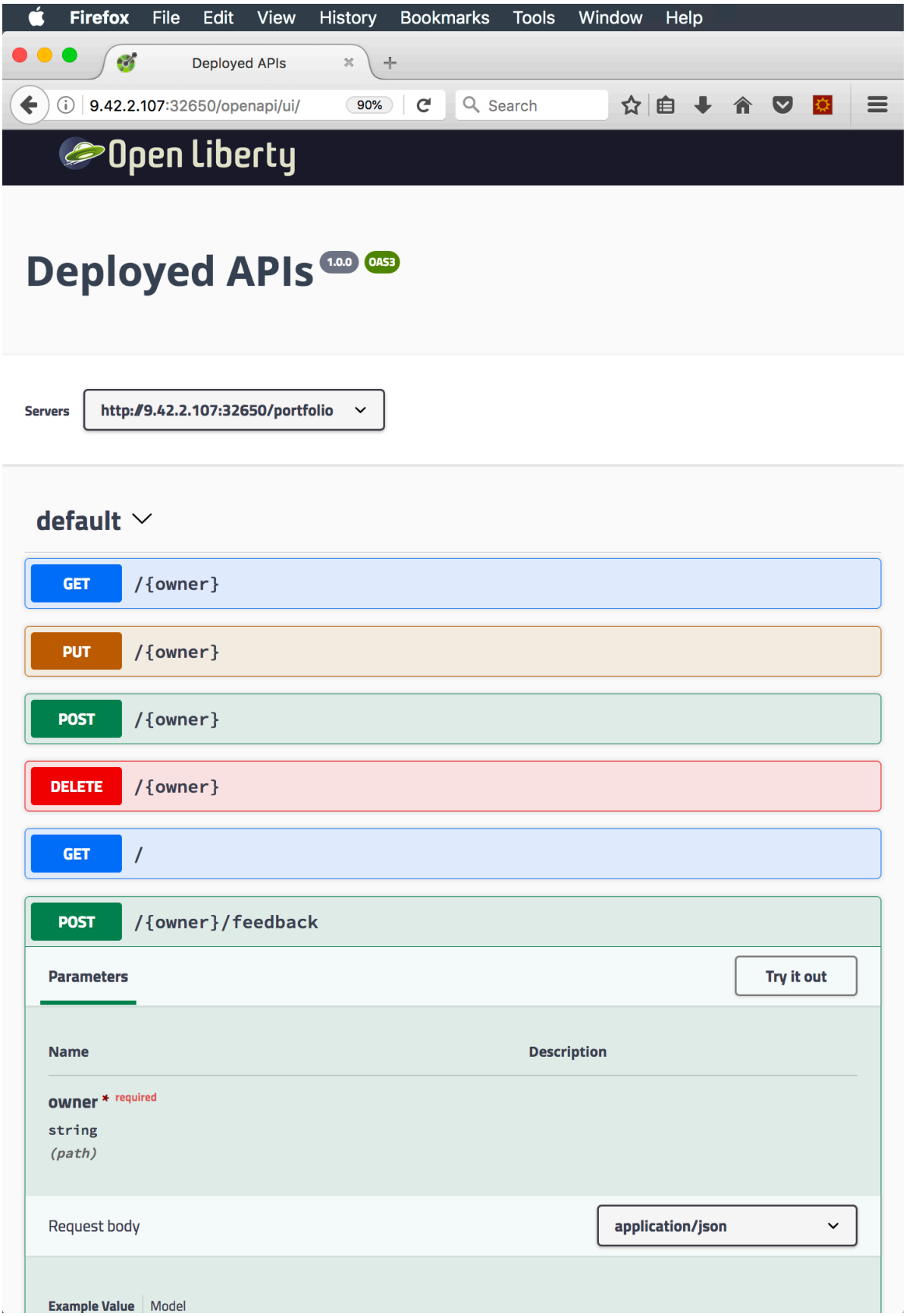
MicroProfile OpenAPI support

MicroProfile 1.3 defines a unified Java API for OpenAPI v3 that allows runtimes like WebSphere Liberty to generate an OpenAPI file (and the HTML produced from that) by introspecting your war file for any JAX-RS annotated classes. In WebSphere Liberty, enable the `mpOpenAPI-1.0` feature (or the `microProfile-1.3` convenience feature) in your server.xml file. The generated raw OpenAPI yaml will be available at `/openapi`, with a UI available for browsing at `/openapi/ui`.

Let's take a look at what this feature generates for our Portfolio microservice. Note that I truncated the results to just show the first of the APIs (it would go on for pages otherwise).

```
Johns-MacBook-Pro-8:StockTrader jalcorn$ curl http://9.42.2.107:32650/openapi
openapi: 3.0.0
info:
  title: Deployed APIs
  version: 1.0.0
servers:
- url: http://9.42.2.107:32650/portfolio
paths:
  /{owner}:
    get:
      operationId: getPortfolio
      parameters:
      - name: owner
        in: path
        required: true
        schema:
          type: string
      responses:
        default:
          description: default response
          content:
            application/json:
              schema:
                type: object
                additionalProperties:
                  $ref: '#/components/schemas/JsonValue'
```

Now let's view the HTML UI produced from that. I expanded the last one - which happens to be the one that uses the Watson Tone Analyzer - to show what's in such a section, such as the "Try it out" button):


Firefox File Edit View History Bookmarks Tools Window Help
Deployed APIs
9.42.2.107:32650/openapi/ui/
90%
Search
Open Liberty
Deployed APIs
1.0.0
OAS3
Servers
http://9.42.2.107:32650/portfolio
default
GET /{owner}
PUT /{owner}
POST /{owner}
DELETE /{owner}
GET /
POST /{owner}/feedback
Parameters
Try it out
Name
Description
owner * required
string
(path)
Request body
application/json
Example Value
Model

Note that all of the above was generated without us doing anything in our code. However, we could annotate our code, using `@APIResponse` to describe the structure of the JSON object being returned, for example, and then we'd get an even more complete OpenAPI v3 document produced (rather than it just showing "default response" of JsonValue, like you see above).

<<Need snippet of mpOpenAPI annotations in the code.>>

Alternatively, place an `openapi.yaml` file in the META-INF (not the WEB-INF) directory of your war file, and the MicroProfile OpenAPI support will use that directly.

It should be noted that the `/openapi` (and`/openapi/ui`) URL is specific to the server running it (which is in a particular pod).

FUTURE: There is nothing today that provides a federated view of all of the APIs across a total application, like our Stock Trader app. Outside of manual effort to put the separate OpenAPI documents together, we have to view the APIs for each microservice separately.

Generating a JAX-RS implementation

As mentioned, you can use the open-source OpenAPI generator to provide a skeleton application implementation from an OpenAPI definition. Specifically, use the following parameters to create a Java project with JAX-RS annotated methods for every operation defined in the API definition (`petstore.yaml`):

```
openapi-generator generate -g jaxrs-cxf-cdi -i ./petstore.yaml -
o petstore --api-package=com.ibm.petstore
```

That says to use the `jaxrs-cxf-cdi` generator against a file named `petstore.yaml`, with an output directory of `petstore`, and a package name for the generated Java of `com.ibm.petstore`. It produces output like the following:

```
Johns-MacBook-Pro-8:OpenAPI jalcorn$ find petstore -type f -
print
petstore/.openapi-generator-ignore
petstore/pom.xml
petstore/.openapi-generator/VERSION
petstore/src/gen/java/org/openapitools/model/Order.java
petstore/src/gen/java/org/openapitools/model/ModelApiResponse.ja
va
petstore/src/gen/java/org/openapitools/model/Tag.java
petstore/src/gen/java/org/openapitools/model/Pet.java
petstore/src/gen/java/org/openapitools/model/Category.java
petstore/src/gen/java/org/openapitools/model/User.java
petstore/src/gen/java/com/ibm/petstore/UserApiService.java
petstore/src/gen/java/com/ibm/petstore/PetApi.java
petstore/src/gen/java/com/ibm/petstore/UserApi.java
```

```
petstore/src/gen/java/com/ibm/petstore/StoreApi.java
petstore/src/gen/java/com/ibm/petstore/PetApiService.java
petstore/src/gen/java/com/ibm/petstore/StoreApiService.java
petstore/src/main/webapp/WEB-INF/beans.xml
petstore/src/main/java/com/ibm/RestApplication.java
petstore/src/main/java/com/ibm/petstore/impl/PetApiServiceImpl.j
ava
petstore/src/main/java/com/ibm/petstore/impl/UserApiServiceImpl.
java
petstore/src/main/java/com/ibm/petstore/impl/StoreApiServiceImpl
.java
Johns-MacBook-Pro-8:OpenAPI jalcorn$
```

This is somewhat similar to what you get when you use IBM Developer Tools (`bx dev create`), except that this only builds application artifacts (the Java source and the Maven pom.xml); it does NOT have Liberty configuration (like server.xml) or deployment artifacts for Docker (like a Dockerfile) or Kubernetes (like a deploy.yaml or a helm chart).

From here on out, develop the implementation of the API using the programming model defined in this guide. Use the tools and pipelines that are selected for the project.

Communications Between Microservices

Up to this point, the programming model definition has focused on the basics of getting a single microservice created. That is a necessary and valuable start. However, microservices are indeed living in an environment that consists of many microservices. In order to make a proper solution from a collection of microservices, they will need to communicate with each other. For various reasons, there are multiple ways to have microservices interact with each other.

Conceptually, there are synchronous calls between microservices and asynchronous calls. Synchronous calls will block a thread while waiting for a response. Asynchronous calls will not block a thread. Asynchronous calls are very valuable when dealing with challenges of scale and erratic request volumes. Synchronous calls are generally easier to program. The reality is that most systems will leverage a combination of synchronous and asynchronous interactions. Asynchronous interactions can be very event based where it is a model of publishing events that zero or more downstream services might be interested in knowing about. Asynchronous calls can also be more point to point oriented, with various techniques used to capture the response.

More technically, synchronous calls in this programming model are recommended to be done using REST, which can be used in almost any programming language. Alternative approaches such as gRPC, RMI-IIOP or others can certainly be leveraged as well for synchronous interactions. Each has specific payload implications and programming models onto themselves. We won't repeat all the literature on those. We will focus on REST and describe language and framework specific approaches.

Asynchronous calls can also be done in a number of ways also. Some of these are enumerated below:

1. Kafka
2. REST Asynchronous calls
3. MQ (which can mean JMS)

Each of these approaches will have different clients or APIs for a given language and framework environment. For example, REST Async can be done in Java using an option on the mpRESTClient. MQ, obviously has language bindings such as the *Java Message Service* 2.0 (JSR 363).

For now, the book will focus on these three with emphasis on Kafka in the main microservice to microservice asynchronous communications section. REST Asynchronous calls and MQ, as well as more advanced asynchronous patterns will be covered later in the book.

Note also that our *Stock Trader* sample referenced throughout this primarily book demonstrates traditional synchronous REST calls, but it does also demonstrate each of the three types of asynchronous calls. It uses Kafka to talk to Trade History, and MQ to talk to Notification. And the Slack version of Notification uses asynchronous REST to call our action sequence in IBM Cloud Functions (aka OpenWhisk). OpenWhisk is inherently asynchronous, just accepting the input data and returning immediately, processing the actual action sequence on another thread. (that said, you can pass the `blocking=true` query param to simulate a synchronous REST call, causing the calling thread to wait for the action sequence to complete – including the call to the Slack web hook).

Java – Microservice to Microservice - Synchronous

When implementing a call to another microservice from a Java based microservice, the best and simplest approach is to use the mpRestClient.

mpRestClient

One of the best features that MicroProfile has brought to Java is the *mpRestClient*. Whereas the JAX-RS 2.0 client is a very awkward interface to code to, the approach introduced by MicroProfile simply reuses the JAX-RS annotations from the implementation, but as a Java interface, and it is easily injectable into the caller via CDI.

This is mostly useful for "green field" scenarios, where you are writing a new client to a microservice. If you are doing a lift-and-shift of an existing application to your private or public cloud, you could just leave its cross-component calls in place initially, as they should still work. But say you did a lift-and-shift and then wanted to put a more modern UI (perhaps with responsive/mobile support) face on that app; in that case, your new client (if written in Java) could be written to use the mpRestClient to access the back-end services.

Of course, if a real SDK is available for the service you want to call, you can just use that. Such an SDK might take care of additional stuff like security for you, like in the Twitter4J SDK for Twitter that we showed above did with the O-Auth stuff.

To see the mpRestClient in action, recall the `createPortfolio` method shown earlier with its JAX-RS annotations. You'd simply put the same thing in an interface, as shown below. This is only one of several API calls made by the Portfolio microservice:

```
import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;

@Dependent
@RegisterRestClient
@Path("/")
public interface PortfolioClient {
   @POST
   @Path("/{owner}")
   @Produces("application/json")
   public JsonObject createPortfolio(@PathParam("owner") String owner) throws
SQLException;
```

Note I added two annotations (`@Dependent` and `@RegisterRestClient`) at the interface level, needed so you can use CDI to inject this into your microservice. Here's how you do that:

```
@Inject @RestClient private PortfolioClient portfolioClient;
```

Now, as you can see, I have a type-safe interface I can invoke like any other interface in Java, which will immediately tell me at compile time if I have the wrong number of parameters, or the wrong data type for them. So then to create a portfolio, I'd just call:

```
String owner = "John";
JsonObject portfolio = portfolioClient.createPortfolio(owner);
String loyalty = portfolio.get("loyalty");
```

Alternatively, you could use JSON-B to define a strongly-typed `Portfolio` POJO class, like with methods for `getOwner`, `getStocks`, and `getLoyalty`, and define your method on `PortfolioClient` to return that, rather than the generic JSON-P `JSONObject`. Then you'd just call `portfolio.getLoyalty()` to get that field (with compile time notification of having the name of the field name wrong, for example).

The punchline is, using mpRestClient makes it almost exactly like having an SDK for the REST service you want to call. All you have to do is author that interface; or better yet, use a code generator to generate it directly off your OpenAPI yaml for the REST API you want to call. Note that MicroClimate should consider integrating such a generator, to avoid the annoyance of having to type in that interface by hand, which would also be error-prone.

There's one last thing you have to do, which is to tell it the URL for the REST service you are calling. This is done by adding a line like the following to your `jvm.options` file (which your `Dockerfile` will need to copy into your Docker image, of course, if your caller microservice doesn't already have one):

```
-Dcom.ibm.hybrid.cloud.sample.portfolio.PortfolioClient/mp-
rest/url=http://portfolio-service:9080/portfolio
```

That is, you need to set a JVM property called `<fully qualified interface name>/mp-rest/url`, and you need to set it to the URL that the calling microservice uses to contact the callee; note in this case I used the Kubernetes DNS name for my portfolio Kubernetes service.

Right now, you also need to manually propagate any HTTP headers or cookies needed by the REST API you are calling. For example, if the microservice you are calling requires a JWT, and that JWT was passed in to the microservice that is calling it, you'll need to put in some explicit code to copy that out of the `HttpServletRequest` and to put it in the call to the service (as a `@HeaderParam` for the Authentication header, in this case). Likewise if you need to forward the OpenTracing correlation headers. I have mpRestClient issue #73 (https://github.com/eclipse/microprofile-rest-client/issues/73) tracking this.

Given the above issue with needing to propagate the JWT, here's the actual `PortfolioClient.java` that I use, where I had to add an `@HeaderParam("Authorization")` to each method for the JWT:

```
package com.ibm.hybrid.cloud.sample.portfolio;

import javax.enterprise.context.Dependent;
import javax.json.JsonArray;
import javax.json.JsonObject;
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.HeaderParam;
```

```
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.Path;
import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;

@Path("/")
@Dependent
@RegisterRestClient
/** mpRestClient "remote" interface for the Portfolio microservice */
public interface PortfolioClient {

@GET
@Produces("application/json")
public JsonArray getPortfolios(@HeaderParam("Authorization") String jwt);

@POST
@Path("/{owner}")
@Produces("application/json")
public JsonObject createPortfolio(@HeaderParam("Authorization") String jwt,
@PathParam("owner") String owner);

@GET
@Path("/{owner}")
@Produces("application/json")
public JsonObject getPortfolio(@HeaderParam("Authorization") String jwt,
@PathParam("owner") String owner);

@PUT
@Path("/{owner}")
@Produces("application/json")
public JsonObject updatePortfolio(@HeaderParam("Authorization") String jwt,
@PathParam("owner") String owner, @QueryParam("symbol") String symbol,
@QueryParam("shares") int shares);

@DELETE
@Path("/{owner}")
@Produces("application/json")
public JsonObject deletePortfolio(@HeaderParam("Authorization") String jwt,
@PathParam("owner") String owner);

@POST
@Path("/{owner}/feedback")
@Consumes("application/json")
@Produces("application/json")
public JsonObject submitFeedback(@HeaderParam("Authorization") String jwt,
@PathParam("owner") String owner, JsonObject input);
}
```

One final note: there are rumors that something very similar will be included in Java 11, or will be added in a new spec version of JAX-RS under Jakarta EE. Even if that happens, you could continue using this MicroProfile feature if you wanted, as it's now a public API that any MicroProfile-compliant app server is required to have and an API which can't have breaking changes going forward.

Java – Microservice to Microservice – Asynchronous

There are a number of scenarios for microservices preferring to use asynchronous interactions over synchronous ones. Synchronous calls are conceptually easier for many people initially, but they block a thread to wait for a response and also expect that both microservices are both available at the same time. There are various techniques for making JAX-RS calls asynchronously, primarily to take advantage of concurrent processing and improve efficiency, but this is just improving the efficiency of processing synchronous calls. These are described at the end of this chapter. A further step, beyond these options is to communicate between the microservices using events, loosening the coupling between the microservices. This is for the most part, the preferred option in almost all cases where high-volume scalable microservices interactions are required. We will explore this first, under the heading of "Event-Driven Interactions".

Event-Driven Interactions

Microservices can communicate using events. The publish/subscribe pattern is used because of the benefits of loose coupling of the microservices. Microservices publish events onto topics for any interested recipients. Each topic contains events of a particular type enabling recipients to select which events they are interested in. A microservice wishing to receive events simply makes a subscription to the appropriate topic without knowledge of where the events are being published. The events are not directed to any particular instance of a microservice; they're just published for any recipients that happen to be interested.

Let's start with some event-streaming concepts.

Event
: a piece of information of interest to the business, often represents a state change

Event Stream
: a continuous, unbounded sequence of events in a predictable order, generally synonymous with a single topic

Stream History
: the historical content of an event stream, facilitating stateless microservices

Comparing message queuing and event streaming

Although traditional message queuing systems such as IBM MQ and RabbitMQ can be used to build event-driven microservices, event streaming platforms such as IBM Event Streams (based on Apache Kafka) and Apache Pulsar are better suited. The key difference between them is the idea of stream history, since in these systems historical events are not immediately discarded when they're received. Event streaming platforms have a stronger focus on storage of large quantities of historical events. This enables significant processing backlogs such as might naturally occur during a maintenance window to be handled more comfortably. The size of the stream history is controlled by setting a retention time, or a capacity limit, and over time more sophisticated storage infrastructure is enabling stream history to be kept indefinitely in a data warehouse.

The topics in an event streaming system are capable of very high throughput but are relatively expensive resources. It's not practical to have a separate topic for every client which rules out efficiently directing events to specific clients.

IBM Event Streams is a new offering based on the open-source Apache Kafka event streaming platform. It's ideal as the messaging backbone for event-driven applications. In comparison with IBM MQ, Event Streams has:

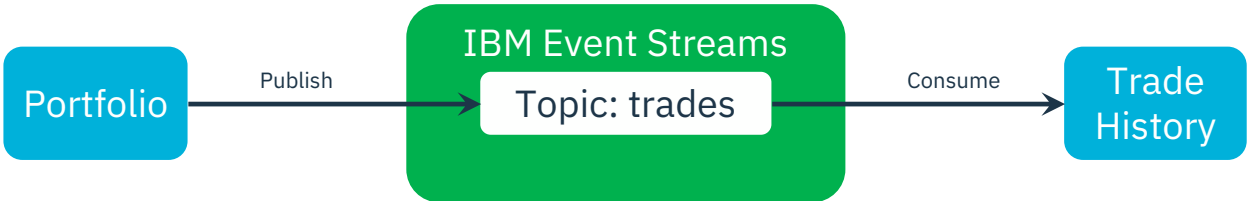
- Publish/subscribe messaging only, with no point-to-point queues
- Stream history so messages are retained even after consumption to allow for late-joining consumers or re-processing of messages
- Local transactions (called exactly-once semantics) aimed primarily at stream processing applications
- No support for XA transactions

Communication patterns

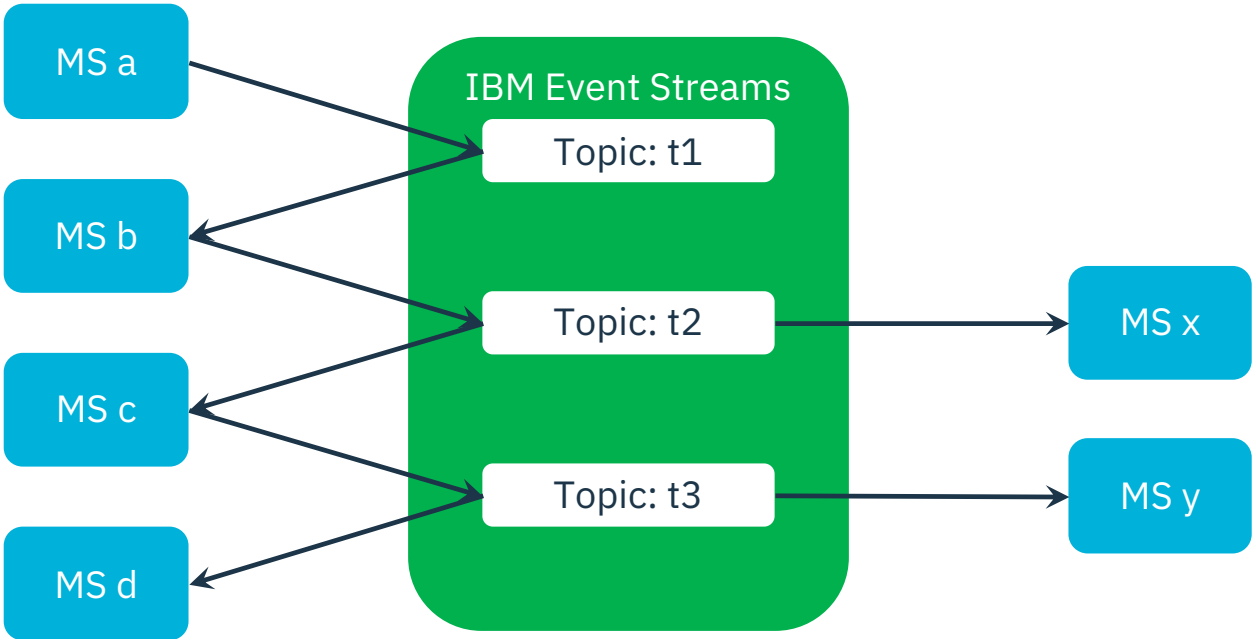
There are two basic patterns for communication using events: one-way and request-completion. These are detailed in the upcoming sections. We will start with "One-way" as this is the foundational building block in Event-Driven architecture.

One-way

Most event communication is one-way meaning that the events are just published onto a topic with no expectation of a response. In the following example from the Stock Trader application, the Portfolio microservice publishes events onto the topic called `trades`. The Trade History microservice subscribes to the topic and consumes the events.



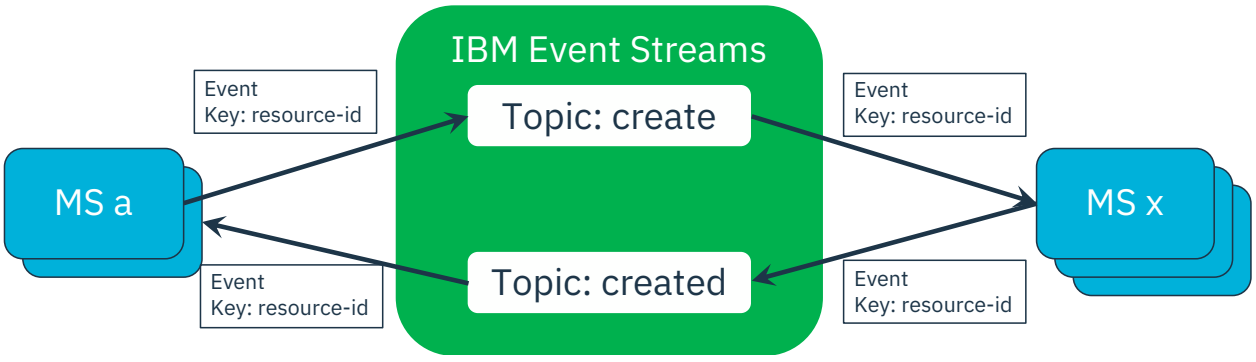
For a complex task, there could be several loosely-coupled stages with events flowing between them, with a topic between each of the stages. Because it's publish/subscribe, each topic might have multiple different kinds of microservices subscribed to it for different purposes.



Request-completion

Sometimes, an event can be used to request a piece of work and another event used to indicate its completion. This is similar to the request-reply pattern familiar from message queuing systems, but the events are delivered using topics. Neither event is targeted as a particular instance of a microservice, so it is more loosely coupled than the request-reply pattern.

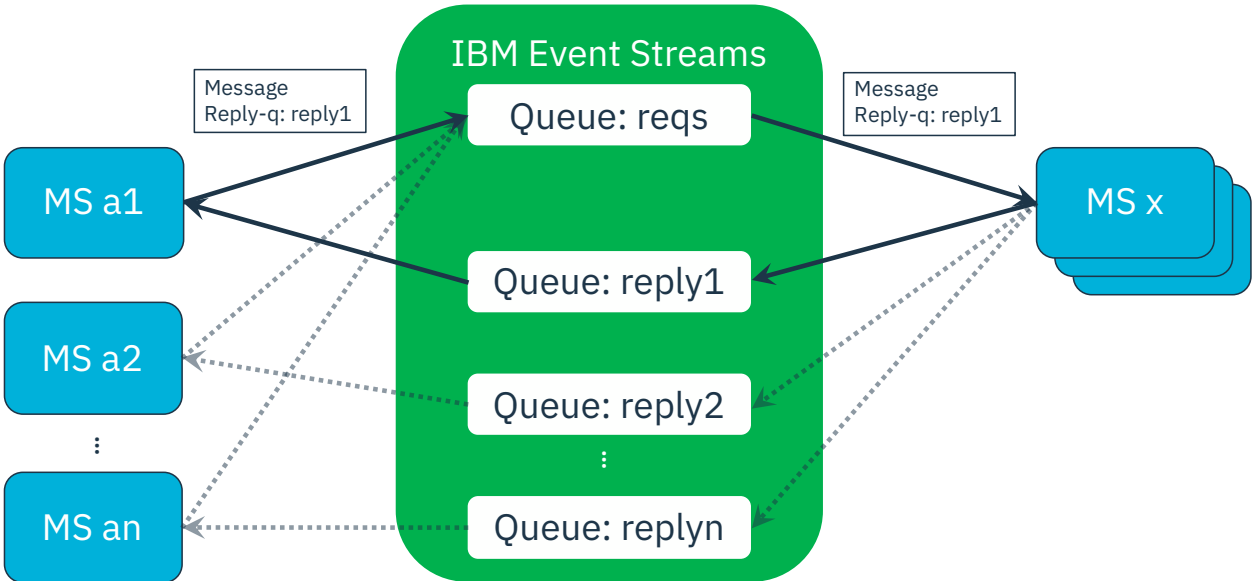
In the example below, a microservice `MSa` consists of several running instances. `MSa` publishes events on topic `create` to request creation of a resource. Each event contains a key `resource-id` for the resource. Another microservice `MSx` subscribes to topic `create` and processes the events. It also has multiple instances. For each event processed, another event is published on topic `created` to indicate that the resource was created successfully. Each event contains the same key `resource-id` that corresponds to the request event. `MSa` is subscribed to topic `created` and receives the completion events. The instance of `MSa` that receives a completion event is not necessarily the same instance that published the request event.



Because this pattern is based on topics, there may be other consumers of the events also for other purposes such as logging. The request event is published to indicate that work is requested. The

completion event is published to indicate that work has been completed, rather than being a targeted response.

Compare this with the request-reply pattern using queues in which each requesting microservice has its own reply queue. It's easy to have multiple instances of `MSx` sharing the processing of the messages on the `req` queue, but the responses are targeted directly to the requesting microservice instance.

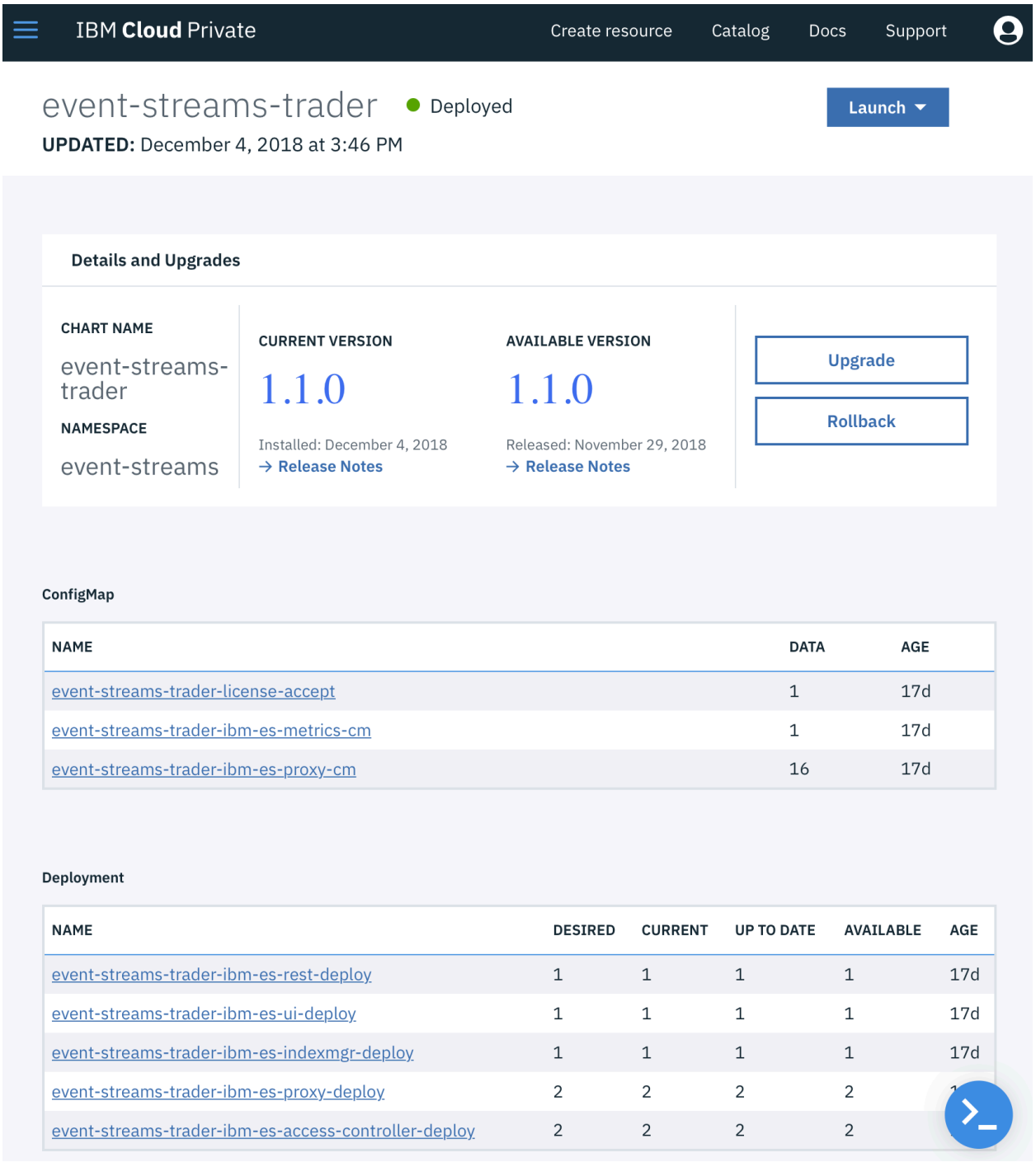


IBM Event Streams

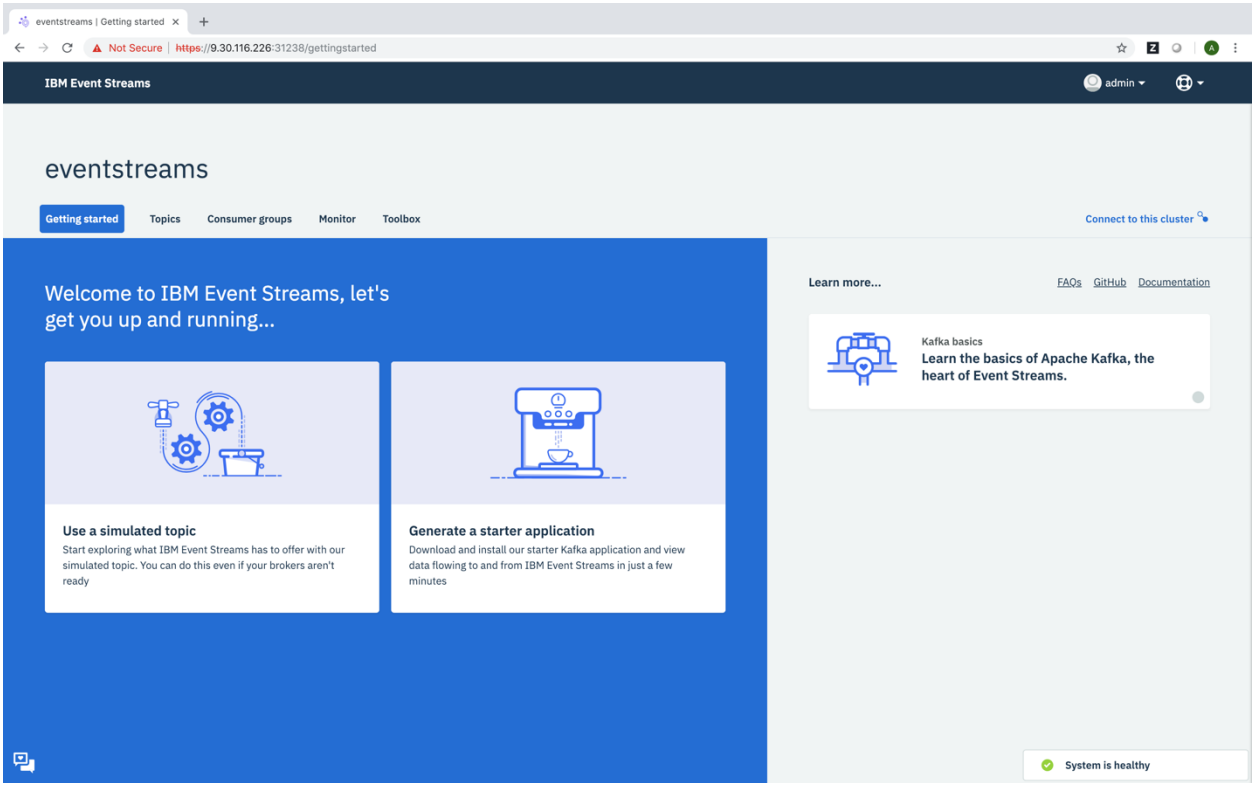
In our Stock Trader sample, we make use of IBM Event Streams to asynchronously invoke a `Trade History` microservice from our `Portfolio` microservice. Whenever a portfolio buys or sells any stock, an event is posted to a topic called `trades`. Then our Trade History microservice will notice that the event appears on the topic and will consume it and store the trade to MongoDB.

We install Event Streams Community Edition in IBM Cloud Private using its Helm chart called `ibm-eventstreams-dev`. First, we need to create a Kubernetes namespace to contain Event Streams. We can do this from the CLI, via `kubectl create namespace <namespace>`, or in the ICP UI. In this example, we call the namespace `event-streams`. The default configuration of Event Streams creates a small Apache Kafka cluster together with supporting microservices for aspects such as networking, administration and security.

Here's the Helm release summary once installed:

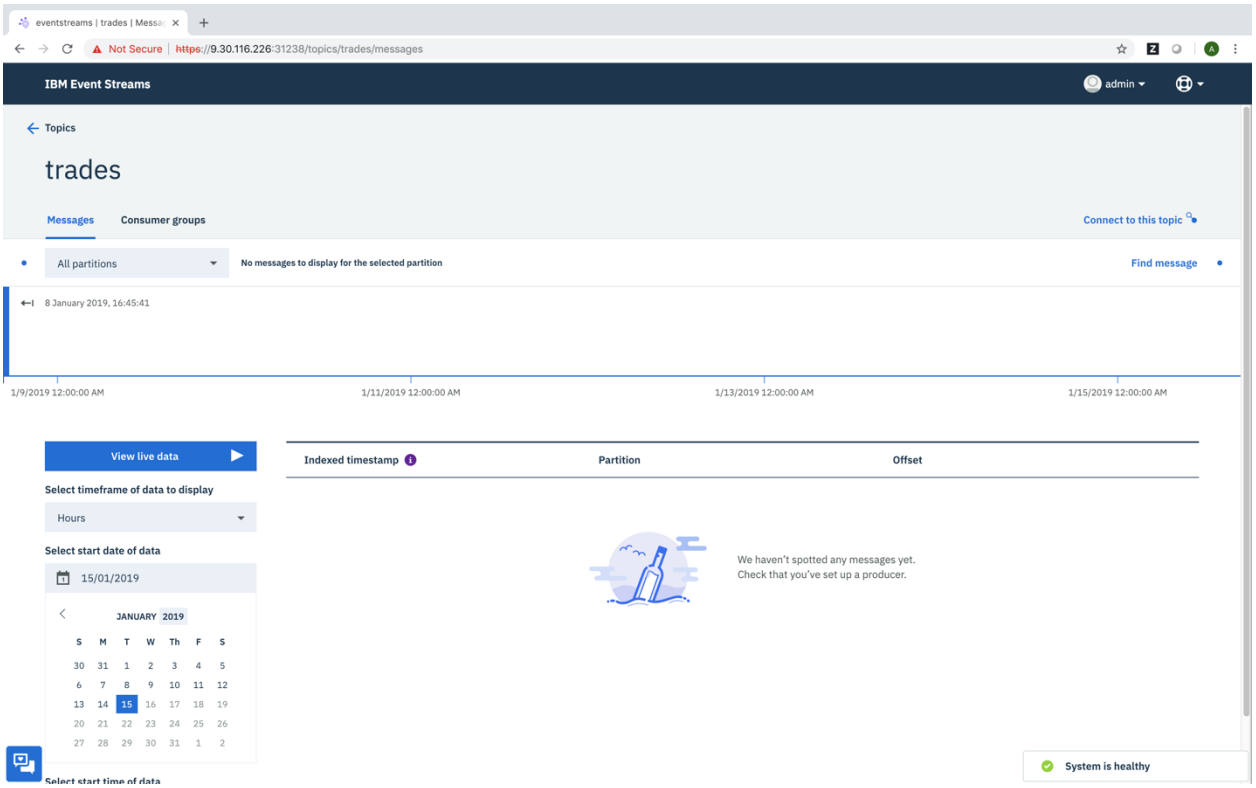


The Event Streams UI can be launched by choosing the `admin-ui-https` entry from the Launch button in the top-right corner.

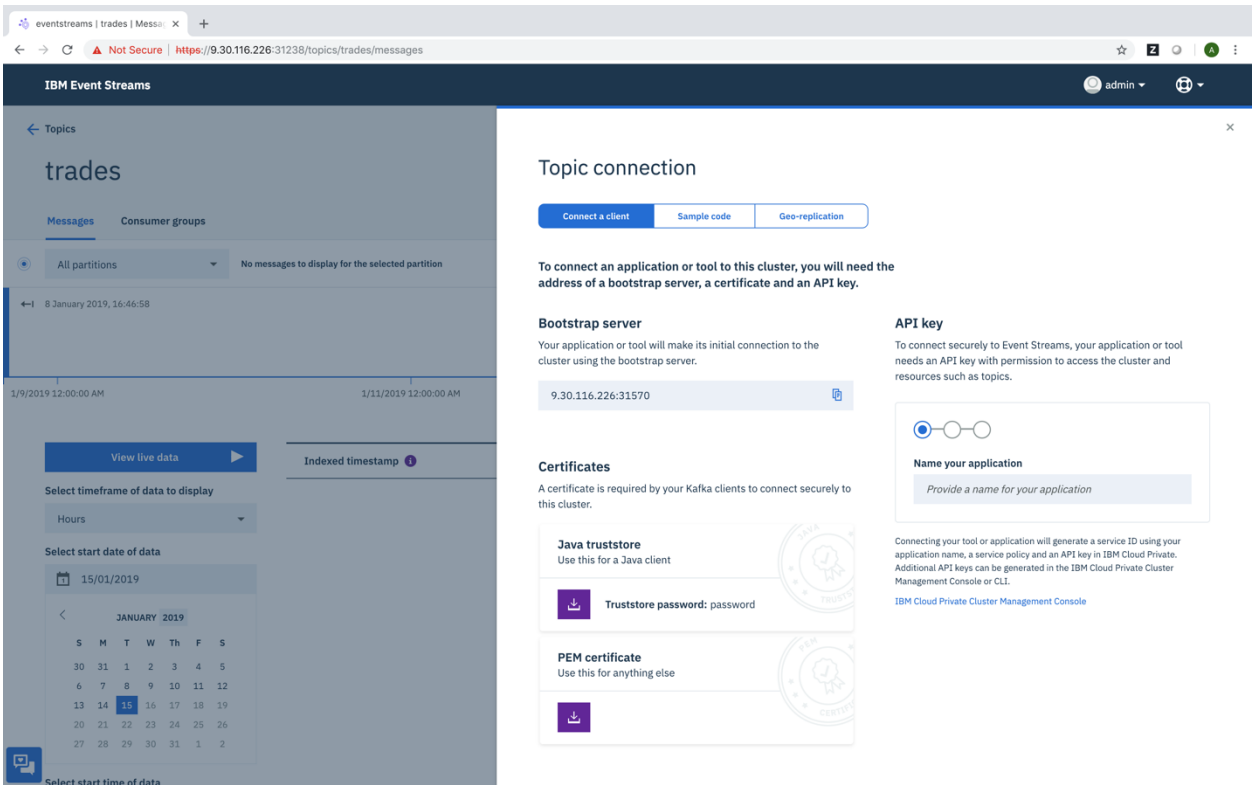


In the Topics tab, create a topic called `trades` and accept the defaults.

Select the new topic in the topics list.



Selecting the `Connect to this topic` link shows the pieces of information needed for connecting applications to use this topic.



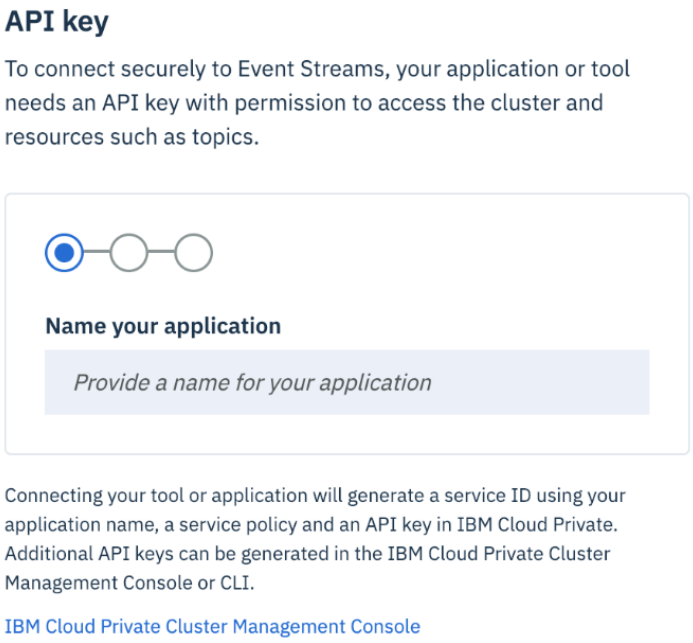
First, the bootstrap server information is used to make the bootstrap connection to Kafka. Copy the information into an environment variable called `KAFKA_BOOTSTRAP_URL`.

Then, a truststore is needed so the Kafka client can make a secure connection to the cluster. This can be downloaded here too – select the download button in the `Java truststore` section. Add the truststore file to the application as `resources/eventstreams.jks` so that it can be used by the Kafka client.

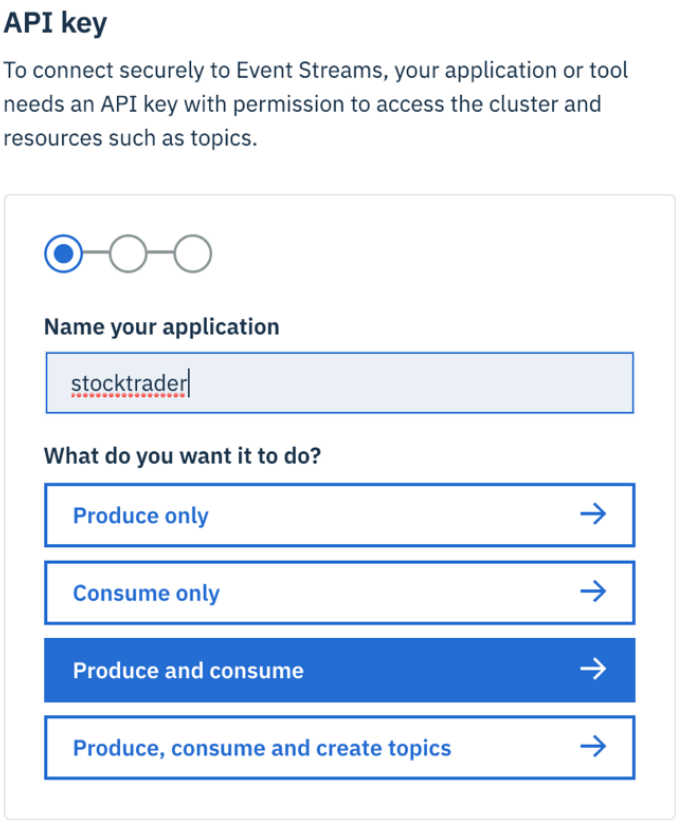
Finally, credentials are needed for the microservices to be able to connect. These can be created right here in the Event Streams UI using the API key tool. It creates:

- A service ID – this is the identity for your application
- A service policy – this grants the service ID permission to use Event Streams
- An API key – this is the credential needed in the application.

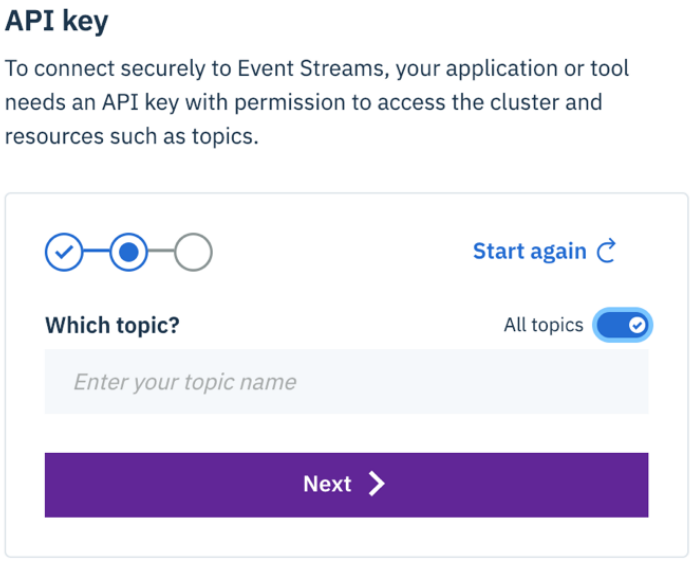
Here's the tool:



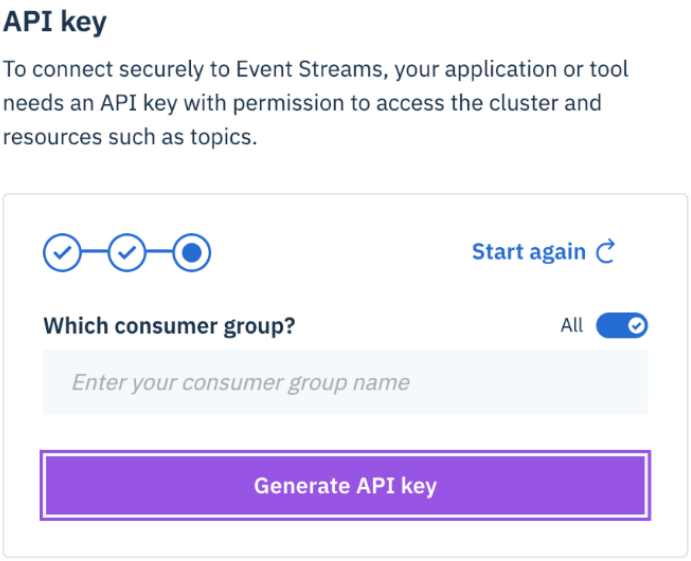
First type `stocktrader` as the name for the application (and service ID). In this tutorial, the service ID has sufficient permissions to be used for all of the stocktrader microservices that connect to Event Streams. In a real deployment, it would be better to create a separate service ID for each microservice.



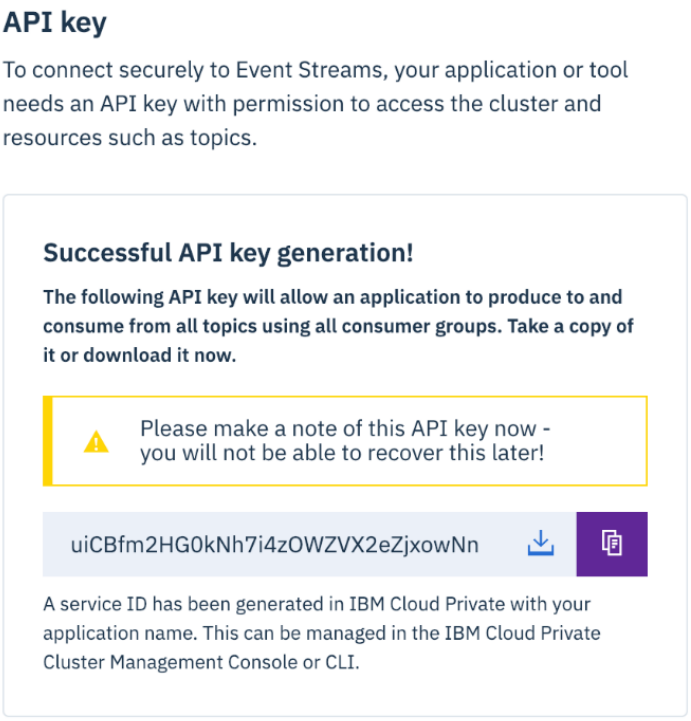
Then, select `Produce and consume` so the service ID is able to read and write to topics.



Then select `All topics`.



Finally, select `Generate API key`.



Copy the API key into an environment variable called `EVENTSTREAMS_API_KEY`. In a proper deployment, this would go into a Kubernetes secret.

Instead of using the API key wizard, you could have used the ICP CLI command `cloudctl es iam-service-id-create`.

Event producer – Portfolio microservice

The auditing code is performed in the portfolio microservice. It uses the Kafka producer API to send events to the topic.

Now let's see how we construct the Kafka producer in the `createProducer()` method:

```
Properties properties = new Properties();
properties.put(CommonClientConfigs.BOOTSTRAP_SERVERS_CONFIG,
brokerList);
properties.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG,
"SASL_SSL");
properties.put(CommonClientConfigs.CONNECTIONS_MAX_IDLE_MS_CONFIG,
10000);
properties.put(ProducerConfig.MAX_BLOCK_MS_CONFIG, 4000);
properties.put(ProducerConfig.RETRIES_CONFIG, 0);
properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
properties.put(SslConfigs.SSL_PROTOCOL_CONFIG, "TLSv1.2");
properties.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG, KEYSTORE);
properties.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG, "password");
properties.put(SaslConfigs.SASL_MECHANISM, "PLAIN");
String saslJaasConfig =
"org.apache.kafka.common.security.plain.PlainLoginModule required
username=\""
    + USERNAME + "\" password=" + API_KEY + ";";
properties.put(SaslConfigs.SASL_JAAS_CONFIG, saslJaasConfig);

kafkaProducer = new KafkaProducer<>(properties);
```

Then the code to send an auditing record looks like this:

```
ProducerRecord<String, String> record = new ProducerRecord<>(topic,
null, message);
RecordMetadata recordMetadata = kafkaProducer.send(record).get();
return recordMetadata;
```

The `send` method returns a `java.util.concurrent.Future` and you can choose whether to wait for its completion (by just calling `get()`, as we do above from `Portfolio`) or not. Of course, you'd only be waiting for the messaging infrastructure to accept the message, not for any consumers to consume and act upon it (which could even happen days later, given how Kafka keeps stream history, allowing one to subscribe to and receive messages sent long ago, as long as you are within the message retention window).

There is also a version of `producer.send` that takes an `org.apache.kafka.clients.producer.Callback` as a second parameter. Of course, you can choose to implement this interface in your class, or create a new class that implements it. That interface has a single method called `onCompletion`, which receives an

`org.apache.kafka.clients.producer.RecordMetadata` as a parameter (the same object that the synchronous `get()` returned above). This metadata includes the offset of the message within the topic (accessed via `metadata.offset()`).

Event consumer – Trade History microservice

The Trade History microservice consumes the events published to the `trades` topic using the Kafka consumer API.

Now let's see how we construct the Kafka consumer in our `createConsumer()` method:

```
Properties properties = new Properties();
properties.put(CommonClientConfigs.BOOTSTRAP_SERVERS_CONFIG,
brokerList);
properties.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG,
"SASL_SSL");
properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
properties.put(ConsumerConfig.GROUP_ID_CONFIG, consumerGroupId);
properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
properties.put(SslConfigs.SSL_PROTOCOL_CONFIG, "TLSv1.2");
properties.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG, KEYSTORE);
properties.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG, "password");
properties.put(SaslConfigs.SASL_MECHANISM, "PLAIN");
String saslJaasConfig =
"org.apache.kafka.common.security.plain.PlainLoginModule required
username=\""
    + USERNAME + "\" password=" + API_KEY + ";";
properties.put(SaslConfigs.SASL_JAAS_CONFIG, saslJaasConfig);

kafkaConsumer = new KafkaConsumer<>(properties);
```

Note how the consumer group ID is being set so that Kafka remembers the current offset for the consumer. Also, if multiple instances of the microservice are started, Kafka will place them all in the same consumer group which enables workload distribution of events across the instances provided that there are sufficient partitions in the topic.

The consumer makes its subscription to the topic like this:

```
consumer.subscribe(Arrays.asList(topic));
```

The events are consumed from the topic like this:

```
ConsumerRecords<String, String> records = POLL_DURATION);
```

Note that the consumer is automatically committing the offsets of the batch just received. In failure situations which prevent the offset information being safely recorded in Kafka, it is possible for the batch of events to be returned to the microservice again.

Reactive Messaging Programming Model Directions

Applications needing to produce/consume events and streams can do so using the Kafka Java client API but this is low-level, evolving, complex, and obviously tied to Kafka. The higher-level and simpler MicroProfile Reactive Messaging API abstracts Kafka as a pluggable Messaging provider and enables the Liberty runtime implementation to take care of logging, monitoring and metrics integration. The MicroProfile Reactive Messaging API is **not yet finalized** but will be used as illustrated below. The signature of message stream methods can have a number of different distinct types, offering differing levels of power and simplicity to application developers. Different shapes are supported depending on whether the method is a publisher, subscriber or processor, for example, a publishing stream supports returning MicroProfile Reactive Streams `PublisherBuilder`, but not `SubscriberBuilder`, the inverse is true for a subscribing stream.

Simple method streams

The simplest shape that an application may use is a simple method. This is a method that accepts an incoming message, and possibly produces an outgoing message:

```
Import org.eclipse.microprofile.reactive.messaging.*;
@Incoming("TopicA")
@Outgoing("TopicB")
public OutgoingMessage processMessage(IncomingMessage msg) {
  return convert(msg);
}
```

In the above example, the stream is both a publishing and subscribing stream, with a 1:1 mapping of incoming to outgoing messages. Asynchronous processing may also be used, by returning a `CompletionStage`:

```
@Incoming("TopicA")
@Outgoing("TopicB")
public CompletionStage<OutgoingMessage> processMessage(IncomingMessage msg) {
  return asyncConvert(msg);
}
```

Reactive streams

For more power, developers may soon use Reactive Streams shapes. Reactive Streams shaped methods accept no parameters, and usually return one of the following:

- `org.eclipse.microprofile.reactive.streams.PublisherBuilder`
- `org.eclipse.microprofile.reactive.streams.SubscriberBuilder`
- `org.eclipse.microprofile.reactive.streams.ProcessorBuilder`

For example, here's a message processor:

```
@Incoming
@Outgoing
public PublisherBuilder<IncomingMessage, OutgoingMessage> processMessages() {
  return ReactiveStreams.<IncomingMessage>builder()
    .map(this::convert);
}
```

The default MessagingProvider provided by Liberty will be for Kafka, requiring the following Liberty server.xml feature declaration:

```
<feature>mpReactiveMessaging-1.0</feature>
```

The connection-specific configuration details for a Kafka provider such as IBM Events Streams remains a work-in-progress.

Asynchronous HTTP Microservice Patterns

Asynchronous JAX-RS

As described elsewhere, in the synchronous part of this Programming Model document, synchronous REST resources on the server-side are declared through JAX-RS annotations and consumed on the client-side through MicroProfile REST client annotations to inject the REST client interface. This provides a simple call-response Java model for both the client and the server, with either side knowing or caring what technology (Java or otherwise) is used by the end. This independence of implementation between client and server can also be extended to whether it processes the REST request synchronously or asynchronously. At the *transport* level there is just an open connection over which a synchronous HTTP request and response flows. At the *application* level the use of asynchronous REST APIs directs the WebSphere Liberty JAX-RS runtime to manage the application threads that access the HTTP connection asynchronously.

Server-side Async JAX-RS

Scalable server-side runtimes like Liberty manage high volume throughputs by internally scheduling application worker threads asynchronously from the threads that process the inbound connections. Applications are unaware of this and may act in a wholly synchronous fashion. There is typically little value in applications that run in environments like Liberty to take responsibility for their own server-side async processing although JAX-RS supports a pattern for doing so. A very simple example is given below for illustrative purposes but this is typically not that useful.

This is a standard part of JAX-RS 2.0 and later and requires a Liberty server.xml configuration that includes
`<feature>jaxrs-2.0</feature>`
or
`<feature>jaxrs-2.1</feature>`

This is achieved using the JAX-RS @Suspended annotation to inject an object implementing the javax.ws.rs.container.AsyncResponse interface that will handle the response on another thread. The Liberty JAX-RS runtime takes care of delivering the response to the appropriate HTTP connection.

For example, a simple group chat pub/sub service could be implemented on the server with the following pattern (over-simplified to highly the async JAX-RS aspects):

```
@Path("groupchat")
public class GroupChatResource {
   protected List<AsyncResponse> responses = new ArrayList<AsyncResponse>();

   // Handle this GET asynchronously. This method is required to return void.
   // The client expects a String response, which is returned on another
   // thread by the resume() processing.
   @GET
   @Produces("text/plain")
   public synchronized void getMessage(@Suspended AsyncResponse response) {
      responses.add(response);
   }

   @POST
   @Consume("text/plain")
   public synchronized void sendMessage(String message) {
      for (AsyncResponse response : responses) {
         response.resume(message);
      }
   }
}
```

Fully working Liberty examples of asynchronous JAX-RS are shown in the following GitHub repo:
https://github.com/WASdev/sample.async.jaxrs

The scenario in the example above can be improved upon using the JAX-RS server-sent events (SSE) API as illustrated below. A problem with the scenario above is that the clients calling the GET method only wait 30 seconds (by default) for a response, so if nobody POSTs in that time, the client will timeout and potentially miss a message while it is reconnecting. SSE's automatically reconnect and have a much more robust broadcast mechanism.

Client-side Async JAX-RS

As described previously in this book, any call-response REST resource can be invoked in a type-safe manner by using the MicroProfile @RestClient annotation to inject a Java interface for the REST resource.

Regardless of the implementation of the resource itself, the client can direct the MicroProfile Rest Client runtime to process the response of any method *asynchronously* by updating the @RestClient interface method's return type to be `CompletionStage`. Liberty integrates the

Apache CXF implementation of MicroProfile Rest Client, enabled through the definition in server.xml of the following feature:

```
<feature>mpRestClient-1.1</feature>
```

The Liberty Rest Client runtime executes the outbound request on the caller's thread using the same context as a synchronous invocation, but control is returned immediately to the caller. By default (in the absence of any application-provided ExecutorService) the response is dispatched on a thread from Liberty's global thread pool. The earlier synchronous client example for the Portfolio service can be turned into an asynchronous client as follows:

```
@RegisterRestClient
@Path("/")
public interface PortfolioClient {
@POST
@Path("/{owner}")
@Produces("application/json")
public CompletionStage<Portfolio> createPortfolio(
      @PathParam("owner") String owner);
```

The client interaction for creating a portfolio for John and having some processing run asynchronously against the result is then:

```
String owner = "John";
portfolioClient.createPortfolio(owner).thenApply(portfolio ->
   portfolio.setLoyalty();  // Asynchronously set initial loyalty
);
```

The result of the createPortfolio() request is a `CompletionStage` for a Portfolio object that will be processed on another thread. In this example the Liberty RestClient runtime will send the createPortfolio() outbound request on the calling thread with that thread's context, and assign the future result to a `CompletionStage` with an executor that processes the result asynchronously on another thread in the Liberty global thread pool. The calling thread proceeds without waiting for the HTTP response from createPortfolio(). In this example, the initial loyalty level for the new portfolio instance is processed on the asynchronous thread.

Thread contexts that are present on the calling thread are not propagated by default to the asynchronous response thread for any further activity directed through the `CompletionStage` and so are not available for the processing of setLoyalty() in the example above. If certain thread contexts are required for asynchronous processing then the client may implement and register a MicroProfile RestClient `AsyncInvocationInterceptor` which provide callbacks that are called on the outbound and asynch response threads. The following example shows how an application-provided AsyncInvocationInterceptorFactory and associated AsyncInvocationInterceptor class could be used to propagate a `ThreadLocal` value from the originating thread to async thread:

```
import org.eclipse.microprofile.rest.client.ext.*;
public class MyFactory implements AsyncInvocationInterceptorFactory {
```

```
    public AsyncInvocationInterceptor newInterceptor() {
         return new MyInterceptor();
    }
}

public class MyInterceptor implements AsyncInvocationInterceptor {
    // This field is temporary storage to facilitate copying a ThreadLocal
value
    // from the originating thread to the new async thread.
    private volatile String someValue;

    public void prepareContext() {
        someValue = SomeClass.getValueFromThreadLocal();
    }
    public void applyContext() {
        SomeClass.setValueIntoThreadLocal(someValue);
    }
}

@RegisterProvider(MyFactory.class)
public interface MyAsyncClient {...}
```

The MicroProfile RestClient API builds upon an underlying JAX-RS 2.1 runtime, and the latter also provides lower-level asynchronous/reactive client APIs. The JAX-RS asynchronous API defines an RxInvoker interface for reactive invocation of HTTP methods with a default implementation provided by CompletionStageRxIvoker, but other reactive frameworks can also be plugged in if they implement RxInvoker. This pluggability simplifies the integration of popular reactive frameworks like RxJava but it is harder to use than the MicroProfile Rest Client and, unlike the RestClient, does not provide a typesafe invocation strategy or a well-defined means to propagate contexts to asynchronous threads. The use of the MicroProfile RestClient API is recommended for calling REST resources, for both synchronous and asynchronous client processing models.

Server-Sent Events and WebSockets

In addition to a reactive client API, JAX-RS 2.1 also adds a Java API for Server-Sent Events (SSE), an event streaming protocol introduced as part of HTML5. This provides a standard way for a Java microservice to declare an SSE source endpoint and for clients to register SSE sinks (consumers) with an SSE source. Each server-initiated event is part of an event-stream that a client consumes in a reactive fashion over a persistent HTTP connection between the client and the server. This is a useful pattern primarily for the UIs of rich web clients that maintain a long-running HTTP connection without needing to poll for each event.

The server-side of the simple chat above might look like this:

```
import javax.ws.rs.sse.*;

@Path("chat")
public class ChatResource {

      @Context
      private Sse sse;
```

```
      private static SseBroadcaster broadcaster;

      private synchronized static SseBroadcaster getOrCreateBroadcaster(Sse
sse) {
            if (broadcaster == null) {
                  broadcaster = sse.newBroadcaster();
            }
            return broadcaster;
      }

      @GET
      @Path("register")
      @Produces(MediaType.SERVER_SENT_EVENTS)
      public void register(@Context SseEventSink sink, @Context Sse sse) {
            SseBroadcaster b = getOrCreateBroadcaster(this.sse);
            b.register(sink);
      }

      @PUT
      public void broadcast(@QueryParam("user") String user,
@QueryParam("message") String message) {
            SseBroadcaster b = getOrCreateBroadcaster(sse);
            ChatMessage chatMessage = new ChatMessage(user, message);
            OutboundSseEvent event =
               sse.newEventBuilder().data(ChatMessage.class, chatMessage)
                  .id(""+chatMessage.getMsgID())
                  .mediaType(MediaType.APPLICATION_JSON_TYPE)
                  .build();
            b.broadcast(event);
}}
```

This code snippet is part of a complete Liberty SSE sample here:
https://github.com/OpenLiberty/sample-sse-chat

SSE is one-way traffic over each long-running HTTP connection – server to client(s) – with each client registering asynchronous SseEventSink handlers with the client-side JAX-RS runtime.

WebSockets is a variation of this that negotiates an upgrade to an HTTP connection, resulting in a two-way persistent connection.

Both of these protocols encourage and enable asynchronous processing of events but are really targeted at rich web client UIs rather than microservice-microservice interactions. Truly event-driven microservices typically do not establish direct connections between even source and sink – instead they use a common eventing infrastructure like Apache Kafka to which they connect and register as publishers or consumers of event streams categorized into topics.

Final Words on Microservices Interactions

With the key patterns of microservices communications now in hand, it should now start to become clear that the interactions between microservices are a very important part of building cloud-native systems. The prior sections have outlined the conceptual approaches supported by

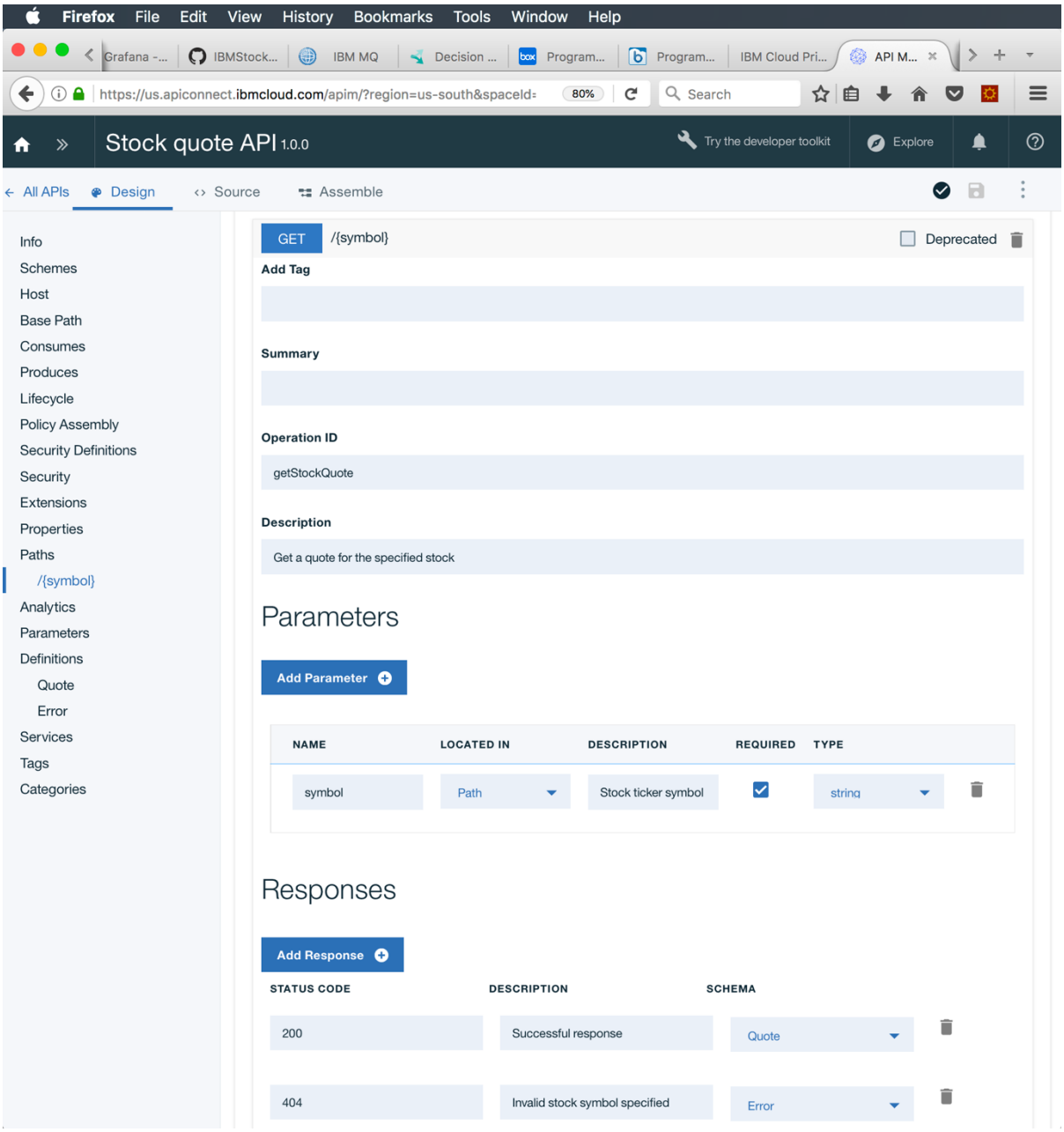
specific patterns and examples. From these, more elaborate and powerful cloud-native systems can be constructed. Advanced patterns and examples will be described later, which look at how these building blocks are assembled to solve more complex business problems.

Exposing and Managing an External API

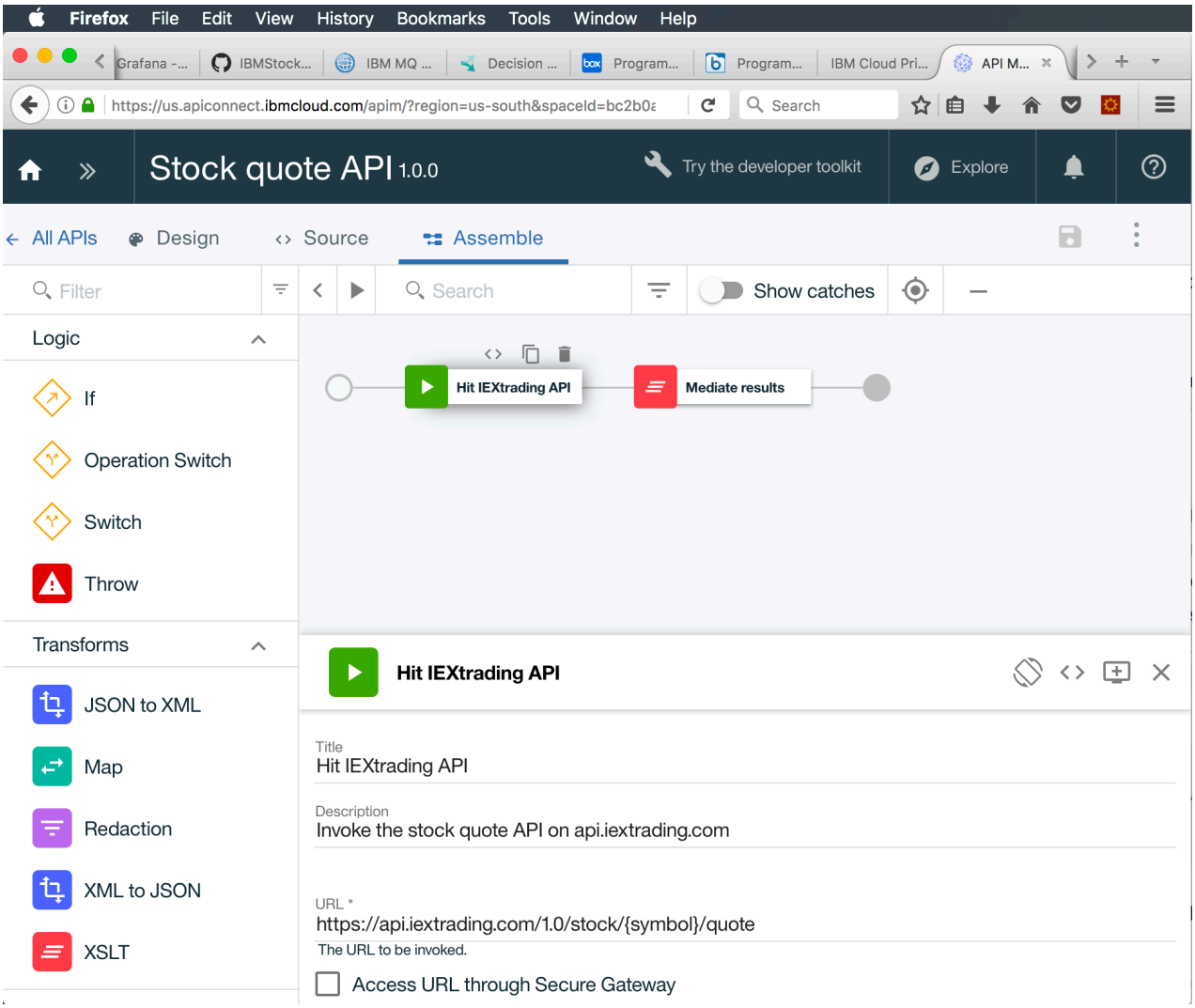
The prior sections on APIs focused on getting an API defined and getting it implemented. This is necessary and is sufficient if you are using REST to communicate between microservices. This section goes beyond that and covers the additional considerations necessary to have one of the APIs which was created and implemented become a managed API.

API Connect

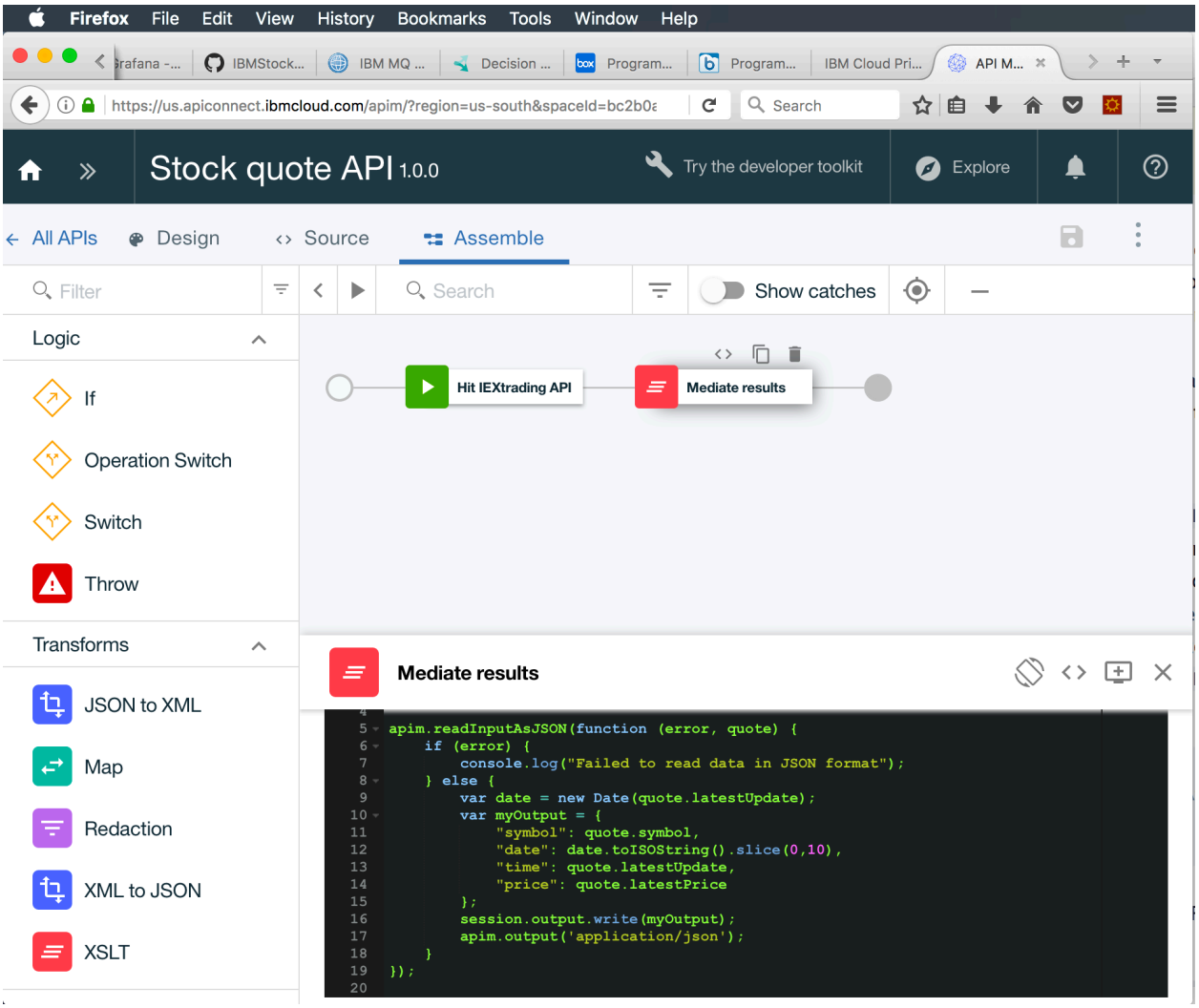
Let's take a look at how we used API Connect to expose a stock quote API backed by IEX Trading (it could just as easily be exposing an API running in your Kubernetes environment). It has four main views: the form-based view to enter stuff in each field, the assembly diagram, the raw OpenAPI v2 view, and the portal. Let's view these in order:



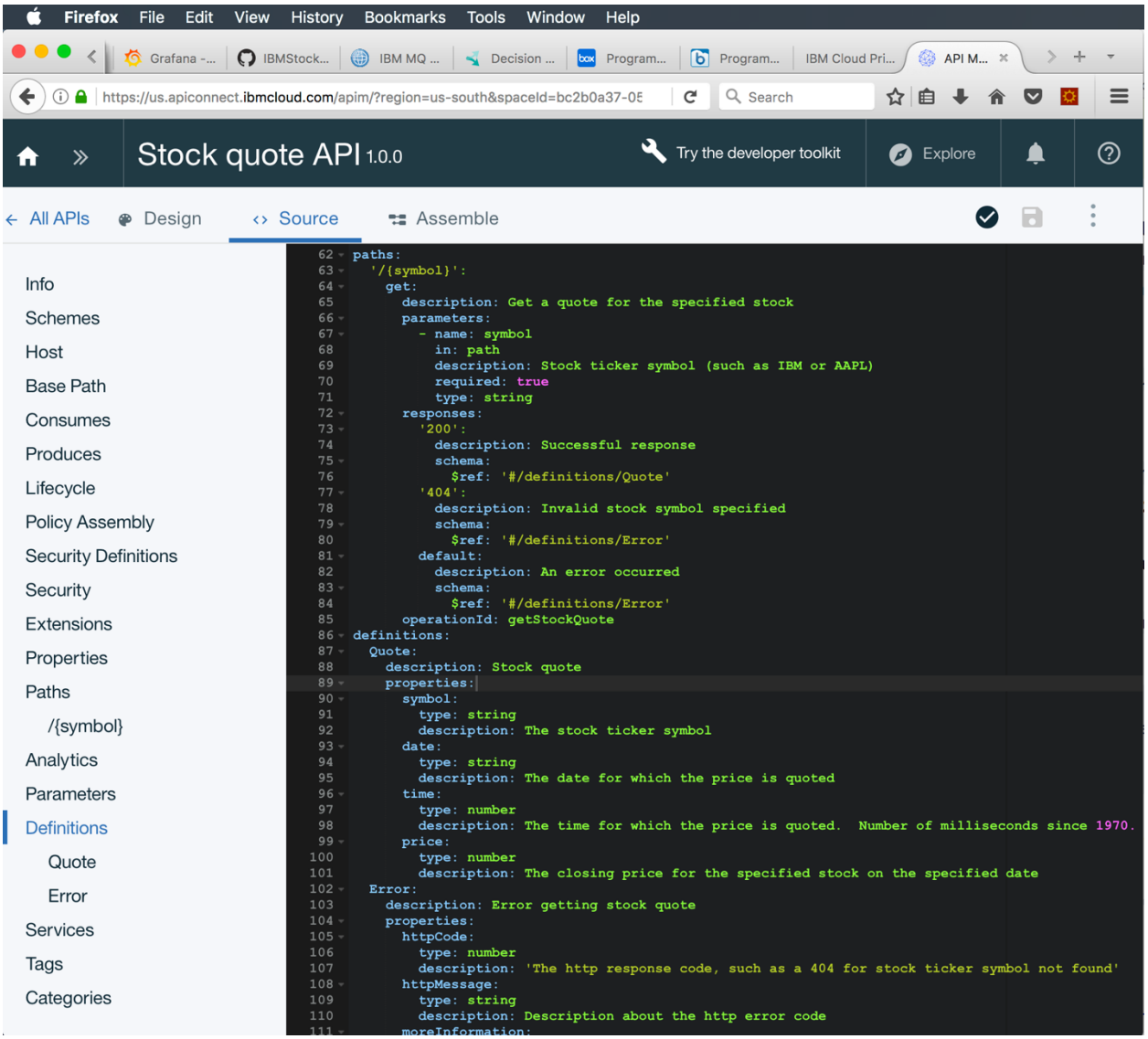
Now the assembly diagram:



Let's also view the (Node.js) mediation node in this assembly diagram:



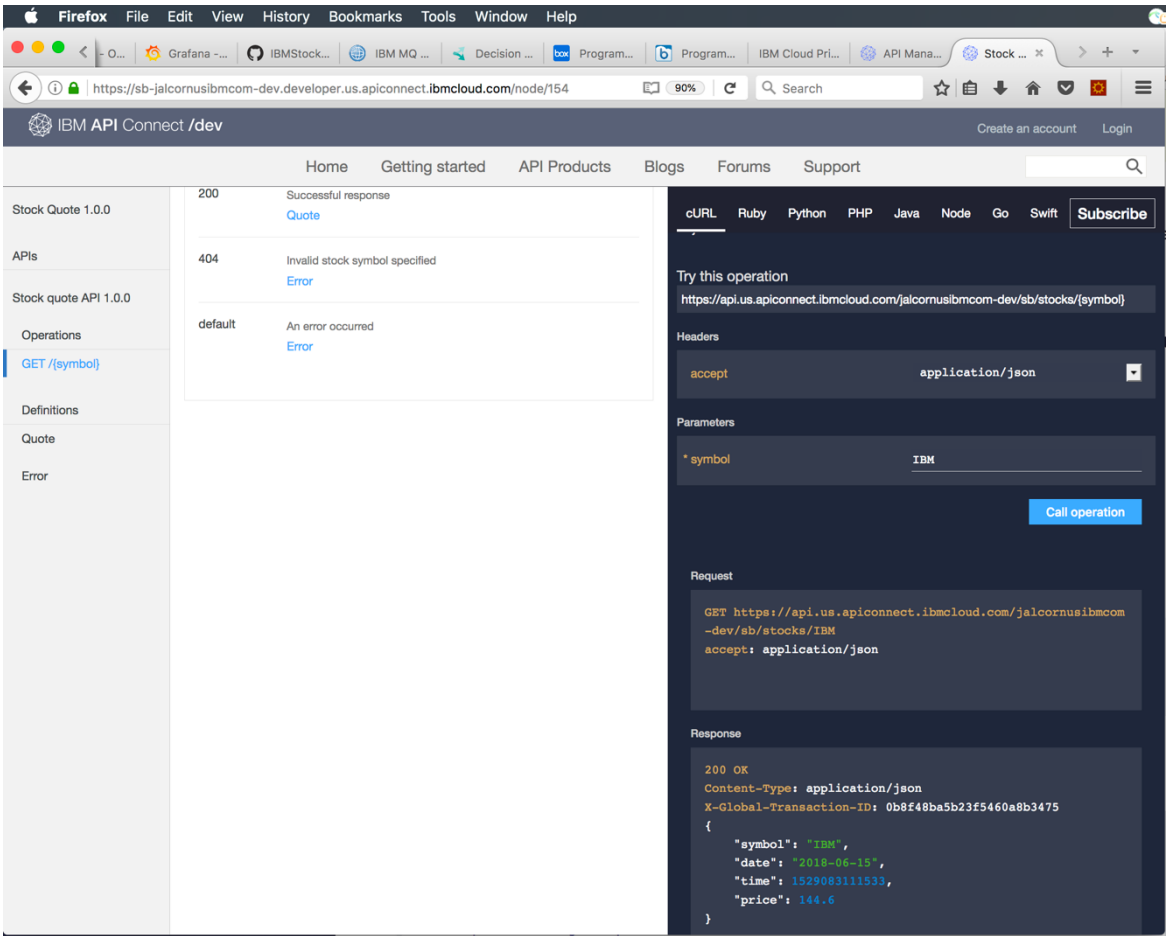
As you can see, it mostly just plucks a few of the dozens of fields for the response. The only one it does any manipulation of is the quote.latestUpdate field, converting it from number of milliseconds since 1970 to a date of the form YYYY-MM-DD. Lastly, let's look at the raw Swagger v2 view:



We can test this URL via `curl`, like this:

```
Johns-MacBook-Pro-8:StockTrader jalcorn$ curl
https://api.us.apiconnect.ibmcloud.com/jalcornusibmcom-dev/sb/stocks/IBM
{"symbol":"IBM","date":"2018-06-15","time":1529083459850,"price":144.615}
Johns-MacBook-Pro-8:StockTrader jalcorn$
```

Or we can do this via the "Call operation" button in the API Connect portal:



Additional considerations for external APIs

When you expose an API, there are a few things to consider. One of them is whether to perform **mediation** on the API. You might want to simplify the input required, or the output returned (as our stock quote example above did – it just picked 4 out of the dozens of fields coming back from the original API to return in the managed/exposed API.

Another is whether you want to adjust how **security** is handled. Maybe the API you want to expose expects to receive a JWT, but you want your managed API to require an API Key instead. You'd need to have a mediation node out in front of the invoke node, that verifies if the API Key is correct, and if so, constructs the expected JWT.

For APIs that are managed, you should leverage the capabilities of the API management product you are using, such as IBM's API Connect, where appropriate. For example, enabling rate limiting, to say, for example, you don't want to allow more than 100 invocations per minute. Or using the built-in analytics to see how your APIs are being used over time.

Programming Model Concept – Consume APIs

Whether a microservice or a serverless function, as the consumer of an API, you need to take a certain amount of care in validating responses you receive from API invocations. The robustness principle (https://tools.ietf.org/html/rfc1122#page-12) provides the best guidance: "Be liberal in what you accept, and conservative in what you send". Assume that APIs will evolve over time and be tolerant of data you do not understand.

As a guideline:

- Only validate the request against the variables or attributes that you need.

  Do not validate against variables just because they are provided. If you are not using them as part of your request, do not rely on them being there.

- Accept unknown attributes.

  Do not issue an exception if you receive an unexpected variable. If the response contains the information you need, it does not matter what else is provided alongside.

This is especially relevant for languages like Java, where JSON serialization and deserialization usually occurs indirectly (e.g. via the Jackson libraries, or JSON-P/JSON-B) and objects are strongly typed. Look for language mechanisms that allow you to specify more generous behavior like ignoring unknown attributes, or to define or filter which attributes should be serialized.

Programming Model for Java – Consume APIs

Introduction

Above, we were talking about how to implement and expose your own APIs. But very often, your microservices will need to invoke other APIs. There are two approaches for how this is done:

1. Use an SDK that you will package inside your microservice.
2. Call the API via the mpRESTClient

SDK

For the first category, you simply need to download the SDK jar file, and code to it directly. For example, I downloaded the Twitter4J SDK jar for talking to Twitter from Java, and my Java code imports interfaces and classes from that SDK, and invokes methods on them, just like you would on any other Java class. In my NotificationTwitter microservice, I have the following import statements:

```
//Twitter for Java (Twitter4J)
import twitter4j.Twitter;
import twitter4j.TwitterException;
import twitter4j.TwitterFactory;
import twitter4j.conf.ConfigurationBuilder;
```

And I use the following code to initialize the SDK:

```
  logger.info("Initializing Twitter API");

  ConfigurationBuilder builder = new ConfigurationBuilder();
  builder.setDebugEnabled(true);
  builder.setOAuthConsumerKey(System.getenv("TWITTER_CONSUMER_KEY"));
  builder.setOAuthConsumerSecret(System.getenv("TWITTER_CONSUMER_SECRET"));
  builder.setOAuthAccessToken(System.getenv("TWITTER_ACCESS_TOKEN"));

builder.setOAuthAccessTokenSecret(System.getenv("TWITTER_ACCESS_TOKEN_SECRET"
));

  TwitterFactory factory = new TwitterFactory(builder.build());
  twitter = factory.getInstance(); //initialize twitter4j
```

And the following `twitter.updateStatus()` code to send a tweet:

```
  Date now = new Date();
  String message = "On "+format.format(now)+", "+owner+" changed
status from "+oldLoyalty+" to "+loyalty+". #IBMStockTrader";

  logger.info("Sending following tweet: "+message);
```

```
  twitter.updateStatus(message); //throws TwitterException on
failure

  logger.info("Message tweeted successfully!");
```

To get this to compile, you need to update your Maven pom.xml to list your SDK as a dependency:

```
    <dependency>
        <groupId>org.twitter4j</groupId>
        <artifactId>twitter4j-core</artifactId>
        <version>[4.0,)</version>
        <scope>provided</scope>
    </dependency>
```

Then you just need to package the jar file in your microservice war file's WEB-INF/lib directory. For example, here's what's in the war file for my `notification-twitter` microservice (see bold emphasis):

```
Johns-MacBook-Pro-8:notification-twitter jalcorn$ jar -tvf
target/notification-twitter-1.0-SNAPSHOT.war
     0 Thu Apr 05 14:30:48 EDT 2018 META-INF/
   133 Thu Apr 05 14:30:48 EDT 2018 META-INF/MANIFEST.MF
     0 Thu Apr 05 14:30:48 EDT 2018 WEB-INF/
     0 Thu Apr 05 14:30:48 EDT 2018 WEB-INF/classes/
     0 Thu Apr 05 14:30:48 EDT 2018 WEB-INF/classes/com/
     0 Thu Apr 05 14:30:48 EDT 2018 WEB-INF/classes/com/ibm/
     0 Thu Apr 05 14:30:48 EDT 2018 WEB-
INF/classes/com/ibm/hybrid/
     0 Thu Apr 05 14:30:48 EDT 2018 WEB-
INF/classes/com/ibm/hybrid/cloud/
     0 Thu Apr 05 14:30:48 EDT 2018 WEB-
INF/classes/com/ibm/hybrid/cloud/sample/
     0 Thu Apr 05 14:30:48 EDT 2018 WEB-
INF/classes/com/ibm/hybrid/cloud/sample/portfolio/
     0 Thu Apr 05 14:30:48 EDT 2018 WEB-INF/lib/
  5220 Thu Apr 05 14:30:48 EDT 2018 WEB-
INF/classes/com/ibm/hybrid/cloud/sample/portfolio/NotificationTw
itter.class
290456 Thu Mar 01 15:07:22 EST 2018 WEB-INF/lib/twitter4j-core-
4.0.4.jar
  1403 Thu Mar 01 15:07:22 EST 2018 WEB-INF/web.xml
  2780 Thu Apr 05 14:30:32 EDT 2018 META-
INF/maven/com.stocktrader/notification-twitter/pom.xml
   134 Thu Apr 05 14:30:48 EDT 2018 META-
INF/maven/com.stocktrader/notification-twitter/pom.properties
Johns-MacBook-Pro-8:notification-twitter jalcorn$
```

As you can see, there's just the one Java class, the web.xml, and the SDK jar (plus a couple of Maven files apparently sneaked in there, but they aren't actually needed at runtime).
The net is, if an SDK is available for the service you wish to use, then you can simply use it like you would any Java library. Note that Stock Trader also uses the Jedis Java SDK for Redis. And though it doesn't use it today, there is a Java SDK for Watson as well, that it could switch over to use when it talks to the Watson Tone Analyzer.

MpRestClient

This topic was covered as part of communication between microservices. You already know what to do here. However, there might be some additional things to consider. For example, you might need a different authorization head or approach to dealing with security when the API you are calling is outside of your control and often times outside your company.

Of course, if a real SDK is available for the service you want to call, you can just use that. Such an SDK might take care of additional stuff like security for you, like in the Twitter4J SDK for Twitter that we showed above did with the O-Auth stuff.

Programming Model Concept – Security

As always, security is an important consideration. In this chapter, we will discuss security requirements within the context of a cloud native microservices architecture.

The dynamic nature of microservice architectures changes how security should be approached. This is specifically relevant to how application or service boundaries might be defined. Figure 1 shows simplified representations of how requests flow in monolithic and microservice architectures. Connections between microservices and persistent storage services (RDBMS, NoSQL) have been hidden for general simplicity, but represent yet another dimension of network traffic.

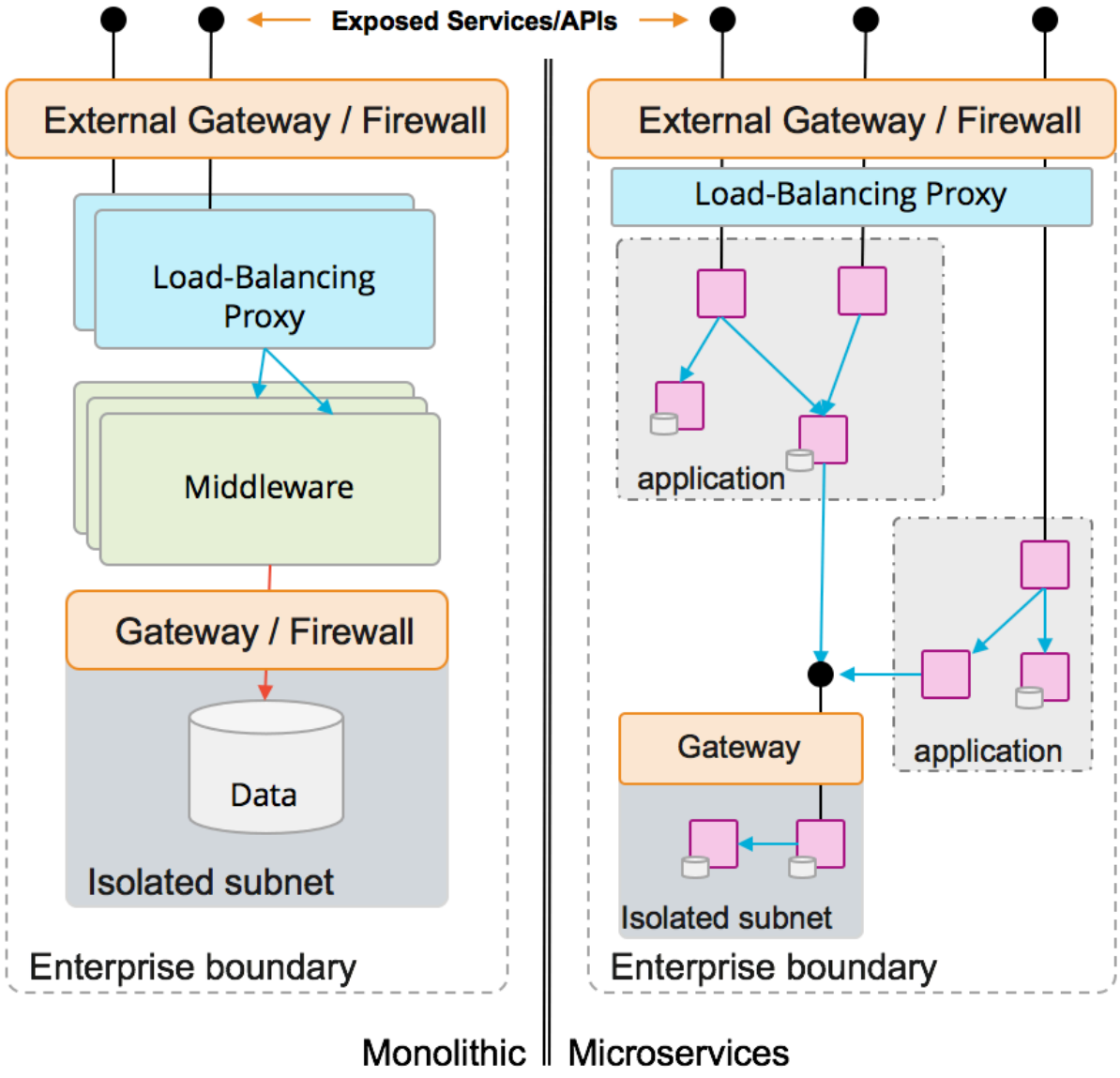


Figure 4 Simplified view of monolithic and microservice architectures

The left side of Figure 4 shows a simplification of a traditional monolithic architecture. Requests come through the gateway, go through a load balancer, then to business logic in the middleware, and then down to the data tier.

On the right side, things are less organized. Collections of microservices are grouped together as applications, and surface external APIs as endpoints through a gateway. One obvious difference is the number of moving parts. The other is how many more requests there are. The less obvious difference, as mentioned previously, is that the composition of services shown on the right hand side are under constant change. Individual service instances will come and go based on load. Microservices will be continuously updated independently. There could be hundreds of updates a day across all of the different services. Securing the perimeter is not going to be enough. The question is how to secure a rapidly changing infrastructure in an organic way that still allows the individual applications to change and grow without requiring central coordination.

Network segmentation

Giving Figure 4 another look, there is an isolated subnet in both systems. Using an additional firewall or gateway to guard resources that require additional levels of protection is a good idea in either environment.

Something else to observe in microservices approach shown in Figure 4, is that the externally exposed APIs are connected to two different microservices-based applications. Defined application boundaries provide a reasonable amount of isolation between independently varying systems and are a good way to maintain a reasonable security posture in a dynamic environment.

Network segmentation happens somewhat naturally in hosted environments: for example, each microservices application shown in Figure 4 could be a separate tenant in a multi-tenant environment. Where it doesn't happen naturally, you might want to encourage it.

Ensuring data privacy

Different kinds of data require different levels of protection, and that can influence how data is accessed, how it is transmitted, and how it is stored.

When dealing with data:

- do not transmit plain-text passwords
- protect private keys
- use known data encryption technologies, don't invent your own
- securely store passwords using salted hashes

Further, sensitive data should be encrypted as early as possible, and decrypted as late as possible. If sensitive data must flow between services, it should only do so while encrypted, and should not be decrypted until the data actually needs to be used. This can help prevent accidental exposure in logs, for example.

Backing services

As mentioned earlier, connections to backing services are additional sources of network traffic that need to be secured. In many cloud environments, backing services are provided by the

platform, relying on multi-tenancy and API keys or access tokens to provide data isolation. It is important to understand the characteristics of backing services, especially how they store data at rest, to ensure regulatory requirements (HIPAA, PCI, etc.) are satisfied.

Log Data

Log data is a tradeoff between what is required to diagnose problems and what must be protected for regulatory and privacy reasons. When writing to logs, take full advantage of log levels to control how much data is written to logs. In the case of user-provided data, consider whether or not it belongs in logs at all. For example, do you really need the value of the attribute written to the log, or do you really only care about whether or not it was null?

Automation

As much as possible will be automated in cloud native, microservice environments, which includes general operations. Repeatable automated processes should be used for applying security policies, credentials, and managing SSL certificates and keys across segmented environments to help avoid human error.

Credentials, certificates, and keys must be stored somewhere for automation to use. Remember:

- Don't store your credentials alongside your applications.
- Don't store your credentials in a public repository
- Only store encrypted values

Identity and Trust

A highly distributed, dynamic environment like microservices architectures place some strain on our usual patterns of establishing identity. We must establish and maintain the identity of users without introducing additional latency and contention with frequent calls to a centralized service.

Establishing and maintaining trust throughout this environment is not the easiest either. It is inherently unsafe to assume secure private network, as the news tells us. End-to-end SSL can bring some benefit in that bytes are not flowing around in plain text, but it doesn't establish a trusted environment on its own, and requires key management.

The following sections will outline techniques for performing authentication and authorization and identity propagation to establish and maintain trust for inter-service communications.

Authentication and Authorization

There are new considerations for managing authentication and authorization in microservice environments. With a monolithic application, it is common to have fine-grained roles, or at least role-associated groups in a central user repository. With the emphasis on independent life-cycles for microservices, however, this dependency is an anti-pattern, as development of a would-be independent microservice is then constrained by and coupled with updates to the centralized resource.

It is common to have authentication (establishing the user's identity) performed by a dedicated, centralized service or even an API gateway. This central service can then further delegate user authentication to a third party, like an Open Authentication (OAuth) provider.

When working with authorization (establishing a user's authority or permission to access a secured resource), it is better in a microservices environment to keep group or role definitions coarse-grained in common, cross-cutting services. Allow individual services to maintain their own fine-grained controls. The guiding principle here is again independence. A balance must be found between what could be defined in common authorization service to meet requirements for the application as a whole, and what authorization requirements are implementation details for a particular service.

Open Authentication (OAuth 2.0)

The Open Authentication protocol has emerged as a standard pattern for delegating authentication to a third-party. Some application servers, like WebSphere Liberty, have OpenID Connect features to facilitate communication with OAuth providers. Cloud-based services like IBM AppID act as an OAuth Identity provider, with federated identity management across a variety of other Open ID Connect or SAML identity providers.

JSON Web Tokens (JWTs)

Identity propagation is another challenge in a microservices environment. Once the user (human or service) has been authenticated, that identity needs to be propagated to the next service in a trusted way. Frequent calls back to a central authentication service to verify identity are inefficient, especially given that direct service-to-service communication is preferred to routing through central gateways whenever possible to minimize latency.

JSON Web Tokens (JWTs) can be used to carry along a representation of information about the user. In essence, you want to be able to:

- Know the request initiated from a user request
- Know the identity the request was made on behalf of
- Know that this request isn't a malicious replay of a previous request

JWTs are compact and URL-friendly. As you might guess from the name, A JWT (pronounced "jot") contains a JSON structure. The structure contains some standard attributes (called claims), like issuer, subject (the user's identity!) and expiration time, which have clear mappings to other established security mechanisms. There is also room for claims to be customized, allowing additional information to be passed along.

While building your application, there likely will be times when you will be told that your JWT is not valid, and you won't understand why. For security and privacy reasons, it is not a good idea to expose (to the front end) exactly what is wrong with the login, as it can leak implementation or user details that could be used maliciously. JWT.IO provides some very nice web utilities for working with JWTs, including a quick way to verify whether the encoded JWT that scrolled by in your browser console or your trace log is valid or not.

Dealing with time

One of the nice attributes of JWTs is that they allow identity to be propagated, but not indefinitely. JWTs will expire, which should trigger revalidation of the identity resulting in a new JWT. JWTs have three fields relating to time and expiry, all of which are optional. It should be considered a best practice to include these fields, and to validate them when they are present as follows:

- The time the JWT was created (iat) is prior to the current time and is valid.
- The "not process before" claim time (nbf) is prior to the current time and is valid.
- The expiration time (exp) is after the current time and is valid.

All of these times are expressed as Unix epoch timestamps.

Signed JWTs

Signing a JWT helps establish trust between services, as the receiver can than then verify the identity of the signe, and that the contents of the JWT have not been modified in transit.

JWTs can be signed using shared secrets, or a public/private key pair (SSL certificates work well with a well-known public key). Common JWT libraries all support signing.

API Keys and Shared Secrets

An API key can be one of a few different things, but the usual usage is to identify the origin of a request. Identifying the origin of a request is important for API management functions like rate limiting or usage tracking. API keys are usually separate from an account's credentials, enabling them to be created or revoked at will.

An API key is sometimes a single string, but unlike a password, these strings are randomly generated and over 40 characters long. Sometimes the string is used directly in the request as either a bearer token or query parameter (over https), however, using an API key is more secure when used as a shared secret for signing tokens (JWTs) or for digest-methods of request authentication (HMACs).

If your service creates API keys, ensure that API keys are securely generated with high entropy. and are stored appropriately. Ensure these keys can be revoked at any time. If your application has distinct parts, consider using different API keys for each part: this can enable more granular usage tracking, and reduces the impact if a key is compromised.

Asymmetric keys can also be used (e.g. SSL), but require a public key infrastructure (PKI) to be maintained. While more secure, this is also more labor-intensive, and can be especially hard to manage when API consumers are third parties.

Consistent with the twelve factors, secrets should be injected configuration when used by services for intra-service communication: the values should never be hard-coded, as that increases the likelihood they will be compromised (e.g. checked into a repository in plain-text), and either makes your code sensitive to the environment, or requires different environments to use the same shared secret, neither of which are good practices.

Programming Model for Java – Security

Authentication

Let's start with **Authentication**. The good news here is that, when using Java-based microservices, you don't need to do much in your Java code; most of this is just configuration in your server.xml and your web.xml. The only thing that can show through in your Java code is if you want to provide your own custom login form (note that if your microservice is configured to use OpenID Connect, that would provide its own login UI); if so, you'd write a servlet or JSP (or whatever) that provides the UI, and you'd reference that login form in your web.xml. OpenID Connect is on open standard and decentralized protocol. It allows end users to be authenticated by co-operating sites. See the `Login.java` in the `trader` microservice for an example of a custom login form; other than the usual UI stuff (`doGet` to produce the HTML, and `doPost` to react to the Submit button being clicked), the one line of security-related code you'd call is `request.login(id, password);`

As for the web.xml, if you want to provide your own login form, you'd have a stanza like the following (note that I have the `/login` path mapped to my `Login` servlet, and the `/error` path mapped to my `Error` servlet – the `Error` servlet (called if a bad id/pwd was entered, or if other problems occurred) just says that something went wrong and to call your Todd for help, and then redirects to the `Login` servlet when they click the OK button):

```
<login-config>
   <auth-method>FORM</auth-method>
   <realm-name>BasicRegistry</realm-name>
   <form-login-config>
      <form-login-page>/login</form-login-page>
      <form-error-page>/error</form-error-page>
   </form-login-config>
</login-config>
```

The rest of the configuration, such as specifying what user registry to authenticate against, is done in the server.xml. Let's first look at the **Basic Registry**, that lets you hard-code IDs, passwords, and groups directly in the server.xml. Note that Liberty has a `securityUtility` tool that you can use to encrypt passwords, etc: https://www.ibm.com/support/knowledgecenter/en/SSEQTP_liberty/com.ibm.websphere.wlp.doc/ae/rwlp_command_securityutil.html.

Though you wouldn't use a Basic Registry in production, it can be a handy way to do unit test without needing all of the overhead of interacting with LDAP, O-Auth, or OpenID Connect.

```
    <basicRegistry id="basic" realm="BasicRegistry">
        <user name="admin" password="admin"/>
        <user name="stock" password="trader"/>
        <user name="read"  password="only"/>
        <user name="other" password="denied"/>
        <group name="StockTrader">
            <member name="admin"/>
            <member name="stock"/>
```

```
        </group>
        <group name="StockViewer">
            <member name="read"/>
        </group>
    </basicRegistry>
```

As you can see, with this in your server.xml for the trader microservice, you'd be able to login with an id/pwd of stock/trader, for example. Note also the groups; we'll talk about that in the **Authorization** section later.

When running in an on-premises private cloud, such as **IBM Cloud Private**, you may want to authenticate against your corporate **LDAP** user registry, such as when providing an app meant for use by your own employees. We have a version of Stock Trader that does this, authenticating against our BluePages LDAP server. Note that even an app in the public cloud, such as the **IBM Kubernetes Service** (IKS), could reach your on-premises LDAP server via a VPN or Secure Gateway.
First, you have to add the `ldapRegistry-3.0` feature to your server.xml feature list, then use a stanza like this:

```
    <ldapRegistry ldapType="IBM Tivoli Directory Server"
host="bluepages.ibm.com" port="389" baseDN="o=ibm.com"
realm="BluePages">
        <idsFilters
userFilter="(&amp;(emailAddress=%v)(objectclass=person))"
groupFilter="(&amp;(cn=%v)(|(objectclass=groupOfNames)(objectcla
ss=groupOfUniqueNames)(objectclass=groupOfURLs)))"/>
    </ldapRegistry>
```

Note the `idsFilters` sub-tag, where we say that the email address field in the LDAP registry will be considered the user ID (alternately, we could have made it the employee serial number, for example).
The above two approaches (`basicRegistry` and `ldapRegistry`) are supported in either Open Liberty or the commercial Liberty. Another option, which is currently only available in the commercial Liberty (though rumor is this will move down into Open Liberty in the 3Q release, 18.0.0.3), is the `socialLogin-1.0` feature. This lets you do your authentication against popular social media providers, such as logging in with your id/pwd for Facebook, Twitter, LinkedIn, Google, or GitHub. For example, here's how to say you want clients to login against **Facebook**:

```
    <facebookLogin clientId="${env.FB_APP_ID}"
          clientSecret="$(env.FB_APP_SECRET}"
          redirectToRPHostAndPort="${env.OIDC_NODEPORT}">
    </facebookLogin>
```

Note the `redirectToRPHostAndPort` parameter. This is needed in a Kubernetes environment, where the native port numbers, like 9443, are mapped to node ports, like 32389. The Liberty server is unaware of the node port value (this happens outside of its understanding), so they added this parameter so that you can tell it the port number (as well as the proxy host

DNS name or IP address, which it also wouldn't otherwise know). The value expected is of the form host:port, without any further path stuff after that.
Or, you can manually specify all of the OpenID Connect parameters yourself, like we do when doing an IBMid login. Note here we are using environment variables (provided to the microservice via entries in the deploy.yaml that reference fields of a Kubernetes secret), rather than hardcoding such values:

```
    <oidcLogin id="${env.OIDC_NAME}" clientId="${env.OIDC_ID}"
displayName="${env.OIDC_NAME}" clientSecret="${env.OIDC_SECRET}"
authorizationEndpoint="${env.OIDC_AUTH}"
tokenEndpoint="${env.OIDC_TOKEN}" scope="openid profile email"
userNameAttribute="email" issuer="${env.OIDC_ISSUER}"
trustAliasName="${env.OIDC_KEY}"
redirectToRPHostAndPort="${env.OIDC_NODEPORT}"/>
```

You've now seen how to authenticate against various user registries, such as a hard-coded one, an LDAP server, an OpenID Connect server, or against various social media providers like Facebook. The nice thing is, we didn't have to change our Java code as we changed authentication approaches! But we did have to rebuild our Docker container each time; we need to decide whether to pursue an approach that provides the server.xml via something like a persistent volume, rather than baking it in to the Docker image.

Single Sign On

That covers authentication; let's move on to **SSO**. The standard cloud-native approach here is to use **JWT** (JSON Web Tokens). JWT tokens become the de-factor standard for access or id tokens. It presents claims to be transferred between two parties. It contains a set of claims as a JSON object that is based64url encoded and digitally signed. It consists of three parts: header, payload and signature, separated by dots. Basically, you create a JWT upon a successful login, and pass it along in the Authorization header on each HTTP/HTTPS request. Those downstream microservices can be configured to reject the request unless the JWT, with the expected claims and encryption, is present.

MP-JWT adds 2 new claims to the standard JWT:

- upn: a unique principal name
- groups: the token subject's group memberships.

To retrieve a MP-JWT from a service invocation, you can use CDI in inject one. First, add the following import statement:

```
import org.eclipse.microprofile.jwt.JsonWebToken;
```

Then just add the following line to your class-level variable declarations:

```
private @Inject JsonWebToken jwt;
```

The claims in this injected JWT will come from the `jwtBuilder` stanza in your server.xml (which can use environment variables to get the values of the claims from a Kubernetes config map or secret):

```
<jwtBuilder id="myBuilder" keyStoreRef="defaultKeyStore"
keyAlias="default" issuer="${env.JWT_ISSUER}"
audiences="${env.JWT_AUDIENCE}" />
```

Then we put that in the Authorization header of the HTTP connection (called "conn" here), as such:

```
conn.setRequestProperty("Authorization", "Bearer "+
jwt.getRawToken());
```

And with that, we've sent the JWT that the downstream microservice, like Portfolio, will use to determine whether to allow the request. Note that each microservice will need to propagate this too. Also, Liberty 18.0.0.2 (in its new `jwtSso-1.0` feature) added support for the JWT being in a cookie, instead of an http header field; with that, it is easier to unit test APIs from a browser, which auto-propagates cookies on http/https requests.

Now let's look at how you configure a Java-based microservice to require a specific JWT in the request. The good news is that you don't have to do anything in your Java code. You just need to put the following in your web.xml:

```
<login-config>
    <auth-method>MP-JWT</auth-method>
    <realm-name>MP-JWT</realm-name>
</login-config>
```

Alternatively, you could do this via an annotation in your Java code if you prefer, instead of the web.xml deployment descriptor approach. You would just import `org.eclipse.microprofile.auth.LoginConfig`, and then add the following annotation to your JAX-RS class:

```
@LoginConfig(authMethod = "MP-JWT", realmName = "MP-JWT").
```

Using the annotation will enable you to have multiple JAX-RS classes in your war file with different settings. For example, you may want to require a JWT for your APIs, but you would NOT want to require a JWT for your readiness or liveness probes (as Kubernetes would not know how to create/populate/sign this and add it to the header, causing the pod to never pass a readiness check and thus be never be ready to accept work).

Then you need to add the `mpJwt-1.1` feature (or the `microProfile-2.0` convenience feature, that gives you all the MP 2.0 features) to your server.xml, and add a stanza like this:

```
    <mpJwt id="myMpJwt" audiences="${env.JWT_AUDIENCE}"
issuer="${env.JWT_ISSUER}" keyName="default"
ignoreApplicationAuthMethod="false"/>
```

Note again that we are using environment variables from Kube secrets, rather than hard-coding the claim values in the server.xml. Also, note the `ignoreApplicationAuthMethod="false"` - this means you don't want all apps in the

server to require a JWT, but rather just the ones with the `MP-JWT` auth-method in their web.xml or their `@LoginConfig` annotation (that way, endpoints like `/metrics`, `/health`, and `/openapi` still work even if no JWT is passed to them).

The above configuration of the `mpJwt` element in the server.xml is a Liberty specific way to specify the public key. A better vendor neutral way is to specify the following 3 properties: `mp.jwt.verify.publickey.location`: it is the relative path or full URL of the public key.

`mp.jwt.verify.publickey`: the full string of the embedded key material of the public key for the MP-JWT signer in PKCS8 PEM or JWK(S) format. If not found, the `mp.jwt.verify.publickey.location` will be used.

`mp.jwt.verify.issuer`: the expected `iss` claim value to validate against an MP-JWT.
The above properties can be specified in Kubernetes Config Map.

Authorization

Now that we've ensured that each microservice knows who is calling it (and doesn't let in people without valid credentials), we can look at specifying who is allowed to call what. By default, everything is callable by anyone that has successfully passed authentication. But you can lock down certain APIs to say they are only callable by members of a specified group. To demonstrate this, I defined two groups: StockTrader and StockViewer. Let's look again at how I defined the `basicRegistry` earlier (LDAP, of course, has its own way to define groups of users):

```
<basicRegistry id="basic" realm="BasicRegistry">
    <user name="admin" password="admin"/>
    <user name="stock" password="trader"/>
    <user name="read"  password="only"/>
    <user name="other" password="denied"/>
    <group name="StockTrader">
        <member name="admin"/>
        <member name="stock"/>
    </group>
    <group name="StockViewer">
        <member name="read"/>
    </group>
</basicRegistry>
```

You can see here that the "admin" and "stock" IDs are members of the StockTrader group, the "read" ID is a member of the StockViewer group, and the "other" ID isn't a member of either. The idea here is that people in the StockTrader group can do everything, people in the StockViewer group can view portfolios but not create/update/delete them, and people in neither group can't do anything (even though they are successfully authenticated – they aren't authorized).
Another way you can specify group membership is in the bindings for the web app. For example, I use the following stanza in my server.xml when I want to say that anyone that has successfully authenticated against IBMid should be a member of the StockTrader group:

```
    <webApplication id="TraderUI" name="TraderUI" location="TraderUI.war"
contextRoot="/trader">
        <application-bnd>
            <security-role id="StockTrader" name="StockTrader">
                <special-subject type="ALL_AUTHENTICATED_USERS" id="IBMid"/>
            </security-role>
        </application-bnd>
    </webApplication>
```

There are two ways to specify such authorization rules in your application. The first, and recommended, is to use the `javax.annotation.security.RolesAllowed` annotation on the JAX-RS class or method in question (if annotations exist at both the class and method level, the more restrictive takes precedence). For example, the `deletePortfolio` method will only allow members of the StockTrader group to call it:

```
@DELETE
@Path("/{owner}")
@Produces("application/json")
@RolesAllowed({"StockTrader"})
public JsonObject deletePortfolio(@PathParam("owner") String
owner) throws SQLException {
```

So if "admin" or "stock" called this API, they'd be allowed in, but if "read" or "other" called it, they'd get a 403 (not authorized) error. For APIs that want to allow multiple roles, they list those as a comma-separated list, like this:

```
@GET
@Path("/")
@Produces("application/json")
@RolesAllowed({"StockTrader", "StockViewer"})
public JsonArray getPortfolios(@Context SecurityContext
securityContext) throws SQLException {

securityContext.getUserPrincipal() //to retrieve the
org.eclipse.microprofile.jwt.JsonWebToken
securityContext.isUserInRole("StockTrader") // to find out the
incoming request's access right.
```

So for this one, "admin", "stock" and "read" would all be allowed in, but "other" wouldn't, since it's not a member of either role that's allowed for this API.
The other way to specify this is the old-fashioned web.xml approach. For example, you first define your roles:

```
<security-role>
   <description>Group with read-only access to stock
portfolios</description>
   <role-name>StockViewer</role-name>
</security-role>
<security-role>
```

```
   <description>Group with full access to stock
portfolios</description>
   <role-name>StockTrader</role-name>
</security-role>
```

Then you define which methods are mapped to which roles. First, here's how we've defined all the GET operations, to allow either role:

```
<security-constraint>
   <display-name>Portfolio read-only security</display-name>
   <web-resource-collection>
      <web-resource-name>Portfolio read-only methods</web-
resource-name>
      <description>Applies to all paths under the context root
(this service specifies the portfolio as a path
param)</description>
      <url-pattern>/*</url-pattern>
      <http-method>GET</http-method>
   </web-resource-collection>
   <auth-constraint>
      <description>Roles allowed to access read-only operations
on portfolios</description>
      <role-name>StockViewer</role-name>
      <role-name>StockTrader</role-name>
   </auth-constraint>
</security-constraint>
```

Then we defined the rule for the other verbs, saying only the one role is allowed:

```
<security-constraint>
   <display-name>Portfolio read/write security</display-name>
   <web-resource-collection>
      <web-resource-name>Portfolio modification methods</web-
resource-name>
      <description>Applies to all paths under the context root
(this
service specifies the portfolio as a path param)
      </description>
      <url-pattern>/*</url-pattern>
      <http-method>POST</http-method>
      <http-method>PUT</http-method>
      <http-method>DELETE</http-method>
   </web-resource-collection>
   <auth-constraint>
      <description>Roles allowed to access read/write operations
on portfolios</description>
      <role-name>StockTrader</role-name>
   </auth-constraint>
</security-constraint>
```

```
<deny-uncovered-http-methods />
```

Note the last line says that any http methods not listed will be automatically denied to everyone.

Of these two approaches – the JAX-RS annotations or the web.xml deployment descriptor, the annotations would be the recommended approach where possible.

Now let's look at how you check in your code whether to offer certain capabilities based on the logged in user and their role/group membership. If you are working with servlets or JSPs, you can use the `isUserInRole()` method on `HttpServletRequest` to check. For example, I do this in my `Summary` servlet, to determine which radio buttons to display based on the logged in user:

```
boolean editor = request.isUserInRole("StockTrader");
if (editor) {
   writer.append("      <input type=\"radio\" name=\"action\"
value=\""+CREATE+"\"> Create a new portfolio<br>");
}
writer.append("      <input type=\"radio\" name=\"action\"
value=\""+RETRIEVE+"\" checked> Retrieve selected
portfolio<br>");
if (editor) {
  writer.append("      <input type=\"radio\" name=\"action\"
value=\""+UPDATE+"\"> Update selected portfolio (add
stock)<br>");
  writer.append("      <input type=\"radio\" name=\"action\"
value=\""+DELETE+"\"> Delete selected portfolio<br>");
}
```

So, a member of the StockViewer group would only get the Retrieve radio button, whereas a member of the StockTrader group would get all 4 CRUD radio buttons.

Additional Security Considerations

Having to deal with SSL certificates is just a reality of using https – which almost all internet-based services require these days. For example, my Stock Trader application talks to IEX Trading to get free stock quotes. But the call to its https URL will fail with an SSL exception that basically says that the SSL certificate for the URL in question is not in the Liberty keystore, and thus isn't trusted. We can fix that by using a couple of command-line tools called `openssl` and `keytool`:

```
openssl s_client -connect api.iextrading.com:443 -showcerts >
iextrading.cert
keytool -importcert -alias iextrading -storetype JKS -storepass
passw0rd –keystore ./key.jks -file ./iextrading.cert
```

The first command reaches out to the domain in question and asks it to present its SSL certificate (just like browsers do, when they give you that "do you want to trust this site" dialog; we're running headless, so have to do that a different way), and redirects that to a file named `iextrading.cert` in this case. The second command adds that certificate to the Liberty

keystore (named `key.jks` by default). I then have the following line in my `Dockerfile` that makes sure this keystore gets copied into the Docker image for my microservice:

```
COPY key.jks /config/resources/security/key.jks
```

And lastly, I have the following stanza in my server.xml that points the server to this keystore:

```
    <keyStore id="defaultKeyStore" password="passw0rd"
type="jks" location="/config/resources/security/key.jks" />
```

Today, I just have this keystore (with the SSL certs I need already added) checked into GitHub. Perhaps a better approach would be to have my Maven pom.xml construct this keystore during the build. That way, I wouldn't have to update GitHub every time some domain updates their SSL certificate (they have expiration dates built in, and won't work once that date arrives). The other thing in a keystore, besides public SSL certificates, is your public or private keys. Obviously, you don't want to compromise your private keys, so those should probably be in some kind of secure key vault. I haven't gone there yet with Stock Trader, so that too is in the keystore currently.

Programming Model Concept – Coordinating Service Interactions

Introduction

Microservices best practices requires that discrete microservices remain independent of one another and the notion of an externally coordinated transaction is anathema. A unit of work context, in any of the traditional senses, should not span multiple microservices. Or to put it another way, a unit of work boundary might reasonably be considered a constraint on the granularity of a microservice such that the scope of a UOW does not extend beyond a single microservice. This is essentially the thinking behind the notion of the *Aggregate*: https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-1-richardson

There are a number of issues for considering how transactions might appropriately be applied across multiple microservices (or aggregates) but there is no ubiquitous industry approach to this yet. A naïve approach might be to consider distributing transactions across REST HTTP but this would break any generally agreed definition of a microservices architecture; we need something more invasive in the application programming model that delivers an eventually-consistent outcome without an external coordinator – potentially based on event sourcing.

Future:
There is a proposal being discussed within the MicroProfile community to introduce a feature to help in this space, called Long Running Activities, or LRA for short, and it's at https://github.com/eclipse/microprofile-lra. This is an app-centric compensation-based approach and is a spec is still under development. It has many of the same issues as ACID transactions when distributed across multiple microservices. The full (but still evolving) spec is here.

A more forward-looking approach to data access requires greater disruption to application design and is a direction we are trying to taking the MicroProfile LRA discussion towards – using CQRS. This favors an event-driven approach to data access in which data consumers are subscribers to domain-specific views on the data model and updates to the data are applied through commands that trigger events for subscribers. This approach is described below. It is followed by a look at more traditional coordination approaches that are ready today, and familiar to many with experience writing enterprise java applications. .

CQRS

This is a forward-looking capability that exists more as a pattern today than a framework, but around which we can construct a framework or consume an existing one as part of our application container (e.g. Liberty) runtime.

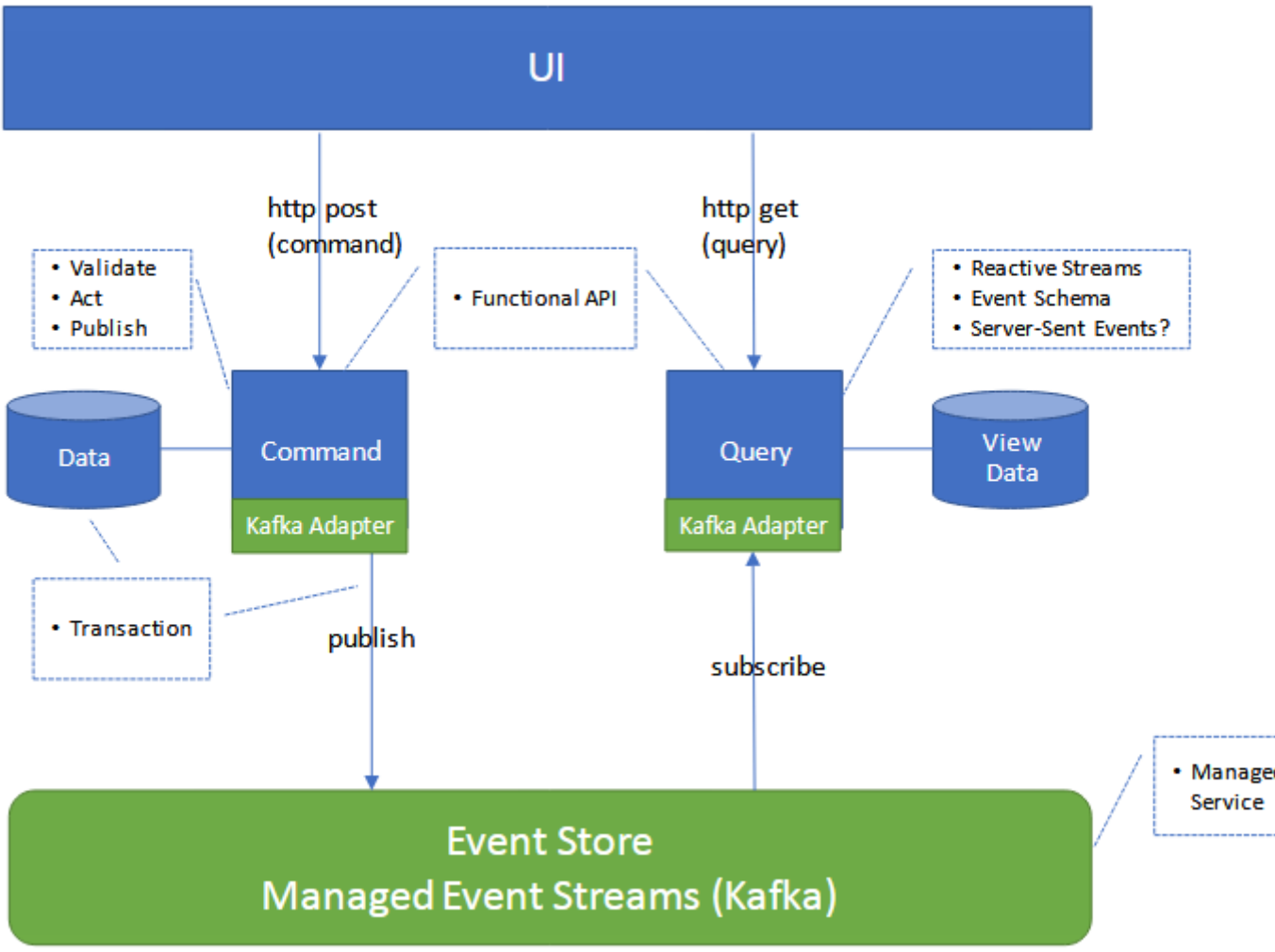
Command Query Responsibility Segregation (CQRS) is a pattern that has been around a long time and is widely recognized as a modern way to design applications that interact with data in a scalable and consistent fashion - but with a different way of thinking about what "consistent" means that requires applications be designed in a different from traditional transactions

applications.

CQRS does not assume an Event Sourcing model for persistence but is often considered alongside it. Some basic microservices assumptions that motivate this discussion:

1. Microservices update data in a manner that needs to be persistent.
2. Microservices retrieve and query data in a way that needs to be performant
3. Both of the above are true for event-based systems in which microservices subscribe to data-related events rather than "push" contextual commands

CQRS patterns are ideally built upon some form of distributed event store as illustrated below:



The main two motivations for CQRS in Liberty/ICP are:

1. We need a strategic direction (in addition to the wholly separate accommodation of traditional transaction models) for interacting with data in a distributed and increasingly asynchronous microservices architecture
2. We also need an answer to the basic question we really do get from customers making forward-looking strategic platform decisions "how do you help me use CQRS beyond just describing a patterns".

We need a solution that is not limited to being Java-centric but we will lead with a Java solution that has a native Java programming model.

Architectural needs:

1. A Java API focused on CQRS that simplifies its use, along with runtime support In Liberty that provides this and pulls in platform components required to implement it and configure the resources required to deliver it with scale, security and availability.
2. An opinionated approach to preferred infrastructure that is "default" but can be replaced. An example is the use of Managed Events Streams (underpinned by Kafka) for the event bus in ICP.
3. A wider-than-IBM community view of the Java programming model. We are trying to guiding the LRA (long-running action) transactions discussion in MicroProfile in this direction at the moment.
4. Simple example that illustrate both getting started (e.g. and Open Liberty guide) and realistic end-to-end use (e.g an updated version of Stock Trader evolved to use CQRS as part of the Architecture Center).

Local Transactions

Let's look again at our A calls B scenario. While A can't coordinate a transaction that includes B, the work that B does can be transactional. For example, our `trader` UI calls the `portfolio` microservice to buy some stock, causing its updatePortfolio method to get called. Within this method, I use JDBC to make a couple of updates – first to the Stock table, inserting a row for the newly purchased stock (or updating a row if they already owned some of that stock), then I update the row in the Portfolio table for that portfolio, to update the total portfolio amount and the loyalty level.
Since I'm using a transactional resource here (DB2, generally), I have the ACID properties, guaranteeing that I won't have, for example, the change to the Stock table happen but the change to the Portfolio table not. I'm either going to get both, or neither, because of how JDBC works. Furthermore, if an exception happened in between the calls to the Stock and Portfolio tables, the local transaction would automatically roll back, ensuring neither change happened. That's a good thing.
So while I don't have a *distributed* transaction here spanning from A to B, I do have the assurance that when I call B, whatever it does either fully happens, or doesn't happen at all. So that's something.
Note you need to use the annotations described below to make it so that the entire method is a single unit of work (otherwise, each call to JDBC would happen in a separately "just in time" started transaction and completed at the end of each specific SQL statement.

XA Transactions

The historical holy grail in the area of ACIDs transactions and distributed systems is XA Transactions – that is, the ability to do distributed two-phase commit across multiple resource managers. So while the above discussion holds that A can't coordinate work happening in B, we are able to do an XA transaction within B. We are in fact doing that in Portfolio: when you purchase a stock, we start a transaction, send a couple of SQL calls to JDBC as described above, and if the loyalty level changes, we also talk to JMS, posting a message about that change. Since both JDBC (backed by DB2 in this case) and JMS (backed by MQ in this case) support XA transactions, we actually are able to get Liberty to drive that distributed transaction with both DB2 and MQ enrolled in the transaction. So if we do our JDBC stuff, but find we're unable to do our JMS stuff for whatever reason, we have the ability to roll back the overall transaction. So we actually are getting the ACID properties here across all of the resources involved in B getting its job done. So transactions aren't in fact dead in this modern microservices world! They are just not distributed across microservices.

Annotations for Transactions

If you wish to scope multiple calls to a transactional resource manager under the same transaction, you need to use the JTA annotations in your code so the app container can take care of transaction management. Otherwise, a transaction will be automatically started for each individual call – meaning that if something goes wrong, you'd not be able to roll back everything to get to a consistent state. To use the annotations, you first need the following import statements:

```
import javax.transaction.Transactional;
import javax.transaction.Transactional.TxType;
import javax.enterprise.context.RequestScoped;
```

You also need to add a CDI "bean-defining" annotation to your class, such as `@RequestScoped`. You would already have that if using mpConfig or mpRestClient:

```
@ApplicationPath("/")
@Path("/")
@RequestScoped //enable interceptors like @Transactional (note you need a WEB-INF/beans.xml in your war)
/** This version stores the Portfolios via JDBC to DB2 (or whatever JDBC provider is defined in your server.xml). */
public class Portfolio extends Application {
```

Like we saw with several of the other MicroProfile features (like mpConfig, mpHealth and mpRestClient), you need a beans.xml in the WEB-INF directory of your war file.
Then you just need to put the `@Transactional` annotation on the method you want to run everything within it under a single transaction. Here we are using the `TxType.REQUIRED` parameter, which means that if a transaction is already in effect, just use that, otherwise it will automatically start one, run the method, and commit it (or do a rollback if the method failed with an exception). There are five other options you could use if your business case calls for them (matching what many of us remember from the EJB spec): `MANDATORY`, `NEVER`, `NOT_SUPPORTED`, `REQUIRED`, `REQUIRES_NEW`, and `SUPPORTS` (see the JavaDocs for this enum at https://docs.oracle.com/javaee/7/api/javax/transaction/Transactional.TxType.html).

```
    @PUT
    @Path("/{owner}")
    @Produces("application/json")
    @Transactional(TxType.REQUIRED) //two-phase commit (XA)
across JDBC and JMS
    @RolesAllowed({"StockTrader"})
    public JsonObject updatePortfolio(@PathParam("owner") String
owner, @QueryParam("symbol") String symbol,
@QueryParam("shares") int shares, @Context HttpServletRequest
request) throws IOException, SQLException {
```

Note that you use the same annotations whether you just want multiple calls to the same transactional resource (like updating both the Stock and Portfolio tables) to happen in a "local" transaction, or if you want to work with multiple transactional resource managers (like JDBC and JMS) in an XA transaction.
Finally, note that if, rather than having the managedBeans stuff wrap the entire method in a transaction, instead you really want to take control of exactly when the transaction begins and commits (or rolls back), you could do an old-fashioned JNDI lookup of `"java:comp/UserTranaction"` and do so yourself (as code you might be lifting-and-shifting might already have been doing, since this has been part of Java EE for many years).

Istio

Getting Started

As introduced earlier, Istio is an open source framework which works with Kubernetes to provide features for managing, securing polyglot microservices. More information can be found on the istio.org website, here: `https://istio.io/docs/concepts/what-is-istio`. Features and techniques described here are applicable independent of which programming language or framework is used to capture business logic in a microservice. This section of the programming model guide provides an elaboration of Istio.

Before using any of the Istio features as part of the cloud programming model, it is necessary to verify that Istio is installed in the target Kubernetes environment. To verify if Istio is installed, in a terminal session logged into the IBM Cloud Private environment:

```
kubectl get pods -n istio-system
```

The pods for Istio Pilot, Sidecar, Telemetry, Gateways and others should be listed. If Istio is not installed, refer to Appendix C – Istio Installation for guidance regarding not only the Istio installation, but IBM Cloud Private considerations.

The following examples also require that the `istioctl` command line client has also been installed:

```
istioctl version
```

If the `istioctl` command line client has not been installed, refer to Appendix C for Istio client installation instructions.

Enabling your application for Istio

Before getting started with Istio sidecar injection, make sure that the istio-user role based authorization is established for your target namespace:

```
cat <<EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: istio-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: ibm-privileged-clusterrole
subjects:
- kind: ServiceAccount
  name: default
  namespace: stock-trader
EOF
```

If ICP 3.1.0 is hosting Istio, change `ibm-privileged-clusterrole` to privileged in the example above.

Automatic sidecar injection for a given namespace by adding the `istio-injection=enabled` label to it. For example:

```
kubectl label namespace stock-trader istio-injection=enabled --overwrite
```

Any deployment to the `stock-trader` namespace will now automatically get an Istio sidecar in each pod. Automatic sidecar injection can be useful with CI/CD pipeline deployments, eliminating the need to put Istio-specific steps into the Jenkinsfile.

Manual sidecar injection can be useful during development and maybe necessary. The `stock-trader` example has containers, like DB2 and MQ that are not Istio sidecar tolerant. Manual sidecar injection requires application yaml be transformed for sidecars by the `istioctl` command. The `stock-trader` example at github provides an all-in-one yaml (a single yaml file with the configuration for each microservice concatenated together), called `stock-trader.yaml` located here: https://github.com/IBMStockTrader/stocktrader/blob/master/stock-trader.yaml)

```
istioctl kube-inject -f stock-trader.yaml -o stock-trader-istio.yaml
```

The "`-f`" parameter specifies the application input yaml file "`-o`" parameter specifies the name of the istio injected output file. Then just deploy your app, using that updated yaml, like usual:

```
kubectl apply -f stock-trader-istio.yaml -n stock-trader
```

After the application, each pod with a sidecar says "`2/2`", meaning there are two Docker containers per pod: one for the app, and one (named "`istio-proxy`") for the Istio sidecar.

```
Johns-MacBook-Pro-8:StockTrader jalcorn$ kubectl get pods -n stock-trader
NAME                                                  READY     STATUS    RESTARTS   AGE
db2trader2-ibm-db2oltp-dev-0                          1/1       Running   0          3d
looper-79f79cd4db-2wpkh                               2/2       Running   0          7m
messaging-6ffcb9c597-6pbf6                            2/2       Running   0          7m
mqtrader1-mqtrader1-0                                 1/1       Running   1          8d
mqtrader2-mqtrader2-0                                 1/1       Running   0          3d
notification-slack-c8496c7df-vcgvz                    2/2       Running   0          7m
notification-twitter-7656cf9978-ccmxr                 2/2       Running   0          7m
odmtrader1-ibm-odm-dev-b5b9b446c-tcsz7                1/1       Running   1          7d
odmtrader2-ibm-odm-dev-54665d97cf-cvtsh               1/1       Running   0          3d
portfolio-7bd5c9957f-vcmx6                            2/2       Running   0          7m
redistrader1-ibm-redis-ha-dev-sentinel-7458b66ccb-jfzqh   1/1       Running   3          8d
redistrader1-ibm-redis-ha-dev-sentinel-7458b66ccb-sg7rk   1/1       Running   3          8d
redistrader1-ibm-redis-ha-dev-sentinel-7458b66ccb-xfh47   1/1       Running   3          8d
redistrader1-ibm-redis-ha-dev-server-5bdb6b76b5-c2p72     1/1       Running   4          8d
redistrader1-ibm-redis-ha-dev-server-5bdb6b76b5-fc5nb     1/1       Running   3          8d
redistrader1-ibm-redis-ha-dev-server-5bdb6b76b5-kj6x7     1/1       Running   3          8d
stock-quote-86f4ddbb55-8c5hz                          2/2       Running   0          7m
trader-58b87989cc-p56zx                               2/2       Running   0          7m
tradr-7ff464966f-8dzgp                                2/2       Running   0          7m
```

Since there are two distinct container types in sidecar injected pods, `kubectl` may require the container name to be specified which disambiguates the container to perform the action. As an example, to display the logs of the portfolio container, the command would look similar to:

```
kubectl logs portfolio-7bd5c9957f-vcmx6 -c portfolio -n stock-trader
```

Egress rules

Prior to Istio 1.1, the default behavior of Istio is to prevent network egress for sidecar injected applications. Each desired egress destination must be whitelisted. For an https site, you need to create two new instances of Istio CRDs: a `ServiceEntry` and a `VirtualService` (an http site only needs the former). Attempts to reach destinations that are not whitelisted will fail, e.g. an SSLException is likely for https: requests.

For Stock Trader, we need connectivity to 5 external https sites:

1. IBMid (unless using the `basicRegistry` version of `trader` for login)
2. API Connect (to get stock quotes – behind the scenes, that API ends up calling IEX)
3. Watson (for tone analysis)
4. IBM Cloud Functions (to call our "serverless" action sequence that posts to Slack)
5. Twitter (for our tweet, obviously)

Following is an example of the yaml for defining the two whitelist CRDs:

```
cat <<EOF | kubectl apply -f -\
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: apic
spec:
  hosts:
  - api.us.apiconnect.ibmcloud.com
  ports:
  - number: 443
    name: https
    protocol: HTTPS
  resolution: DNS
  location: MESH_EXTERNAL
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: apic
spec:
  hosts:
  - api.us.apiconnect.ibmcloud.com
  tls:
  - match:
    - port: 443
      sni_hosts:
      - api.us.apiconnect.ibmcloud.com
    route:
    - destination:
        host: api.us.apiconnect.ibmcloud.com
        port:
          number: 443
      weight: 100
EOF
```

Note: The VirtualService and ServiceEntries can be found in the `stock-quote` repository, named: `apic-egress.yaml`.

Now that the `apic-egress` documents have been whitelisted, the `stock-quote` pod can successfully reach the target API connect service.

The Istio host names in the VirtualService and ServiceEntries are literal: `google.com` is a different destination than `www.google.com`. If `www.google.com` is specified, any egress attempts to `google.com` will be blocked.

Debugging Tip: Add `curl` to egress application containers

When verifying or diagnosing egress issues, it can be useful to leverage the `curl` command to test accessibility to the destination. `curl` is not installed in the default Liberty image, but can be added to the Liberty `Dockerfile` (for each Stock Trader microservice) by adding the following statements:

```
RUN apt-get update
RUN apt-get install curl -y
```

Unfortunately, the `curl` command likely needs to be added by modifying the `Dockerfile`, rather than doing it when after the pod is deployed. Post deployment there are two potential problems: 1) unauthorized to use `apt-get`, and 2) no egress rule defined to execute `apt-get`.

***Note**: `apt-get` is used for Ubuntu distributions, leverage `yum` for Red Hat distributions.*

Having installed `curl`, ensure the container can successfully access the service endpoint:

```
kubectl exec -it stock-quote-54b5c6ff9f-5gdtk -c stock-quote -n stock-trader -- curl
https://api.us.apiconnect.ibmcloud.com/jalcornusibmcom-dev/sb/stocks/IBM
```

By leveraging the `curl` command in the application container, Istio will constrain its egress exactly the same way as it constrains the application itself. This technique can simplify the diagnosis of the egress behavior by isolating Istio configuration from the microservice business logic or from Liberty configuration issues, like an SSL cert problem in the keystore.

Fault Tolerance

Having verified that the application is working in the presence of Istio, we can start to see some of the benefits it brings to the table. It can basically be a "traffic cop", allowing you to specify policies that will be enforced when a microservice in your application gets invoked over http. These policies include **Retries**, **Timeouts**, **Circuit Breakers**, **Bulkheads**, and **Rate Limits**.

I would also highly recommend Emily's recent article comparing Istio fault tolerance to MicroProfile fault tolerance, which covers many of the same topics as below (we just have a Stock Trader spin on our examples below):
https://www.eclipse.org/community/eclipse_newsletter/2018/september/MicroProfile_istio.php

We'll use the `stock-quote` microservice as our example (you can find all of its Istio rules at `https://github.com/IBMStockTrader/stock-quote/tree/master/istio`). Note that it has 3 special stock ticker symbols that do something other than calling API Connect to get a stock quote: TEST, SLOW, and FAIL. TEST always returns a hard-coded quote of $123.45 with today's date (great for testing when you aren't sure about your egress policies, or if API Connect or IEX (or your internet connection) are down. SLOW sleeps for a minute, then returns what TEST does. And FAIL just throws a `RuntimeException`, resulting in a 500 error.

Retries

One of the easiest ways to "tolerate" a fault is to simply try the call again. Maybe there was some momentary glitch, and stuff might work if you try it again later. Rather than altering your application logic to code in such retry logic, you can let Istio do this for you. Just define a **Retry** policy, and that will control how many times the sidecar tries calling your container before giving up, and how long to wait between attempts. For example, we could define a policy like the following for our `stock-quote` microservice:

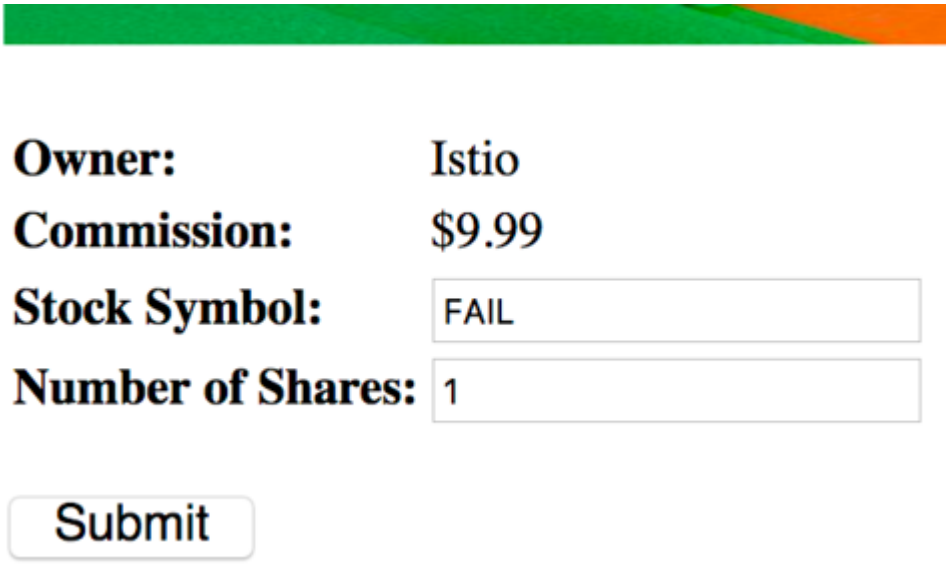
```
apiVersion: networking.istio.io/v1alpha3
```

```
kind: VirtualService
metadata:
  name: stock-quote-retries
spec:
  hosts:
    - stock-quote-service
  http:
  - route:
    - destination:
        host: stock-quote-service
    retries:
      attempts: 3
      perTryTimeout: 5s
```

As you can see, this says to retry the call to get the stock quote up to 3 times, with a 5 second delay between each attempt. Then we just apply that like we would any other yaml. We saved the above to a `retry-policy.yaml` in the `stock-quote` repository, then ran:

```
kubectl apply -f retry-policy.yaml -n stock-trader
```

We can now call our FAIL symbol to get some failing requests that will thus get retried:



With the above retry policy in effect, we see the following appear in the logs for our `stock-quote` pod when we click Submit:

```
[INFO    ] Initialization complete!
[INFO    ] Throwing a RuntimeException for symbol FAIL!
[INFO    ] Initialization complete!
[INFO    ] Throwing a RuntimeException for symbol FAIL!
[INFO    ] Initialization complete!
```

```
[INFO    ] Throwing a RuntimeException for symbol FAIL!
[INFO    ] Initialization complete!
[INFO    ] Throwing a RuntimeException for symbol FAIL!
```

Note that it showed up four times; once for the original request, and then once more for each of the three retries. This means the actual call from `portfolio` to `stock-quote` took just over 15 seconds, given the 5 second delay we specified between each attempt. So be careful with your *Retry* policies that you don't violate your *Timeout* policy; more on that in the next section.

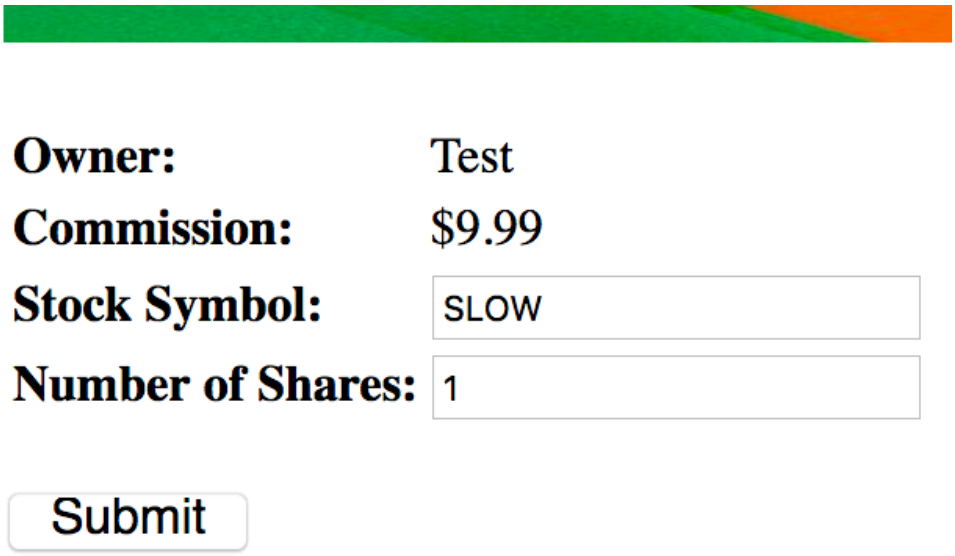
Timeouts

You can specify how long to wait on a REST request before giving up. The default is 15 seconds; any call that takes that long is assumed to be a failure (this can be another thing that mysteriously "breaks" with Istio enabled, if you'd been living with calls taking over 15 seconds prior to giving your app Istio sidecars).

However, in our above retry example, we saw how that would take just over 15 seconds before giving up, so someone waiting on all of that (like the UI waiting on the call to Portfolio, which calls Stock Quote for each stock in the portfolio) would see a failure, even if the final retry had actually worked. For that, or any other case where you want to change away from the 15 second default (like for our SLOW stock ticker symbol, that sleeps for a minute), you can define a **Timeout** policy, like this:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: stock-quote-timeout
spec:
  hosts:
    - stock-quote-service
  http:
  - route:
    - destination:
        host: stock-quote-service
    timeout: 90s
```

Obviously, this says to allow 90 seconds before giving up. We'll save the above to a `timeout-policy-yaml` in our `stock-quote` repository. Note that if you want both the *Retry* and *Timeout* policies on the same Kubernetes service, they actually need to be combined into the same yaml file; Istio will fail in bad ways if you try to define multiple `VirtualService` entries for the same `Service`. But first, before applying the *Retry* yaml, let's consider what happens without it:



What will happen is that the client will get back an error message, saying that the attempt to purchase the stock failed. But in reality, it probably did succeed, it just took so long to do so that the client had given up on it (the client giving up on waiting does not stop what the server was doing in response to the call from that client). So even though you got the failure message on attempting to buy the stock, you would in fact see the stock in your portfolio the next time you viewed it, which could be confusing (that's just the nature of network timeouts). Anyway, let's go ahead and apply our new timeout policy:

```
kubectl apply -f timeout-policy.yaml -n stock-trader
```

And now, when we buy the SLOW stock, it will work (after waiting in the browser for just over a minute), and you'll get no error message, and in fact will see the new stock immediately (because the app immediately redirects you to the `viewPortfolio` servlet when the REST call to purchase the stock returns).

Stock Portfolio for **Istio**:

Symbol	Shares	Price	Date Quoted	Total	Commission
GOOG	7	$1,087.71	2018-10-18	$7,613.97	$9.99
IBM	42	$130.25	2018-10-18	$5,470.50	$9.99
SLOW	1	$123.45	2018-10-18	$123.45	$8.99

Total Portfolio Value:	**$13,207.92**
Loyalty Level:	**BRONZE**
Account Balance:	**$3.05**
Total commissions paid:	**$46.95**
Free Trades Available:	**1**
Sentiment:	**Joy**

OK Submit Feedback

Note I had to put the same timeout policy on `portfolio-service`, so that the `trader` UI wouldn't timeout waiting on it (which was waiting on the `stock-quote-service`).

Circuit Breakers

We can also define policies that say "if X consecutive calls to my microservice have failed in the past Y <time units>, then stop routing traffic to it for Z <time units>". For example, we could define a **Circuit Breaker** policy that says "if 3 consecutive calls to `stock-quote` have failed in the past 5 seconds, then stop routing traffic to it for 5 minutes". This can provide a way to quickly short-circuit calls to a microservice that has some prerequisite that is temporarily unavailable.

For example, my `stock-quote` microservice depends on *API Connect*, and its implementation there depends on *IEX*; so if either of those were unavailable, I could define a policy that would just automatically block all traffic to `stock-quote` for a while (note that my `portfolio` microservice caches stock quotes in DB2 and falls back to using those cached values – which could be quite stale if that portfolio hasn't been accessed in a few days, and is why the quote date is shown in the UI so the user is aware – if it can't get a fresh one from `stock-quote`).

Let's define the following in a `circuit-breaker-policy.yaml` in the `stock-quote` repository, with the values described above:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
```

```
  name: stock-quote-circuit-breaker
spec:
  host: stock-quote-service
  trafficPolicy:
    outlierDetection:
      consecutiveErrors: 3
      interval: 5s
      baseEjectionTime: 5m
      maxEjectionPercent: 100
```

We then apply that policy as seen in earlier examples:

```
kubectl apply –f circuit-breaker-policy.yaml -n stock-trader
```

Then, once the circuit breaker has been tripped, any attempted traffic to `stock-quote` for the next 5 minutes would immediately get back a "503 - Service Unavailable". Of course, if the needed prereqs still aren't back after 5 minutes, the circuit breaker will get tripped again after 3 failures, and that loop will continue until the microservice starts returning success (an http 200 return code).

Bulkheads

Sometimes you want to limit how many concurrent calls to allow through your microservice. One use case would be to protect yourself against Distributed Denial of Service (DDoS) attacks (which of course is more likely if your microservice is out on the public internet, than if it's on a private cloud behind your corporate firewall).

We can enforce such protection for your microservice via a **Bulkhead** policy in Istio, such as this one that we'll save to a file named `bulkhead-policy.yaml` in the `stock-quote` repository:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: stock-quote-bulkhead
spec:
  host: stock-quote-service
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 10
```

As before, we'd apply this bulkhead policy via the following:

```
kubectl apply –f bulkhead-policy.yaml -n stock-trader
```

With this in place, we are limiting the maximum number of concurrent calls to our `stock-quote` service to 10; any beyond that will get a "503 – Service Unavailable" response (which would then get retried after your `perTryTimeout`, if you had a retry policy, of course).

Rate Limits

You can also define **Rate Limits** with Istio. These are similar to bulkheads, but are defined over a time window, rather than only limiting concurrent calls being processed at the same instant.

For example, we could say we don't want to allow more than 1 tweet per second (or 100 per hour, or whatever). This is actually a real-world example, as Twitter has automatically locked out our `@IBMStockTrader` account in the past when we have run stress tests (like my Looper client) against Stock Trader that have resulted in several tweets per second over a span of a few minutes.

<Need example. Note that rate limits require us to introduce the concept of Mixer Adapters, like `memquota` or `redisquota`; there's more to them than the earlier policies.>

Request Routing

Istio will let you dynamically define, at runtime, what version of a microservice will be used, under what circumstances. For example, in our Stock Trader sample, we have two different versions of our `notification` microservice: `notification-slack`, that posts to a Slack channel (`stock-trader-messages` on `ibm-cloud.slack.com`), and `notification-twitter`, that sends a tweet (under the `@IBMStockTrader` account on Twitter, with an `#IBMStockTrader` hashtag). If Istio isn't installed, then you want to only install one or the other of these two versions of this microservice.

In Kubernetes terminology, there's one *Service*, named `notification-service`, and clients of this service, like the `onMessage()` method of our `messaging` microservice (see the *JMS – Receiving a Message* section for details), invoke that service by hitting the Kube DNS name for that service. For example, they hit `http://notification-service:9080/notification` (the `/notification` at the end is the context-root for that microservice).

So, the caller is unaware there are multiple implementations; they just ask to make a call to `notification-service`, and Istio takes care of things from there on, consulting the registered routing rules to determine which implementation to call at runtime.

To continue with Kubernetes terminology, there are two *Deployments* (each with one or more *Pods*): `notification-slack` and `notification-twitter`. Each deployment has Kubernetes *Labels* on it saying the "`app`" is "`notification`" and the "`version`" is either "`slack`" or "`twitter`" (we also specify the "`solution`" is "`stock-trader`" on each of the *Stock Trader* deployments, for use in log filtering in Kibana and in "application view" in Prism). For example, the `notification-twitter` deployment specifies this (and `notification-slack` is the same, except for the last line):

```
spec:
```

```
  replicas: 1
  template:
    metadata:
      labels:
        app: notification
        solution: stock-trader
        version: twitter
```

So far, everything we've discussed is standard Kubernetes concepts, and is valid even if Istio isn't present in the environment. Note, however, that if Istio is not in place, it is important that you install only one of the two `notification` versions (otherwise, it is random/undefined which version will get called each time).

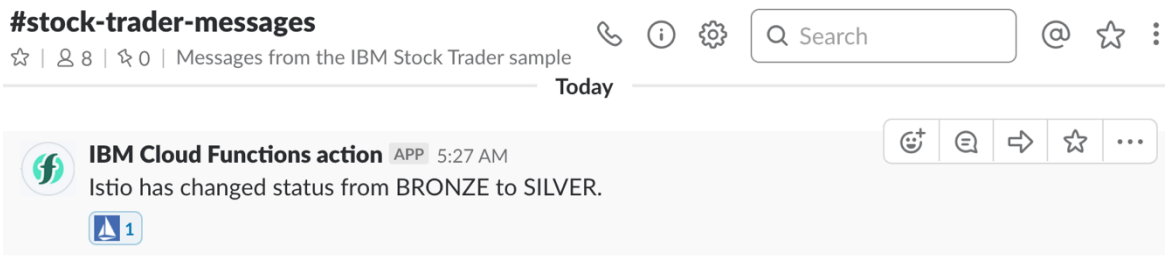
Now let's look at what we need to do in Istio to say which version is used when. We do that via routing rules. For example, we have a default routing rule saying, unless otherwise specified, all notification messages should be sent to our "slack" version. The routing rule for this looks like the following, and we'll save it to a file named `route-rule-notification-slack.yaml` in the `notification-slack` repository:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: notification-slack
spec:
  hosts:
  - notification-service
  http:
  - route:
    - destination:
        host: notification-service
        subset: slack
```

As before, we apply this route rule like any other Kubernetes yaml:

```
kubectl apply –f route-rule-notification-slack.yaml -n stock-
trader
```

With this rule in place, if I buy some stock for the "Istio" portfolio (enough to cause a change in its loyalty level), I will indeed get a notification message sent to our `#stock-trader-messages` channel on `ibm-cloud.slack.com`:



Now let's define another routing rule, that says a portfolio named "John" will send its notifications to "twitter" instead:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: notification-twitter
spec:
  hosts:
  - notification-service
  http:
  - match:
    - headers:
        owner:
          exact: John
    route:
    - destination:
        host: notification-service
        subset: twitter
```

We'll save that to a file named `route-rule-notification-twitter.yaml` in the `notification-twitter` repository, and run it like before:

```
kubectl apply -f route-rule-notification-twitter.yaml -n stock-trader
```

This rule says that if a request has an http header field named "`owner`" with a value of "`John`", then that request will get routed to the "`twitter`" version instead of the "`slack`" version:

We can also see which version of `notification` is getting used per message received by `messaging` by checking its log:

```
[INFO    ] Sending
{"id":"stock","owner":"Istio","old":"BRONZE","new":"SILVER"} to
http://notification-service:9080/notification
[INFO    ] Received the following response from the Notification
microservice: {"message":"Istio has changed status from BRONZE to
SILVER.","location":"Slack"}
[INFO    ] Sending {"id":"stock","owner":"John","old":"BASIC","new":"BRONZE"}
to http://notification-service:9080/notification
[INFO    ] Received the following response from the Notification
microservice: {"message":"On Friday, October 19, 2018 at 10:45 AM UTC, John
changed status from BASIC to BRONZE. #IBMStockTrader","location":"Twitter"}
```

It's worth pointing out that natively, the "`owner`" field is in the JSON in the http body, so I'm actually having to copy that field into the header to appease Istio. There is an issue (`https://github.com/istio/istio/issues/4460`) open requesting that Istio add support for routing rules being able to be based on the http body as well (note this would have performance implications, which is why it hasn't been implemented yet), but in the meantime, I need to define this extra header param in my `NotificationClient` *mpRestClient* interface:

```
public NotificationResult notify(@HeaderParam("Authorization")
String jwt, @HeaderParam("owner") String owner, LoyaltyChange
loyaltyChange);
```

And I need to pass the value of that header param when I call it, of course (note that `LoyaltyChange` and `NotificationResult` are *JSON-B* objects representing the http request body and the http response body, respectively):

```
NotificationResult result = notificationClient.notify("Bearer "+jwt,
owner, loyaltyChange);
```

Obviously, we could define additional routing rules to make portfolios with other owners get their notifications via Twitter as well.

One last thought: so far, I've discussed routing based on a plain-text header field added to the http request. But my colleague Ozzy has prototyped a cool plugin to Istio that will let you route based on the value of a claim (such as the logged in user or their group membership) in the JWT attached to the request – for more on this cool work, see his Git repo at `https://github.com/BarDweller/istio-content-based-routing`.

Infrastructure pieces with value-add UIs

When you install the Istio helm chart, you will get several subcharts installed that provide the infrastructure needed by Istio. Some of these are the plumbing that makes Istio work, like **Pilot**, **Mixer**, **Envoy**, and **Gateways**. Everything we've discussed so far depends upon on those core pieces of the Istio infrastructure.

Now let's take a look at the other open-source components that get installed by Istio, and briefly look at the user interfaces they provide. Those include **Jaeger**, **Prometheus**, **Grafana**, and **Kiali**. Some of those you may have seen in contexts outside of Istio (like ICP installs

Prometheus and Grafana itself; with Istio, you get a second copy of each of those, in the `istio-system` namespace).

Jaeger

In earlier pre-releases, Istio had integration with *Zipkin* (it's actually still there, but that subchart is not installed by default anymore). Instead, it has moved up to *Jaeger*, which is endorsed by the Cloud Native Computing Foundation (CNCF), as its tracing infrastructure, as described at https://istio.io/docs/tasks/telemetry/distributed-tracing/.

MicroProfile has a feature (`mpOpenTracing-1.1`) to get tracing to happen. But you can also get tracing, of each trip through the side car for your Docker container (basically, each call from one microservice to another), via Istio. This approach is preferred. The nice thing about this is that it is polyglot – so whether you are calling a service implemented in Java, Node, Swift, Go, or whatever, you will automatically get tracing for that call, as long as each microservice has had an Istio side car injected.

It's not as configurable (no annotations for getting additional things other than JAX-RS calls being traced, for example), but it all just happens magically for free, without you needing to do anything in your source code or your server.xml. The exception to that "no touch" rule is if you want the "A called B called C" scenario to be a single trace span, then you would need to ensure those http headers discussed earlier (in the MicroProfile tracing chapter) are propagated (otherwise, you'll get separate, uncorrelated spans for "A called B" and "B called C").

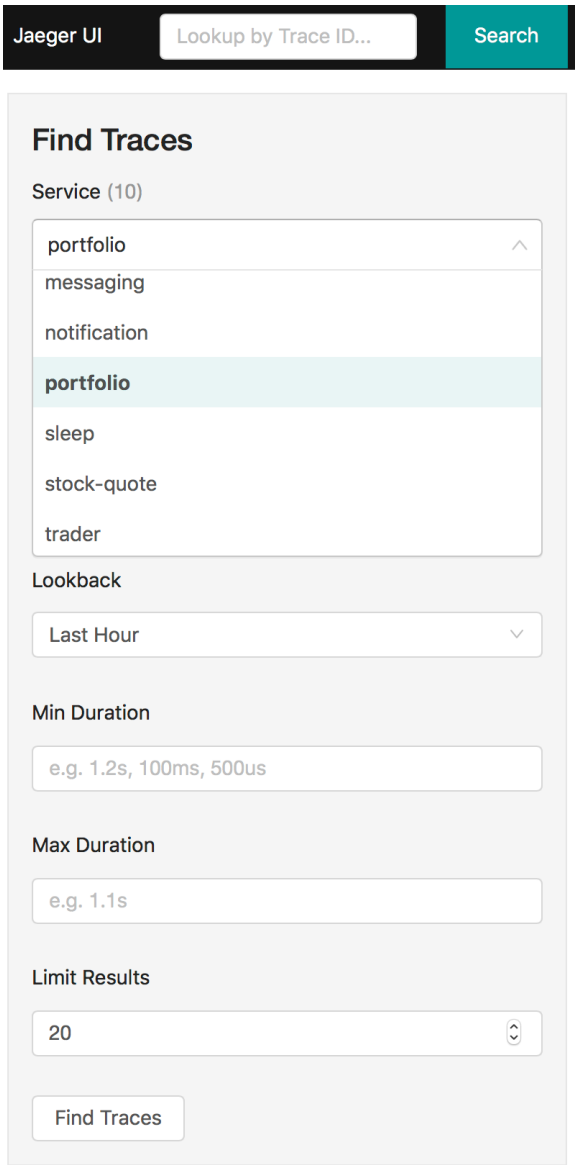
Here is where Istio and MicroProfile play well together. Simply by enabling the `mpOpenTracing-1.1` feature in your server.xml (or the `microProfile-2.0` feature, which enables all of the MP features), you will get the open tracing headers automatically injected into any HTTP requests made via the JAX-RS 2.0 client package (or via the *mpRestClient*, which is implemented atop that). So, while just Istio alone gives you something (the separate, uncorrelated trace span for each microservice to microservice call), and just using mpOpenTracing alone gives you something (if you enable extra tracer impls in your server.xml), you really get the best of both worlds by combining Istio and MicroProfile! That, in fact, is what I have done here with Stock Trader.

The actual services for Jaeger are `jaeger-agent`, `jaeger-collector`, and `jaeger-query`, all in the `istio-system` namespace. You can launch the Jaeger UI by clicking the node port (that it named "query-http") for the `jaeger-query` service in the ICP console.

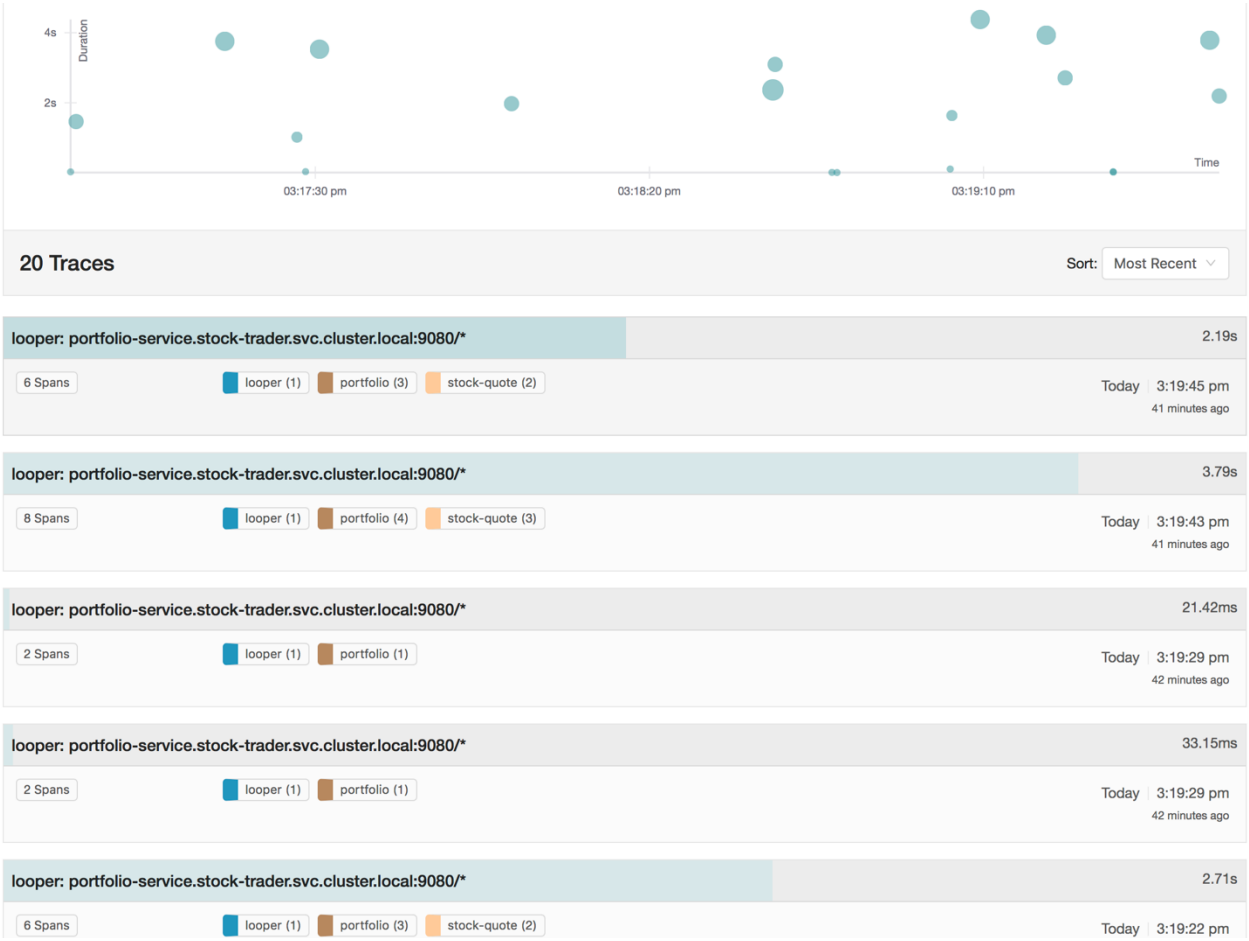
Service details

Type	Detail
Name	jaeger-query
Namespace	istio-system
Creation time	Apr 10th 2018 at 12:25 PM
Type	NodePort
Labels	app=jaeger,jaeger-infra=jaeger-service
Selector	jaeger-infra=jaeger-pod
IP	10.0.0.26
Port	query-http 80/TCP
Node port	query-http 32278/TCP
Session affinity	None

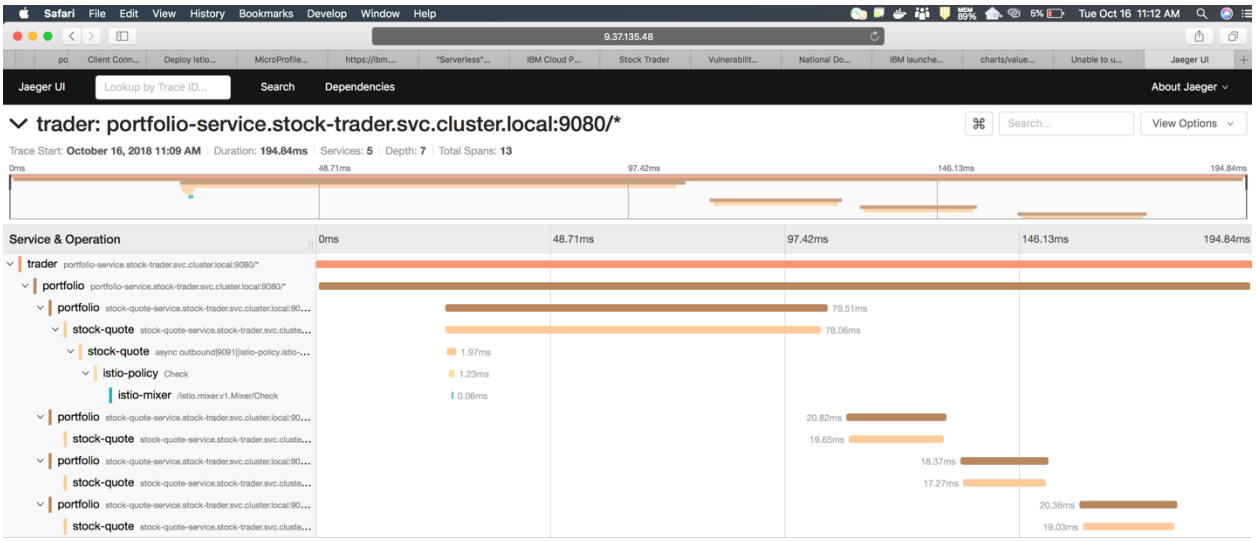
That will take you into the Jaeger UI, where the first thing you see will be a search dialog (I expanded the Service field to show the available services):



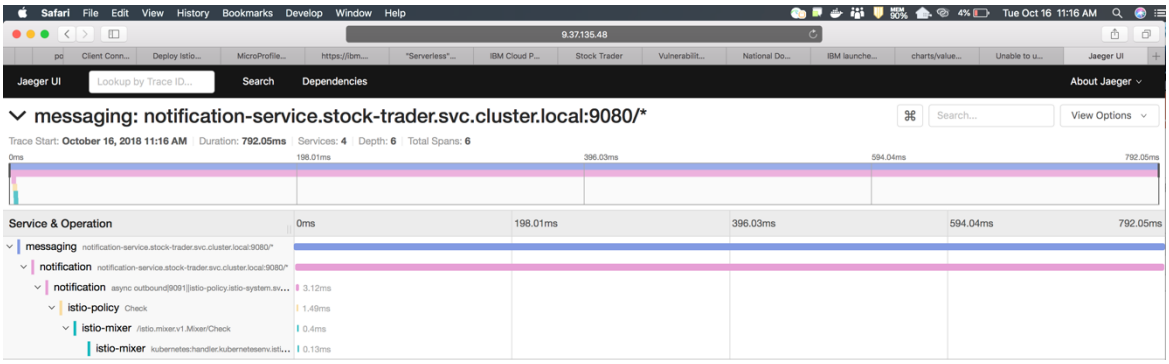
If you pick a service (I chose `portfolio`), you'll see the following table of trace spans:



If you click on one (pick one that also includes `stock-quote`; some don't, like a portfolio creation or deletion), you'll see the details of that trace span. Note that the above screenshot actually is making me aware of a problem; the calls that include a `stock-quote` are taking multiple seconds to return, meaning we're not getting a cache hit in Redis and thus are having to go outside of our private cloud, onto the internet (to API Connect, which goes on to IEX) for each stock in the portfolio in sequence. Apparently my Redis pod had gone bad, resulting in exceptions in my `stock-quote` pod's logs on each call to Redis. Killing that pod (thus causing Kubernetes to start a fresh one) appeared to fix the problem. It is kind of cool that the automatic trace record telemetry in Istio made me aware of such a problem! Anyway, let's look at a particular trace span:



Here you can see that `trader` called portfolio, which called `stock-quote`, along with timings (here I had Redis working again, so the whole thing across the multiple stocks in the portfolio was less than 200 milliseconds). Note you will only see REST (http or https) calls, so we aren't seeing the calls to DB2 or MQ, for example. And the trace spans won't be able to stretch across an asynchronous break, like we have between `portfolio` and `messaging`, where a trip through MQ happens (and `messaging` processes the MQ message on its own thread). But once we are back in `messaging`, we can see how it then calls on to `notification`:

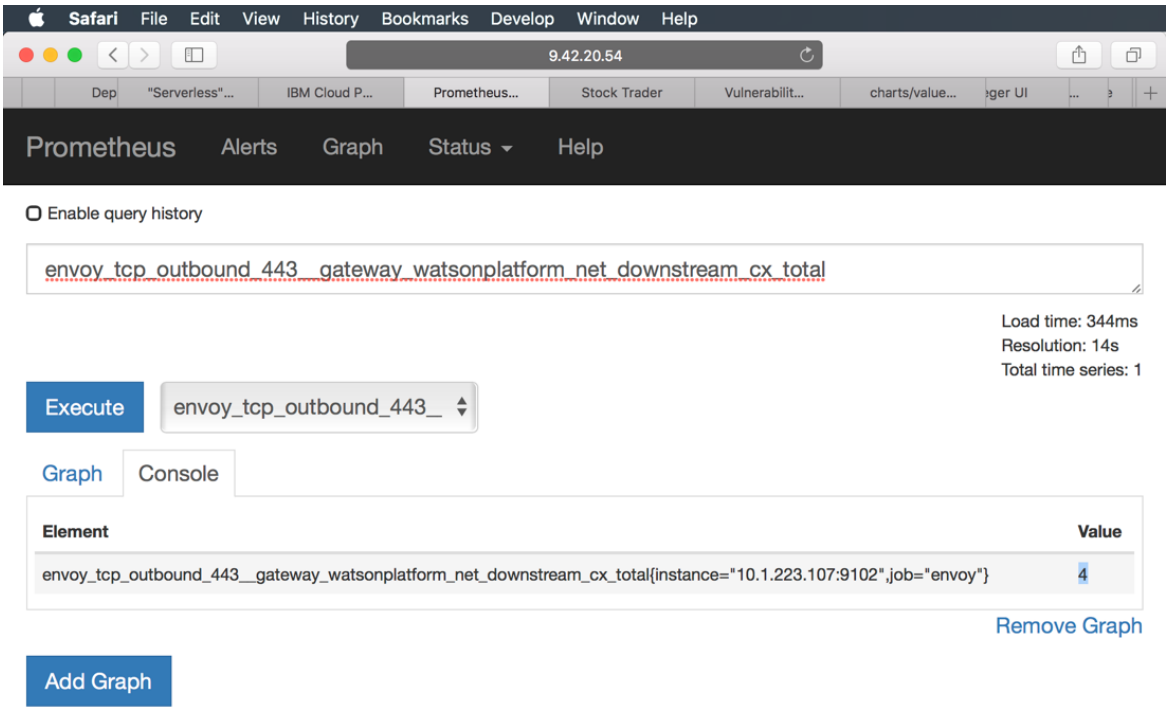


Note that it just knows it called a service named `notification`. It is unaware of the Istio routing magic going on to either the `notification-slack` or `notification-twitter` "version".

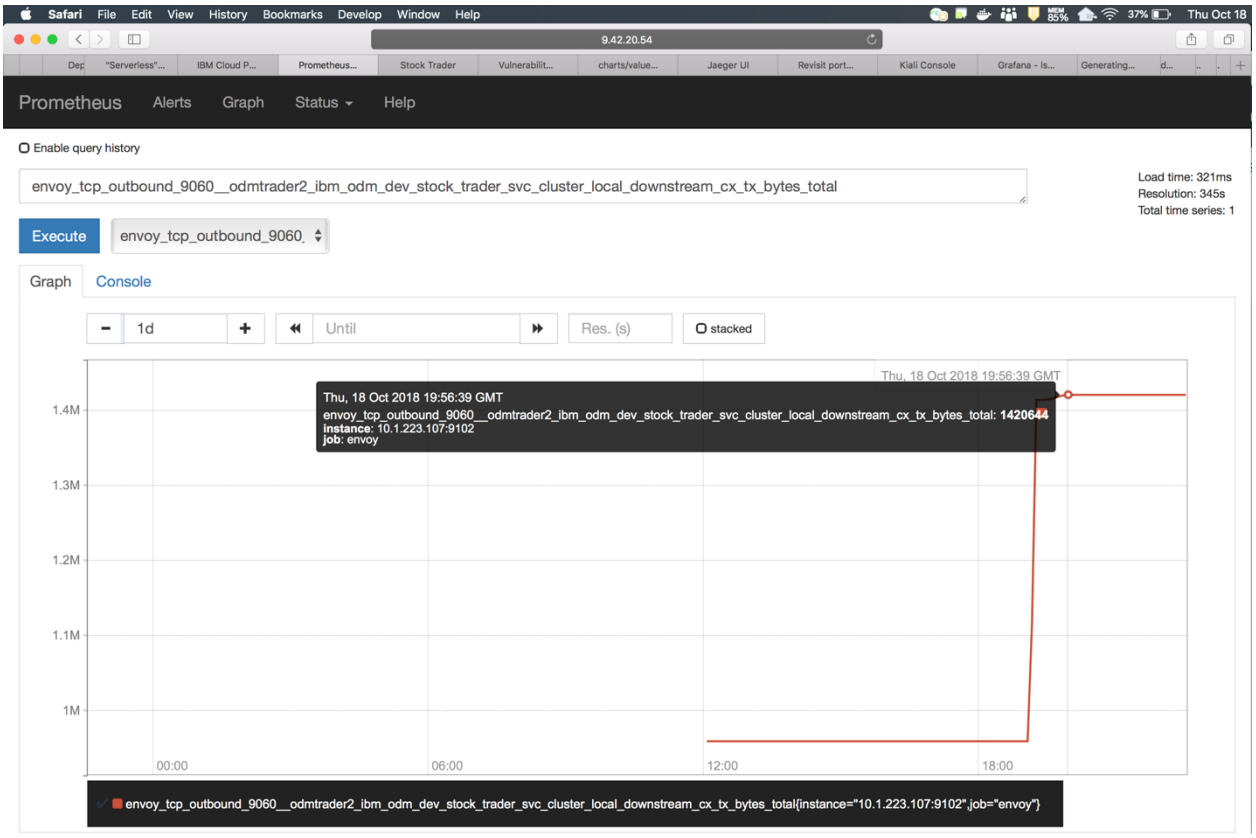
The net is, the programming model is totally unaffected here. Jane didn't have to change one line of Java code in her microservices to get this trace data. Just the fact that each microservice's container has a sidecar (which sends telemetry to Jaeger) means we get the above insights for free, which in this case led the Todd persona to recognize an infrastructure problem in Redis and to resolve it. Whereas in mpOpenTracing we are discussing stuff specific to Java-based microservices using MicroProfile, what we just saw here was truly polyglot and would all be true regardless of the language in which the microservices were written.

Prometheus

Istio produces a wide variety of metrics about what's going on in the service mesh. These get scraped by Prometheus, and you can see the raw values of thousands of such metrics in the Prometheus UI. As an example, you can see that we have done an egress from `portfolio` to the *Watson Tone Analyzer* 4 times in this screenshot:



And here you can see a graph of how many total bytes have been received from our "loyalty level" business rule in ODM over time; it apparently jumped from a little under a MB to over 1.4 MB at around 7 PM GMT today:

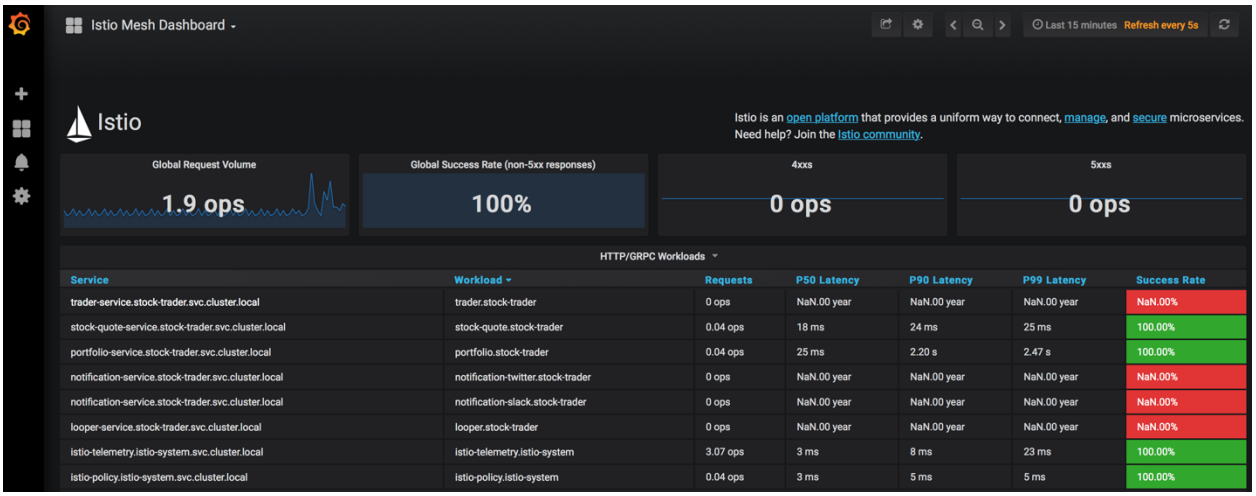


Prometheus will keep track of such data over time; it remembers data it has been sent, even if the pod that sent it is no longer available. However, it tends to show data at such a low level that it isn't often that useful during normal business operations; for that, let's move on to *Grafana*.

Grafana

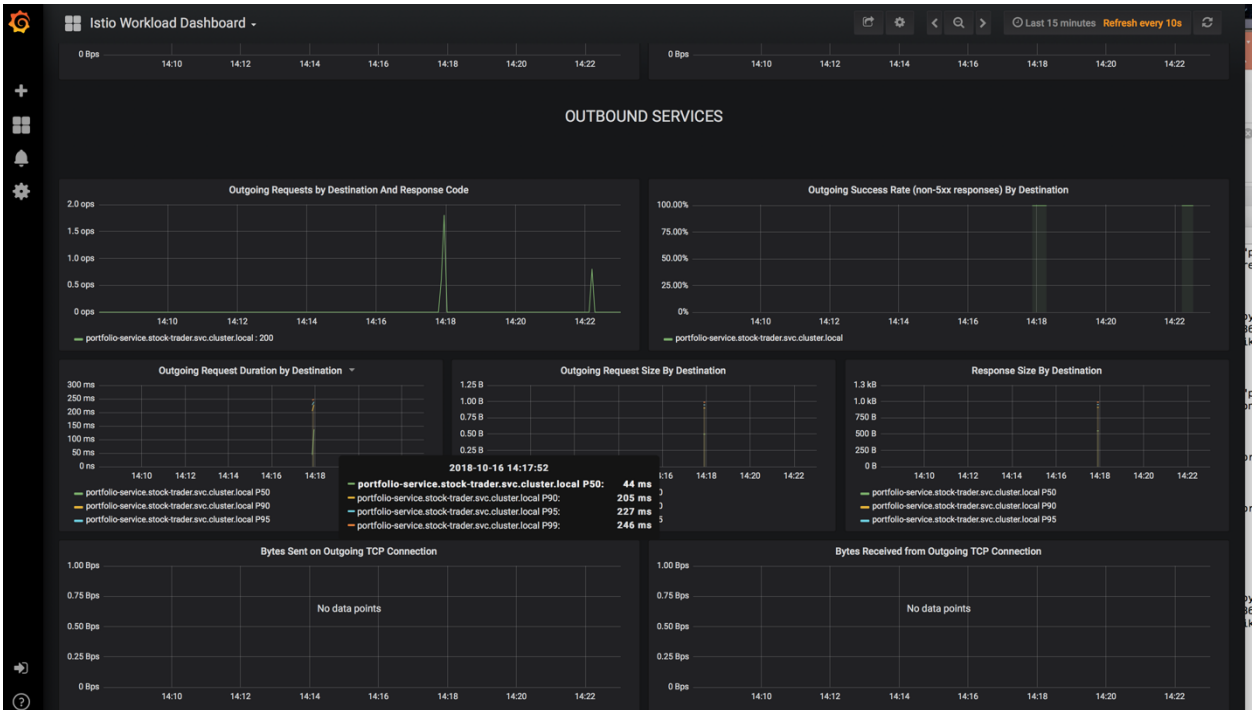
For those familiar with the ELK stack, I often think of Prometheus being to ElasticSearch like Grafana is to Kibana. And just like how someone doing cross-microservice troubleshooting would likely go to Kibana to see data at a higher level than raw ElasticSearch, they would usually go to Grafana to see data presented at a level that is more useful to someone needing to understand the metrics gathered by Istio rather than viewing the raw metric values in Prometheus.

Of course, you can manually define your own graphs in Grafana if you want, but there are some built-in "dashboards" that Istio preloads into Grafana that show some generally useful/interesting data about what has been going on in the service mesh. Let's take a look at a few. We'll start with the "Istio Mesh" dashboard:



As you can see, this shows us what percent of REST calls have been successful (100% in this case!), latency experienced by various microservices (note it shows both the "slack" and "twitter" versions of the "notification" service), etc. Of course, if no data was received from a given microservice during the time window specified, you will see no data for such rows.

Now let's look at the "Istio Workload" dashboard. Here, I've filtered it to my `portfolio` microservice:



We can see some activity spikes here, like at 2:18 PM. Obviously, the graphs aren't that interesting when the system isn't under much load. Therefore, I fired up my `looper` client, to run some steady traffic on 10 concurrent threads, and now we see some more interesting data, like in the "Mixer" dashboard:

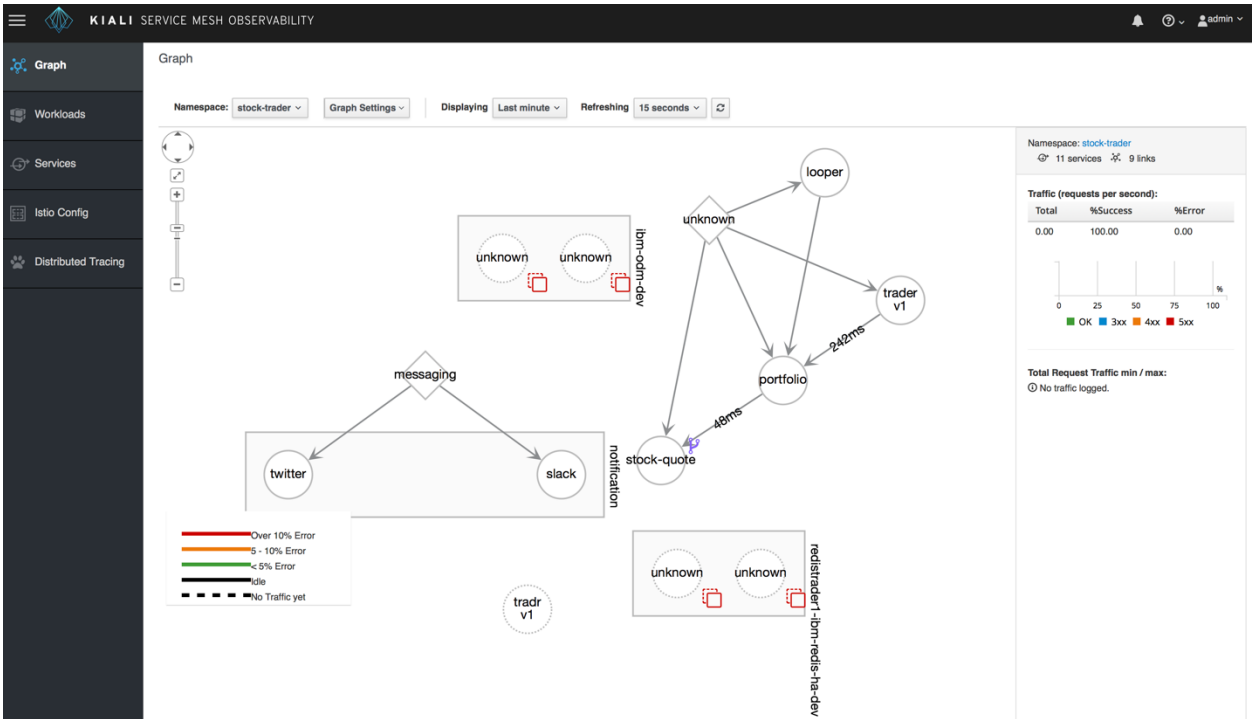
I won't go into explaining each "portlet" - most are pretty self-explanatory. I did find the "Pilot" dashboard to be interesting, as far as how much memory and CPU was being used by Istio infrastructure:

The net is, there's a LOT of metric data available with Istio, and both Prometheus and Grafana are automatically wired up to receive that and to allow you to visualize it in a variety of ways, and then act upon the insights you receive.

Kiali

One component I hadn't dealt with before is Kiali. Somewhat like the work we had done with Weave earlier, it attempts to provide visualizations of the interactions between microservices in

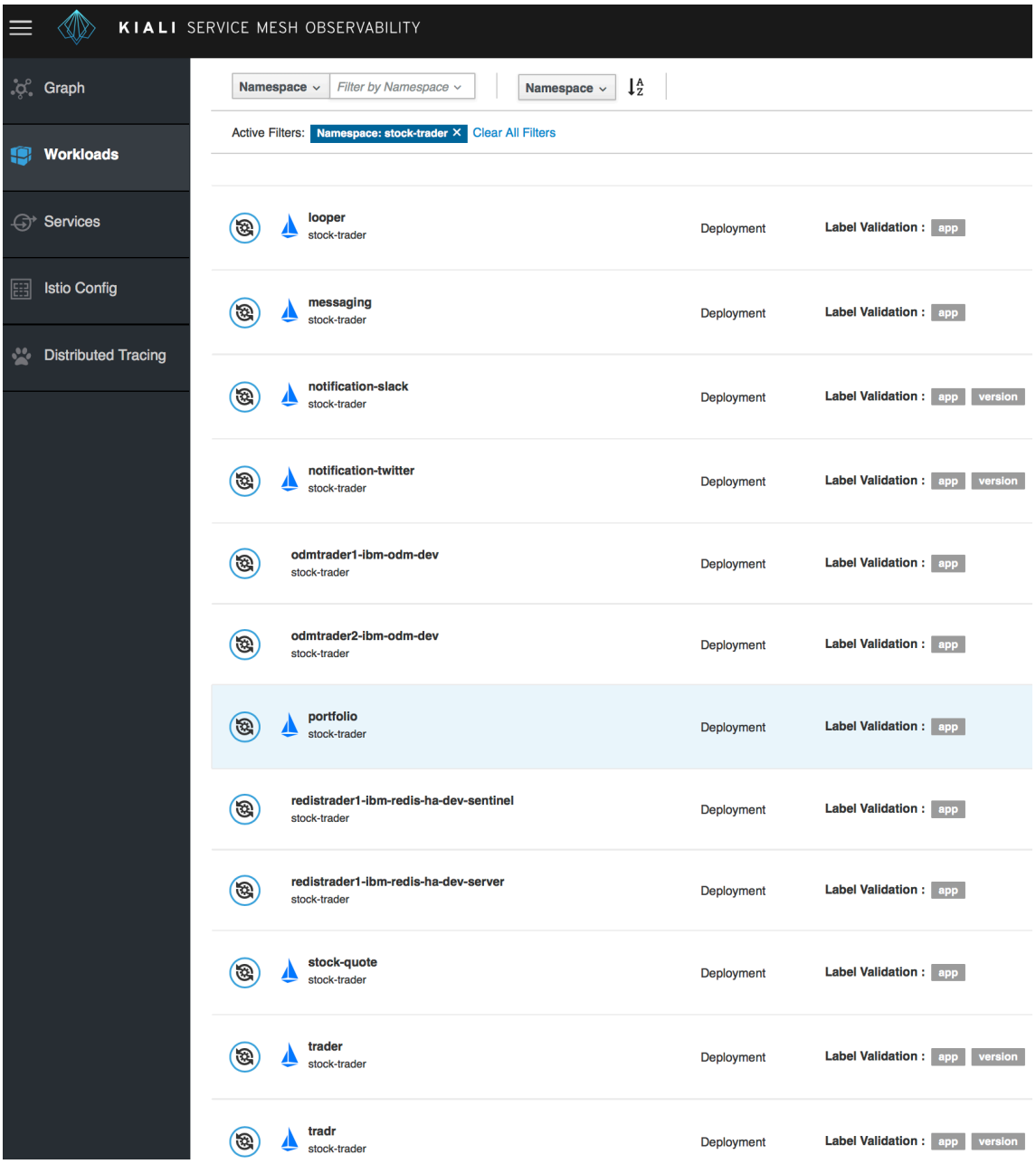
your mesh. For example, I had it show me its visualization for stuff in my `stock-trader` namespace, and found that it had a rather novel approach to presenting such relationships:



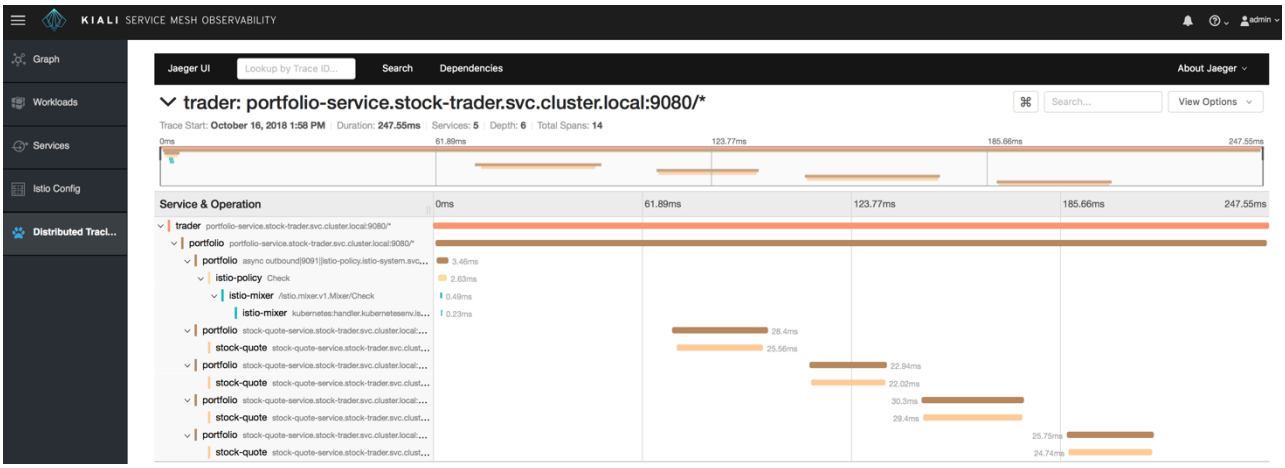
For example, it shows that both `trader` and `looper` talk to `portfolio`, which talks to `stock-quote` (I've manually made various PowerPoint slides showing such "architectural" diagrams; it is generating something similar for free for me). It also shows average response times on such calls; this is a live diagram updating in real-time, not some static thing produced once and which eventually gets out of date (unlike my PowerPoint slides).

It also shows that my `messaging` microservice calls a `notification` service, and shows inside that box how it consists of two versions: `slack` and `twitter` – that seemed pretty cool, and it only knows that because Istio understands the routing rules I have in place.

That was from the Graph tab on the left. If we move on to the Workloads tab, we can see a table that shows which of the deployments in my `stock-trader` namespace have had sidecars injected (via the blue Istio sailboat icon) and the various labels present on them:



Interestingly, the Kiali “console” also provides a way to see the Jaeger stuff we discussed earlier, as an embedded frame when you click its “Distributed Tracing” tab:



Kiali seems like a nice addition to our arsenal of tools that help us understand what's going on in our Kubernetes environment.

Security

Discuss *Mutual TLS* and *sidecar-enforced Role-Based Access Control (RBAC)* here.

Discovery and Load Balancing

A/B Testing must go here.

Fault Injection

You can use Istio to deliberately cause faults, to see how your code would respond to them....

Other Topics

Whitelisting, Blacklisting, etc.

Creating Microservices – Data Access and Management

Almost every microservice will have to access or manage data. Some microservices will simply rely on a REST interface to another microservice that serves up the data and manages is appropriately. Another way to acquire data is via listening to events. These approaches have largely been covered via prior topics. What this chapter is focused on are those services that directly managing data. How are these implemented? What are the interactions that these services are likely to have with the various data management services which are available?

One of the principles of microservices is to keep the data which a particular service needs isolated. While this isn't always possible, it should be the intention for net new microservices to embrace this.

When deciding on what approach to use for persistence and management of data, several factors come into play:

- Quantity of data
- Type of data – structured, unstructured
- Typical usages of data including query and updates
    - How rich of a query interface is needed?
    - Are updates in bulk or a single object at time?
- Performance requirements
- Required consistency
- Availability requirements

There are a few underlying technologies that can and should be leveraged when dealing with data. These are:

1. No-SQL databases
2. Cloud Object Storage
3. Using Streams as Data Sources
4. Relational databases

These technologies can all support the requirements a microservice has for storing and managing new data. It is also the case that some new microservices will need to leveraging existing stores. There is a lot of existing information in relational databases. After covering the 4 topics above, an extra section on accessing existing relational stores is also provided.

No-SQL Databases

Since JSON is our "native" notation for data that callers of our microservices will generally expect (if we have the `@Produces("application/json")` annotation on our JAX-RS method, which will usually be the case), being able to store that JSON directly, without mapping it to something else like we did above in the JDBC/SQL section, would make our job easier. Also, some of the No-SQL datastores scale MUCH better than you'll get with relational databases (though you give up some qualities-of-service in order to make such scalability gains). One example is the ElasticSearch No-SQL datastore, that our logging infrastructure in IBM Cloud Private uses; it is able to perform queries and filters – even when many millions of log

entries have been received from across dozens or hundreds of pods – at very fast speeds within Kibana, due to the nature of how the underlying ElasticSearch works.

That isn't to say that working directly with JSON is the right method. Many of the common NoSQL databases provide either analogs of the JDBC/SQL functions or there exists an ORM API for mapping POJO objects to/from JSON.

IBM provides a wide variety of NoSQL databases in both public and private cloud flavors.

IBM Public Cloud

Cloudant

IBM Cloudant, compatible with Apache CouchDB and PouchDB, is a fully managed JSON document database from IBM that is eventually consistent (all nodes will eventually contain the same data) and schema-less (you don't have to declare your database schema up-front or inform the database that you are going to a add new field). Cloudant is known for its powerful replication capabilities that allow users to create sophisticated hybrid, multi-region, or multi-cloud topologies. It is highly available out of the box and scales from an experimental tier to web-scale with zero downtime.

A Cloudant instance can contain many databases_ each of which is a collection of small JSON documents representing the application's _Objects_ - purchase orders, receipts, refunds etc. Each document has a unique identifier: the document `_id` which is either generated by Cloudant or provided by the application. Different object types can co-exist in the same database and can be queried using Cloudant's declarative query language or by building MapReduce views to aggregate and group data.

There are three different flavors of Cloudant, depending on the use case required by your microservice, however all feature triplicate replication across three availability zones, along with encryption in-flight and at-rest. Cloudant stores data in RAID-10 configuration preserving 6 copies of your data at minimum.

These three tiers are:
• Lite (free tier with limited data and throughput for experimentation),
• Standard (metered storage and flexible provisioned throughput capacity rates, 99.95% SLA)
• Dedicated Hardware (Provides single tenant hardware for Cloudant Standard plans, HIPAA-Ready, and supports Bring Your Own Encryption Key).

Regardless of which tier used there is a single management pane and all interactions with databases and documents occurs over HTTPS. All Cloudant instances are SOC 2 Type 2 and ISO 27001 compliant. There are many ways to interact with Cloudant besides doing direct Java-over-HTTP calls using something like Jersey or JAX-RS. Since Cloudant is compatible with CouchDB most, if not all, can be used with Cloudant. Cloudant also has a native Java client library, Java-Cloudant, that is fully supported by IBM

Compose for MongoDB

MongoDB is very similar to Cloudant in that it is a document store. However, while Cloudant databases have no strict schema (different documents can live in the same database) MongoDB databases have a set design schema which all documents must adhere to (similar to relational databases). Another distinction is that while JSON documents are sent to both Cloudant and MongoDB, MongoDB stores that data as BSON (Binary JSON) which allows them to support datatypes which JSON does not natively support, such as Date or binary. This is used as communication with MongoDB is via various driver technologies (very similar to the JDBC driver of relational databases) which map the various language-native constructs to BSON under the covers. That said it is convenient to think of the documents as JSON at a fundamental level.

As a Compose (an IBM Company) offering there are different prices levels depending: where you deploy MongoDB, how much storage is used, how much RAM is needed, and what the isolation requirements are (multi-tenant vs dedicated). These deployments all support automated backups, autoscaling, and high availability out of the box.

Note that the Stock Trader sample demonstrates usage of MongoDB in its `Trade History` microservice. Whenever it receives a `StockPurchase` message from the `stocktrader` Kafka topic in IBM Event Streams, it will insert it into MongoDB. Here's an example from its `com.ibm.hybrid.cloud.sample.stocktrader.tradehistory.mongo.MongoConnector` class:

```
//{ "id": "1234-abcd", "owner": "John", "symbol": "IBM", "shares": 42, "price": 120,
"when": "now", "commission": 0 }
public void insertStockPurchase(StockPurchase sp, DemoConsumedMessage dcm) {
   MongoCollection<Document> collection = database.getCollection("test_collection");
   Document doc = new Document("topic", dcm.getTopic())
      .append("id", sp.getId())
      .append("owner", sp.getOwner())
      .append("symbol", sp.getSymbol())
      .append("shares", sp.getShares())
      .append("price", sp.getPrice())
      .append("when", sp.getWhen())
      .append("commission", sp.getCommission());
   collection.insertOne(doc);
}
```

Compose for RethinkDB

RethinkDB, another JSON document store, supports dynamic schemas very similar to Cloudant. One interesting feature is the ability to publish real-time query updates to applications making this a hybrid storage and messaging system. A commonly cited example of using RethinkDB is storing a leaderboard for an online game. Any connected client will receive the updated leaderboard while they are connected to the internet. A second example could be pushing the latest stock prices out to all stock brokers currently online.

<code sample for RethinkDB>

Compose for ElasticSearch

Elasticsearch is a search engine and JSON store. Typically we see this used for log analysis and fuzzy search but more traditional use cases can be supported, too.

<code sample for elasticsearch>

Compose for ScyllaDB

ScyllaDB is a C++ implementation of Apache Cassandra, a hybrid NoSQL/SQL database. Data consists of rows, columns, and tables sharded across many nodes where no single node is a point of failure (data is replicated across all nodes, and new nodes can be added with no downtime). Instead of using SQL as the querying language it uses CQL (Cassandra SQL) for querying and Apache Thrift for binding data into data structures. Since it is a reimplementation of Cassandra any Cassandra API or library is also compatible with ScyllaDB.

<code sample for scyllaDB>

Compose for JanusGraph

If your data contains many complex or interconnected data then a graph database, such as JanusGraph, is the right choice. One good example is a social connectedness graph between different people. In a traditional database this would be stored as a series of tables all using many foreign keys to show these relationships. In graph databases the entire graph as-is can be stored and manipulated without worrying about foreign key constraints.
<sample JanusGraph code>

Compose for etcd

etcd is a distrubted key/value store typically used for configuration management by using the RAFT consensus protocol to assure that data is updated consistently and in the correct order across all the etcd nodes. You could use this database to coordinate work across different services.

<sample etcd code>

Compose for Redis

Redis is a memory-based key/value store but can also act as a cache, a queue, or a transient data store. Nearly any piece of data can serve as either the key or the value. Interacting with Redis is done via commands such as `APPEND key value` to update a given key's value or `RPUSH key value` to append a value to some key and `RPOP key` to pull the value back off.

Note that the Stock Trader sample demonstrates usage of Redis from its `Stock Quote` microservice, which it uses to cache stock quotes. It uses the Redis for Java, or `Jedis`, SDK (version 3.0) to communicate with Redis. It uses the `Jedis` connection pool class, `JedisPool`, to get a connection from the pool:

```
String redis_url = System.getenv("REDIS_URL");
URI jedisURI = new URI(redis_url);
logger.info("Initializing Redis pool using URL: "+redis_url);
jedisPool = new JedisPool(jedisURI);
```

It then checks whether the given stock is in the Redis cache (and later checks to see if it's less that one hour stale) via:

```
Jedis jedis = jedisPool.getResource(); //Get a connection from the pool
String cachedValue = jedis.get(symbol); //Try to get it from Redis
```

```
   Jsonb jsonb = JsonbBuilder.create();
   Quote quote = jsonb.fromJson(cachedValue, Quote.class);
```

And it adds a stock quote to the Redis cache via:

```
   jedis.set(symbol, quote.toString()); //Put in Redis so it's there next
time we ask
```

Cloud Object Storage

https://console.bluemix.net/docs/services/cloud-object-storage/libraries/java.html#using-java

Streams as Data Sources

New Relational Databases to Support Microservices

The traditional Java EE programming model for accessing relational databases is Java DataBase Connectivity (JDBC). Note that JDBC version 4.2 (part of the new Java EE version 8) just became available in Liberty 18.0.0.2, that just appeared on DockerHub.

Server configuration

First, you'll want to enable the `jdbc-4.2` feature (or the `javaee-8.0` convenience feature) in your server.xml. Then you need to add the `datasource` and `connectionManager` stanzas (of course, you might want to replace the literal values shown below in the `connectionManager` stanza with environment variables fed by config maps/secrets; likewise with the `isolationLevel` in the `datasource`):

```
    <connectionManager id="DB2-Connections" minPoolSize="5" maxPoolSize="50"/>
    <dataSource id="PortfolioDB" jndiName="jdbc/Portfolio/PortfolioDB"
connectionManagerRef="DB2-Connections" isolationLevel="TRANSACTION_READ_COMMITTED">
        <jdbcDriver>
            <library name="DB2" description="DB2 JDBC driver jar">
                <file id="db2jcc4" name="/config/db2jcc4.jar"/>
            </library>
        </jdbcDriver>
        <properties.db2.jcc serverName="${env.JDBC_HOST}"
portNumber="${env.JDBC_PORT}" databaseName="${env.JDBC_DB}"
            user="${env.JDBC_ID}" password="${env.JDBC_PASSWORD}"/> <!-- It won't use
a containerAuthDataRef for some reason, so defining credentials here instead -->
    </dataSource>
```

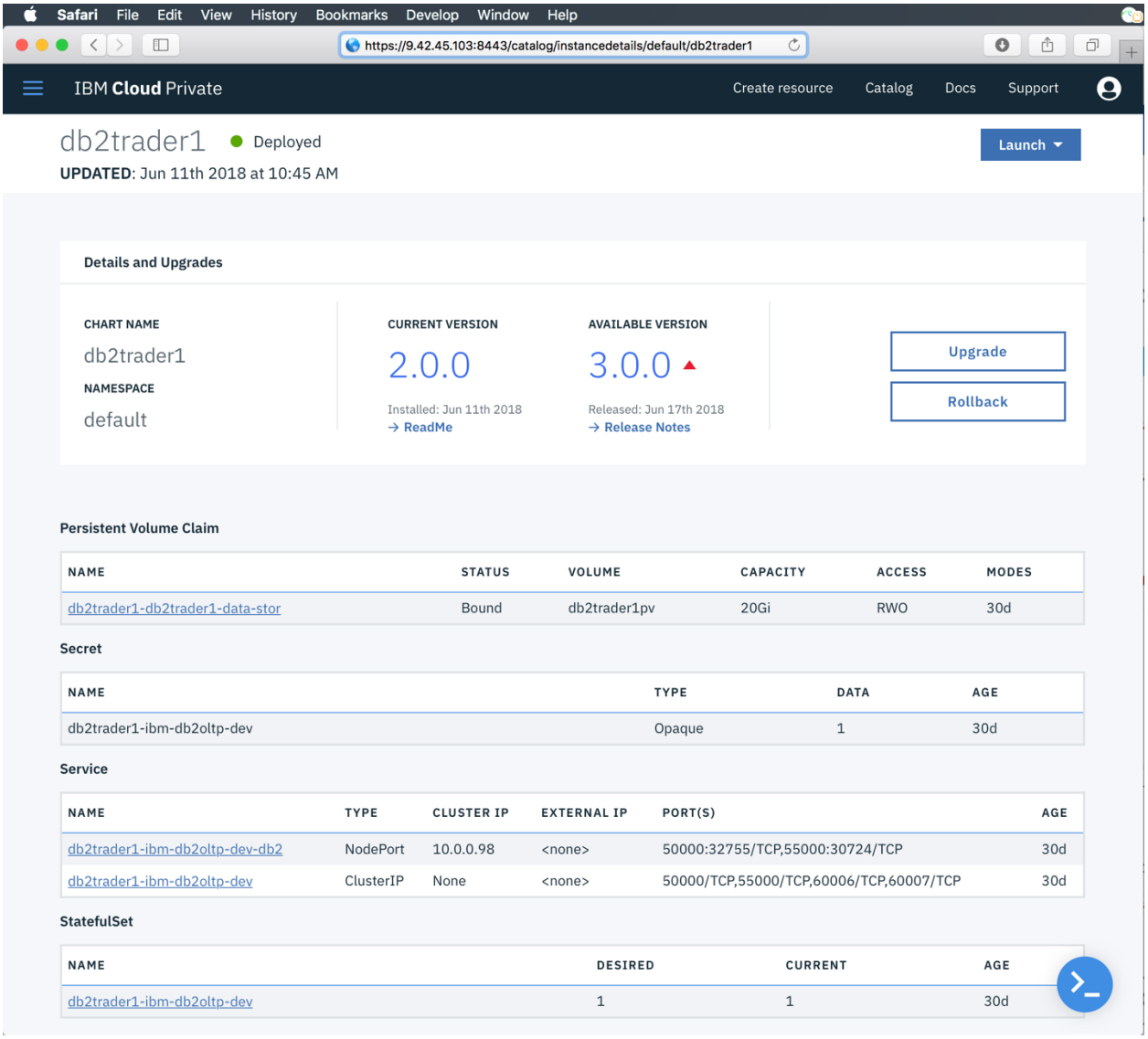
You also need to ensure that the JDBC jar file (`db2jcc4.jar`, in the example above) is copied to the specified place in your Docker image. I have the following line in my `Dockerfile` to make sure this happens:

```
COPY db2jcc4.jar /config/db2jcc4.jar
```

DB2

We are using **IBM DB2 version 11.1.4.4** as our JDBC provider. You can install it via its helm chart; the dev version is preloaded in the ICP helm catalog, and the prod version can be

obtained from Passport Advantage. We have tested Stock Trader with both the dev and the prod versions, and with DB2 HADR. Here's an example of its helm release:



It should be noted that the only thing you'd need to change to switch JDBC providers (like to Oracle, SQL Server, etc.) is the above line in the `Dockerfile`, and the `jdbcDriver` stanza in the server.xml; nothing at all changes in the Java code itself. In fact, I initially tested with **Derby** as my JDBC provider, before the person in the Todd persona got DB2 stood up in my private cloud, and I literally did not need to change anything in my Java code when I switched to DB2.

SQL statements

When working with any environment that can take objects through their entire lifecycle, we often talk about its "**CRUD**" methods, which stands for CREATE, RETRIEVE, UPDATE, and DELETE. For example, with REST, those CRUD methods map to the POST, GET, UPDATE and

DELETE http verbs. In SQL, the equivalent CRUD methods are INSERT, SELECT, UPDATE, and DELETE. Let's look at actual examples of each of these, first for the *Portfolio* table:

- `INSERT INTO Portfolio VALUES ('John', 0.0, 'Basic', 50.0, 0.0, 0, 'Unknown')`
- `SELECT * FROM Stock WHERE owner = 'John'`
- `UPDATE Portfolio SET total = 12345.67, loyalty = 'BRONZE' WHERE owner = 'John'`
- `DELETE FROM Portfolio WHERE owner = 'John'`

Now let's look at actual examples for the *Stock* table:

- `INSERT INTO Stock (owner, symbol, shares, commission) VALUES ('John', 'IBM', 10, 9.99)`
- `SELECT * FROM Stock WHERE owner = 'John' and symbol = 'IBM'`
- `UPDATE Stock SET shares = 10, commission = 8.99 WHERE owner = 'John' AND symbol = 'IBM'`
- `DELETE FROM Stock WHERE owner = 'John' AND symbol = 'IBM'`

So those are the kinds of SQL statements we need to execute to interact with our Portfolio and Stock tables. Now let's look at how we code up doing so in Java.

Java coding via JDBC

To execute SQL like above, you'll want to add the following import statements in your Java code:

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import javax.sql.DataSource;
```

To get the datasource in your code, just do a JNDI lookup (requires the `jndi-1.0` feature), specifying the value of the `jndiName` field from the `datasource` in the server.xml:

```
    logger.info("Obtaining JDBC Datasource");
    context = new InitialContext();
    datasource = (DataSource) context.lookup("jdbc/Portfolio/PortfolioDB");
    logger.info("JDBC Datasource successfully obtained!"); //exception would
have occurred otherwise
```

Then you just use that datasource to get a connection and a statement, and run either `executeQuery` or `executeUpdate`, passing in your SQL statement like shown in the previous section:

```
    @Traced
    private void invokeJDBC(String command) throws SQLException {
        try {
            initialize();
```

```
        } catch (NamingException ne) {
            if (datasource == null) throw new SQLException("Can't get
datasource from JNDI lookup!", ne);
        }

        try {
            logger.info("Running SQL executeUpdate command: "+command);
            Connection connection = datasource.getConnection();
            Statement statement = connection.createStatement();

            statement.executeUpdate(command);

            statement.close();
            connection.close();

            logger.info("SQL executeUpdate command completed successfully");
        } catch (SQLException sqle) {
            logException(sqle);
            throw sqle;
        }
    }
```

If you went the `executeQuery` path, it would have returned you a JDBC `ResultSet` that you could iterate over to access the results:

```
    logger.fine("Running following SQL: SELECT * FROM Stock
WHERE owner = '"+owner+"'");
    ResultSet results = invokeJDBCWithResults("SELECT * FROM
Stock WHERE owner = '"+owner+"'");

    int count = 0;
    logger.fine("Iterating over results");
    while (results.next()) {
        count++;
        String symbol = results.getString("symbol");
        int shares = results.getInt("shares");
        double commission = results.getDouble("commission");
        ... //do stuff with results
    }
    logger.info("Processed "+count+" stocks for "+owner);
    results.close();
```

One note: if I were especially concerned about performance, I would convert over to using a unique `PreparedStatement` per call, allowing the database to cache and re-use each statement. But for the simplicity of a sample, I preferred being able to have the generic `invokeJDBC` method that will just run any arbitrary SQL statement that I pass it. Also note that, again for simplicity of having an easily understood sample, I'm not addressing the possibility of SQL injection attacks here, like if someone entered malicious SQL statements in one of the entry fields in the UI.

Java coding via JPA

An alternative to coding to raw JDBC to talk to a relational datastore is to instead use the higher-level *Java Persistence API*, or *JPA*. This will let you represent each table as a Plain Old Java Object (POJO), with getter/setter methods for each column (very similar to the JSON-B POJOs). For example, you'd have a class like the following to represent the *Portfolio* table described above:

```
@Entity
@Table(name="Portfolio")
public class PortfolioTable implements Serializable {
  @Column(name="owner") private String owner;
  @Column(name="total") private double total;
  @Column(name="loyalty") private String loyalty;
  @Column(name="balance") private double balance;
  @Column(name="commissions") private double commissions;
  @Column(name="free") private int free;
  @Column(name="sentiment") private String sentiment;

  public PortfolioTable() {}
  public PortfolioTable(String newOwner) { setOwner(newOwner); }

  public String getOwner() { return owner; }
  public void setOwner(String newOwner) { owner = newOwner; }

  public double getTotal() { return total; }
  public void setTotal(double newTotal) { total = newTotal; }

  public String getLoyalty() { return loyalty; }
  public void setLoyalty(String newLoyalty) { loyalty = newLoyalty; }

  public double getBalance() { return balance; }
  public void setBalance(double newBalance) { balance = newBalance; }

  public double getCommissions() { return commissions; }
  public void setCommissions(String newCommissions) { commissions =
newCommissions; }

  public int getFree() { return free; }
  public void setFree(String newFree) { free = newFree; }

  public String getSentiment() { return sentiment; }
  public void setSentiment(String newSentiment) { sentiment = newSentiment; }
}
```

See the Liberty JPA guide for further details on how to configure and make use of such a JPA POJO: https://openliberty.io/guides/jpa-intro.html

Referential Integrity

The point of a relational database is to define the *relationships* between the tables, and to have the database be able to enforce the constraints you define around those relationships. Database administrators call this referential integrity.

In our Stock Trader example, there is logically a 1-to-many relationship between Portfolio and Stock. That is, each Portfolio can contain many stocks, but any given stock only belongs to one

portfolio. We express this via the foreign key constraint on our Stock table. Here are the SQL statements to create our two tables – I've emphasized the relevant relationship part in bold:

```
CREATE TABLE Portfolio(owner VARCHAR(32) NOT NULL, total DOUBLE,
loyalty VARCHAR(8), balance DOUBLE, commissions DOUBLE, free
INTEGER, sentiment VARCHAR(16), PRIMARY KEY(owner));
CREATE TABLE Stock(owner VARCHAR(32) NOT NULL, symbol VARCHAR(8)
NOT NULL, shares INTEGER, price DOUBLE, total DOUBLE, dateQuoted
DATE, commission DOUBLE, FOREIGN KEY (owner) REFERENCES
Portfolio(owner) ON DELETE CASCADE, PRIMARY KEY(owner, symbol));
```

The "**`FOREIGN KEY (owner) REFERENCES Portfolio(owner)`**" part means that the owner column in the Stock table is actually a pointer (a *foreign key*, in SQL terminology) to a row in the Portfolio table. Furthermore, defining this foreign key relationship means that you won't be allowed to insert or update a row into the Stock table that points to a non-existent row in the Stock table; that is, a stock can't be created or updated such that it would be orphaned from its containing portfolio.
The "**`ON DELETE CASCADE`**" part means that if I delete a portfolio, all of the stocks that have a foreign key pointing to that portfolio will also get deleted. This saves the programmer from having to issue a "DELETE FROM Stock WHERE owner = <portfolio>", and avoids the possibility of causing orphaned rows in the stock table due to deleting a portfolio.

Object to Relational Mapping

Note that there's a non-trivial object-to-relational (O-R) mapping here. The actual native data structure, like that a caller gets back from `GET /portfolio/{owner}`, is JSON of this form (I've manually formatted the object here for easier human consumption):

```
{"owner": "John",
 "total": 1647.60,
 "loyalty": "BASIC",
 "balance": 30.02,
 "free": 0,
 "sentiment": "Unknown",
 "stocks": [
    {"symbol": "IBM",
     "shares": 10,
     "price": 144.39,
     "total": 1443.90,
     "dateQuoted": "2018-07-09",
     "commission": 9.99 },
    {"symbol": "MSFT",
     "shares": 2,
     "price": 101.85,
     "total": 203.70,
     "dateQuoted": "2018-07-09",
     "commission": 9.99 }
 ]
```

```
}
```

Now let's look at how that gets persisted, across both the `PORTFOLIO` and `STOCK` tables in DB2 (note I had to escape both the asterisk and the quotes when using the db2 CLI below):

```
Johns-MacBook-Pro:portfolio jalcorn$ kubectl exec db2trader1-
ibm-db2oltp-dev-0 -it /bin/bash
[root@db2trader1-ibm-db2oltp-dev-0 /]# su db2inst1
[db2inst1@db2trader1-ibm-db2oltp-dev-0 /]$ db2 connect to trader

   Database Connection Information

 Database server        = DB2/LINUXX8664 11.1.3.3
 SQL authorization ID   = DB2INST1
 Local database alias   = TRADER

[db2inst1@db2trader1-ibm-db2oltp-dev-0 /]$ db2 list tables

Table/View                      Schema          Type  Creation
time
------------------------------- --------------- ----- ----------
--------------
PORTFOLIO                       DB2INST1        T     2018-06-
15-01.07.41.927261
STOCK                           DB2INST1        T     2018-06-
15-01.08.03.666591

  2 record(s) selected.

[db2inst1@db2trader1-ibm-db2oltp-dev-0 /]$ db2 select \* from
Portfolio where owner=\'John\'

OWNER                            TOTAL
LOYALTY  BALANCE                  COMMISSIONS              FREE
SENTIMENT
-------------------------------- ----------------------- ------
-- ----------------------- ----------------------- -----------
---------------
John                               +1.64760000000000E+003 BASIC
+3.00200000000000E+001   +1.99800000000000E+001           0
Unknown

  1 record(s) selected.

[db2inst1@db2trader1-ibm-db2oltp-dev-0 /]$ db2 select \* from
Stock where owner=\'John\'

OWNER                            SYMBOL   SHARES      PRICE
TOTAL                   DATEQUOTED COMMISSION
```

```
------------------------------ -------- ----------- ----------
------------- ---------------------- ---------- -------------
-----------
John                                    IBM                10
+1.44390000000000E+002   +1.44390000000000E+003 07/09/2018
+9.99000000000000E+000
John                                    MSFT                2
+1.01850000000000E+002   +2.03700000000000E+002 07/09/2018
+9.99000000000000E+000

  2 record(s) selected.

[db2inst1@db2trader1-ibm-db2oltp-dev-0 /]$ exit
```

The fact that I had to split the JSON above into two tables was something the O-R mapping forced upon me, so that I could implement the 1-to-many relationship between a portfolio and the stocks it contains.

Note that there isn't actually an "`owner`" field in each Stock object. That's something I'm having to add in my SQL statement, to get the foreign key to appear (note it is a `NOT NULL` field, due to being part of the *primary key*), which is how the 1-to-many relationship is manifested in a relational database.

Right now, in my `getPortfolio` JAX-RS method, I have to issue multiple SQL commands to get the data for a portfolio (one to the Portfolio table, and another to the Stock table), iterate over the JDBC ResultSet, then manually copy each field from each entry in the ResultSet into a JsonObjectBuilder, so that I can "just in time" produce the JSON that the caller expects. This is an unnatural mapping from the SQL world to the JSON world that is a fact of life when persisting your JSON objects to a relational database.

I'll point out that did NOT have to do an O-R mapping in the original version of Portfolio that I wrote, which just inserted the javax.json.JsonObject instance for each portfolio into an in-memory hashmap, with the "owner" string as the key (of course, I lost the data if my pod was bounced, or rescheduled onto a different node). This provides a good segue into our next topic, which is what if we wanted to use a form of persistence that isn't based on mapping the native JSON data format into a SQL-defined relational datastore?

Accessing Existing Relational Databases

In all but the most greenfield of solutions, there is a strong likelihood that existing relational databases will provide a source of necessary information and might even need to be updated by a new cloud native application. There are many approaches and many opinions on how to tackle this reality. Some of these are discussed in this section:

1. using APIs that wrapper the existing schemas to access the tables.
2. directly accessing the tables via JDBC
3. building APIs that are specific to a given need
4. shared static data – make it a service

Note: In this case, the columns and the SQL elements in the schema in general are fixed and immutable. Thus, this becomes a bottom up challenge.

Advanced Data Management Topics

This section is not intended to detail out all these advanced application design patterns, but should mention the key ones and highlight our IBM support for these things.

- In memory caches, having them keep a URL to real data as part of the data structure
- Versioning, tolerant reads, etc..
-

Advanced Asynchronous Patterns

There are two principal scenarios for microservices using messaging for asynchronous interactions. First, a microservices solution which is otherwise composed using synchronous APIs might have an need to trigger a piece of work that it need not or must not wait for. This is referred to as the worker offload pattern. Second, a microservices solution might be loosely coupled to the extent that the majority of communication between the microservices is in the form of events. This is referred to as event-driven microservices. Event-driven microservices using Kafka were covered earlier in the book, so here, we will focus on alternatives for Event-driven microservices and more advanced patterns built on top of Kakfa, as well as looking at the roles that MQ and JMS can play.

IBM offers two types of messaging technology in IBM Cloud Private, with equivalents on public IBM Cloud too.

IBM MQ is a solid, long-standing member of the IBM middleware family, with strong qualities of service, like "assured persistent messaging" and "once and only once delivery" of messages. It integrates well with other enterprise middleware and can act as a resource manager in XA transactions.

IBM Event Streams is a new offering based on the open-source Apache Kafka event streaming platform. It's ideal as the messaging backbone for event-driven applications.
In comparison with IBM MQ, Event Streams has:

- Publish/subscribe messaging only, with no point-to-point queues
- Stream history so messages are retained even after consumption to allow for late-joining consumers or re-processing of messages
- Local transactions (called exactly-once semantics) aimed primarily at stream processing applications
- No support for XA transactions

Both IBM MQ and IBM Event Streams are available in the IBM Cloud Private catalog.

Worker Offload

In this scenario, a microservices application is largely wired together using synchronous calls but some aspects of the application are more loosely coupled.

For example, when calling out to a third-party service, it is often advisable to consider the availability characteristics of that service such as maintenance windows and how they might affect the entire solution's reliability. By invoking the service indirectly through sending a message, and having the actual service called by a separate microservice as it processes the messages, the availability of the entire solution is insulated from the third-party service's behavior.

A real-world example from retail is a loyalty scheme which required interaction with a handful of partner companies, each with a service to be called during customer enrolment. Each

enrolment involved sending a message to request a call to each partner's service, but not waiting for the services to be called directly.

The messaging pattern to use here is essentially a queue. That's really a function of how the messages are processed more than how they are sent. In the loyalty scheme example earlier, imagine that there are 4 third-party services A, B, C and D to be called for each customer enrolment. You could achieve the messaging in a variety of ways, such as:

- Using IBM MQ, you could have 4 separate queues, one for each of A, B, C and D. Each message then represents a specific call to the third-party service.
- Using IBM MQ, you could use a topic with 4 durable subscriptions, one for each of A, B, C and D. Each subscription is handled by a separate microservice, one per third-party service. Each message then represents a customer enrolment and the subscribing microservices make the requests accordingly.
- Using IBM Event Streams, you could use a topic for customer enrolments with 4 consumer groups, one for each of A, B, C and D. Each consumer group is handled by a separate microservice (potentially multiple instances of each). Each message represents a customer enrolment.

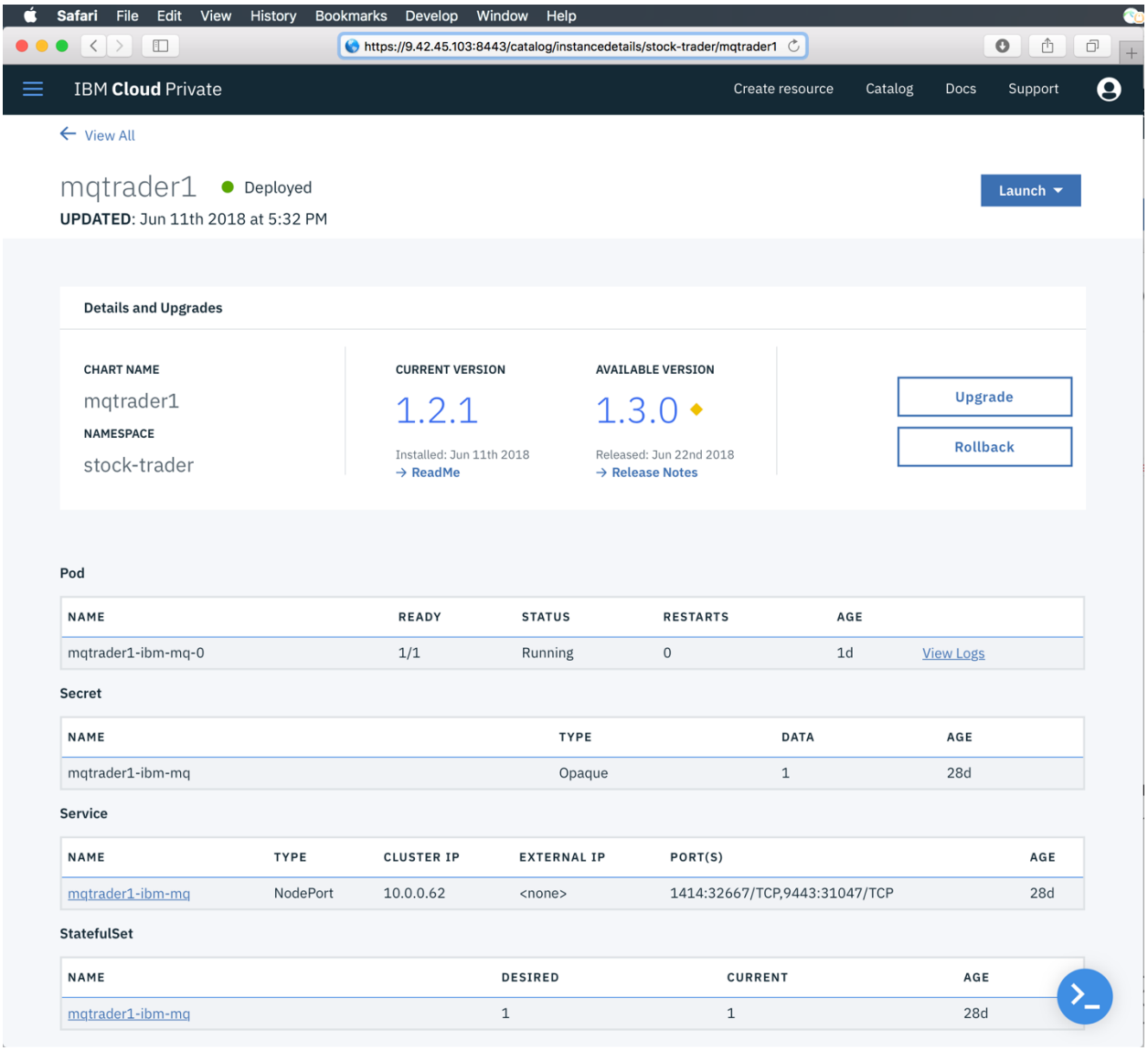
In practical terms, all of these behave like queues in terms of how the messages are processed, even though the last 2 are actually publish/subscribe. The publish/subscribe technique is more loosely coupled because you can easily add more back-end services just by adding more subscriptions or consumer groups.

Of course, the downstream effects of the asynchronous interactions are not quite consistent with the rest of the application and lag slightly behind while the messages are being processed, but it's normal for consistency of a microservices application to be looser than traditional monolithic applications. As a result, transactions and exactly-once delivery are less prevalent than in monolithic applications. The advantage of separating the application into independent pieces often has the side-effect of loosening consistency among the pieces, at least temporarily. There can be pieces of the application for which data integrity and consistency is key, but it's unlikely that all of the components of the entire application are consistent at all times.

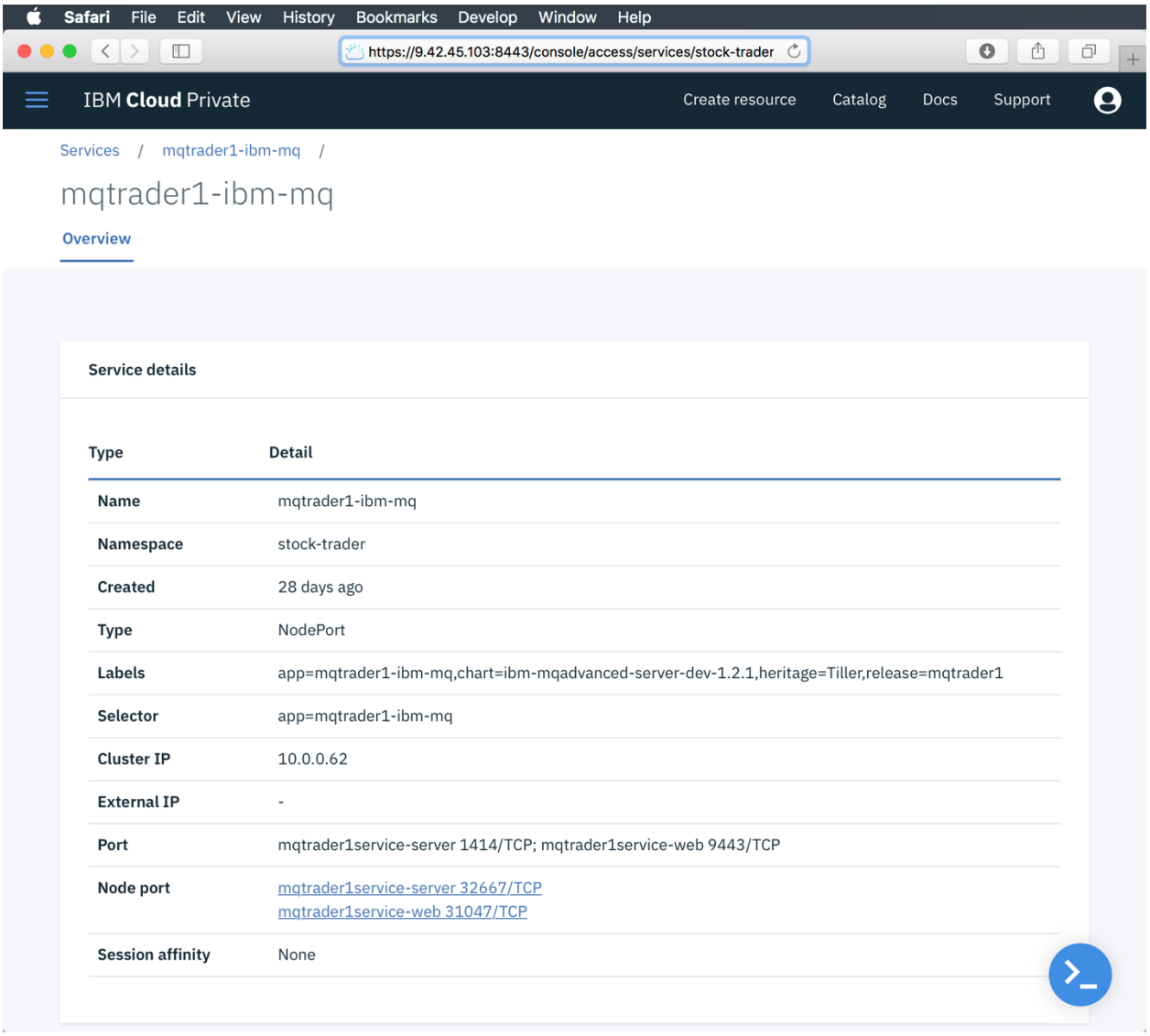
In our *Stock Trader* example, we currently do exactly this, when we detect a change in loyalty level for a portfolio. Our portfolio microservice builds up a small JSON object, and drops it onto a queue, then moves on, unaware of, and not carrying about, what actually receives that message and acts upon it. We are also about to implement an auditing feature, that will involve us sending a message, whenever a portfolio buys or sells any stock. We will demonstrate two different approaches to how to send and respond to such messages.

MQ

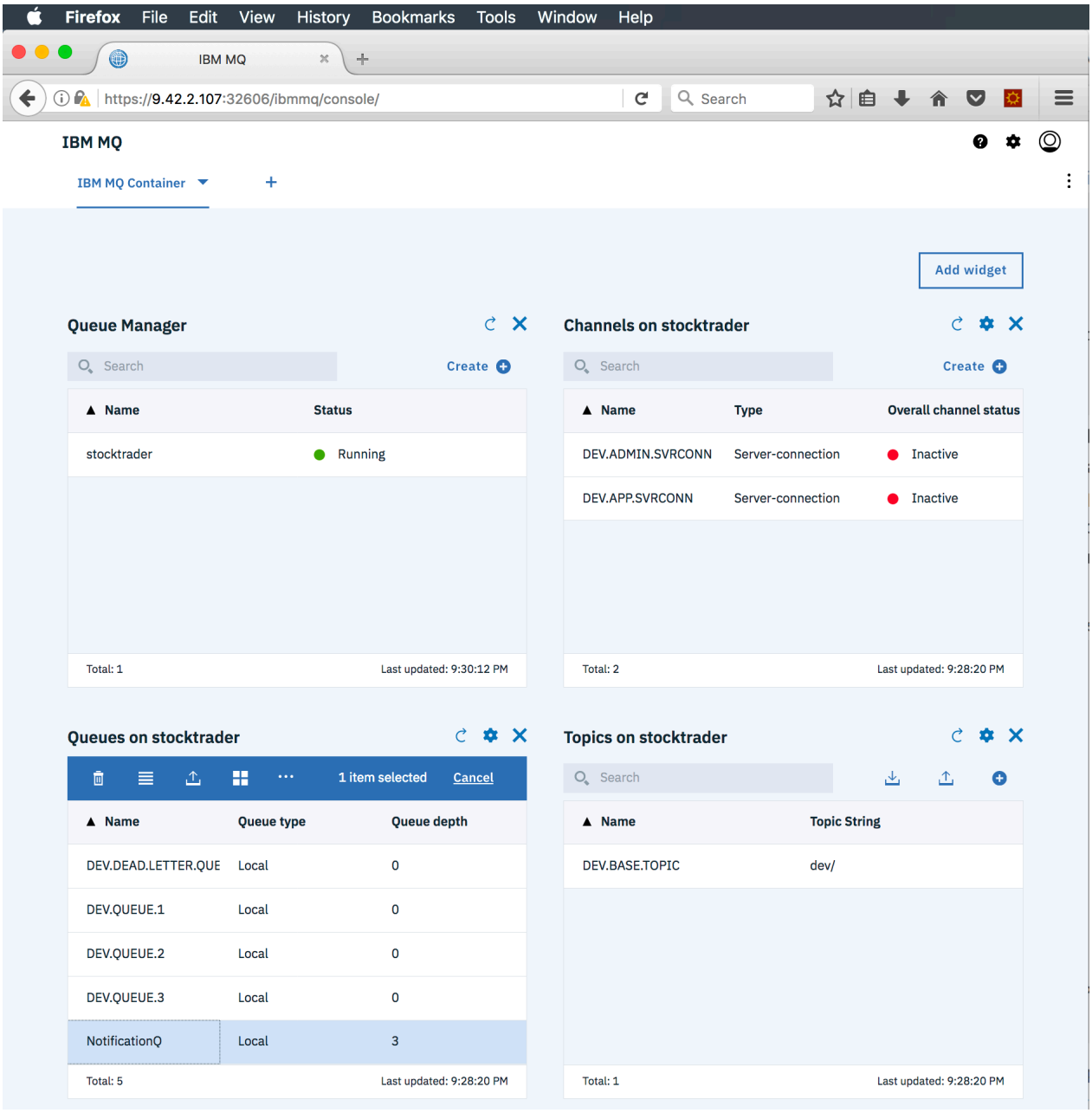
For the first case, responding to a loyalty level change, we have decided to use IBM MQ as our messaging infrastructure. IBM MQ has strong qualities of service, like "assured persistent messaging" and "once and only once delivery" of messages.
We installed MQ via its helm chart; note that like many of our IBM products, there's the dev chart preloaded into the ICP catalog, and a prod chart you can access via Passport Advantage – we've tested *Stock Trader* with each. Here's the helm release summary for MQ:



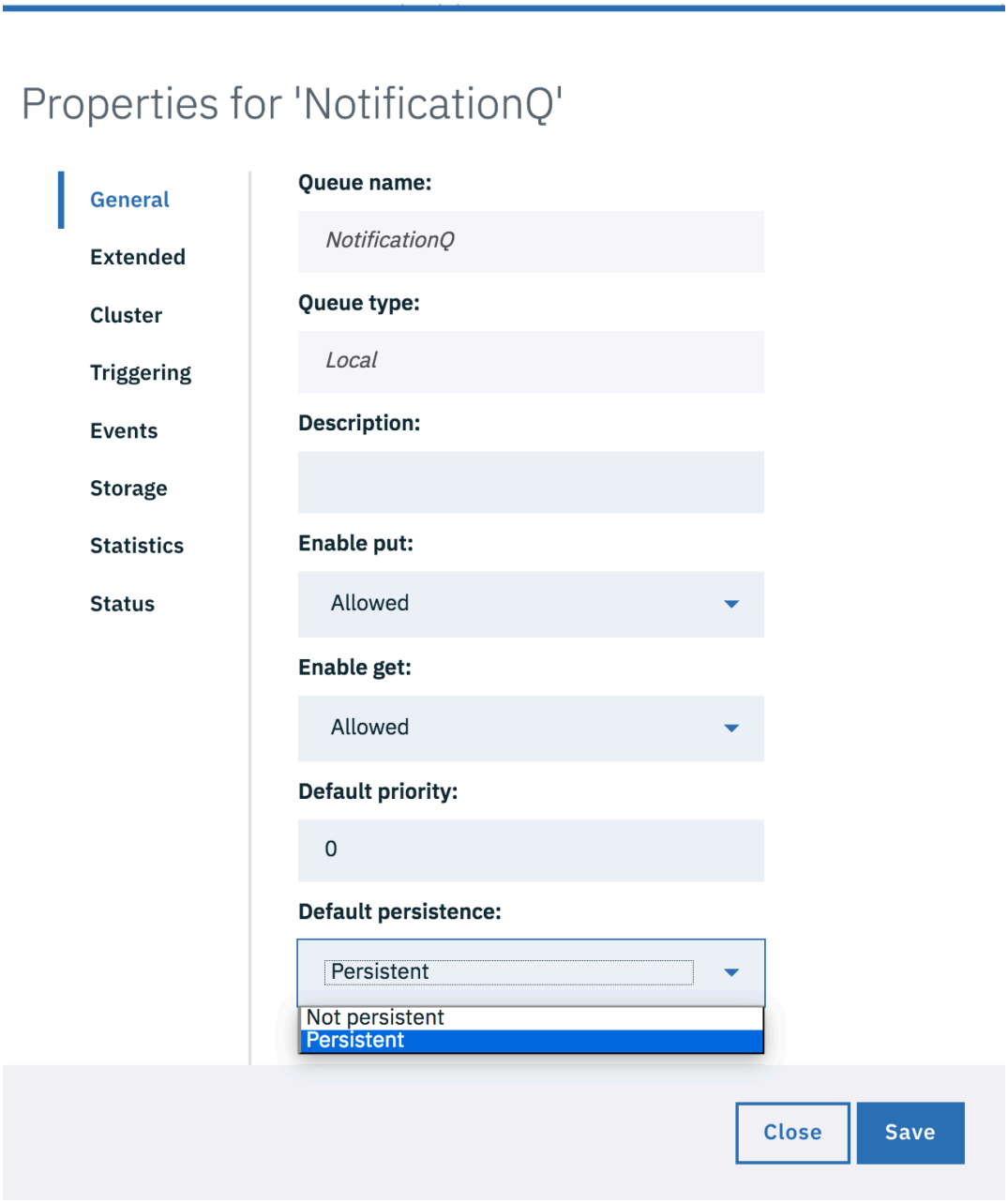
We can access the MQ console via the node port for the MQ service, by choosing the one with "web" in the name (the other one is the one you'd use to issue API requests). In this case, it's the one named "`mqtrader1service-web 31047/TCP`". Note I had to edit the URL it took me to, changing the http to https, to actually get there:



Here's an example of the MQ console (from clicking on that link above), with our `NotificationQ` highlighted – note that it shows 3 messages are currently sitting on the queue, waiting for someone (like our `messaging` microservice) to act upon them.



With our `NotificationQ` selected, we can select the four-horizontal-lines icon (second from the left in the toolbar) to view the Properties for the queue. We can adjust the `Default persistence` value on the queue here, like to set it to `Persistent`, so that messages won't be lost if the MQ pod dies (they'll be written to the Persistent Volume configured for MQ with this set).



Note that while we are using MQ most of the time, we can actually swap JMS providers whenever we want just by editing the server configuration and the Dockerfile (to copy the JMS provider's rar file into the Docker image); nothing in the Java code itself is specific to the use of MQ as the JMS provider. In fact, when MQ isn't available (like if someone in the Todd persona hasn't installed it yet to my private cloud), I sometimes use Liberty's built-in JMS provider (as a separate `messaging-engine` deployment/pod) instead, and it works just fine. In fact, you could use the `wasJmsServer-2.0` feature (Liberty's built-in JMS provider) in Dev, but use MQ in Test/Prod, if desired, with zero change to your Java code.

Server configuration

First, like we did with the JDBC jar, we need to make sure our `Dockerfile` copies the JMS provider's rar file into the Docker image, to the location specified in the `resourceAdapter` stanza of the server.xml discussed below. We do this via:

```
COPY wmq.jmsra.rar /config/wmq.jmsra.rar
```

What you put in your server.xml depends on whether you want to send or receive messages. For example, our `portfolio` microservice is a message producer, meaning it sends messages to our `NotificationQ`. Whereas our `messaging` microservice is a message consumer, meaning it receives messages from our `NotificationQ`. In either case, you'll want the `jms-2.0`, `jca-1.7`, and `jndi-1.0` features in your server.xml (or the `javaee-8.0` convenience feature, which includes these and all of the other features of Java EE v8).
Let's look at what a message producer needs to configure first:

```
    <authData id="MQ-Credentials" user="${env.MQ_ID}" password="${env.MQ_PASSWORD}"/>
    <resourceAdapter id="mq" location="/config/wmq.jmsra.rar"/>
    <jmsQueueConnectionFactory id="NotificationQCF"
jndiName="jms/Portfolio/NotificationQueueConnectionFactory" containerAuthDataRef="MQ-
Credentials">
        <properties.mq
            transportType="CLIENT"
            hostName="${env.MQ_HOST}"
            port="${env.MQ_PORT}"
            channel="${env.MQ_CHANNEL}"
            queueManager="${env.MQ_QUEUE_MANAGER}"/>
    </jmsQueueConnectionFactory>
    <jmsQueue id="NotificationQ" jndiName="jms/Portfolio/NotificationQueue">
        <properties.mq baseQueueName="${env.MQ_QUEUE}"
baseQueueManagerName="${env.MQ_QUEUE_MANAGER}"/>
    </jmsQueue>
```

So there are four parts to that:

1. First we define the `authData` stanza, which has the credentials for how we access MQ.
2. Next is the `resourceAdapter` stanza, which is where you specify the path to the JMS provider rar file, and give it an id.
3. Third is the `jmsQueueConnectionFactory` stanza, which is where we specify the JNDI name for the factory, and which has a nested `properties.<id>` stanza, where `id` is the id we assigned the JMS provider adapter (so it is `properties.mq` when we are using MQ), and this is where we provide properties specific to your JMS provider implementation, such as the `queueManager` for MQ (note we are using environment variables, fed to the pod via a config map or secret, so that we can set such values at deployment time, without having to edit our server.xml and rebuild our Docker container and redeploy it to Kube).
4. Finally, there's the `jmsQueue` stanza, where we specify the JNDI name for the queue, and again we have a nested `properties.<id>` stanza (like `properties.mq` for MQ), where we can set things like the name of the queue (like `NotificationQ`, again fed in as an env var).

In the message consumer case, items 1, 2, and 4 above remain the same, but instead of the `jmsQueueConnectionFactory` from 3, we instead have a `jmsActivationSpec`:

```
    <jmsActivationSpec id="Messaging/MessagingEJB/MessagingMDB"
authDataRef="MQ-Credentials" maxEndpoints="1">
        <properties.mq
            transportType="CLIENT"
            hostName="${env.MQ_HOST}"
            port="${env.MQ_PORT}"
            channel="${env.MQ_CHANNEL}"
            queueManager="${env.MQ_QUEUE_MANAGER}"
            destinationRef="NotificationQ"
            destinationType="javax.jms.Queue" />
    </jmsActivationSpec>
```

As you can see, it is nearly identical to a `jmsQueueConnectionFactory`; the main difference is that a JMS activation spec specifies the name of a message driven bean (in the format `ear_name/ejb_jar_name/mdb_name`, without file name extensions like .ear or .jar specified), which is an EJB whose `onMessage` method gets invoked whenever a message arrives on the specified destination (which can be a queue or topic) - more on that later, in the "*JMS – Receiving a message*" section. Note we also need the `mdb-3.2` feature in our server.xml for this MDB to work.

JMS – Sending a message

Now that our server is properly configured, let's look at how we send a message from our Java code. First let's look at the import statements we need (from our `portfolio` microservice):

```
import javax.jms.DeliveryMode;
import javax.jms.JMSException;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.Session;
import javax.jms.TextMessage;
```

Now let's see how we get at our queue and its factory, from our `initialize()` method:

```
     logger.info("Looking up our JMS resources");

     QueueConnectionFactory queueCF = (QueueConnectionFactory)
context.lookup(NOTIFICATION_QCF);
     Queue queue = (Queue) context.lookup(NOTIFICATION_Q);

     logger.info("JMS Initialization completed successfully!");
//exception would have occurred otherwise
```

Now let's look at actually sending a message. Note that the message we are sending is a TextMessage that contains a JSON object with information about the change in loyalty level; an example message would be: `{"owner": "John, "old": "Silver", "new": "Gold"}`. Now let's look at how to send that JSON message:

```
    /** Send a JSON message to our notification queue. */
```

```
    @Traced
    private void invokeJMS(JsonObject json) throws JMSException,
NamingException {
        if (!initialized) initialize(); //gets our JMS managed
resources (Q and QCF)

        logger.info("Preparing to send a JMS message");

        QueueConnection connection =
queueCF.createQueueConnection();
        QueueSession session =
connection.createQueueSession(true, Session.AUTO_ACKNOWLEDGE);

        String contents = json.toString();
        TextMessage message =
session.createTextMessage(contents);

        logger.info("Sending "+contents+" to
"+queue.getQueueName());

        //"mqclient" group needs "put" authority on the queue
for the following to work
        QueueSender sender = session.createSender(queue);
        sender.setDeliveryMode(DeliveryMode.PERSISTENT);
        sender.send(message);

        sender.close();
        session.close();
        connection.close();

        logger.info("JMS Message sent successfully!"); //a
JMSException would have occurred otherwise
    }
```

Basically, we just get a connection to our queue, start a session (which is essentially a local transaction scope; all messages sent under the same session will be sent atomically; that is, you'll either get all of them on a commit, or none of them on a rollback; note that if you want to do an XA transaction, you'll need to follow the advice in the *Unit of Work* section of this book), and then send our text message under that session.
Note I'm explicitly setting the delivery mode to `PERSISTENT`; technically, that isn't needed if I remembered to set that `Default persistence` property on my MQ queue from the earlier screenshot, but I like to explicitly set it in code in case I forget to configure it in MQ. I want it to be persistent so that the message won't get lost if the MQ pod dies for whatever reason before my `messaging` microservice can receive it and act upon it.

JMS – Receiving a message

Now that we've seen how to send a message in Java, let's look at how we can receive such a message in Java. Here, we'll use an old-fashioned Enterprise Java Bean (EJB) - specifically a Message Driven Bean (MDB). Our `messaging` microservice is the only one in *Stock Trader* that uses old-fashioned Java EE, rather than just MicroProfile; as such, the first line of its

`Dockerfile` is `FROM websphere-liberty:javaee8`, rather than the usual `FROM websphere-liberty:microProfile`.
First, let's look at the import statements we'll need:

```
import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;
```

Now let's look at the class definition:

```
@MessageDriven(name = "MessagingMDB", mappedName =
"jms/Portfolio/NotificationQueue")
public class MessagingMDB implements MessageListener {
```

As you can see, we have the `@MessageDriven` annotation on our class, which announces that this class is a Message Driven EJB, and which says it is mapped to the JNDI name for our queue. Our `MessagingMDB` class implements the `MessageListener` interface, meaning it has an `onMessage` method that will get invoked whenever a message arrives on the destination (like our `NotificationQ`) specified by the `mappedName` field of the `@MessageDriven` annotation. Let's take a look at that method:

```
    public void onMessage(Message message) {
        if (message instanceof TextMessage) try {
            TextMessage text = (TextMessage) message;
            String payload = text.getText();

            logger.fine("Sending "+payload+" to
"+NOTIFICATION_SERVICE);
            JsonObject output = invokeREST("POST",
NOTIFICATION_SERVICE, payload);
            logger.info("Received the following response from
the Notification microservice: "+output);
        } catch (Throwable t) {
            logger.warning("An error occurred processing a JMS
message from the queue");
            logException(t);
        } else {
            logger.warning("onMessage received a non-
TextMessage!");
        }
    }
```

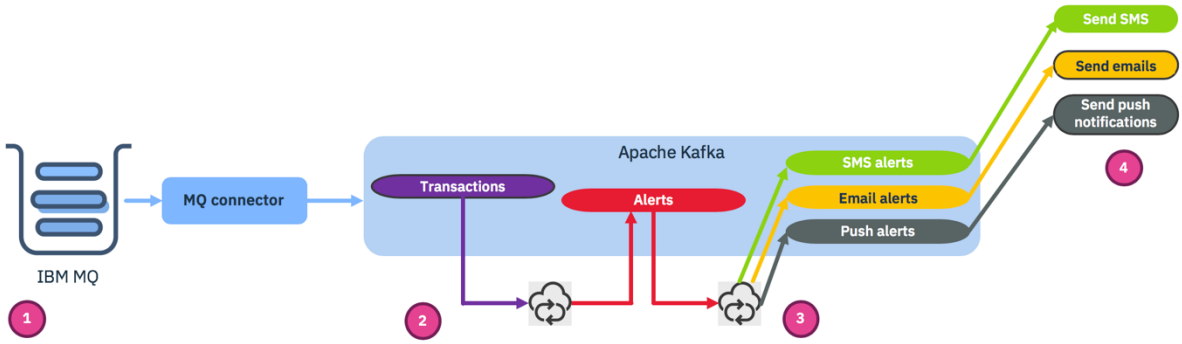
Basically, we just cast the message to a `TextMessage`, get the (JSON) text out of it, and send that to our `notification` microservice (which you'll see in our *Istio* section has two versions: one that posts to a Slack channel, and one that sends a tweet). Of course, this `onMessage` method is getting called on a separate thread (actually, in a whole separate address space/pod), under a separate transaction, from the code that sent the message. Even if our MDB isn't

deployed or is having a problem, that won't affect the main-line path in `portfolio` that sends the messages; they'll just queue up until our MDB is ready to process them.

Notice that there is the possibility of repeated calls to the REST API in unusual situations. The REST API is not transactional, and cannot be readily coordinated with the message's receipt in MQ. This is just part of the looser semantics that are normal in microservices. For the Stock Trader loyalty example, a simple way to overcome this is to make the REST API idempotent so that it can be called safely multiple times.

Financial Alert example

Imagine that Cash Inc want to modernize their customer experience in response to competitors. They already have messages that represent completed financial transactions in an MQ system and they want to hook this up to their customers' preferred notification mechanisms to alert them of any notable transactions. They decide to use an event-driven microservices pattern.



1. A connector is used to feed messages from MQ onto an Apache Kafka topic called "Transactions". Each event represents a completed transaction.
2. A microservice reads the transaction events and filters out those which are for small amounts or which occur regularly. The remaining events are published onto another topic called "Alerts".
3. Another microservice reads the alert events, correlates them with the notification preferences of the customers and publishes an another event onto a topic corresponding to that particular customer's preferred mechanism, either SMS, email or mobile push.
4. Microservices process each of the notification topics and notify the customers about their transactions.

There is a clear flow of events from topic to topic. Each microservice can be independently operated and scaled. Stream history means that a consumer joining a topic late has the option of processing the historic data.

Advanced Asynchronous Topics

This section is not intended to detail out all the these advanced application design patterns, but should mention the key ones and highlight our IBM support for these things.

This is where the realities of async need to go. Things such as dead letter queues, retries, message browsers must be covered here. There will be some specifics to the IBM platform here to cover.

Application Modernization and Cloud Native

Modernization is a large topic taken on for a number of reasons. This section links modernization to the cloud native programming model in a number of ways.

Figure 3- Modernization Journey shows the big picture of modernization:

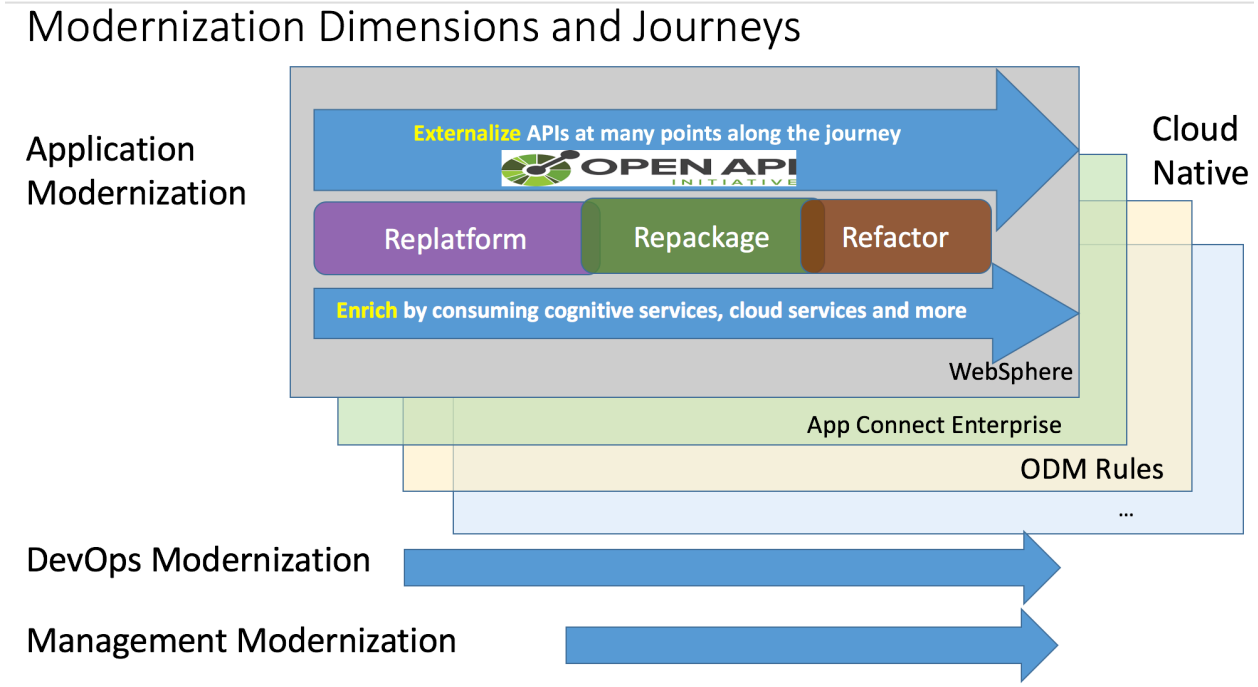


Figure 5- Modernization Journey

There are many dimensions to consider when taking on a modernization effort. First and foremost, from a programming model perspective is that applications will need to be modernized. That application modernization journey will include Externalize, Enrich, Replatform, Repackage, and Refactor activities. Application modernization is then complemented by DevOps modernization and Management modernization throughout the entire modernization process as shown in the Figure 3- Modernization Journey. This diagram shows that application modernization alone often cannot deliver the desired benefits.

Conceptually, this approach applies to all kinds of existing systems, whether they be WebSphere Application Server-based or are constructed using other tools and middleware. Note that on the right side of the diagram lies the words "Cloud Native". This is to suggest that as applications progress through the modernization journey, they take on more and more elements found in a cloud native programming model. Essentially, the goal of application modernization from an application architecture and programming model perspective is to pursue a cloud native and microservices based structure. The rate and pace of that evolution is gated by the benefits enjoyed by those modernization activities. Some applications may not be completely modernized. However, the cloud native programming model should serve as the guide or direction in which those application modernization activities should drive toward.

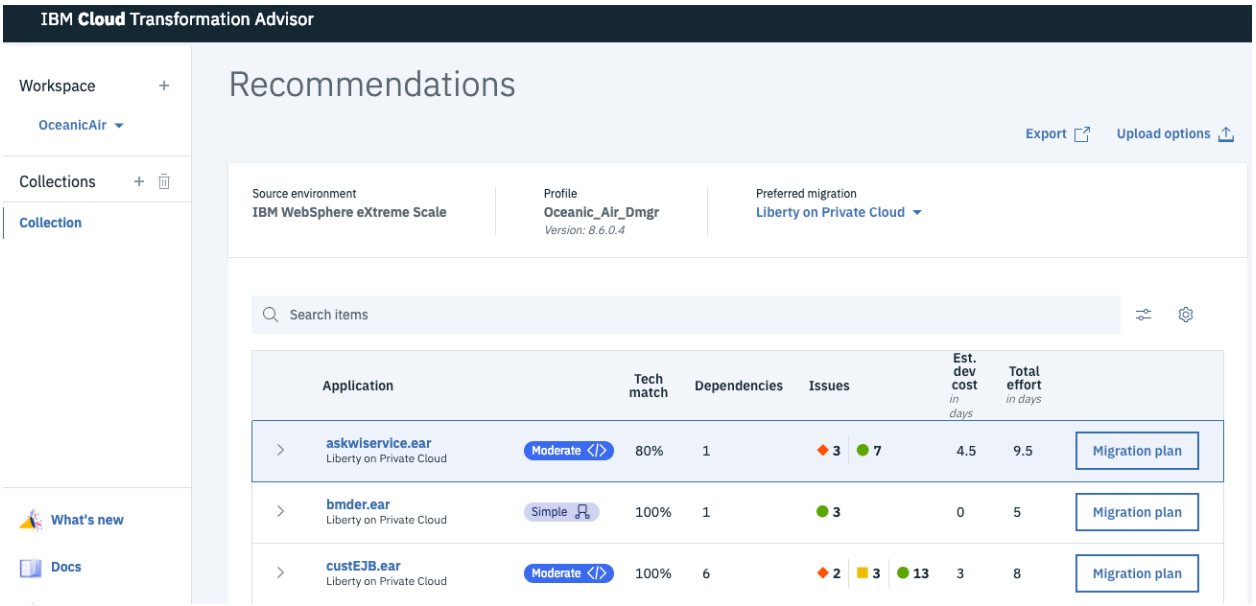
With that big picture outlined, each of the application modernization topics deserves a bit of additional detail. Specific focus on the programming model elements leveraged during each

activity is also provided. There is a mixing of techniques already described previously in this book combined with complementary and supplemental techniques needed to properly deal with that existing application, in whatever form it might be in and wherever it might be along the application modernization journey.

Replatform, repackage, refactor, externalize and enrich can be done in any number of sequences and from a variety of starting points. The order chosen for the remainder of this chapter is more optimized to efficiently explaining these things versus being suggestive of a precise order. Replatform is first and is the followed by repackage. These activities affect the program logic and likely the packaging of that logic from both a source code and configuration file perspective. Externalize works best within a replatformed and repackaged structure showing how to expose the value of the evolved capabilities. These externalized capabilities are then not only building blocks for to be used for new business capabilities, but also are enablers of some of the refactoring patterns techniques that are described. This is why refactoring and enriching coming after externalizing makes the most sense when explaining this overall modernization topic.

Introducing IBM Cloud Transformation Advisor

IBM Cloud Transformation Advisor (TA) is a free-to-run offering available with IBM Cloud Private which helps you modernized your JavaEE and MQ applications and get them running in the cloud (either public or private). TA gathers data about your JavaEE application (WebSphere, JBoss, and WebLogic are all supported) or MQ infrastructure and provides you with an at-a-glance table to see how much effort is required to bring those applications to a new cloud-based runtime.



Transformation Advisor also generates modernization artifacts, such as WebSphere Liberty configurations, Dockerfiles, Kubernetes deployment.yaml, etc., which help kick-start the modernization journey.

In each of the upcoming sections covering replatform, repackage, and refactor will highlight the features available in Transformation Advisor which provide guidance with each technical aspect of modernizing.

Replatform

Introduction

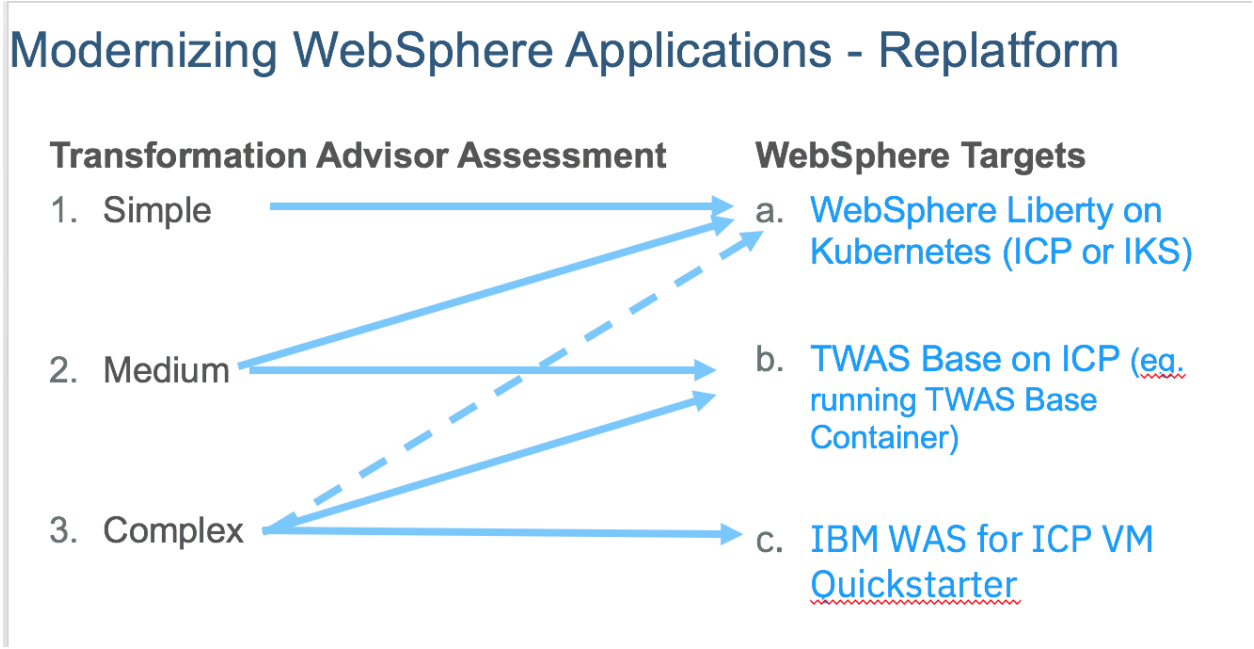
Replatform, at the highest-level means "get the application running on containers with the minimum amount of change." We know the most about this in the context of WebSphere (WAS), where the replatform is a little more complex because we're trying to reach WebSphere Liberty from Traditional WAS as our lead option. For many other runtimes, no application coding changes should be the norm for replatform. This aspect modernizes the execution environment but does not fundamentally alter the application logic or the size and shape of the executable units.

Mapping from Traditional WebSphere to WebSphere Liberty

Kubernetes introduces a number of concepts that are essential to understand and leverage when replatforming a traditional WebSphere application. Most of these concepts have been described earlier in this programming model guide. There a number of things that Traditional WebSphere did, especially WebSphere Network Deployment that map pretty easily to concepts in Kubernetes.

Replatforming Options.

While the conceptual model of Replatform is clear, the detailed options and underlying mechanics require some more detailed explanation. For WebSphere, there are now a couple of replatform options to consider. The following figure provides a way to look at these options:



While the classification of traditional WebSphere applications into categories of simple, medium and complex by Transformation Advsior is not perfect, it provides a starting point to consider the

options. These classifications are based upon the amount and type of source code changes required to move to change the application to support your targeted cloud and runtime platform. For example, applications with minimal to no code changes and targeting a private cloud are classified as Simple since they should run in Liberty as-is. Conversely, if they application is moving to public cloud that is immediately classfied as complex as there may be connections back to services within the corporate firewall that must be negotiated. A few other examples: an application using JAX-RPC and staying in the corporate data center is classified as moderate and an application making use of WorkManager-based asynchronicity (again within the data center) is classifed as complex. These classifications are a best-guess based upon IBM Lab Services field experiences. More information about application classification and complexity can be found in the IBM KnowledgeCenter at https://www.ibm.com/support/knowledgecenter/SSAW57_9.0.0/com.ibm.websphere.nd.multiplatform.doc/ae/covr_modernapp.html

The primary option for 'Simple' applications is to target running WebSphere Liberty based containers. This is also the recommended approach for 'Medium' complexity applications. As applications move from 'Simple' to 'Medium' and 'Medium' to 'Complex', the cost of moving to WebSphere Liberty based containers grows. Sometimes, as indicated by the dashed line, the cost will exceed the benefits to be gained. Therefore for many of the 'Complex' applications and some of the 'Medium' applications, IBM supports running traditional WebSphere Base in a containerized format. For some of the most complex applications, there is another option, which does not leverage containers fully, but also provides a modernization path to be considered. For more on the IBM WAS for ICP VM Quickstarter, go here: https://developer.ibm.com/recipes/tutorials/setting-up-websphere-application-server-on-ibm-cloud-private-with-the-was-vm-quickstarter/

The upcoming sections will go deeper into these first two options. First, the 'Replatform to WebSphere Liberty' section will deep dive on many of the important things that might be encountered on that journey as it relates to the programming model. Then, the 'Replatform using the WAS Base container' will explain this approach and essentially explain what this means in terms of the programming model artifacts.

If you need a deeper dive into the evoluton of Traditional WebSphere over the years and insights as to what changes and additions have taken place, go here: https://www-01.ibm.com/support/docview.wss?uid=swg27008724&aid=9

Replatform to WebSphere Liberty

From a programming model perspective, the Transformation Advisor tool plays a role in describing the programming model that is necessary to have the application run in WebSphere Liberty. In many cases the changes in the source code that are necessary involve moving from an older API or construct to one that is supported by WebSphere Liberty.

The following sections cover some topics related to getting from the old world to the new cloud native world. The Transformation Advisor tool has significant advice and guidance. What follows is not meant to be comprehensive. It highlights some areas which are likely to be commonly encountered during modernization efforts and some areas where simple mappings are

not available. Some of these topics are more aligned towards just updating an application to the latest standards while others are necessary to achieve proper execution within the context of a container execution. Each application being modernized will have a unique set of existing conditions to be assessed and updated. The list of source code changes required by an application can be found in the individual application report page of Transformation Advisor by clicking on the application name and reviewing both the list of code changes here and on the much more detailed Analysis Report at the bottom. It is likely that a few of these topics will apply to your existing applications. The list to be elaborated upon from a replatforming perspective is:

1. Logging
2. Tracing
3. Session management
4. JAX-RPC to JAX-WS
5. Removal of WebSphere-specific extensions to JavaEE
6. Removal of WebSphere APIs and SPIs

Logging

There are a multitude of different logging frameworks available to application developers. Two of the more popular options are Splunk and log4j. Splunk, and other log aggregators and forwarders, take log data and pass it along to another service for storage and further processing. Log4j and other frameworks allow applications to define logging, debugging, and tracing points within application code while also allowing the data to be sent to a multitude of targets, including log files on a file system. Each of these has different considerations when modernizing to containers.

If you are using Splunk or some other logging platform you will need to install and configure the appropriate log forwarder in WebSphere Liberty. This may be as simple as adding a jar (or set of jars) to the classpath or it may involve installing and configuring an additional EAR or WAR file on the server.

Configuring a programmatic logging framework, such as log4j, can be more involved. If, for example, the existing logger writes to a specific location on the file system then that location must exist in the Liberty container. However, this raises an interesting question: how are these log files used? Remember that container file systems are ephemeral, and the contents will disappear if the running container is restarted. This provides an opportunity to investigate other ways of disseminating this logging data. Liberty containers running in IBM Cloud Private can forward their logged messages to the internal IBM Cloud Private logging framework (commonly called Prometheus).

In both cases, there are opportunities to modernize the logging mechanisms currently by evaluating the new logging technologies available and determining if they fill the niche your pre-existing technologies provide.

Tracing

Along with a logging framework, existing WebSphere applications will likely have a different and older tracing approach. It is likely that you will find use of the package

`com.ibm.websphere.logging` as the main logging mechanism. Typically, the WsLevel class will have been using Level.FINE, Level.FINER and Level.FINEST for tracing purposes. Here is an example of some logging that might be found in older WebSphere applications:

4. Use the isLoggable method to avoid creating data for a logging call that does not get logged. For example:

```
if (logger.isLoggable(Level.FINEST)) {
        String s = dumpComponentState();        // some expensive to compute method
        logger.logp(Level.FINEST, className, methodName, "componentX state
dump:\n{0}", s);
 }
```

Once an application contains calls to trace execution, the output is disabled until the specific trace is activated in the WebSphere Admin Console. Once enabled, the trace output is placed into the log file located at $WASHOME/<profileName>/logs/server/trace.log.
All of this mechanism will still work, in Liberty. However, these trace log files are hard to get at and not consolidated across pods. There is no federation such as that provided by the microprofile based approach. This is the key drawback.
The recommendation from a modernization perspective would be to pursue one of the options listed below:

1. Have TA identify these tracing statements in existing applications and suggest they be turned into the new tracing specification.
2. Leave them as is and see only these, on a pod by pod based (not federated). Kubectl exec into the pod. These are hard to get at. They won't hurt anything but they are out in the cold and isolated from the go forward way to do this.

There was also an eclipse plug-in that would generate rich trace statements at certain points (entry, exit, catch block), so when you are looking at old code, you might find evidence of this phenomena. Show example of this here:
<example goes here>
We probably need to broaden this topic, understand what https://www.ibm.com/support/knowledgecenter/SSEQTP_9.0.0/com.ibm.websphere.base.doc/ae/ttrb_addtrace.html really means and predict exactly what we might find and then show the target programming model.

Session Management

Legacy WebSphere applications will leverage and exploit session management in a number of different ways. A suitable approach must be provided for each different historical approach to managing session. If you are coming from WebSphere ND then there are built-in configuration options which handle session replication for you. When moving to Liberty in containers a new solution is required, typically with Hazelcast as the session storage provider.

The other assumption that becomes relevant is that replatforming also implies leveraging the ingress capabilities of ICP or IKS and removing the request routing role of the web server (eg.. IHS) from the equation. Sticky sessions, for example, need to be configured within the ingress to ensure proper routing.

JAX-RPC to JAX-WS

Liberty does not provide an out-of-the box JAX-RPC runtime for those older webservices. This leaves you with three options, as suggested by Transformation Advisor:

1. **Package a JAX-RPC provider with your application**, such as Apache Axis 1. This is the simplest path with no code change (probably) but may introduce security risks.
2. **Migrate to JAX-WS**. This will require code changes on both the client and server to support JAX-WS. This is an option if you can change both sides at the same time
3. **Migrate to JAX-RS**. Similar to JAX-WS but using JSON instead of SOAP/XML messages.
4. Leave the JAX-RPC as-is in tWAS and follow either recommendation 2 or 3 to create new versions of the services in Liberty while planning to phase out JAX-RPC over time.

Removal of WebSphere-specific extensions to JavaEE

As WebSphere and JavaEE progressed point-in-time shortcomings to JavaEE were solved by vendors in different ways. For example: WebSphere 7 solved the "Do not start threads in JEE" problem by including the WorkManager API as a sanctioned way to perform asynchronous operations without resorting to JMS and message-driven beans. As JavaEE continued to evolve a new concurrency spec was introduced and the old WorkManager API was deprecated (but not removed) from traditional WebSphere. Liberty started from a fresh slate so there is no WorkManager API Liberty. This means the code must be updated to use the JavaEE concurrency API to continue functioning. Another example is that for some older web services IBM Rational Application Developer generated stubs and skeleton files which include `com.ibm.websphere.webservice.*` classes. Again, this was to address a shortcoming (at the time) of the various specifications WebSphere implemented at the time but is no longer an issue. IBM Transformation Advisor provides alternatives and suggestions for many of the SPI and API usages identified in a given application.

Removal of WebSphere SPIs and APIs

Similarly, to the WebSphere-specific extensions there are features in WebSphere that IBM made available or included as part of any generated artifacts which were tied to the WebSphere runtime. One example is the unit of work API at `com.ibm.wsspi.uow`. This SPI was removed from Liberty and replaced with a different transaction management API. Again, the code making use of this SPI must either:

1. Be rewritten to make use of the Liberty API which provides similar functionality
2. Be rewritten to use some 3rd party library which provides similar functionality
3. Be left in place on tWAS

IBM Transformation Advisor provides alternatives and suggestions for many of the SPI and API usages identified in a given application.

Evolving the Pipeline

Getting an existing application replatformed requires more than just the necessary code changes to allow WebSphere Liberty to be the target runtime. The pipeline leveraged to build replatformed applications will need to be updated and adjusted as well. From a programming model perspective, this does affect the artifacts that developers must be aware of.

Replatform using the WAS Base Container

In December 2018 IBM released a WebSphere Base container to Docker Hub. This new platform fills a gap when moving 'Moderate' or 'Complex' WebSphere applications to containers because we're no longer changing the underlying application server from Traditional WebSphere to Liberty. This means there are likely few to no source code changes required when moving the application to containers. However, if the source versions of WebSphere being analyzed are very old, then there will still be a set of APIs and SPIs no longer supported in the newest WebSphere Base release. The WebSphere Base support is for a container based on WebSphere Base 9.0 and beyond. A list of WebSphere deprecations is located in the KnowledgeCenter at:

https://www.ibm.com/support/knowledgecenter/en/SSAW57_9.0.0/com.ibm.websphere.nd.multiplatform.doc/ae/rmig_depfeat.html

while a list of WebSphere removals is available here:
https://www.ibm.com/support/knowledgecenter/SSAW57_9.0.0/com.ibm.websphere.nd.multiplatform.doc/ae/rmig_remfeat.html

Using the WAS Base Container as the target of a replatform does affect the programming model in the area of the pipeline, similar to that of moving to the WebSphere Liberty container. There will need to be proper creation of the docker images and those artifacts that help put these containers into our Kubernetes environments. For example: building EAR/WAR files is only the first step of the overall build pipeline. Now you must build, configure, and install the application into the WAS Base container. Things to consider include:

- Where is the dividing line between the 'base' container configuration and the application-centric configuration? What artifacts/resources are used by all applications and which are the concern of individual applications?
- Overall ownership and maintenance of the 'base' container image
    - Installation and configuration of any 3rd party monitoring agents. Are they needed any more or is the built-in Kubernetes/IBM Cloud Private monitoring capabilities enough?
    - Configuration of any shared libraries or resources. Shared Libraries could be removed and merged back into the application if you stick to the "one application to one container" methodology. Global resources could be removed and a more application-targeted tWAS configuration could be created and maintained.
    - Security configuration along with trust and keystore management
    - Basic tuning and configuration information (threadpools, heap size, etc)
- Configuration of application specific resources
    - Installation of the application and any application
    - Configuration of application-specific resources such as JDBC connections, JMS resources, etc.
    - Finer-grained tuning for a given application, such as customized threadpool settings.

Thankfully many of the existing WebSphere wsadmin management scripts can be reused with some tweaks with the WebSphere Base container. For example, if you have WAS ND wsadmin

scripts you'll need to alter the scopes within the scripts to work with server scope instead of cluster.

There are also considerations on how you manage running WebSphere Base containers. A good reminder is that you must now treat any administration change as code which must be committed and included in a Docker build. While you can access the WebSphere administration console within the container and make changes to the configuration it's important to remember that any changes are tied directly to that running instance. If you're running the container within Docker on a developer's laptop those changes are tied directly to a running container instance..The implication here is that if you restart the same docker container (docker run ad9d902) your changes are persisted but if you create a new container from ad9d902's source you will not see the changes from ad9d902. If you're running in Kubernetes, any configuration changes are lost once a pod goes away (due to a restart, a pod dying, etc). These occur regularly and naturally in most Kubernetes based systems. The WebSphere admin console now becomes a way to generate wsadmin scripts (via the context assist functionality) which can then be fed back to the `docker build` steps which bakes that new configuration into the WebSphere docker container.

Repackage

Repackage simply says look for ways to break monoliths into pieces that are suggested by the programming model. For WAS, this means breaking monoliths into macroservices that package servlets separately into different WAR files or EJBs into different EAR files versus having many of each all bundled together. Usually, this means no code change. For other things in our hybrid cloud integration portfolio, they also have higher level programming models such as defining rules, defining integrations, or processes which can be split apart into the various different, native, packaging mechanisms supported by each.

Repackage is suggested by the existing programming models and frameworks. Even underneath our graphical tools, there are files with such language elements in them. Common repackaging challenges and the associated techniques to consider when facing these challenges are described in the upcoming sections. The following will be covered:

1. Traditional Websphere – shared library handling
2. Traditional WebSphere – 3rd party Jar handling
3. Splitting EARs into constituent WAR and EAR files

Shared Library Handling

Shared libraries were introduced in traditional WebSphere to work around limitations and realities of the JEE packaging model. For example, you may have applications A, B, C, D, E and F all installed on the same system, but applications A-D depend on jars 1-3 while E and F depend on 4-6. If these are large jar files with many classes, operators and administrators would notice the same classes loaded in heap within different classloaders, creating bloated heap. One solution would be to place all those JARs on the JVM classpath but you're then allowing applications E &F access to those jars which can cause unintended consequences. Shared libraries allow operators to define a set of jar files or directories which an application depends on but without packaging those files with the application.

When moving to containers keeping the mental model of one microservice or application per container provides a few different options for moving shared libraries. First, and simplest, you can continue to use the shared library construct in Liberty by using the `<library>` and `<classloader>` elements to define the shared library and the mapping of that library to an application, respectively. For example:

```
<library id="someLibrary">
   <!-- Location of XML and .properties files in the file system for easy
editing -->
   <folder dir="${server.config.dir}/editableConfig" />

   <!-- Location of some classes and resources in the file system -->
   <folder dir="${server.config.dir}/extraStuff" />

   <!-- A zip file containing some resources -->
   <file name="${server.config.dir}/lib/someResources.zip" />

   <!-- All the jar files in ther servers lib folder -->
   <fileset dir="${server.config.dir}/lib" includes="*.jar" scanInterval="5s"
/>
</library>

<application location ="webStore.war">
   <classloader commonLibraryRef="someLibrary" />
</application>
```

However, if you remember that the original creation of shared library was to prevent unnecessary heap usage and if you stick to the "one application/microservice per container" then there is no longer a need to utilize shared libraries as there is no fundamental difference between a server with a single application with a single library and a single application containing all its dependencies within a single package. The second option is to package those shared library dependencies with the application. This requires changes to the build process but there should be no behavior difference between the shared library and large all in one packaged application.

3rd Party Jars

This is a variation of one of the earlier replatform changes when moving to Liberty. Traditional WebSphere packaged many open-source and third party jar files as part of the runtime. Features such as log4j or Apache Axis are commonly-used third party functionality which, while included in traditional WebSphere, are no longer available or no longer exposed in Liberty. The solution here is to include those third-party jar files as part of the application.

Splitting into separate WAR and EAR files

Features of the programming model provide seams from which we can tease apart the application into different packages. Historically we see JEE applications containing a heterogeneous mix of EJBs, servlets, message driven beans, and web services. In some cases, these artifacts are packaged together purely for convenience purposes and there is no technical linkage between the pieces and parts. In these cases, it is possible to split these artifacts into

separate packages and, ultimately, into separate Liberty instances (and Docker containers). This also allows lifecycle decoupling which was not possible when the various artifacts were still packaged together in a monolith.

Externalize

Externalize at the highest level of the modernization journey is all about making the business value of an application easily available to other consumers.

At any point along the replatform, repackage and refactor journey it is reasonable and expected that the business value contained in an application be **externalize**d as APIs. This allows new applications to leverage the capability in exposed in an easily accessible way. Typically, this activity will involve having a REST interface. This will be an OpenAPI specified REST interface. This then allows the interface to be discovered and managed. If this is done early in the modernization life-cycle, it can insulate consumers from further modernization activities which are undertaken in pursuit of a more complete cloud-native implementation.

There are a few programming model considerations and approaches to cover when externalizing the value of an application that is somewhere on the journey to being a modernized application. These approaches and techniques are described in more detail in the upcoming sections:

1.

Externalize – REST Approach and Considerations

One approach is to do a top-down definition of the desired REST API as explained earlier in this book. This essentially involves creating a new microservice ……..point back to JAX-RS section.

Notes if we assume old Java code is still running LTPA:

1. One way to do this is to take the API Key in the JAX-RS implementation Java code and somehow create what's needed to talk LTPA to the downstream Java code. To do this, you need the server's private key.
2. The next alternative is have API Connect terminate the API Key based request and magically create an LTPA token to go downstream. The implementation of this would need your private key. You probably want API Connect running in the same place as the implementations. You don't want this in your DMZ as someone might get your private key.

Notes if we assume old Java code has been modernized to JWT and we get an API key

1. One way to do this is to take the API Key in the JAX-RS implementation Java code and somehow create what's needed to talk JWT inside an http authorization header to the downstream Java code. To do this, you need the server's private key.
2. The next alternative is have API Connect terminate the API Key based request and magically create an JWT token to go downstream.

Need to cover security implications and constraints. What is exposed to consumers? What is used in the new microservice versus that is in the existing code that has the implementation that we want to expose? What role does API management play here? What's the minimum code way of doing this?

Next, we need a section that talks about a more bottom up API, for example taking SOAP interfaces that exist and make them into REST APIs. Do we do some mechanical bottom up and write some code? Do we do this all in API Connect? Are there parameter mappings that to be done? What code goes with this? How much of that can I do in API Connect?

Externalize – Queues and Topics

In an earlier chapter, the role of messaging in microservice to microservice communications was explained. This was mostly done in the context of building out new microservices that are loosely coupled.

When considering a modernization scenario, the context is different. There are likely to be existing queues and topics, which are being used to loosely couple the existing system. These are externalizations of existing capabilities. From these, both refactor and enrich modernization tasks can essentially hook in and enable various things to happen.

Let's look at enrichment first. For example, a message on a topic can easily be read by multiple consumers. Note, there might be a need to study the current messaging architecture and change a queue to a topic so that multiple consumers are enabled. From there, create a new microservice that either provides new business value itself or is the starting point to a composition of services that add enriched value. One can imagine hooking onto these queues and starting analytics activities, loading data lakes, training machine learning models and more, just based on hooking into the messaging infrastructure and the data which is already available. Current producers and consumers should not have to change.

Messaging can also be a critical element in refactoring. In situations where parts of a system are being re-written into microservices and parts remain the same, at least during the strangling or during the transition, a messaging approach can be applied in a few ways:

1. Producer unchanged, consumer is new refactored code - Similar to the enrichment story, a replacement set of function can be launched via consuming messages that already exist, but this time instead of the original consumer.
2. Synchronizing data stores that enable reads on the new datastore – If a refactoring effort is targeted at first replacing the read and reporting types of functions, hooking into existing messaging infrastructure or even modifying existing systems to produce messages can help create a new data store or subset of datastores for new reporting and read oriented operations to run against.
3. Synchronizing data stores to preserve reads on the old data store – This inverse of the above is also possible.
4. Synchronizing data stores that receive different update actions – If refactoring of an existing system leaves a state where there are two data stores, an old and a new, with existing code driving changes to the old data store and the new refactored code driving changes to the new

datastore, a messaging approach can be used to keep both data stores up to date. This is the extreme case.

In almost all, if not all cases enumerated above, there is a need for guaranteed delivery and once and only once delivery. When in a transition between an old data architecture and a new data architecture, preserving integrity is mandatory.

Externalize – Additional Considerations

Strong consideration must also be given for the non-functional requirements (nfrs) that accompany exposing existing capabilities. Is the current business value properly able to handle an increased volume of traffic? The answer might be no or that it is unknown, especially if the application being modernized is early in the replatform, repackage and refactor journey. This means that these nfrs might require visiting or revisiting timeouts, fault tolerance and caching.

Refactor

Refactor means getting serious about the really big monolithic things and refactoring them to be cloud native. This is where we tackle the huge .java and .js files that exist with our clients. This is where we approach cloud-native. This may be more important for BPM and TWAS than others, and we are still thinking about refactor in those other programming models. Repackage might be enough for some. Specific topic to be covered related to Refactoring include:

1. Front End Modernization
2.

Front End Modernization

Enrich

At any point along the replatform, repackage and refactor journey it is possible to **enrich** an existing application by consuming new services. Typically, this will involve having the existing application consume a REST interface. There are a wide variety of services available in the cloud via REST API, including cognitive services and various industry and domain services. At the simplest level, this means applying the techniques described in "Consuming APIs" earlier in this book. Once APIs are invoked and responses are received, the rest of the programming model and approach described in this book should be used to enrich the application as necessary. This might mean using new datastores, combining existing data with the results of API calls and adding new interfaces for consumers to get at this enriched value

Managing Multi-Component Applications

Intro

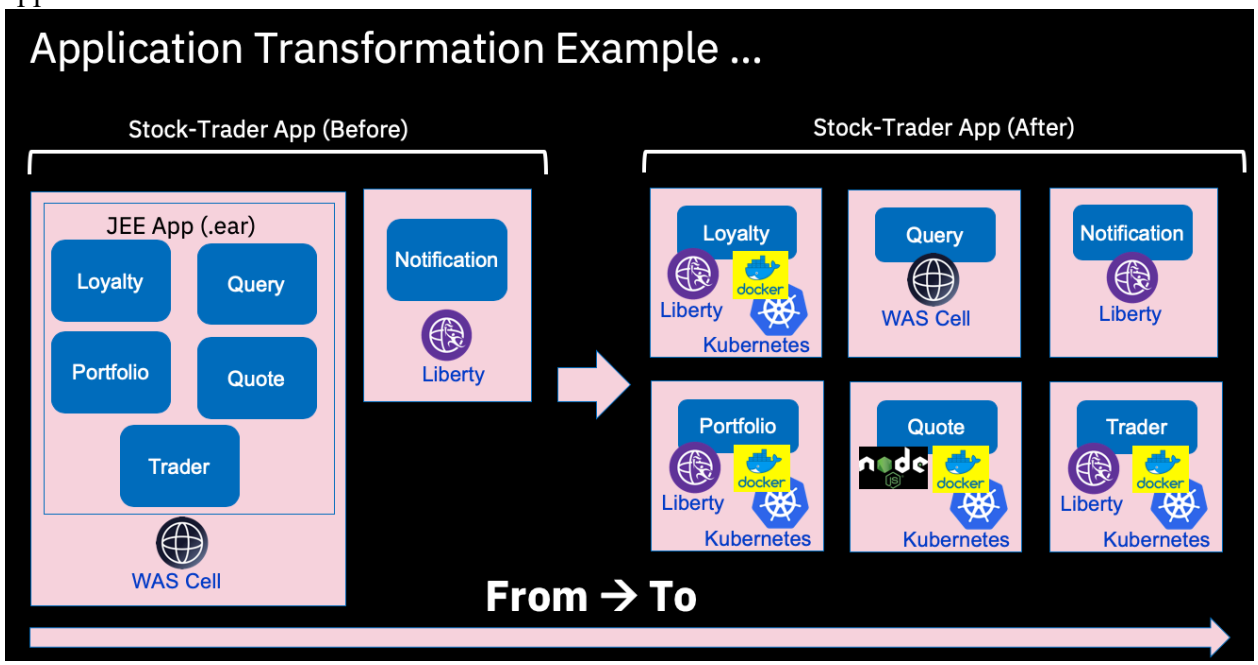
As mentioned earlier in the end-to-end overview, most cloud native applications are composed of multiple components: microservices, container-based middleware, and even existing VM-based

middleware. Because of this complexity, there needs to be a way to gather all of these components into a single meta-object that represents the application in one view.

New in IBM Cloud Private is the "Application Navigator" (tech preview), a helm chart that provides visualization for applications.

This is an optional component, and you can always add the labels and annotations described below without installing "Application Navigator" because even basic labeling will help you run Kubernetes commands to view resources with a common label. That said, Application Navigator does provide far more value for you to consider.

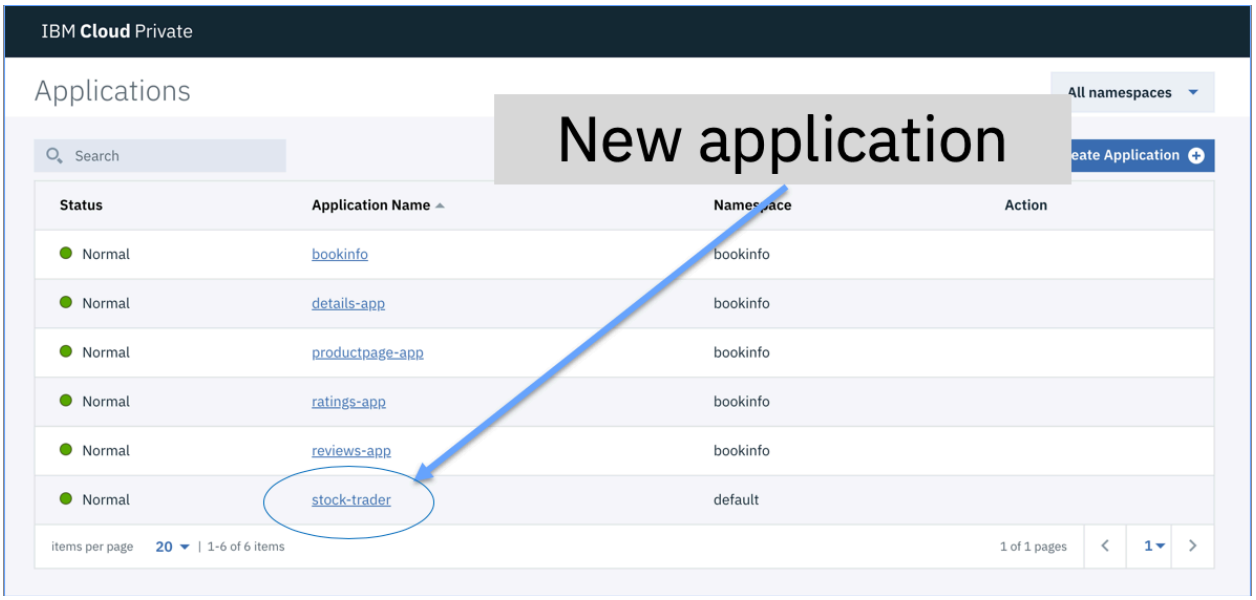
For example, let's take a "Stock Trader" application. On the left is what it looked like running in traditional WebSphere, and on the right is how it was transformed into a (mostly) containerized application.



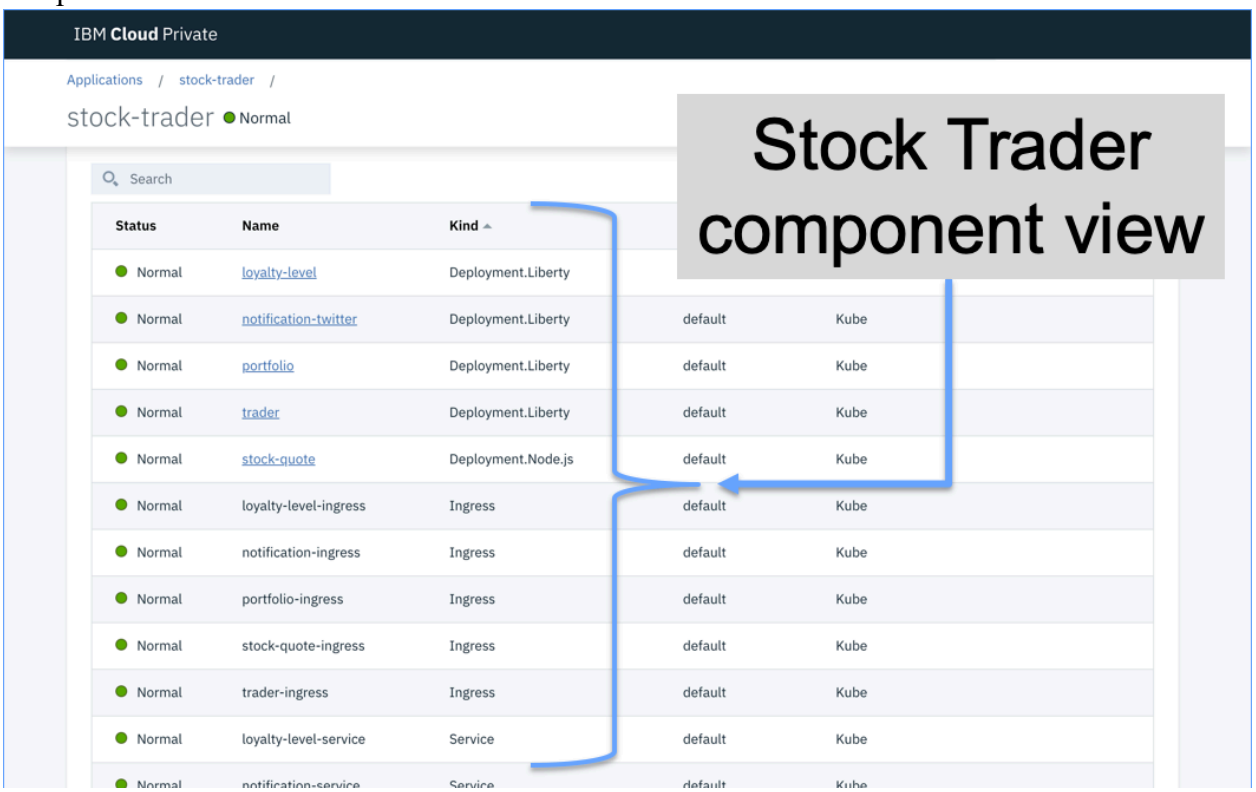
The resulting application can now take advantage of Kubernetes, Liberty, and polyglot runtimes, but as you can see, it has also become more difficult to manage because the logical application is composed of many different components spanning multiple architectures and runtimes. As a result, there are many different logging sources, monitoring views, and when you get a call saying "My stock trader app doesn't work right", it can become quite a job to troubleshoot.

With Application Navigator, you can quickly see a holistic view of all your applications running in your cluster, and then click to drill down to see individual components.

In this view I can see my applications:



In this view I can see individual components of my Stock Trader application, even the "Kind" of component.



The above graphics show a summary of the value of Application Navigator.

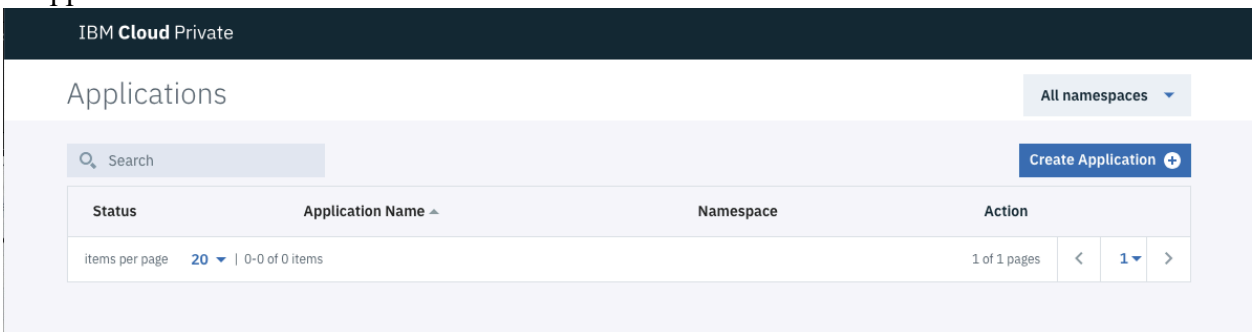
In the following sections, learn the steps required to prepare your own application to use Application Navigator.

Install Application Navigator

To get started, follow the instructions here: [Application Navigator Installation Instructions]
It's fairly straight forward:

- Create a namespace called "prism" (do it from the UI and add the pod security policy "ibm-anyuid-psp")
- Run the helm command:
  ```
  helm install --name=application-navigator --namespace=prism app-nav-helm-chart/stable/prism --tls
  ```

Once installed, you will see a helm release with a "Launch" button and you'll get an empty list of applications:



Next we will add your application, starting with the "Application" resource, which is the resource defining your multi-component application.

Define Your Application Resource

Application Navigator will look for all "Application" custom resources and use them to scan the rest of the Kubernetes cluster to find any resources with that label. [Learn more about Application resources]

Here is how the Application resource is defined for "Stock Trader" [View in GitHub]:

```yaml
apiVersion: app.k8s.io/v1beta1
kind: Application
metadata:
  name: "stock-trader"
  labels:
    app.kubernetes.io/name: "stock-trader-app"
    app.kubernetes.io/version: "3"
spec:
  selector:
    matchLabels:
     solution: "stock-trader"
  componentKinds:
    - group: deployments
      kind: Deployment
    - group: TWAS
      kind: WebSphere-Traditional
    - group: Liberty
      kind: WebSphere-Liberty
```

```
    - group: ingress
      kind: Ingress
    - group: service
      kind: Service
```

Notice a few things:

- "metadata" is where you give your application a name, and where you give the application labels (the internal app name and the version in this example).
- "matchLabels" is where you give it the label you need to match in order for a component to become a member of the "Stock Trader" application. In this case, you're telling Application Navigator to add any Kube resources with the label "solution: "stock-trader""
- "componentKinds" lets you define the kinds of resources Application Navigator should look for.

Label your Kubernetes Resources

For each Kubernetes deployment, you will need to label the deployment and annotate to clarify what kind of deployment. View in GitHub

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: portfolio
  labels:
    solution: stock-trader
  annotations: {
     prism.subkind: Liberty
  }
```

In this example, our "portfolio" deployment is written in Liberty so Application Navigator provides a subkind of "Liberty" so its default actions can be used. Further, additional resources like Ingress, Service that are created for "portfolio" to work are labeled so they show up in the view as well.

By default, Application Navigator can optimize the UI (links to details, actions to launch to management consoles) when you label the following resources:

- Deployment
- Ingress
- Service
- Pod

If you want to add other types, simply label the resources. If you want to customize the details link or the action menu, then you need to create a configmap specifying the kind of Kubernetes resource you want Application Navigator to include. For example, here is a link to a configmap-statefulset.yaml that tells Application Navigator to add actions to any StatefulSet (in our case, it adds a details link of the name of the StatefulSet).

Create and Label your "Non-Kubernetes" Resources

For "non-kube" resources like VMs, SaaS APIs (Watson services for example), you can create custom resources that represent that external resource, along with a link to get to its management UI. Application Navigator provides the following custom resource definitions by default:

- WebSphere-Liberty
- WebSphere-Traditional

Steps to create your own:

- Create a custom resource definition (CRD) Learn more
- Create a custom resource, add your application label, and annotate with any menu items you want to add to help troubleshoot.

In this example, a "Watson-Tone-Analyzer" CRD is being created. The CRD is not the resource, just the definition: View in GitHub

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  creationTimestamp: null
  labels:
    api: default
    kubebuilder.k8s.io: 0.1.10
  name: watson-tone-analyzers.prism.io
spec:
  group: prism.io
  names:
    kind: Watson-Tone-Analyzer
    plural: watson-tone-analyzers
    singular: watson-tone-analyzer
```

Tips:

- "name": must use the "plural" version of the names
- "group": use "prism.io" since that is the current Application Navigator API name.

In this example, a "Watson-Tone-Analyzer" resource is being created, labeled with "stock-trader" application, and URL to manage this resource is added. View in GitHub

```
apiVersion: prism.io/v1beta1
kind: Watson-Tone-Analyzer
metadata:
    name: watson-tone-analyzer
    labels:
        solution: stock-trader
    annotations: {
        prism.platform.kind: Public,
        prism.platform.name: IBM-Cloud,
        prism.platform.console-url: "https://cloud.ibm.com/services/tone-
analyzer/crn:v1:bluemix:public:tone-analyzer:us-
south:a%2fb71ac2564ef0b98f1032d189795994dc:195be178-5f78-467f-8355-
99ee5772f4c2::?paneId=manage"
```

```
    }
```

Once you define your custom resource, you need to create a configmap.yaml to tell Application Navigator what menu items you want to show in the UI. In this example, the configmap is telling Application Navigator that it should show one menu item called “Service Instance” which will launch the URL specified in the specific custom resource (notice how the annotation “prism.platform.console-url” is used in both places):

```
apiVersion:  v1
kind: ConfigMap
metadata:
  name: prism.actions.watson-tone-analyzer
  namespace: prism
  labels:
data:
  url-actions: |
      [
        {
          "name": "Service Instance",
          "text": "Service Instance",
          "description": "View service instance in IBM Cloud",
          "url-pattern": "${resource.$.metadata.annotations['prism.platform.console-
url']}",
          "open-window": "tab"
        }
      ]
```

Once you have created them, add the .yaml files to you deployment manifest (either your helm chart or manually deploy using kubectl).

Deploy Your Application

Finally, deploy your application.

Steps:

- Deploy Application.yaml
- Deploy Custom Resource Definitions
- Deploy Custom Resources
- Deploy your labeled microservices

Lastly, if you also want any containerized middleware you rely upon to be listed in the application view, you can label your middleware “pets” that your application will use. In Stock Trader’s example, we labeled ODM, Redis, MQ, Db2, and Event Streams with these label commands (substitute the resource name and namespace that you used in your deployment):

```
kubectl label deployments odm-trader1-ibm-odm-dev solution=stock-trader -n trader-odm
kubectl label statefulsets trader-redis-master solution=stock-trader -n stock-trader
```

```
kubectl label statefulsets mq-trader1-ibm-mq solution=stock-trader -n trader-mq
kubectl label statefulsets db2trader1-ibm-db2oltp-dev solution=stock-trader -n default
```

3 event streams stateful sets:

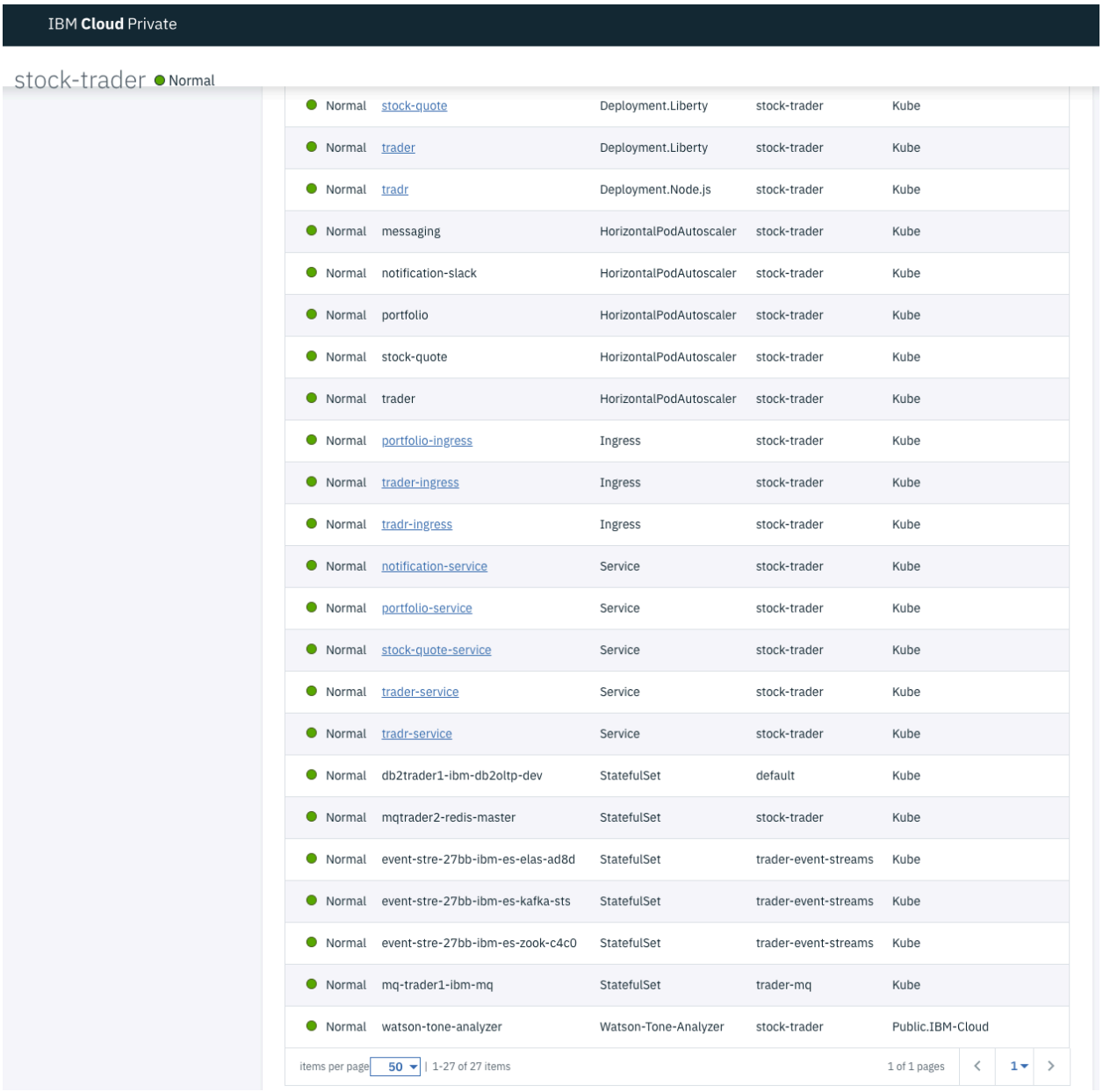
```
kubectl label statefulsets event-stre-27bb-ibm-es-elas-ad8d solution=stock-trader -n trader-event-streams
kubectl label statefulsets event-stre-27bb-ibm-es-kafka-sts solution=stock-trader -n trader-event-streams
kubectl label statefulsets event-stre-27bb-ibm-es-zook-c4c0 solution=stock-trader -n trader-event-streams
```

Limitation*: Currently, labels can have only one value, so you can label middleware to be used by one application at a time.*

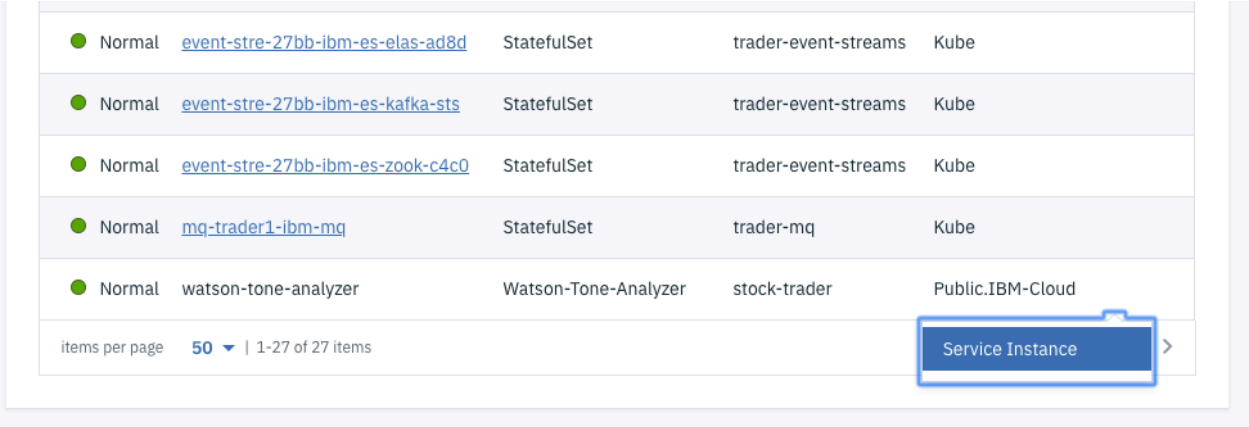
The end result looks like this: All components contributing to Stock Trader showing in this single view:

- Stock Trader microservices Deployments
- HPAs controlling Stock Trader auto-scaling
- Ingress for Stock Trader
- Services for Stock Trader pods
- Middleware Deployments and Stateful Sets
- External resources Stock Trader is dependent on

This graphic shows a diverse set of components that make up the application. Each kind of component has different actions which can be applied to it. An application also is made up of components that can transcend a specific namespace.



Finally, the following image shows a custom action added to the Watson service called “Service Instance”, which launches into the IBM Cloud UI to view the specific instance the application is using.



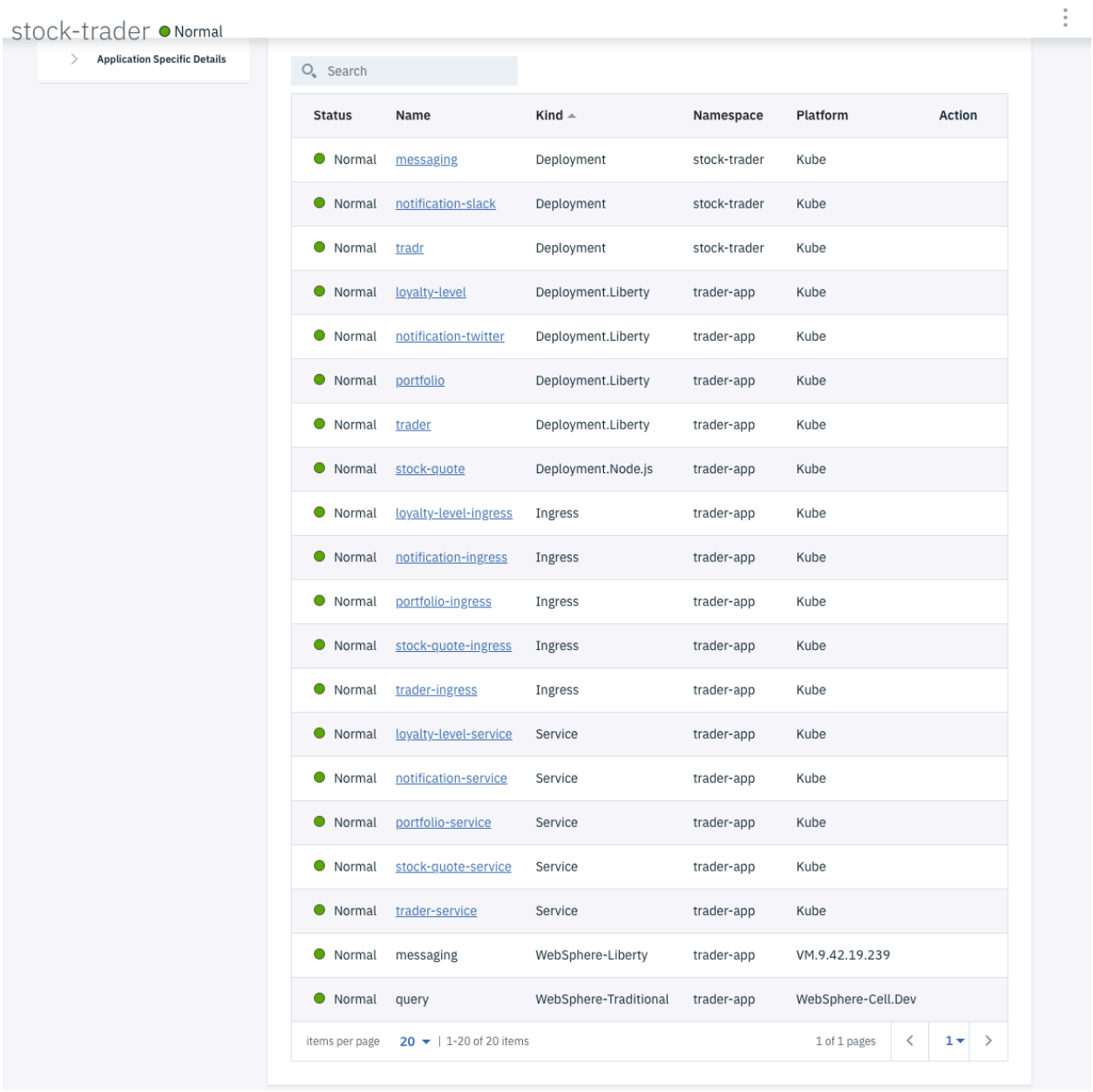
Deploy Samples

Application Navigator comes with some basic samples: Learn more here. In this sample, a single helm chart contains the deployments for both the application custom resource as well as all the application components. In your case, you may want to separate the application deployment from individual microservices.

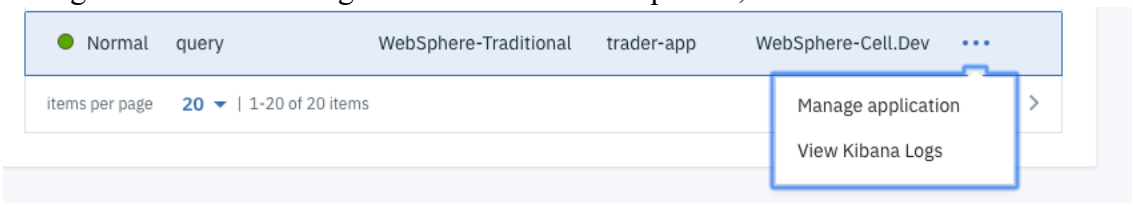
NOTE: If you want to use this sample, prepare your cluster by creating a namespace and use "cloudctl" to login and set context to that specific namespace.

```
helm install --name=stock-trader-appnav --namespace=trader-app
samples/stocktraderAppNavChart -tls
```

Now you can see all components for Stock Trader, including external components running in traditional WebSphere:



Even better, each component can add menu items so if the need arises to launch into the management console or logs for an individual component, the link is embedded into the menu:



Tools Support for the Programming Model

While a programming model of any sort should largely stand independent of specific tools, it is absolutely the case that for a cloud-native project to succeed, a proper and tooled approach to leveraging the programming model is essential. With automation as a foundation to doing cloud-native, how could it be any other way? Throughout this book, we have been as minimalist as possible when it comes to recommending and describing tools. We have focused on the artifacts. In this chapter, the tables will be turned, and tools are the featured subject. We will cover the spectrum of tools required and provide examples and suggestions along the way.

The tools we will discuss are:

- Helm
- Application Cloud Pak
- Microclimate
- Jenkins
- Knative

A key success factor for any cloud-native project is automation. Automation begins with a developer's first exposure to the programming model. It continues through the developer's authoring experience through to initial testing of a microservice. That testing activity necessitates tools that help to build and assemble the artifacts of the programming model into executables. Once initial testing is complete, additional automation via tools chains, will further drive a new microservices or a change to existing microservices through additional tests, scans and other processes until production is achieved. All along the way, there is tooling that enables feedback not only to operations, but also to developers. These tools are enabled by elements of the programming model which have already been described. The upcoming sections will pass through this development cycle describing the tools and how they enable the true velocity desired and expected from a proper cloud-native approach to development.

First, we'll look at Helm, often called the package manager for Kubernetes. This will let us gather up all of the microservices that collectively comprise our application and install them all at once, while also gathering any configuration data needed, such as how to connect to prerequisite middleware, like a database or a messaging service. We can also manage our application as a unit, by working with its *helm release*, and drive updates to it via the *helm upgrade* process.

Building upon helm, we'll next look at the use of an Application Cloud Pak as an accelerator and supplement to the core programming model. By leveraging an Application Cloud Pak, a number of typical use cases related to standing up applications in part or in total is simplified through a higher level of automation and specification than allowed by basic Docker and Kubernetes vocabulary.

Next, a more holistic of tools accelerators called Microclimate are introduced. Special care will be taken to ensure that the use cases addressed by Microclimate are called out and the additional value gained by using Microclimate is explained.

It is expected that consumers of the programming model will embrace anywhere from all of the tools described to an extreme case where none of them are leveraged. This selection will depend in part on the existing platforms which might be in use and how amenable the tools in those platforms are to supporting the needs of a modern cloud native programming model.

Helm

Most *off-the-shelf* or *packaged* applications for Kubernetes will be installed via *Helm*. At its core, helm is simply a basic set of standards for collecting various yaml files in a single unit, and having a way to parameterize values in those yaml files, so that you can pass in different values on different installations of a helm chart.

Archive format

A helm chart can be envisioned as a kind of like a special format of a zip file of yamls. In reality, it is a tarred gzip, or .tgz file, of the contents of a defined directory structure. Let's get specific and look at the contents of the Stock Trader helm chart:

```
Johns-MacBook-Pro-8:stocktrader jalcorn$ tar tvf stocktrader-0.1.0.tgz
-rwxr-xr-x  0 0      0         202 Dec 31  1969 stocktrader/Chart.yaml
-rwxr-xr-x  0 0      0        3597 Dec 31  1969 stocktrader/values.yaml
-rwxr-xr-x  0 0      0         264 Dec 31  1969 stocktrader/templates/NOTES.txt
-rwxr-xr-x  0 0      0         530 Dec 31  1969 stocktrader/templates/_helpers.tpl
-rwxr-xr-x  0 0      0        1984 Dec 31  1969 stocktrader/templates/config.yaml
-rwxr-xr-x  0 0      0        1591 Dec 31  1969 stocktrader/templates/credentials.yaml
-rwxr-xr-x  0 0      0        2979 Dec 31  1969 stocktrader/templates/messaging.yaml
-rwxr-xr-x  0 0      0        1059 Dec 31  1969 stocktrader/templates/notification-service.yaml
-rwxr-xr-x  0 0      0        2193 Dec 31  1969 stocktrader/templates/notification-slack.yaml
-rwxr-xr-x  0 0      0        2462 Dec 31  1969 stocktrader/templates/notification-twitter.yaml
-rwxr-xr-x  0 0      0        6591 Dec 31  1969 stocktrader/templates/portfolio.yaml
-rwxr-xr-x  0 0      0        2707 Dec 31  1969 stocktrader/templates/stock-quote.yaml
-rwxr-xr-x  0 0      0        2804 Dec 31  1969 stocktrader/templates/trader.yaml
-rwxr-xr-x  0 0      0        3797 Dec 31  1969 stocktrader/templates/tradr.yaml
-rwxr-xr-x  0 0      0        3902 Dec 31  1969 stocktrader/README.md
Johns-MacBook-Pro-8:stocktrader jalcorn$
```

You build the archive file by running `helm package <directory>` from just above this. For example, if you `git clone https://github.com/IBMStockTrader/stocktrader` and then `cd stocktrader`, you would run `helm package stocktrader`, which will produce the .tgz file, named for the chart name and version in the `Chart.yaml`.

The `Chart.yaml` simply specifies high-level info about the helm chart, such as its name, version, and icon. Let's take a look at the `Chart.yaml` for our Stock Trader sample:

```
Johns-MacBook-Pro-8:stocktrader jalcorn$ cat stocktrader/Chart.yaml
apiVersion: v1
description: A Helm chart for the IBM Stock Trader sample
name: stocktrader
version: 0.1.0
icon:
https://raw.githubusercontent.com/IBMStockTrader/stocktrader/master/st
ock-trader-icon.png
```

```
Johns-MacBook-Pro-8:stocktrader jalcorn$
```

There is also a `README.md`, which (among other things) is what will be shown if you do a `helm inspect <chart>`, such as `helm inspect stocktrader-0.1.0.tgz`. You also see this readme if using the helm UI in IBM Cloud Private. And upon successful installation of the helm chart, the `NOTES.txt` will be displayed (which is where you would often provide instructions on how to launch the newly installed application in a browser).

There is a `templates` directory, which contains whatever yaml files you want executed when the chart is installed. Inside each yaml file, you can refer to parameterized values, rather than hardcoding each string, by wrapping it in double curly braces. For example, to refer to the `db2.host` field of your `values.yaml`, you'd use `{{ Values.db2.host }}`. There are also some predefined variables, like for the name of the helm release given when installing this helm chart, where you would use `{{ Release.name }}`, such as when constructing the name of a given object; like if we want to create a portfolio service with our helm release name as the prefix, we would use `name: {{ .Release.Name }}-portfolio-service`.

Lastly, there is the `values.yaml`. Let's look closer at it in the next section.

values.yaml

Everything you want displayed in the helm UI for your helm chart goes here. This is where you specify what entry fields you want displayed, and the variable name associated with each entry field, that you can refer to from within the various yaml files in your `templates` directory. Or if using the helm CLI, these are the values that can be specified via `--set` parameters to `helm install`, such as `helm install stocktrader-0.1.0.tgz --set db2.host=1.2.3.4` from our example above. This is also where you can specify default values for each field, where appropriate.

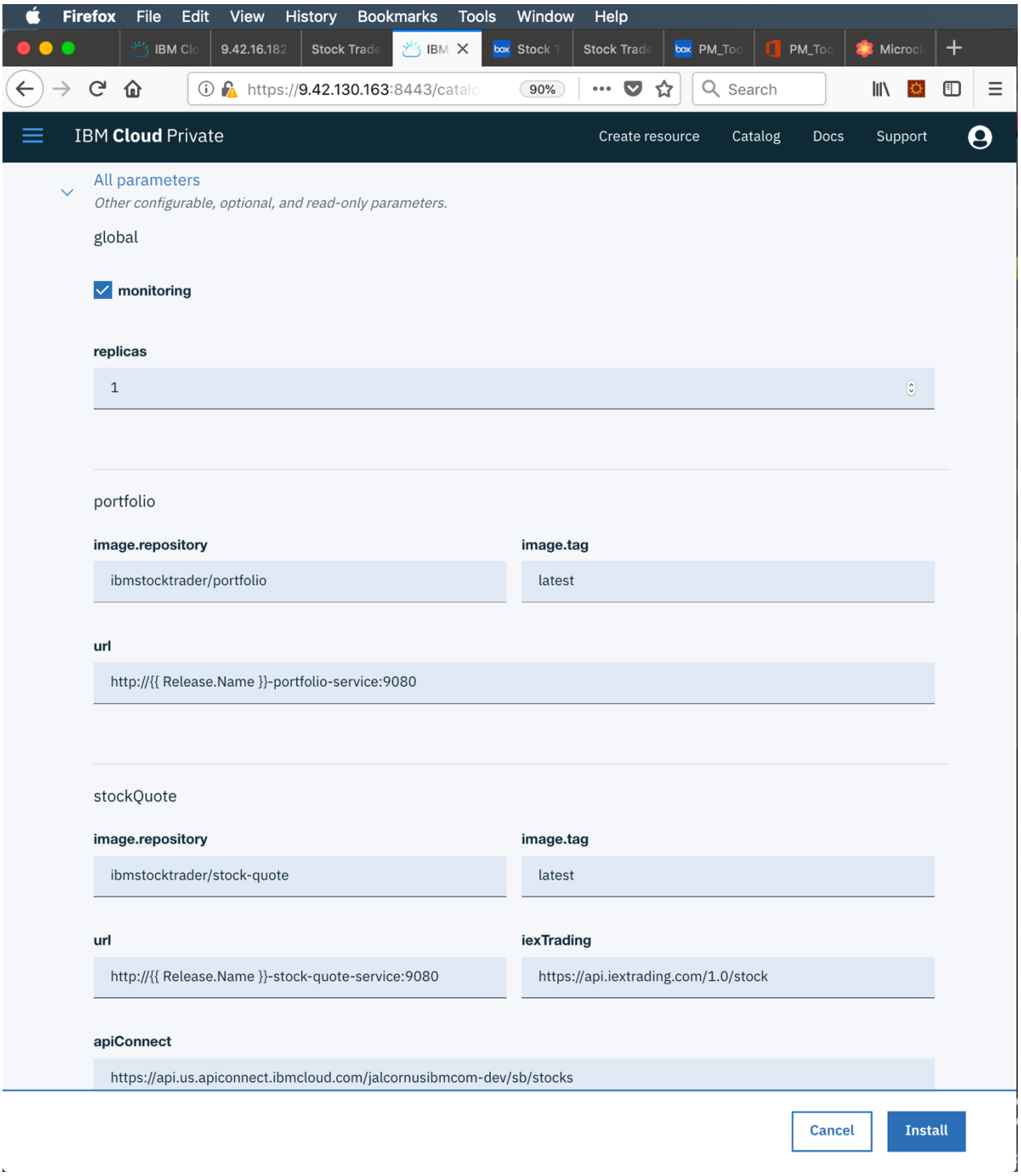
Generally, you would have a section for each microservice you want installed, plus a section for any configuration to talk to prerequisite middleware. Let's take a look at the `values.yaml` for our Stock Trader sample; it's a bit long, but is illustrative of the concepts:

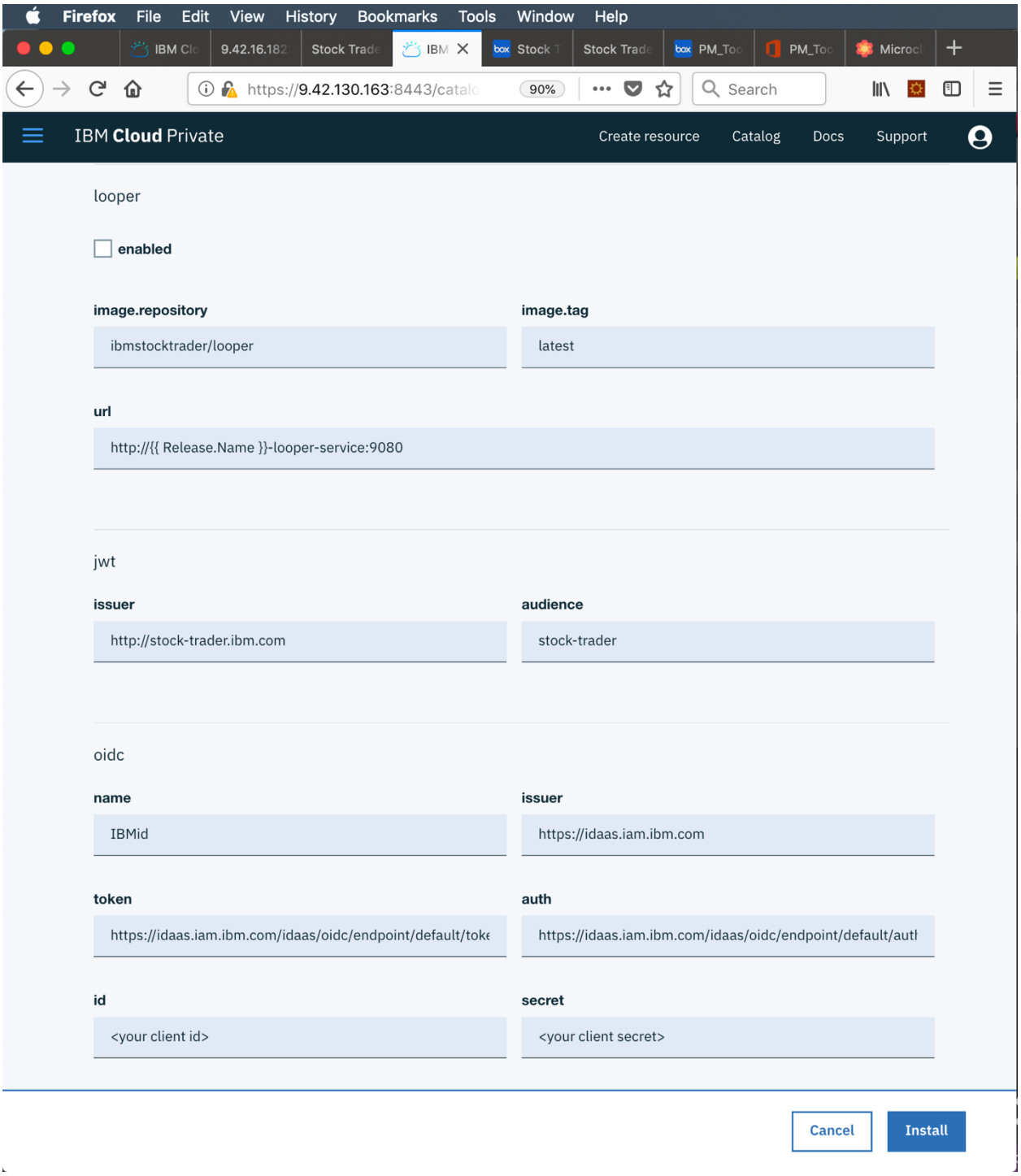
```
global:
  monitoring: true
  replicas: 1
portfolio:
  image:
    repository: ibmstocktrader/portfolio
    tag: latest
  url: http://{{ Release.Name }}-portfolio-service:9080
stockQuote:
  image:
    repository: ibmstocktrader/stock-quote
    tag: latest
  url: http://{{ Release.Name }}-stock-quote-service:9080
  iexTrading: https://api.iextrading.com/1.0/stock
  apiConnect: https://api.us.apiconnect.ibmcloud.com/jalcornusibmcom-dev/sb/stocks
trader:
  enabled: true
```

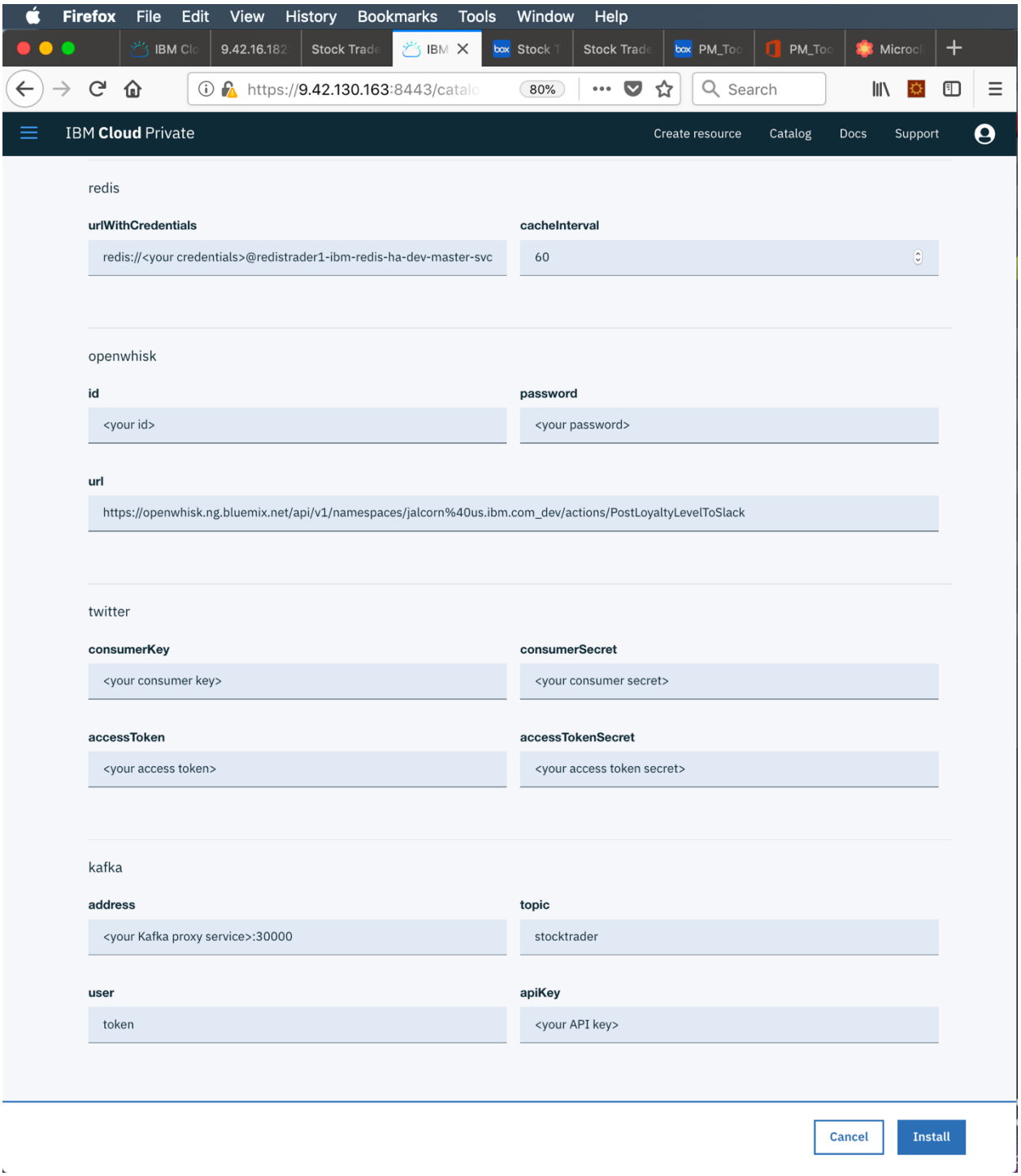
```
  image:
    repository: ibmstocktrader/trader
    tag: basicregistry
tradr:
  enabled: false
  image:
    repository: ibmstocktrader/tradr
    tag: latest
messaging:
  enabled: false
  image:
    repository: ibmstocktrader/messaging
    tag: latest
  notification:
    url: http://{{ Release.Name }}-notification-service:9080
notificationSlack:
  enabled: false
  image:
    repository: ibmstocktrader/notification-slack
    tag: latest
notificationTwitter:
  enabled: false
  image:
    repository: ibmstocktrader/notification-twitter
    tag: latest
tradeHistory:
  enabled: false
  image:
    repository: ibmstocktrader/trade-history
    tag: latest
  url: http://{{ Release.Name }}-trade-history-service:9080
looper:
  enabled: false
  image:
    repository: ibmstocktrader/looper
    tag: latest
  url: http://{{ Release.Name }}-looper-service:9080
jwt:
  issuer: http://stock-trader.ibm.com
  audience: stock-trader
oidc:
  name: IBMid
  issuer: https://idaas.iam.ibm.com
  token: https://idaas.iam.ibm.com/idaas/oidc/endpoint/default/token
  auth: https://idaas.iam.ibm.com/idaas/oidc/endpoint/default/authorize
  id: <your client id>
  secret: <your client secret>
  key: blueidprod
  nodeport: <your ingress URL>
ingress:
  address: <your proxy node IP>:443
watson:
  id: apikey
  passwordOrApiKey: <your API key>
  url: https://gateway.watsonplatform.net/tone-analyzer/api/v3/tone?version=2017-09-
21&sentences=false
db2:
  host: db2trader1-ibm-db2oltp-dev
  port: 50000
  id: db2inst1
  password: db2inst1
  db: trader
odm:
```

```
  id: odmAdmin
  password: odmAdmin
  url: http://odmtrader1-ibm-odm-
dev:9060/DecisionService/rest/ICP_Trader_Dev_1/determineLoyalty
mq:
  host: mqtrader1-mqtrader1
  port: 1414
  id: app
  password: ""
  queueManager: stocktrader
  queue: NotificationQ
  channel: DEV.APP.SVRCONN
redis:
  urlWithCredentials: redis://<your credentials>@redistrader1-ibm-redis-ha-dev-master-
svc:6379
  cacheInterval: 60
openwhisk:
  id: <your id>
  password: <your password>
  url:
https://openwhisk.ng.bluemix.net/api/v1/namespaces/jalcorn%40us.ibm.com_dev/actions/Po
stLoyaltyLevelToSlack
twitter:
  consumerKey: <your consumer key>
  consumerSecret: <your consumer secret>
  accessToken: <your access token>
  accessTokenSecret: <your access token secret>
kafka:
  address: <your Kafka proxy service>:30000
  topic: stocktrader
  user: token
  apiKey: <your API key>
```

As you can see, we have a `global` section with settings that apply to all of the microservices. Then there is the Docker image repository name and tag for each microservice, plus an `enabled` boolean per optional microservice, used to indicate which ones you want to install. Lastly, we have sections for connecting to each piece of prerequisite middleware. Note that indented items under a heading will be shown as a grouped set in the helm UI, with a thin horizontal separator between top-level sections. Let's look at a few screenshots showing portions of the UI resulting from the above `values.yaml`:

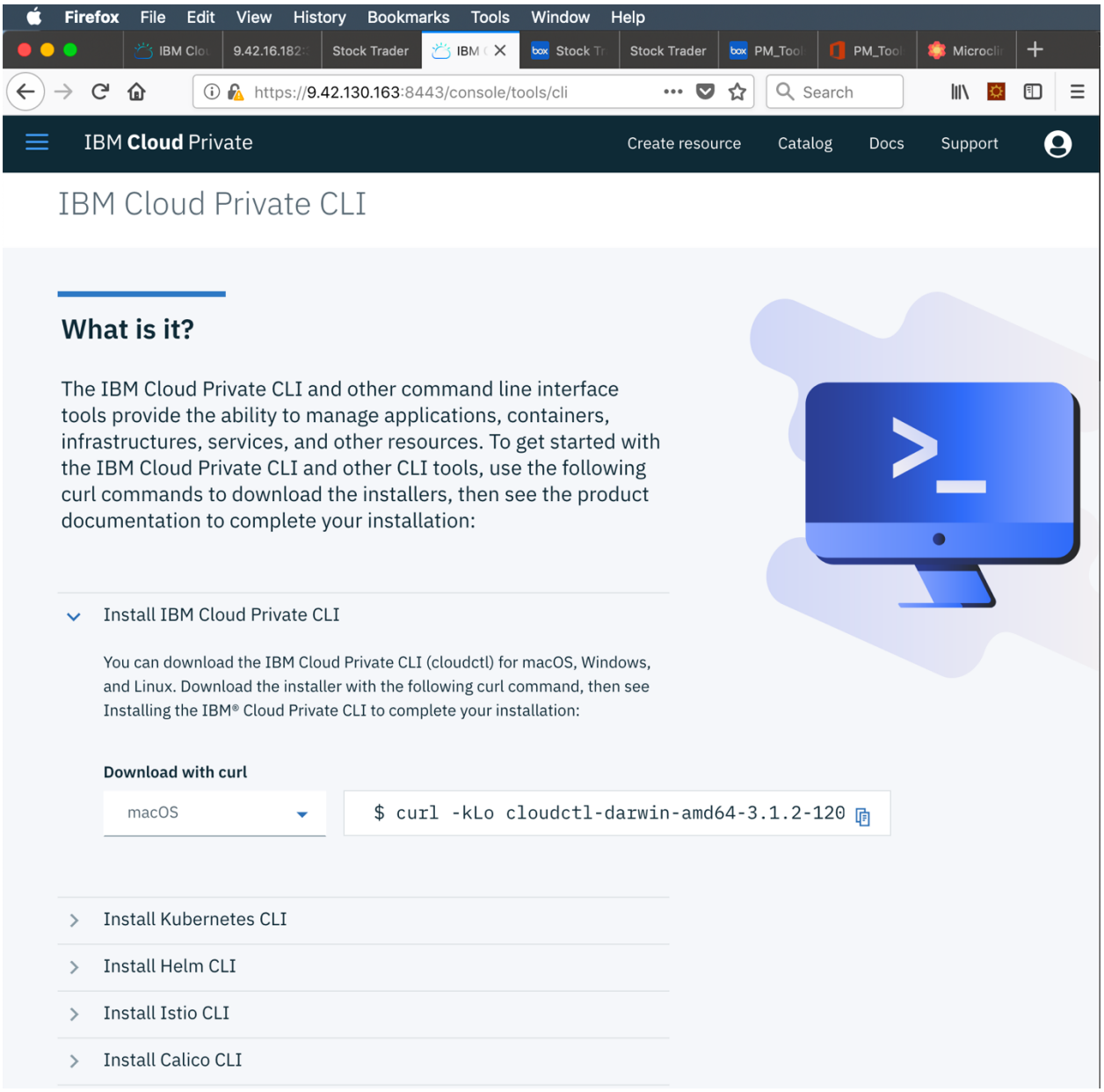

Firefox File Edit View History Bookmarks Tools Window Help
https://9.42.130.163:8443/catalo
90%
Search
IBM Cloud Private
Create resource
Catalog
Docs
Support
All parameters
Other configurable, optional, and read-only parameters.
global
monitoring
replicas
1
portfolio
image.repository
ibmstocktrader/portfolio
image.tag
latest
url
http://{{ Release.Name }}-portfolio-service:9080
stockQuote
image.repository
ibmstocktrader/stock-quote
image.tag
latest
url
http://{{ Release.Name }}-stock-quote-service:9080
iexTrading
https://api.iextrading.com/1.0/stock
apiConnect
https://api.us.apiconnect.ibmcloud.com/jalcornusibmcom-dev/sb/stocks
Cancel
Install


Firefox File Edit View History Bookmarks Tools Window Help
https://9.42.130.163:8443/catalo
90%
Search
IBM Cloud Private
Create resource
Catalog
Docs
Support
looper
enabled
image.repository
ibmstocktrader/looper
image.tag
latest
url
http://{{ Release.Name }}-looper-service:9080
jwt
issuer
http://stock-trader.ibm.com
audience
stock-trader
oidc
name
IBMid
issuer
https://idaas.iam.ibm.com
token
https://idaas.iam.ibm.com/idaas/oidc/endpoint/default/toke
auth
https://idaas.iam.ibm.com/idaas/oidc/endpoint/default/autl
id
<your client id>
secret
<your client secret>
Cancel
Install



Deploying your helm chart

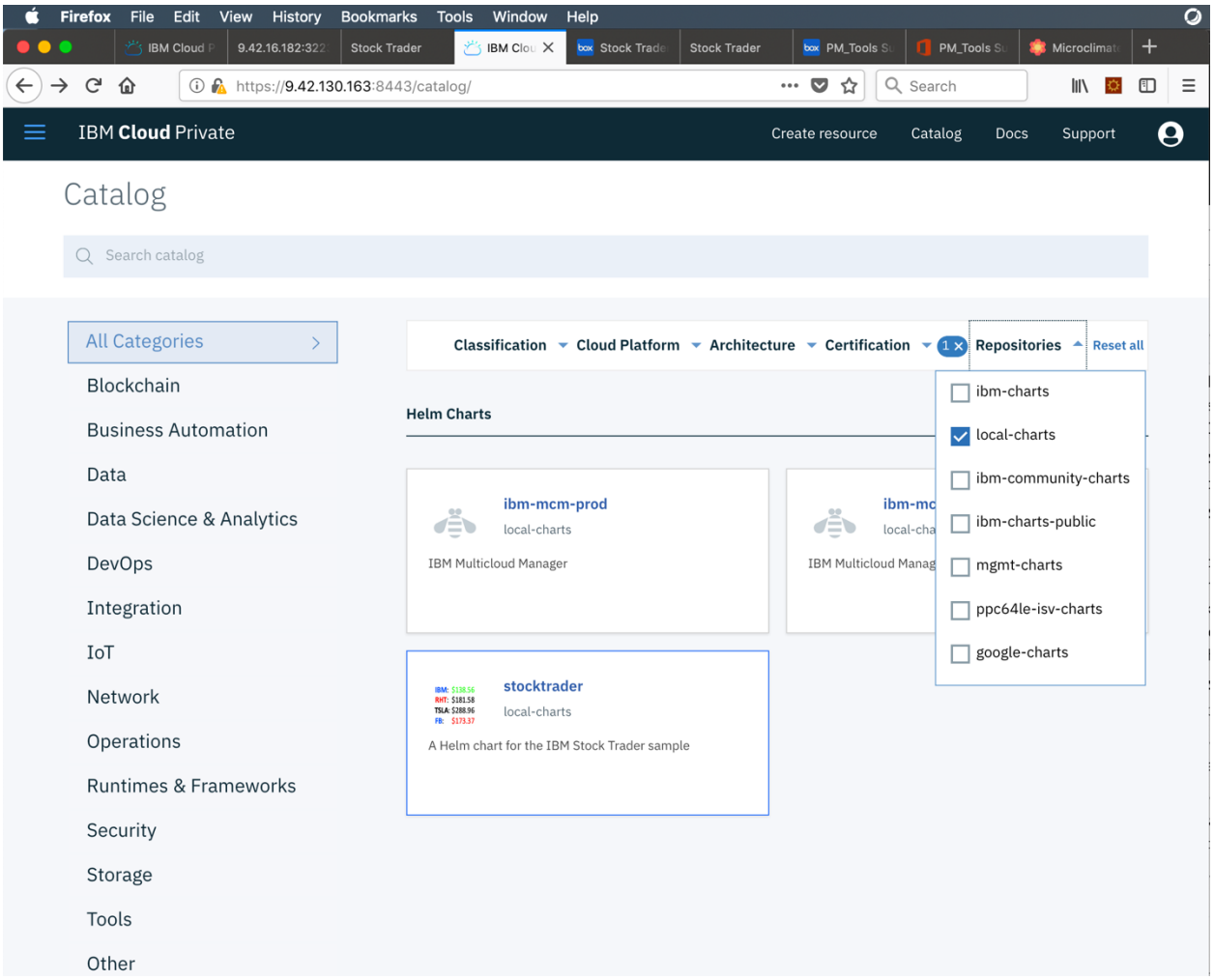
Once you have your helm chart packaged as a .tgz file, you need to use the `cloudctl` CLI to load it into the ICP catalog. You can download this from the `Command Line Tools->Cloud Private CLI` page of the ICP catalog.



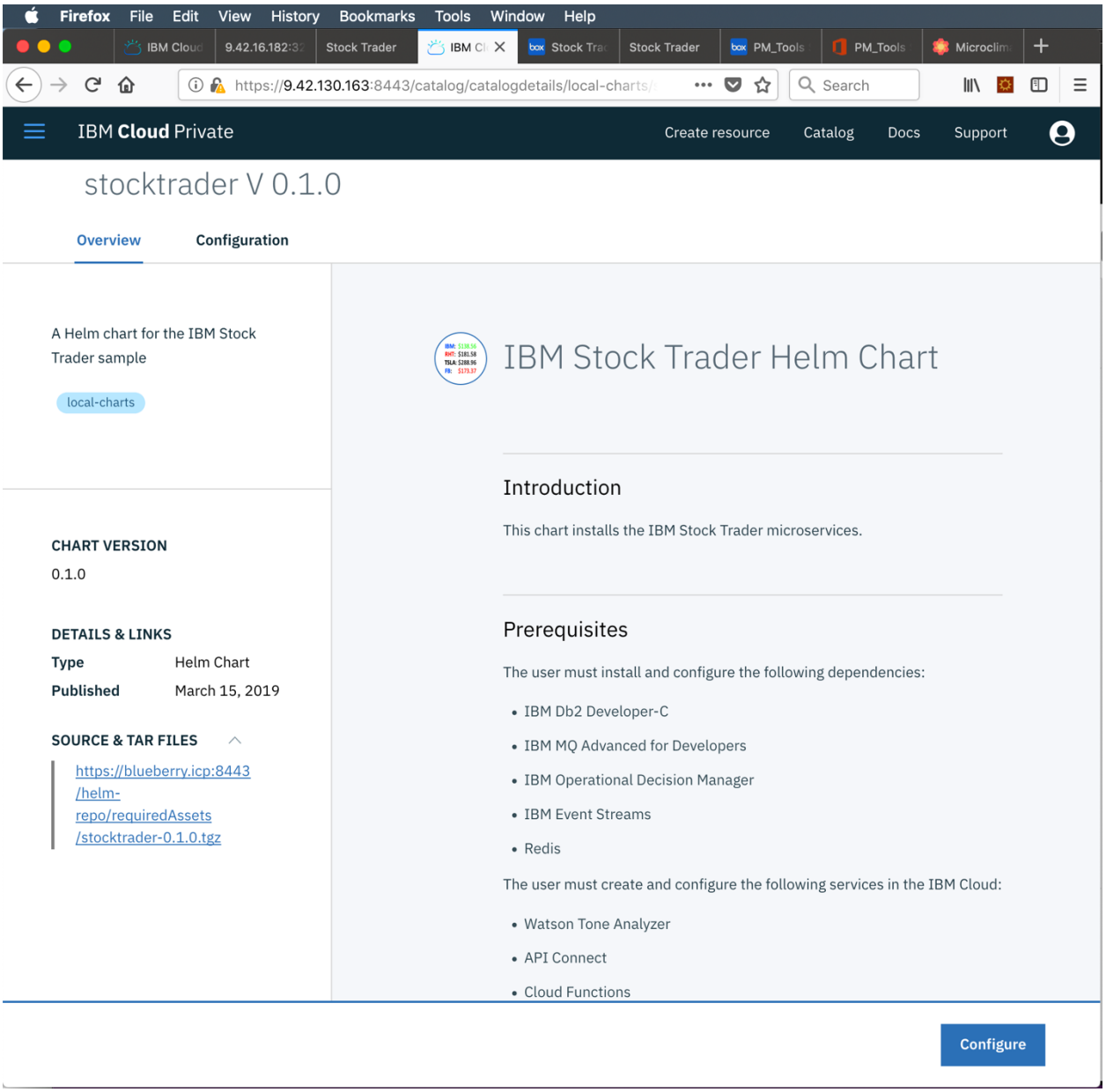
Then do a `cloudctl login` to get authenticated, and you are ready to go. The command to load the Stock Trader helm chart into the ICP catalog is as follows:

```
cloudctl catalog load-chart --archive stocktrader-0.1.0.tgz --repo local-charts
```

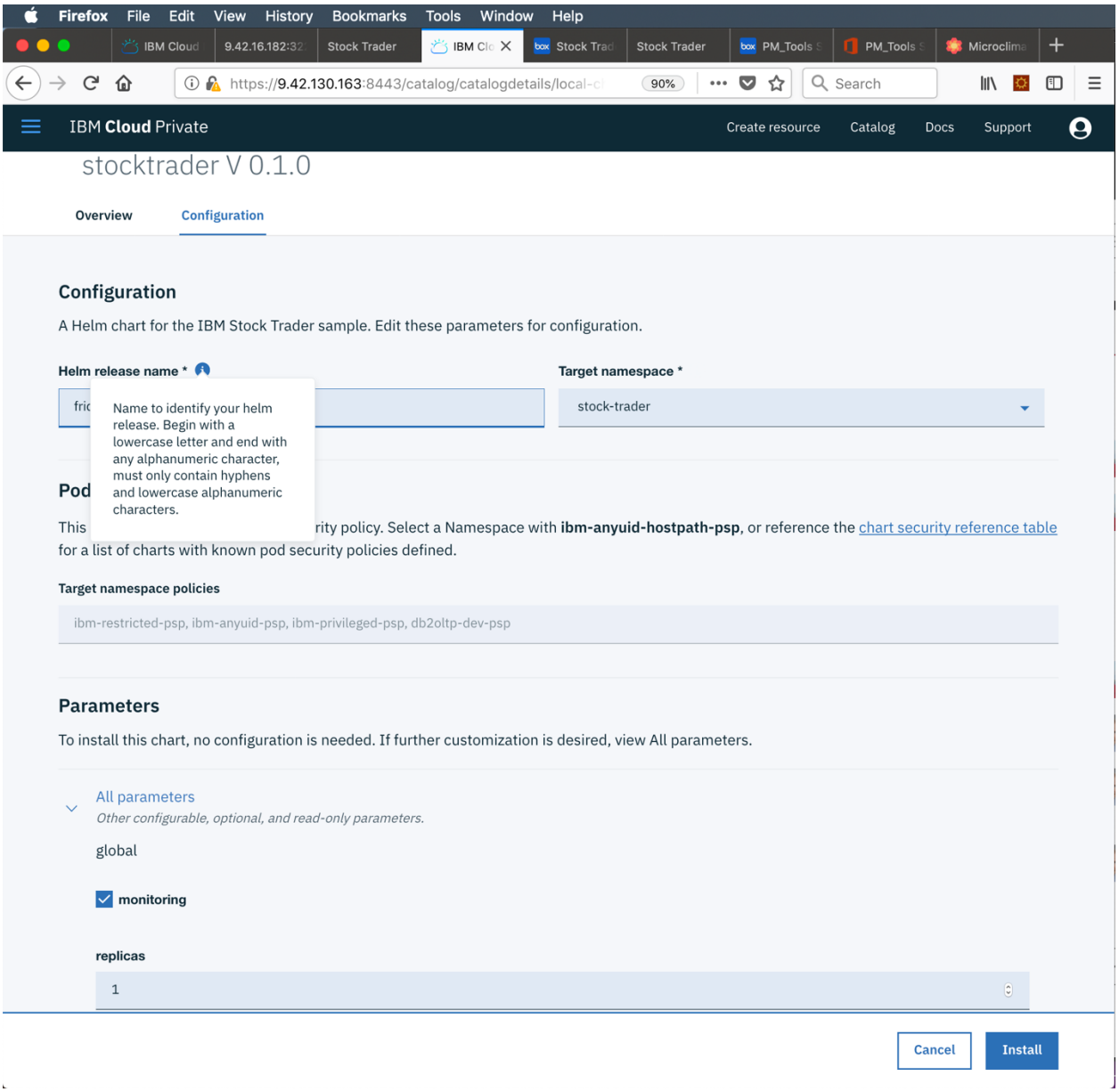
Then, in the ICP console, go to `Manage->Helm Repositories` and click `Sync repositories`, and give it a few minutes to process. Once complete, you will see your new helm chart in the list (here, I filtered the list to just those in my `local-charts` repo):



Clicking on it will first show the readme page for your chart:



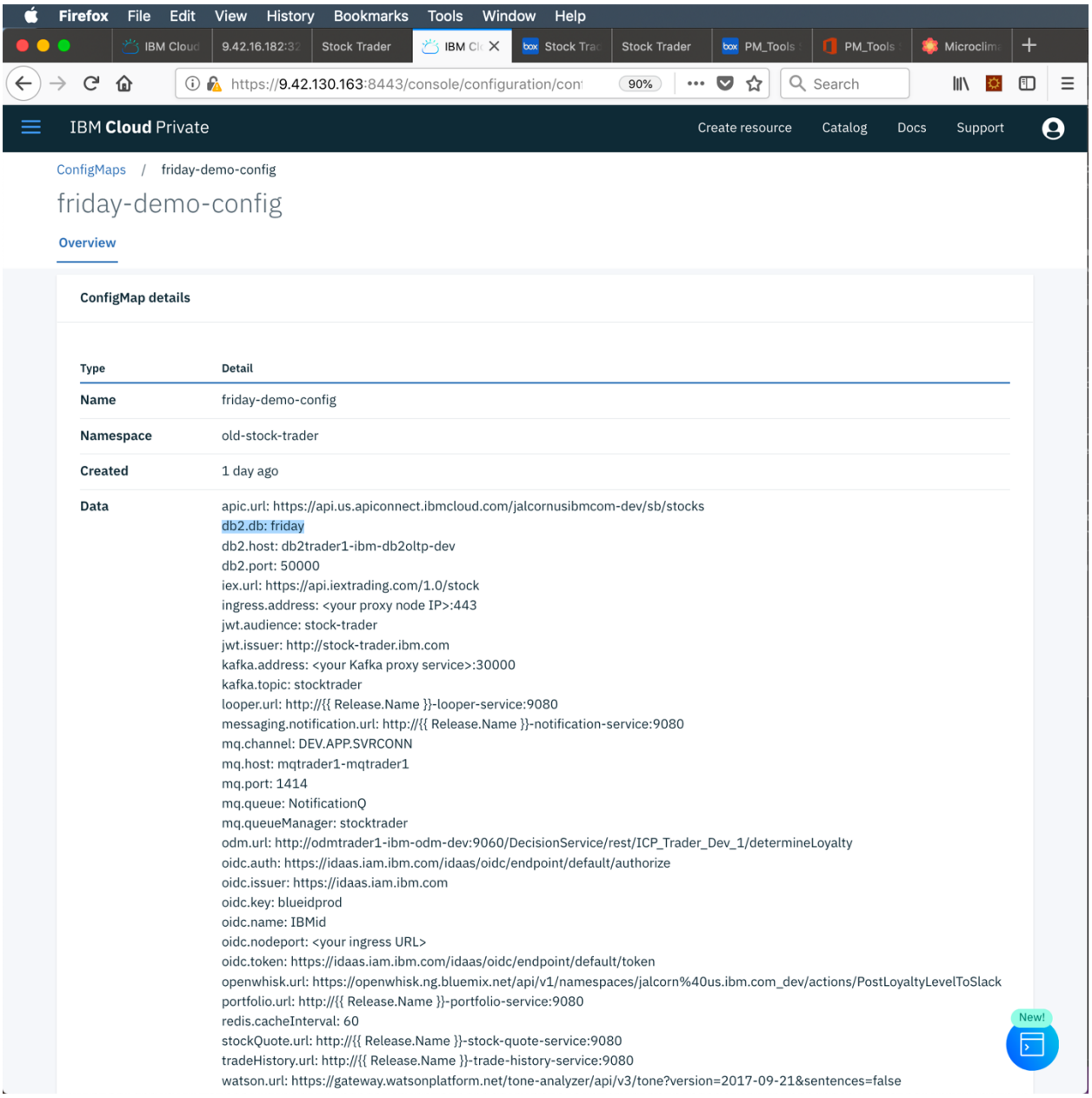
Once you click on Configure, you will receive the page where you can enter the name of the helm release and choose your namespace, and the `All parameters` twisty that, when expanded, shows the `values.yaml` fields shown previously.



Configuration values

While some of the fields from the `values.yaml` are used to locate the Docker images to download or other things in the yaml for say a deployment or service or ingress definition, most of the fields end up in a *config map* or *secret*, that ultimately get fed to each microservice as environment variables.

For example, here are the contents of the `friday-demo-config` from running the helm chart shown above (where I had customized the DB2 database name to `friday`) - note it alphabetizes the entries, so having prefixes helps keep related items shown together:

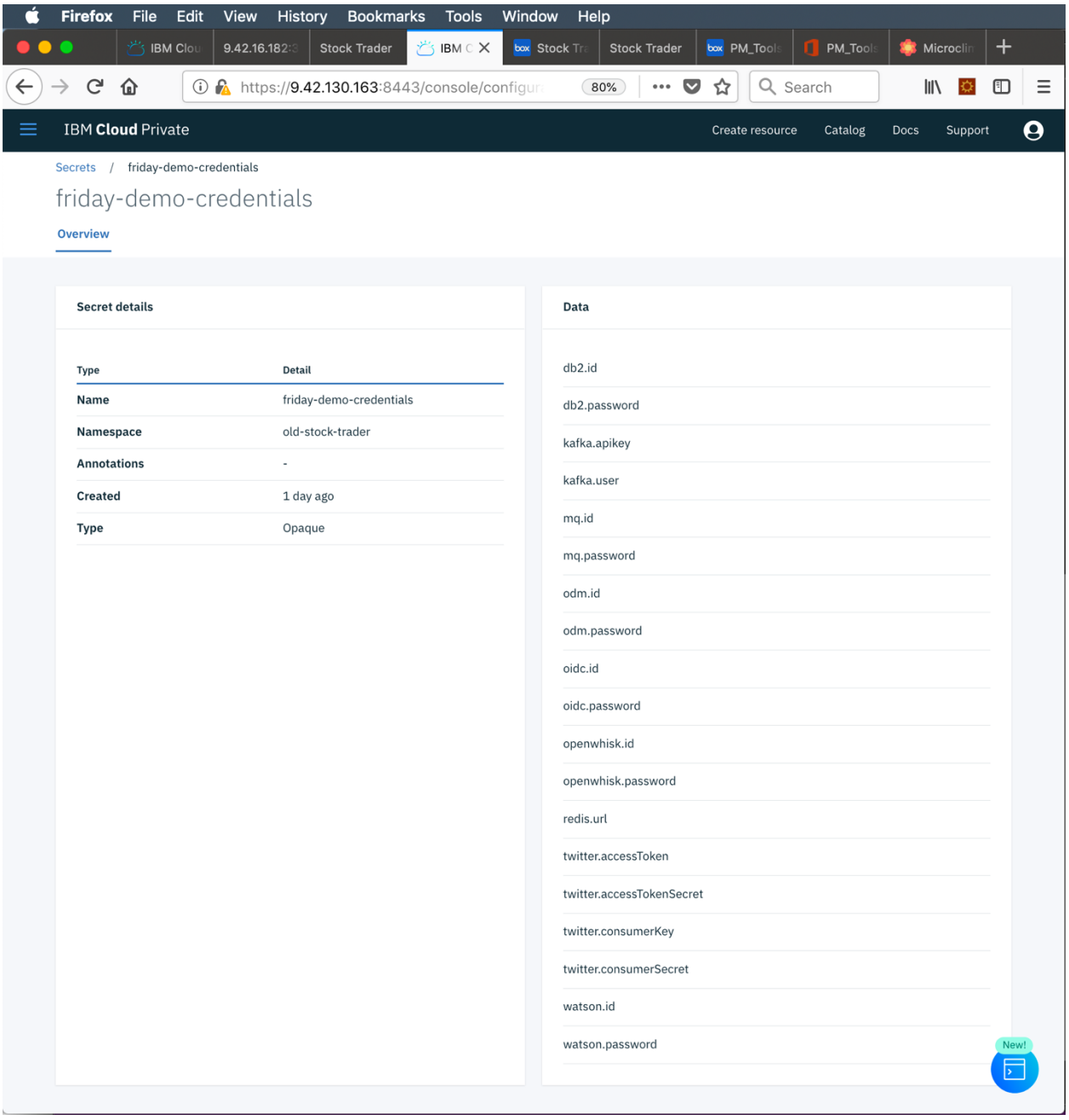


Here's the associated `config.yaml` from the `templates` directory. Note that numbers, like a port number or cache interval, need to have quotes added around them, to avoid a parsing error:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-config
type: Opaque
data:
  portfolio.url: {{ .Values.portfolio.url }}
  stockQuote.url: {{ .Values.stockQuote.url }}
  watson.url: {{ .Values.watson.url }}
  tradeHistory.url: {{ .Values.tradeHistory.url }}
  messaging.notification.url: {{ .Values.messaging.notification.url }}
```

```
  looper.url: {{ .Values.looper.url }}
  apic.url: {{ .Values.stockQuote.apiConnect }}
  iex.url: {{ .Values.stockQuote.iexTrading }}
  odm.url: {{ .Values.odm.url }}
  openwhisk.url: {{ .Values.openwhisk.url }}
  ingress.address: {{ .Values.ingress.address }}
  kafka.address: {{ .Values.kafka.address }}
  kafka.topic: {{ .Values.kafka.topic }}
  db2.host: {{ .Values.db2.host }}
  db2.port: "{{ .Values.db2.port }}"
  db2.db: {{ .Values.db2.db }}
  mq.host: {{ .Values.mq.host }}
  mq.port: "{{ .Values.mq.port }}"
  mq.queueManager: {{ .Values.mq.queueManager }}
  mq.queue: {{ .Values.mq.queue }}
  mq.channel: {{ .Values.mq.channel }}
  oidc.name: {{ .Values.oidc.name }}
  oidc.issuer: {{ .Values.oidc.issuer }}
  oidc.token: {{ .Values.oidc.token }}
  oidc.auth: {{ .Values.oidc.auth }}
  oidc.key: {{ .Values.oidc.key }}
  oidc.nodeport: {{ .Values.oidc.nodeport }}
  jwt.issuer: {{ .Values.jwt.issuer }}
  jwt.audience: {{ .Values.jwt.audience }}
  redis.cacheInterval: "{{ .Values.redis.cacheInterval }}"
```

All of the credentials entered when deploying the helm chart are off in a *secret* named `friday-demo-credentials` (note that values of a secret are not displayed in the UI, as these are sensitive data that you don't want the casual observer to be able to view):



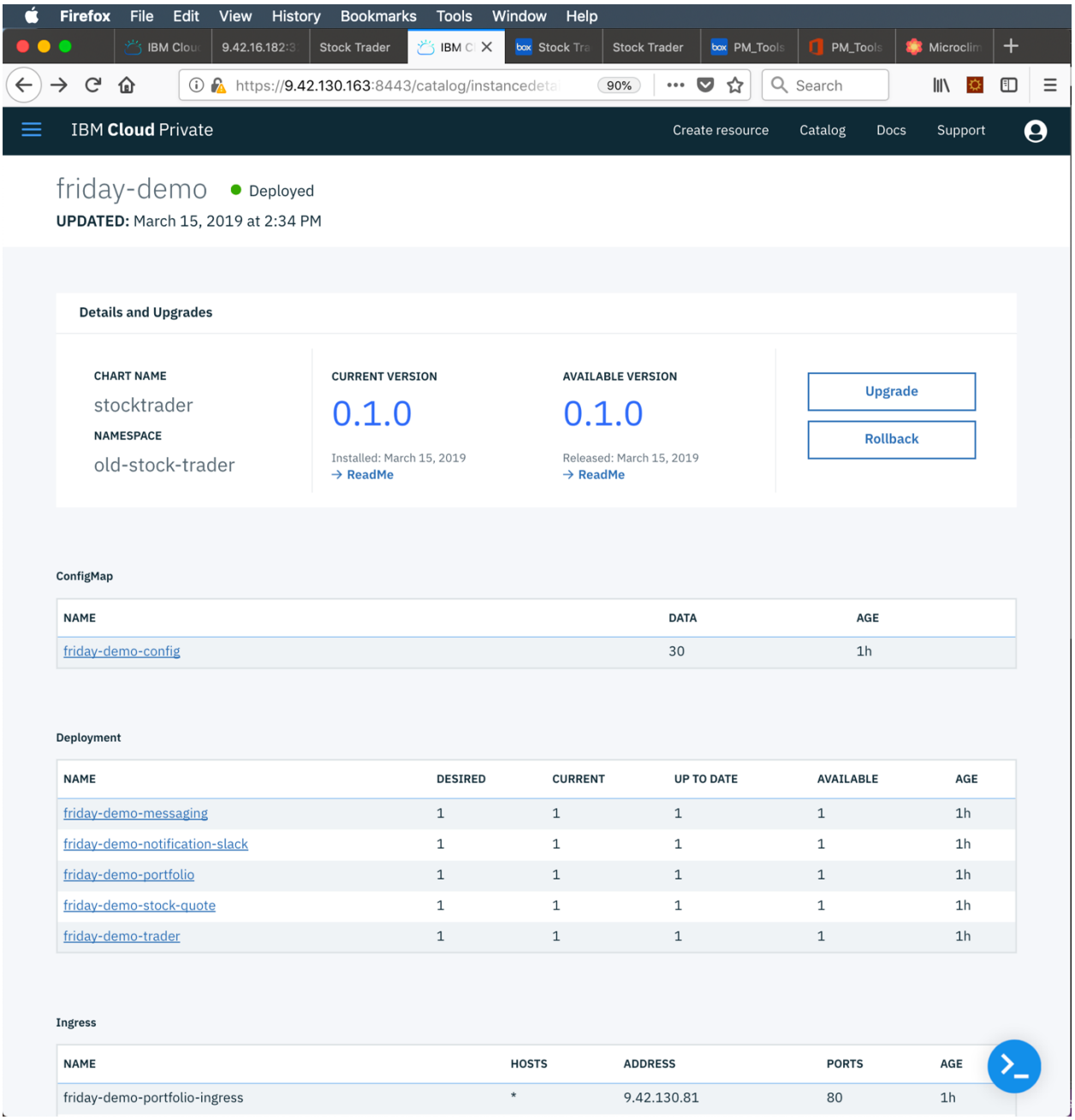
Let's look quickly at the `credentials.yaml` in the `templates` directory, that defines that secret. Note we use `stringData`, which is a convenience data type, saving us from having to base64 encode each value (Kubernetes will automatically do this encoding once the yaml is executed when the helm chart is installed).

```
apiVersion: v1
kind: Secret
metadata:
  name: {{ .Release.Name }}-credentials
type: Opaque
```

```
stringData:
  db2.id: {{ .Values.db2.id }}
  db2.password: {{ .Values.db2.password }}
  mq.id: {{ .Values.mq.id }}
  mq.password: {{ .Values.mq.password }}
  odm.id: {{ .Values.odm.id }}
  odm.password: {{ .Values.odm.password }}
  openwhisk.id: {{ .Values.openwhisk.id }}
  openwhisk.password: {{ .Values.openwhisk.password }}
  watson.id: {{ .Values.watson.id }}
  watson.password: {{ .Values.watson.passwordOrApiKey }}
  redis.url: {{ .Values.redis.urlWithCredentials }}
  oidc.id: {{ .Values.oidc.id }}
  oidc.password: {{ .Values.oidc.password }}
  kafka.user: {{ .Values.kafka.user }}
  kafka.apikey: {{ .Values.kafka.apiKey }}
  twitter.consumerKey: {{ .Values.twitter.consumerKey }}
  twitter.consumerSecret: {{ .Values.twitter.consumerSecret }}
  twitter.accessToken: {{ .Values.twitter.accessToken }}
  twitter.accessTokenSecret: {{ .Values.twitter.accessTokenSecret }}
```

Helm release

Each installation of a helm chart is called a helm release. You can go to the UI page for a helm release to see everything that got installed for that helm chart, and to see the status of each item, with hyperlinks to reach further details. For example, if we install our Stock Trader helm chart, giving it a name of `friday-demo`, we will see the following under `Workloads->Helm releases` when clicking on `friday-demo`:



That goes on for several screens, showing the deployments, pods, services, ingresses, config maps, and secrets needed by our Stock Trader app.

You can also upgrade a helm release, like to change the value of a field entered for the `values.yaml`, or to get the latest fixes from a new version of the helm chart. And you can delete the helm release if you want to remove your application. Being able to act upon the entire application as a whole, rather than having to do so one microservice at a time, greatly simplifies administration and maintenance for your application.

Application Cloud Pak

When running Docker containers in an orchestration environment such as Kubernetes, you also need a set of related artifacts—ingress setup, horizontal auto-scaling policies, persistent transactions, dynamically mounted configuration, logging/metrics dashboards, session caching, node affinity rules, etc. You also need these artifacts to be easily configurable, shareable, and deployable.

Enter IBM's Cloud Paks—a set of pre-packaged enterprise capabilities for deployment, lifecycle management, and production use cases. The strategy is to leverage the best tools available in the industry, such as Docker and Helm, to create and manage these artifacts.

A Certified IBM Cloud Pak:

- ✓ Provides enterprise capabilities for deployment, lifecycle management, and production use cases
- ✓ Unlocks the value of IBM Cloud Private, out-of-the-box integration with core operational services
- ✓ Accelerates time to Production for Enterprise client use cases

	Ad hoc containers Client created	IBM provided containers On non-IBM platforms	IBM Cloud Paks On IBM Cloud Private or IBM Kubernetes Service
Deployment/Orchestration (Helm Chart)	Not provided	Not provided	Supported by IBM
IBM Software (core product functionality)	Supported	Supported	Supported by IBM
Base OS container image		Supported	Supported by IBM
Platform Services (logging, monitoring, etc)			Supported by IBM
Cloud Platform (Kubernetes+)			Supported by IBM
Operating System & Hypervisor			Supported by IBM for Linux on Power and Z Supported by RH when running certified content

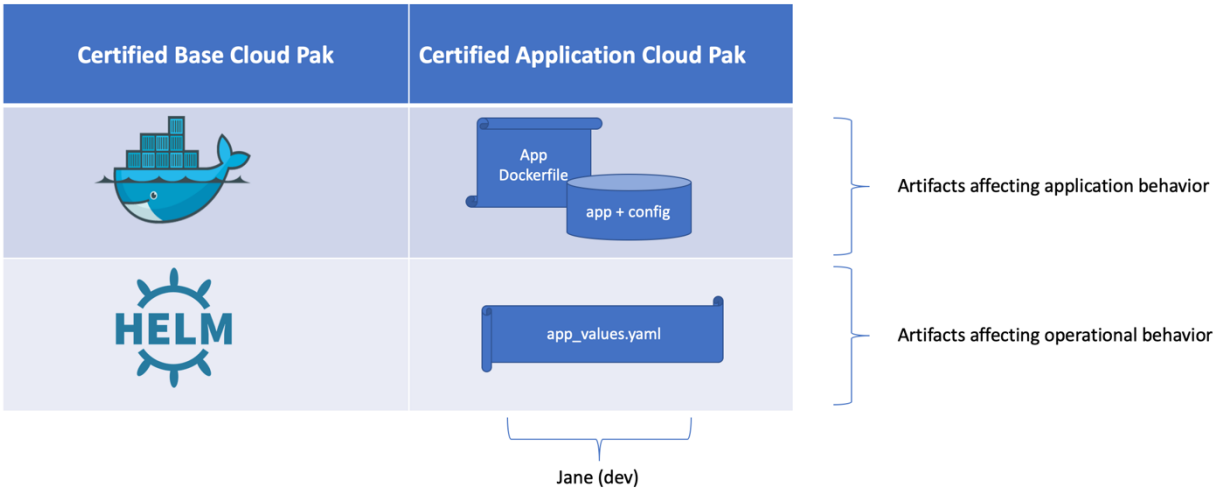
These capabilities include the Docker images, logging/metering dashboards, operators, and helm charts - Kubernetes' native packaging format and a part of the Cloud Native Computing Foundation.

However, Cloud Paks for runtimes, such as Java / Liberty, are a bit different than Cloud Paks for other middleware such as MQ and Db2. A runtime developer will always need to customize their Docker image and helm chart for their application's specific needs. Each microservice will

bring different business logic with it, may require different features enabled in the underlying runtime and may configure those features differently.

Luckily both Docker and Kubernetes have standard customization mechanisms which are exploited by our **Application** Cloud Pak.

At a high-level view, an Application Cloud Pak consists of:



Application Cloud Pak- Docker

From a Docker perspective, the developer (Jane persona) will extend the certified WebSphere Application Server (Liberty or traditional) Docker image from Docker Hub and add their custom application and configuration, such as this example:

```
FROM websphere-liberty:kernel

# Add my app and config
COPY --chown=1001:0  myApp.war /config/dropins/
COPY --chown=1001:0  server.xml /config/

# Optional functionality
COPY --from=hazelcast/hazelcast --chown=1001:0 /opt/hazelcast/*.jar
/opt/ibm/wlp/usr/shared/resources/hazelcast/
ARG HZ_SESSION_CACHE=client
ARG MP_MONITORING=true



# This script will add the requested XML snippets and grow image to be fit-
for-purpose
RUN configure.sh
```

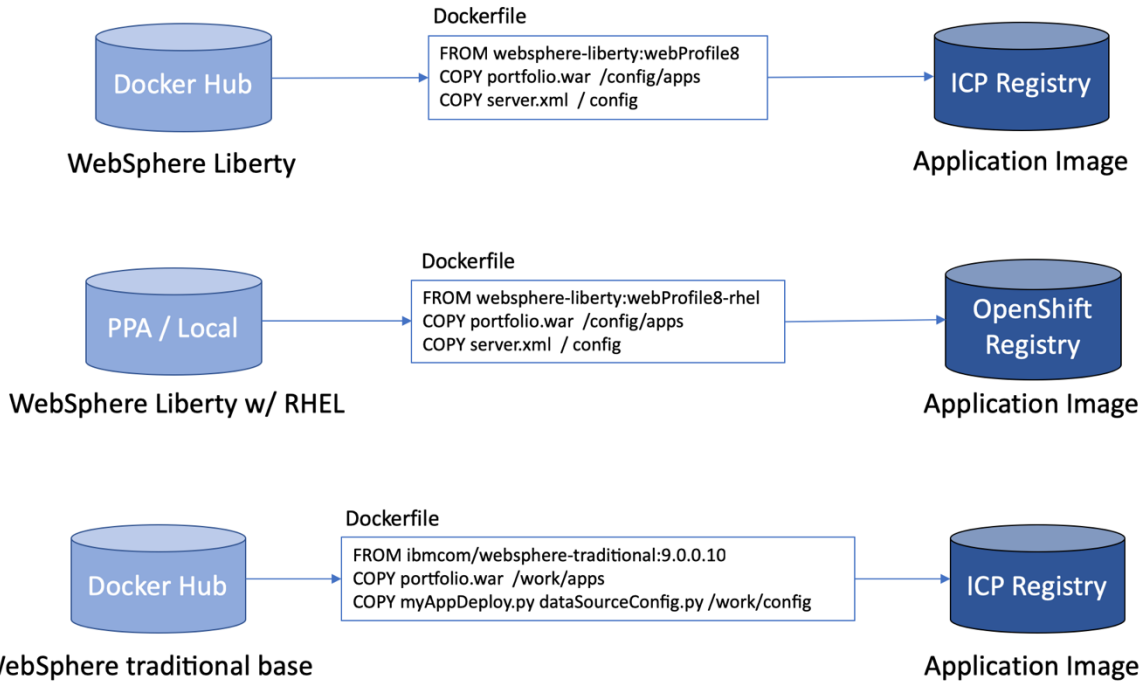
As can be seen from the above example, the base Docker image included in the Cloud Pak has built-in production-grade functionality, such as enabling MicroProfile Metrics and HTTP Session Cache that developers can toggle. The base image also runs by default as non-root,

following industry's best practices, and has helpful *symlinks* for logs and configuration. All these attributes make the WebSphere Docker image a great vehicle to deploy production-grade applications – even independently of a Helm chart.

Detailed guidance for each Cloud Pak's Docker image is provided in their corresponding GitHub repositories for Liberty (https://github.com/WASdev/ci.docker/blob/master/README.md) and traditional WAS (https://github.com/WASdev/ci.docker.websphere-traditional#building-an-application-image).

A public example of extending the Liberty Docker image can be found in Stock Trader Portfolio microservice: https://github.com/IBMStockTrader/portfolio/blob/master/Dockerfile

The following diagram illustrates the flow between the base Cloud Pak Docker images and their corresponding application Docker images via Dockerfiles:



Application Cloud Pak – Kubernetes / Helm

From a Kubernetes perspective, the developer (Jane persona) will create a YAML file that customizes and overrides the default configuration of the WebSphere (Liberty and traditional) Helm chart. As we saw in the previous section, this is often referred to as *values.yaml* and is the standard overriding mechanism built into Helm. The official Helm documentation provides a detailed explanation of this file's usage, but basically this override allows each developer to completely customize the exact same base chart.

The most common customizations are the environment variables that the application is expecting – which can be set directly, from a ConfigMap or from a Secret. Another common customization point are labels and annotations for Ingress and Service resources.

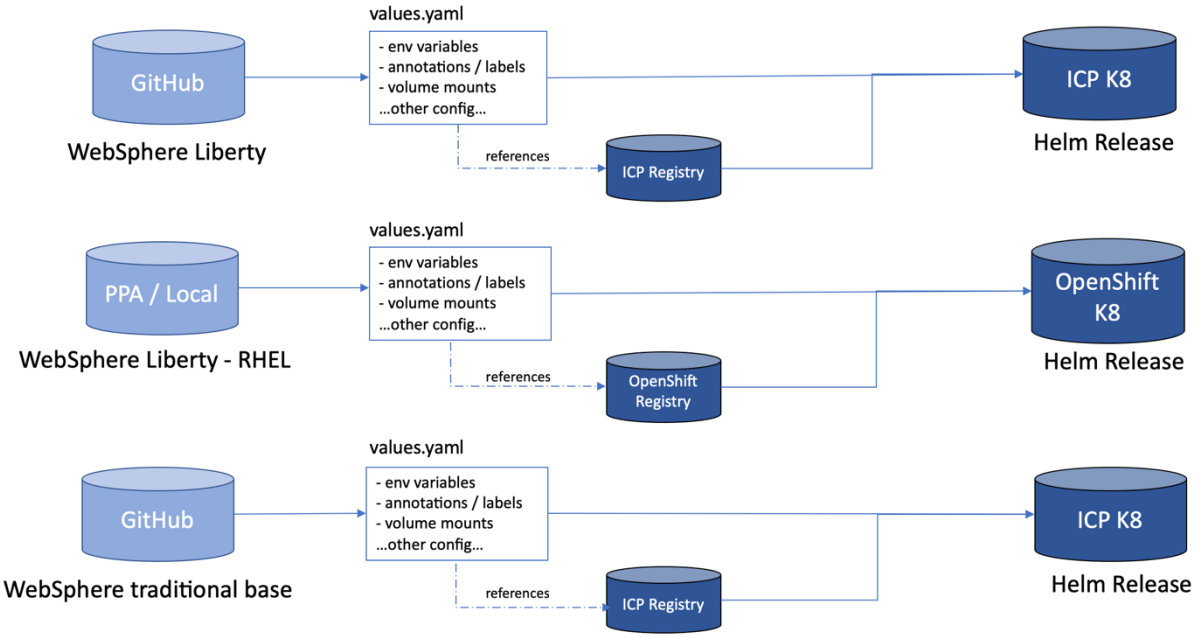
By extending from a common, certified Helm chart, developers can take advantage of built-in functionality such as the setup of a production-grade Pod Security Policy, cluster-wide monitoring, and transactional recovery using StatefulSets and Persistent Volumes.

These Helm charts are available out-of-the-box with IBM Cloud Private, but can also be retrieved manually via the public IBM Charts repository: https://github.com/IBM/charts/tree/master/stable

For more information on the usage of each chart, please see the corresponding document for Liberty and traditional WAS.

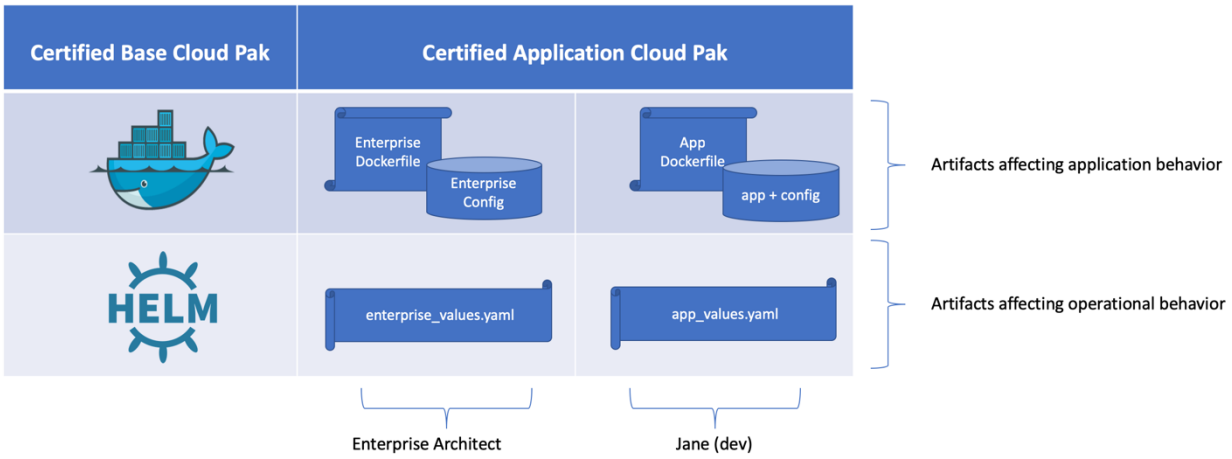
The Stock Trader Portfolio microservice demonstrates this pattern: `https://github.com/IBMStockTrader/portfolio/blob/master/manifests/portfolio-values.yaml`

The following diagram illustrates the flow where a fully customized and app-specific Helm chart is deployed by starting with the base Cloud Pak Helm charts and applying configuration values with values.yaml:



Enterprise Application Cloud Pak

While the above pattern works very well for Jane to develop and deploy an application, in an enterprise there will often be a set of specific settings and configuration that must be applied consistently across the company.



This is achieved by having an internal base Docker image that is defined by an enterprise architect which must be used by all developers from within that enterprise. For example, the enterprise architect may create an image that always has monitoring and health checks turned on, in addition to a pre-defined set of features:

```
FROM websphere-liberty:kernel

COPY --chown=1001:0  enterprise_server.xml /config/configDropins/default/

COPY --from=hazelcast/hazelcast --chown=1001:0 /opt/hazelcast/*.jar
/opt/ibm/wlp/usr/shared/resources/hazelcast/
ARG HZ_SESSION_CACHE=client
ARG MP_MONITORING=true


RUN configure.sh
```

This image can be tagged with *websphere-liberty:myEnterpriseWeb*, and required to be used by the developers. Jane's Dockerfile would then become:

```
FROM websphere-liberty:myEnterpriseWeb

# Add my app and config
COPY --chown=1001:0  myApp.war /config/dropins/
COPY --chown=1001:0  server.xml /config/
```

Similarly at the Helm chart level, the enterprise architect can create an *enterprise_values.yaml* file which is applied to the developer's chart during *helm install*. For example:

```
ingress:
   annotations:
     nginx.ingress.kubernetes.io/enable-cors: "true"
     nginx.ingress.kubernetes.io/cors-allow-methods: "PUT, GET, POST,
OPTIONS"

logs:
```

```
  persistLogs: true
  persistTransactionLogs: false
  consoleFormat: json
  consoleLogLevel: info
  consoleSource: message,trace,accessLog,ffdc
```

The developer, Jane, can deploy her Helm chart via the following command (noticed that the enterprise values comes after, since precedence is given to the right-most set of values)

```
helm install ibm-charts/ibm-websphere-liberty -f
myApp_values.yaml -f enterprise_values.yaml -n myRelease --tls
```

Another usage of this pattern is the definition of enterprise-wide configuration for specific target environments, such as different cloud vendors or public vs private environment, as well as for different stages of the application life-cycle (dev, QA, prod). For example: the Kubernetes secret representing a database credential that gets bound into an environment variable will differ between these scenarios.

For example, there can be a specific *enterprise_values.yaml* that business critical applications must include which setup more strict Pod Security Policy, or different Ingress configurations depending on the target cluster.

Application Cloud Pak – Dashboards

The WebSphere Liberty Cloud Pak includes a set of dashboards for Kibana (logging captured from the built-in ELK stack) and Grafana (metrics data loaded from Prometheus) which is optimized to work with data generated from WebSphere Liberty containers.

These dashboards are included in the Helm chart inside the *dashboards* directory:
https://github.com/IBM/charts/tree/master/stable/ibm-websphere-liberty/ibm_cloud_pak/pak_extensions/dashboards

Application Cloud Pak for OpenShift

You can get a special version of the WebSphere Liberty Cloud Pak for OpenShift, which uses Red Hat Enterprise Linux (RHEL) as the base operating system, from Red Hat's catalog:
https://access.redhat.com/containers/#/search/websphere%2520liberty

This Docker image and Helm chart follow the same patterns described in previous sections of this programming guide.

Application Cloud Pak – Upgrading

To upgrade an existing docker container, such as patching a new security vulnerability or fixing a defect in the application, the developer starts by rebuilding their application docker image.

If an ifix was provided by IBM, a simple *RUN java -jar …* instruction will expand the code, otherwise the same application Dockerfile can be run to generate the updated image – which should have a different tag (i.e. myApplication:v102).

To replace the current running container with the updated version the command *helm upgrade* can be used. This command takes into consideration the upgrade strategy of the Helm chart, which by default is *RollingUpdate*, and creates a new *revision* of the Helm release – this # can be used to *rollback* into a particular revision.

This pattern can also be used if a particular configuration of the Helm release needs to be updated, by simply updating the customized *values.yaml* before using *helm upgrade*.

If a canary deployment is desired – having multiple application versions at the same time, with distributed routing – then Istio, as described in previous chapter, is the right tool.

Application Cloud Pak - Certification

IBM's Cloud Paks are called "Certified" because of the strict set of rules and best practices that our Docker images, Helm charts and dashboards must abide to. Given that an Application Cloud Paks extend our Certified Cloud Pak, to be able to still have the "certified" status there are two requirements:

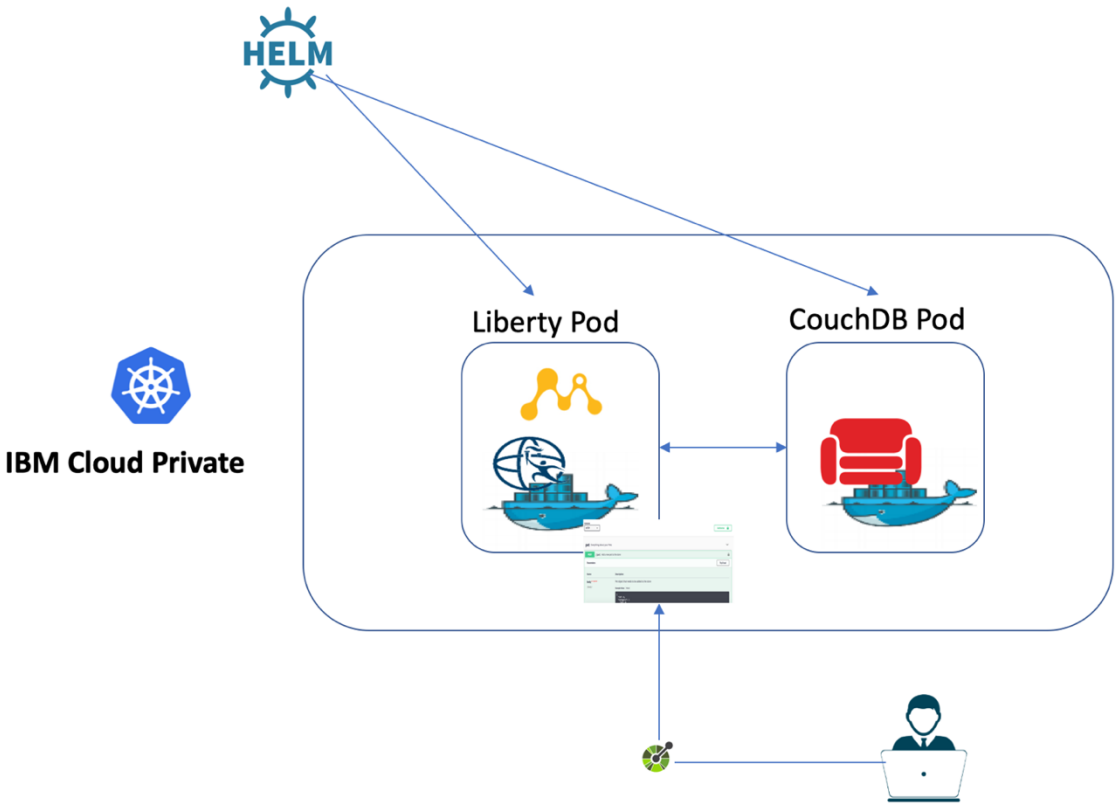
- The application Docker image must pass the Vulnerability Advisor scan and must continue to pass that scan. For example, if after a week the deployment has gone red because of a new vulnerability the deployment will temporarily lose its certification status until it is patched.

- The Cloud Pak Helm chart cannot be modified. The only configuration of the Helm chart is via the *values.yaml* override.

Application Cloud Pak – Examples

The best example of the Application Cloud usage is the Stock Trader application, discussed throughout this programming guide.

Another example is a virtual public lab, https://microservices-api.github.io/kubernetes-microprofile-lab, which walks through the usage of an umbrella chart deploying both Liberty and CouchDB:



Microclimate

Microclimate is a full development experience tool that runs on your laptop or in IBM Cloud Private. To learn about its capabilities, go here: `https://microclimate.dev/`. The focus of this section is how we used Microclimate to build our existing Stock Trader application.

Import and Augment

While some developers may start with a sample application, we already had a microservices-based application in GitHub so we imported each Git repo into Microclimate as a project. After the import Microclimate noticed changes we had to make to various files in order for the project to build. Here's what we changed:

- **server.xml** - Location needed to change to /src/main/liberty/config
- **Jenkinsfile** – We needed to use the Microclimate-generated Jenkins file since it uses a library it assumes is in the Jenkins server. Additionally, we added test = 'false' so we could bypass build tests (not a best practice but we did to temporarily verify the build)
- **pom.xml** - Additional dependencies and parent declarations were needed (we ended up generating a new project to compare the POM.xml files and add what was missing. The resulting POM.xml was quite large and could be pruned down to what you need)
- **Dockerfile, Dockerfile-build** – These files were updated to support security enhancements in Websphere Liberty. See this troubleshooting section for details. `https://microclimate-dev2ops.github.io/troubleshooting#microclimate-liberty-projects-are-broken`

- **Helm charts** – The biggest change was while earlier versions used "kubectl apply deploy.yaml" commands, Microclimate uses helm charts to deploy into IBM Cloud Private. As a result, we copied a new project's helm folder and augmented with our information.

Build Through Pipeline

One great thing about Microclimate is how easy it is to set up a Jenkins pipeline. We quickly set up a pipeline for each microservice and deployed it into the desired namespace.

One major consideration is that since Microclimate's use of Jenkins is baked into a library located on the Jenkins server deployed by Microclimate, there is no current way to add build logic into the Jenkinsfile itself. Therefore, if you need to integrate with other build tooling (Urban Code Velocity, for example), you may want to build your own Jenkins server and use it independently.

Jenkins

There are two paths you can go down if you want to deploy your own Jenkins server: Inside IBM Cloud Private (or any containerized platform), or on a VM. There are benefits to both; and complexities to both paths.

Jenkins in ICP: The advantage is that it's container-based and running in the same cluster as your deployment. However, this requires configuration to run "Docker on Docker" since your Jenkins container will need to be running a Docker engine to build the container image. There are helpful guides on various forums to get this running.

Jenkins in a VM: The advantage we found is that in a VM you can set up Jenkins exactly how you want, and create scripts to deploy into multiple targets. However, this requires you to set up secure access into the target Kubernetes environment as well as the image repo to push your newly built image into.

We chose Jenkins in a VM, which allowed us to:

- Add a Maven plug-in to build the Java files
- Add our own Docker engine to build the Docker image
- Create a script with sensitive information inside the VM so that the Jenkins file can call the script and no sensitive information is located inside the Jenkinsfile
- Utilize build tests, and based on success, harness IBM Multicloud Manager CLI to select which cluster to deploy into
- Add Urban Code Velocity integration so that we could monitor each pipeline through a common UI

For our Jenkins install (learned here), we configured the VM with the following:

- Installed Java JRE for Ubuntu 16.04 (learned here)
- Installed Maven (learned here)
- Configured Jenkins "Global Tools" and added the Java JDK and Maven in order for Jenkins to recognize the maven build command
- Installed Docker (learned here)

- Added Jenkins to a Docker Group (learned here)
- Install ICP CLI, Kubectl CLI, helm CLI , MCM CLI onto VM running Jenkins so we could use scripts to deploy (learned here)
- Created scripts so we could call Maven, Docker, Kube CLI at appropriate times with proper credentials/certificates without exposing them in the Jenkinsfile
- Connected to Private ICP image repo (learned here). This was the trickiest part. Since our ICP clusters are uniquely named, the deploy.yaml that pulled the image needed to know the specific cluster name. This is where helm comes into play where you can pass in variables like image location.

A few notes worth mentioning:

- We were able to add multiple Jenkinsfile instances into our GitHub repo so that the VM-based Jenkins would look for "Jenkinsfiledemo" and the Microclimate Jenkins would look for "Jenkinsfile".
- Our scripts that used "`kubectl apply deploy.yaml`" were able to use variables for everything except the location of the image, requiring us to initially have multiple deploy.yaml files...one for each unique image repo location. A better way is for us to use helm charts where I can customize the location of the image repo in the deploy.yaml so that I can pull from different private image repos depending on which cluster I'm deploying to.

Knative

An emerging technology (currently at 0.4) that is designed to help developers "get back to development" is Knative (https://github.com/knative/docs/). A pain point of developing in Kubernetes is the steep learning curve required for developers. While developers want to focus on business logic in their application, with Kubernetes developers need to understand quite a bit about deploying and operating applications. Knative simplifies the process by helping developers build their code, serve the code as an app (using serverless technology), and drive an event-driven model between components (using Kafka or other eventing services).

More examples will come but for now, check out this IBM blog for a great summary of Knative and its value: https://www.ibm.com/blogs/bluemix/2018/12/video-what-is-knative/

Conclusion

There are a large number of topics to consider and concepts to embrace when plotting a course towards truly leveraging a cloud-native approach to building systems. The ability to create microservices and leverage them in production systems is at the heart of the journey to cloud-native. A new programming model is necessary to build out microservices and enjoy the benefits that they offer.

The cloud-native programming model has some language specific elements as well as a lot of programming language independent elements. This combined set of elements are specified in a number of files. The presence of some specific files and the definition of the content of each of these files is the programming model. Pipelines, authoring tools and an overall cloud platform aim to make this journey of delivering microservices based systems fast and effective.

This book has touched on the concepts of cloud-native, but has majored on helping designers and developers as they begin to really build systems based on a cloud-native approach. Choices made in the book along the way have blended in ingredients available in an opinionated way that is proven to work. Different choices in the selection of technologies and techniques could also be made. Given the fast-paced nature of change in the world of cloud-native and in the platforms upon which these applications run, today's right choices may become upstaged by new approaches tomorrow. Go forth and build cloud-native systems and keep an eye on our web site for new material that evolves the approach laid out.

Appendix A: Java – Summary of Files

This appendix contains all of the mandatory and optional files that make up the artifacts of the programming model. Each of them has been introduced and explained throughout the course of the book. The purpose of this appendix is to show them in a consolidated way, taking more of a 'files' view of the programming model.

Mandatory files

File name	File type	Description or role
yourCode.java	Source code	This is unique for each microservice implementation language. Java, for example, will have a number of .java files, dependent on some specifics about the kind of Java microservice you are writing. Location: `/src/main/java/`
Healthcheck.java	Health Checks	These are special-purpose files designed for declaring health of the application. Location: `/src/main/java/`
server.xml	Configuration file	This describes how your Liberty container will be configured. For example, any Kubernetes Secrets will be pulled in as environment variables using this file into the liberty runtime. There is only one per microservice. Location: `/src/main/liberty/config/`

Dockerfile Dockerfile-build Dockerfile-lang Dockerfile-tools	Build files	Every microservice will have a Dockerfile. The contents will vary based on the runtime that is required to run the specific kind of microservice. Every Dockerfile will have a FROM statement. This specifies the base image from which you are building your docker image as well as the tag. For example: `FROM websphere-liberty:microProfile2` says to pull the Liberty image with the microProfile content. Location: root of git repo
pom.xml	Image Build	Defines maven build Location: root of git repo
Deployment.yaml	Yaml file	It is typical to have at least two .yaml files. One specifies the "kind" of Kubernetes resources that will be deployed via `kubectl create -f <directory>` or `kubectl apply -f <directory>` commands. These are often called deployment .yaml files. The other typically specifies the values that are inserted into the places in the deployment .yaml files where substitution values are specified. It should be the case that kind: Deployment is there, and kind: Service and often kind:Ingress will be required. Location: Depends on deployment technology. Microclimate helm charts location is `/chart/<chartName>/templates/deploy.yaml`
Docker Image	Image binary (generated)	This is the actual docker image and needs to be stored in an image repo in order for the deploy.yaml to pull it and deploy.

Optional files

File name	**File type**	**Description or role**
Jenkins file	Pipeline file	Defines how to deliver and build through Jenkins. One jenkins file can feed many pipelines, but you can also have multiple jenkins files if you need customized builds for different targets. Location: root of git repo
Other .yaml files	Kubernetes config files	Any optional config required for Kubernetes can be done through additional .yaml files
Istio files	Istio configuration and routing files	These istio files control the behavior of the Istio sidecar into each pod, and provide the routing rules for Istio to follow. The rules could control routing, circuit breakers, retrys, and timeouts. Location: `/istio/` is ideal, but no specific location required. The files are usually paired with the microservices it affects.
Application Tags	.yaml files	This is a Kubernetes custom resource definition, which is created during the 'Define' activity. It is a way to collect together all of the microservices which are part of an application or solution. The application CRD is defined then deployed using an application.yaml file.
key.jks	SSL Certificate	A java key store that stores certificate for APIs like Watson Tone Analyzer. Ideally these would be stored in a key store like Vault or a key service like Key Protect.

		Location: `src/main/liberty/config/resources/security`
Helm Chart	.yaml dedicated to deploy via Helm	A collection of .yaml files with the ability to deploy via helm cli. Not required but valuable in that it can support templatizing deployments and support variables via config.yaml files or command line variables.

File Changes for Microclimate

If you are importing your existing GitHub repo into Microclimate, you may need to make some changes.

For a summary of changes, study this link:
`https://microclimate-dev2ops.github.io/troubleshooting#creating-a-new-project`

This will talk about editing your `Dockerfile` and `Dockerfile-build` file

Further changes involve the following files:

- `pom.xml` - template has more than needed, trim down to just what you need
- `Jenkinsfile` - add `test = 'false'` if you don't have any build tests
- `server.xml` needs to move to `/src/main/liberty/`**`config`**`/`

Appendix B: Additional Features of Note for Java

This section contains additional topics which are not necessarily required to build microservices that are implemented in Java and intended to run in a Kubernetes environment. For the most part, these are alternatives which can be considered, but are not the primary approach that is recommended.

MicroProfile Tracing

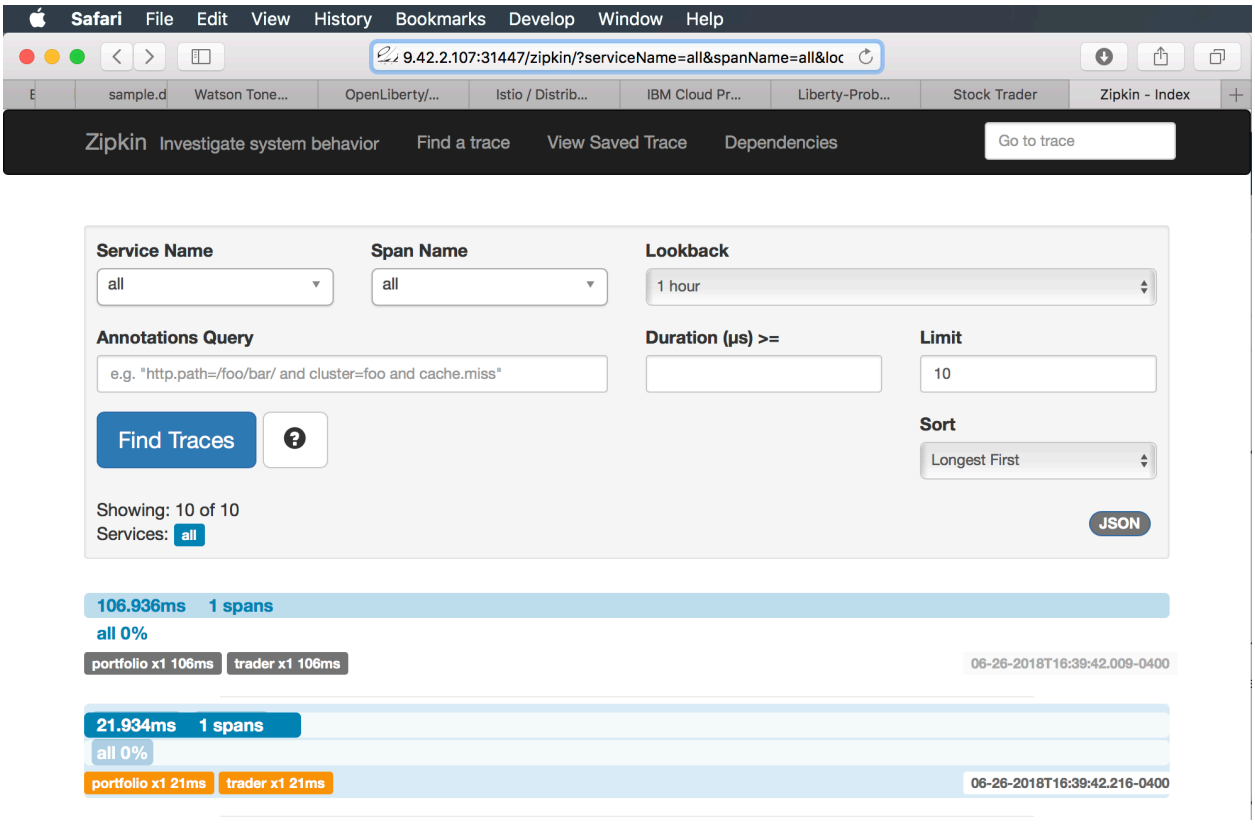
Unlike logging, which requires coding effort on the part of the developer, tracing is something that can be configured to happen automatically (though you can augment the trace programmatically if desired, as described below). You won't get as much info – just when various methods got called, and by whom – but sometimes just knowing you made it to a certain point in the code – and how long that code took to execute – can be helpful. MicroProfile 1.3 introduced an industry-standard way to work with tracing from Java-based microservices (and this was slightly tweaked in MicroProfile 2.0).

MP Open Tracing happens automatically, for JAX-RS methods as far as entry and exit tracing goes (and timings for each), as long as your server.xml enables the `mpOpenTracing-1.1` feature (or the convenience feature `microProfile-2.0`, that gives you all of the MP 2.0 features).

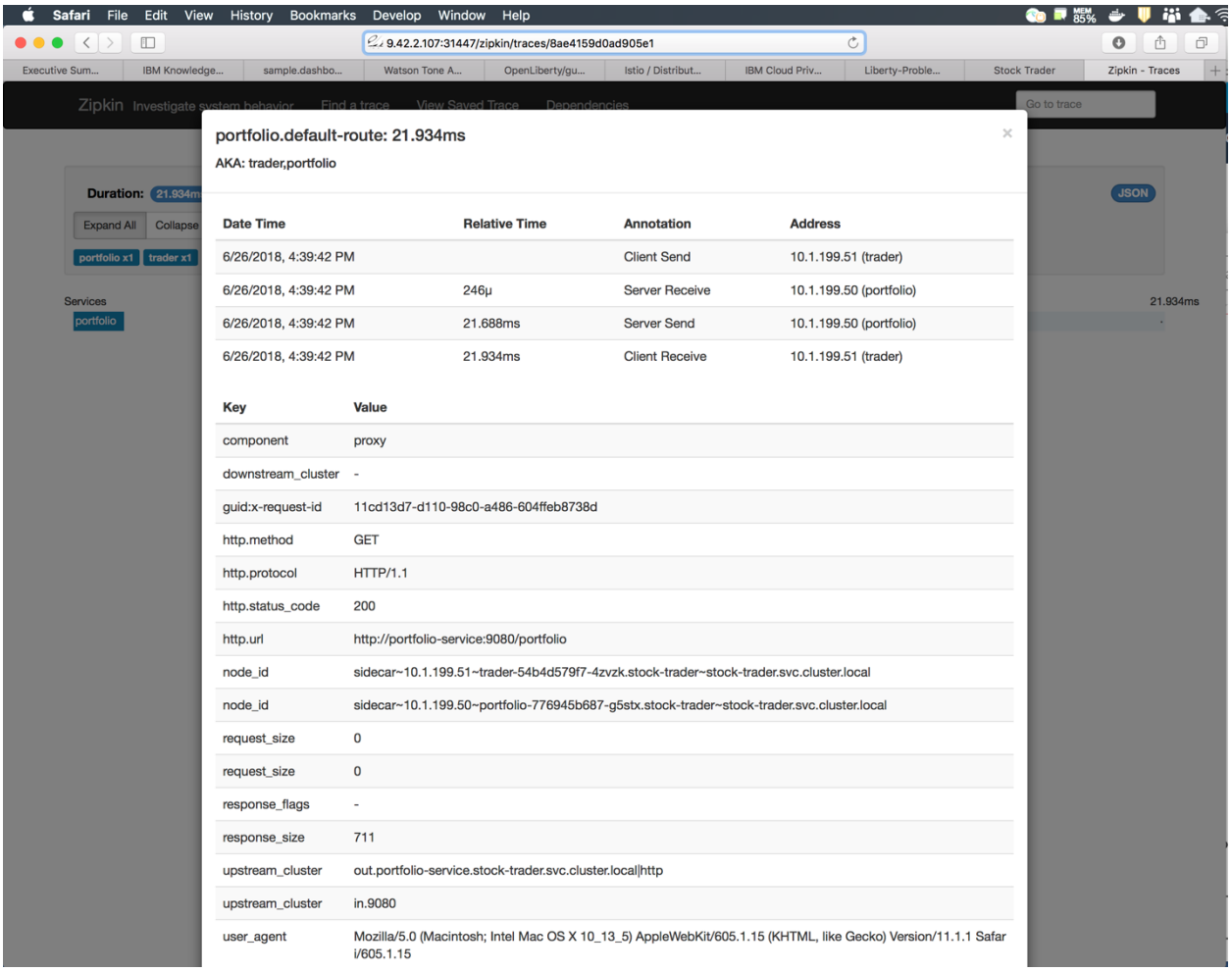
Zipkin

Note that, besides enabling this feature, you also need to enable a feature for the particular tracer implementation you want to use. For example, if you choose **Zipkin** (https://zipkin.io/pages/architecture.html) as your tracing server/UI, your server.xml would need to specify the `usr:opentracingZipkin-0.31` feature, which you'll need to have your Dockerfile download from Maven Central by having it get https://repo1.maven.org/maven2/net/wasdev/wlp/tracer/liberty-opentracing-zipkintracer/1.1/liberty-opentracing-zipkintracer-1.1-sample.zip and unzip it.

Once you do this, if you run your application, and then open the Zipkin dashboard (you can find its URL in the IBM Cloud Private console by going to Network Access->Services and clicking the link for Node Port for the "zipkin" service), you will see something like this:



If you click on one of the trace spans, it will show further information, like this:



mpOpenTracing annotations

You can trace non-JAX-RS methods by adding the `@Traced` annotation to such methods (you need to import `org.eclipse.microprofile.opentracing.Traced`). For example, I could get tracing for my private `invokeJDBC` method by annotating it like this:

```
@Traced
private void invokeJDBC(String command) throws SQLException {
```

If you want to take direct control of trace spans (an advanced topic you wouldn't often need, but which is possible), you can import the `io.opentracing.Tracer` and `io.opentracing.ActiveSpan` classes, and use CDI to inject an instance of the tracer, like this:

```
@Inject Tracer tracer;
```

And then you can use code like the following to start a new trace span called "Demo Span":

```
ActiveSpan childSpan = tracer.buildSpan("Demo
Span").startActive();
```

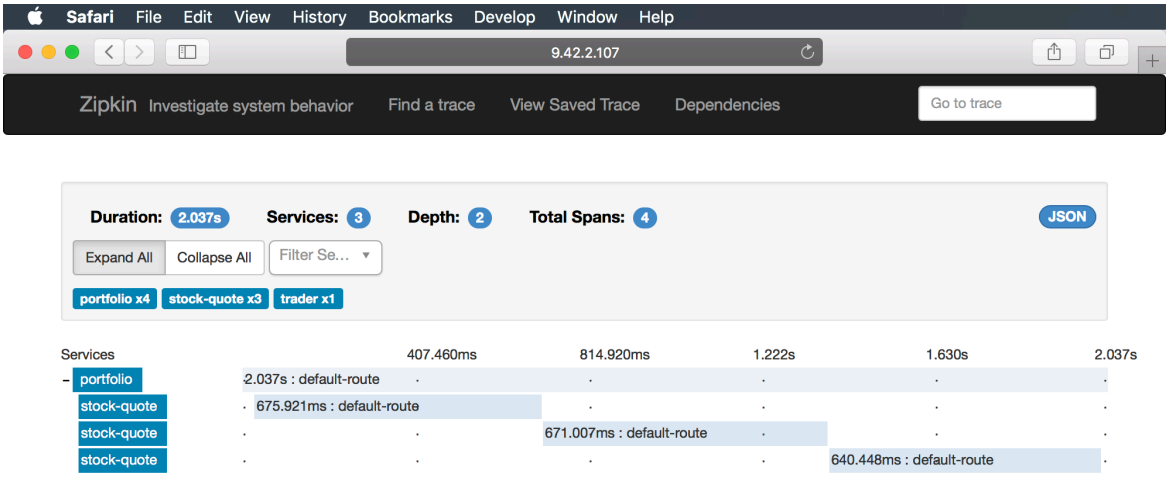
To build such code via Maven, you need to add the following dependency stanzas to your pom.xml (note the first one is only needed if doing the advanced topic of the manual trace span coordination):

```
<dependency>
    <groupId>io.opentracing</groupId>
    <artifactId>opentracing-api</artifactId>
    <version>0.31.0</version>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>org.eclipse.microprofile</groupId>
    <artifactId>microprofile</artifactId>
    <version>2.0.1</version>
    <type>pom</type>
    <scope>provided</scope>
</dependency>
```

See https://openliberty.io/guides/microprofile-opentracing.html for the full source code and how to build and run their example.

Performance tracing

One thing you can use the tracing data for is investigating performance issues. For example, here you can see that the `portfolio` microservice from Stock Trader is talking to the `stock-quote` microservice:



Among other things, you can see request timings. As you can see, the call to get the portfolio took just over 2 seconds, which is a long time to sit waiting in a browser. But we can see here that at total of 1.99 seconds of that was spent across three calls (one per stock in that particular portfolio) to the stock-quote microservice (which reaches out to API Connect in the cloud, which reaches out to IEX Trading on the internet). This is actually why we usually use Redis caching of stock quotes, to avoid this cost of multiple trips out on to the public internet to retrieve stock price data – but if the value isn't in Redis, or is too stale, you'll see the behavior above. As an aside, this trace suggests that perhaps we should go multi-threaded (via the Java EE Concurrency

API – JSR 236) and get each stock for a portfolio in parallel, rather than waiting for the first one to finish before starting the next one (especially if your portfolio had 100 stocks, for example).

Trace correlation

Note that basic tracing of "A called B" and "B called C", as two separate trace spans, happens automatically when mpOpenTracing is enabled. But if you want one trace span for "A called B called C", you need to explicitly propagate several http headers that get used for trace correlation. Here is the list of headers that need to be forwarded (which is reminiscent of the old ECSCurrentID/ECSParentID used in correlation of CEI events from tWAS):

- x-request-id
- x-b3-traceid
- x-b3-spanid
- x-b3-parentspanid
- x-b3-sampled
- x-b3-flags
- x-ot-span-context

See the mpRestClient discussion in the **Consuming APIs** section for a discussion on propagating headers and cookies on REST calls.

Jaeger

Finally, note that we've used **Zipkin** as our example tracing server/UI. That has been popular for a while now. But it appears that the industry is in the process of switching over to **Jaeger** (https://www.jaegertracing.io/docs/), which came from Uber and is endorsed by the Cloud Native Computing Foundation (CNCF), as the strategic tracing server (among other things, Zipkin didn't support proper security).

Switching to Jaeger should only require changing the Zipkin feature in your server.xml to a Jaeger feature (no Java code changes needed at all). See the *Istio* chapter for screenshots that show tracing the Stock Trader sample via Jaeger.

Note that there is currently not a tracer implementation built in to Liberty. We are working with the Liberty team to get a Jaeger tracer delivered as part of the app server. Due to this, ***we are currently recommending people avoid use of mpOpenTracing (using Istio's OpenTracing support instead)***.

MicroProfile Health API

The Health API feature of MicroProfile is a framework and related support that will accelerate the implementation of the **liveness** check of Kubernetes (https://www.ibm.com/support/knowledgecenter/en/SSEQTP_liberty/com.ibm.websphere.wlp.doc/ae/twlp_microprofile_healthcheck.html).

It is up to *Jane* the developer to decide what logic to implement in a liveness check. General guidance is to not do anything expensive, such as making a remote call, during a liveness (or readiness) check. Nor would Jane want to do any logging during a successful check, unless at

trace level not usually enabled, such as *fine*. Of course, she definitely would want to log something when deciding to indicate the health check has failed.

In our Stock Trader example, we took a simple approach of just keeping a counter of consecutive failures across any of our REST APIs. If something goes wrong and an http 200 is NOT going to be returned by a given operation, then we increment the counter. If a method completes successfully and is going to return a 200, we reset the counter to zero. Our liveness check therefore is simply comparing this counter to a threshold. Others may choose to implement more sophisticated logic for a liveness check, but we kept it simple, only telling Kube to kill us if we are consistently being unable to fulfill our API contract. In our `PortfolioService.java`, our liveness method is just a single line of code:

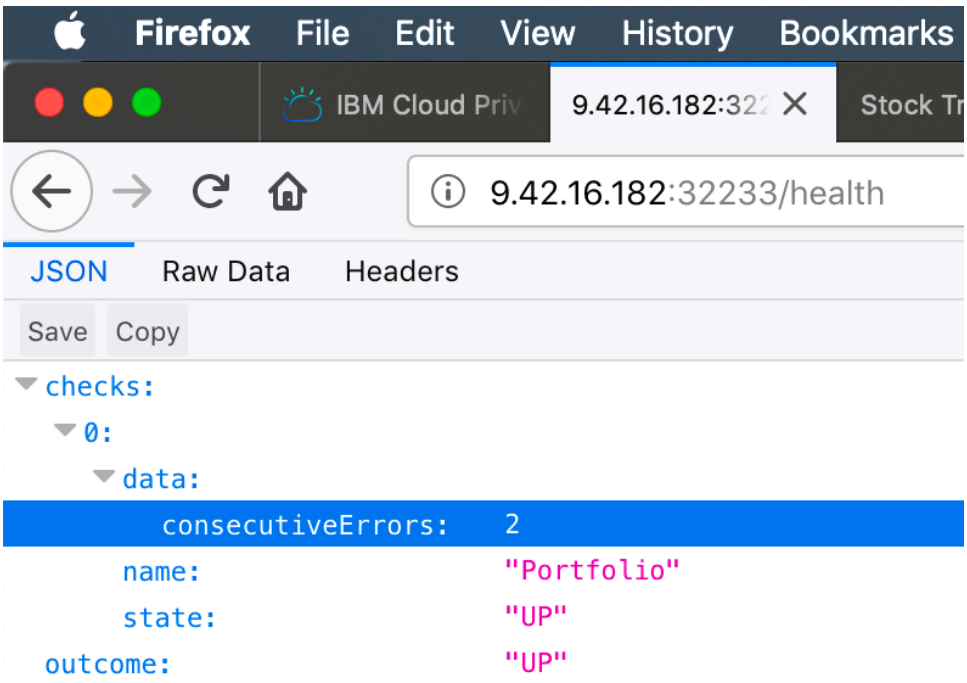
```
    //determines answer to livenesss probe
    public static boolean isHealthy() {
        return consecutiveErrors<MAX_ERRORS;
    }
```

With *mpHealth*, you implement the `org.eclipse.microprofile.health.HealthCheck` interface (which has a single `call()` method), and add the `@Health` annotation (from the same package) to the class implementing that interface (which could be your main JAX-RS class, if you wish). This will make a `/health` endpoint available in your server, which will return an http 200 if you answer that your app is healthy, or an http 503 if not. You can also choose to embed other info in the health response – in our case, we include the value of the *consecutiveErrors* counter:

```
    //mpHealth probe
    @Override
    public HealthCheckResponse call() {
        HealthCheckResponse response = null;
        try {
            HealthCheckResponseBuilder builder =
HealthCheckResponse.named("Portfolio");

            if (PortfolioService.isHealthy()) {
                builder = builder.up();
                logger.fine("Returning healthy!");
            } else {
                builder = builder.down();
                logger.warning("Returning NOT healthy!");
            }

            builder = builder.withData("consecutiveErrors",
PortfolioService.consecutiveErrors);

            response = builder.build();
        } catch (Throwable t) {
```

```
            logger.warning("Exception occurred during health
check: "+t.getMessage());
            logException(t);
            throw t;
        }

        return response;
    }
```

You can of course hit this `/health` endpoint directly (in a browser, or via `curl`), to see what is being returned. You'd want to use the node port for your service, since you wouldn't have an ingress definition for this (after all, the endpoint is named the same, `/health`, in each of your Liberty-based microservices using *mpHealth*). For example:



You tell Kubernetes to enable the liveness check via a stanza in your deployment yaml (which could be embedded in a helm chart, of course). Note you can specify that it has to fail several checks in a row (the `failureThreshold`) before actually killing the pod if desired; in our case, this would allow a window during which perhaps a successful API request would occur, which would reset the counter and effectively cancel the killing of the pod. This delay also allows a brief period of time to grab logs off of the pod before it is killed (in case you aren't directing your logs to a persistent volume).

```
    imagePullPolicy: Always
    livenessProbe:
      httpGet:
        path: /health
        port: 9080
      initialDelaySeconds: 90
```

```
      periodSeconds: 10
      successThreshold: 1
      failureThreshold: 5 # Allow some time to kubectl exec into
the pod (or kubectl cp from it) to grab FFDCs before it's killed
```

Note that the `/health` URI is NOT under your web app's context root – which can actually be a good thing, since your web app may specify security settings you wouldn't want in place for the health check, such as requiring a JWT on the request (Kubernetes would have no way to create/populate/sign such a JWT to include in the header to the request).

You can also choose not to implement the above in your Java code, but still turn on the `mpHealth-1.0` feature (or the `microProfile-2.1` "umbrella" feature) in your server.xml. In that case, the `/health` endpoint will always return a 200, meaning that if the server is healthy enough to respond to this endpoint, then your pod will be considered healthy (and otherwise, Kubernetes will kill the pod and start a fresh one – all without you having to do anything at all in your code).

Note that Kubernetes defines two endpoints: **liveness** and **readiness**. Currently mpHealth only defines a single endpoint. There is a proposal that the next version of mpHealth have two separate endpoints, one for each expected by Kubernetes. There is also a proposal to have some automatic liveness checks in Liberty that you can enable in your server.xml without writing any code, like that would base the response on whether the server has been able to make a successful connection to certain features enabled for that server, like `jdbc-4.2` or `jms-2.0`, or base it on *mpMetrics* values like JVM memory usage or even custom metrics. ***At this time, we are recommending avoiding use of mpHealth until such a newer version becomes available.***

MicroProfile Fault Tolerance

Before introducing and explaining MicroProfile fault tolerance, realize that our recommended approach for fault tolerance is to leverage Istio. Istio provides a language independent approach for the various features aggregated under fault tolerance. That said, there is one feature, namely fallback, from MicroProfile Fault tolerance which can be used in conjunction with the Istio features. This is the `MP_Fault_Tolerance_NonFallback_Enabled` configuration property. Set this to false. Then it is possible to leverage Istio and the fallback feature of MicroProfile fault tolerance together, without the features conflicting with each other.

The following sections then should be embraced with the following guidance in mind:

1. Read about Fallback if your project has embraced Istio and is in need of a fallback capability, or might be in need of one. Skip the rest of this section.
2. Read the whole section on MicroProfile Fault Tolerance if your project is not using Istio at this time, or you are looking for a comprehensive view of what MicroProfile Fault Tolerance is capable of and how it should be leveraged

Introduction to MicroProfile Fault Tolerance

MicroProfile defines an mpFaultTolerance feature, which lets you define policies that the runtime should enforce around your microservice's JAX-RS operations. You can define annotations that control define how to respond to various kind of failure situations, such as:

- that you'd like the runtime to enforce automatic retries on failures
- how long to wait before giving up on a request (timeouts)
- disable an operation for a while if it has repeatedly failed (circuit breaker)
- how many concurrent callers should be allowed to a given JAX-RS method (bulkheads)
- an alternative implementation to invoke if the main one isn't working (fallback).

Like many of the other MicroProfile technologies, you specify Fault Tolerance properties via method-level annotations. Each of the annotations come from the `org.eclipse.microprofile.faulttolerance` package, which you'd need to import, and you'd need the same MicroProfile dependency stanza in your Maven pom.xml as shown in earlier MicroProfile sections (like mpConfig) to get such code to compile.

Note that by combining mpFaultTolerance with mpConfig, you can make it so things like retry counts, for example, can be passed in from Kube config maps/secrets, so that changes could be made without having to rebuild and redeploy the Docker container.

Of all of the MicroProfile technologies, this is the one that is most directly in conflict with capabilities from the underlying infrastructure of Kubernetes and Istio. That being said, this does offer a programmatic approach, via Java annotations, to addressing fault tolerance issues, as opposed to the declarative approach offered by Istio policy rules, and some people prefer the programmatic approach.

As stated before, our general recommendation is to use the (polyglot) Istio approach if possible, and only to drop down into the language-specific when you have specific reasons to do so.

Retry

Let's look at *Retry* first, as applied to our `getStockQuote` method. Here we specify that if the method fails, we'll automatically retry it up to two times before giving up:

```
    @GET

    @Path("/{symbol}")

    @Produces("application/json")

    @Retry(maxRetries = 2)

    /**  Get stock quote from API Connect */

    public JsonObject getStockQuote(@PathParam("symbol") String symbol)
throws IOException {
```

Timeout

Now let's add a *Timeout* to the above example. Let's say that if the method takes longer than a second to execute, it should be automatically aborted (and then retried, per above, due to that failure), rather than just hanging.

```
    @GET

    @Path("/{symbol}")

    @Produces("application/json")

    @Retry(maxRetries=2)

    @Timeout(value = 1, unit = ChronoUnit.SECONDS)

    /**  Get stock quote from API Connect */

    public JsonObject getStockQuote(@PathParam("symbol") String symbol)
throws IOException {
```

Note that there is currently an implementation issue in Liberty that keeps calls made via mpRestClient from timing out (something about the underlying CXF library not being interruptible), so given that we are using mpRestClient to call our API Connect-based stock quote REST service, this Timeout would not be honored. Note that the Istio flavor of Timeout would be honored, as there it is implemented in the sidecar's call to its container.

Circuit Breaker

Another thing we can do is to say that, if a specified number of consecutive requests have failed, disallow all use of that method until a specified amount of time has passed. For our same example as before, let's say we want the circuit to break if we've had three failures in a row, and not allow any traffic for 5 seconds in such a situation:

```
    @GET

    @Path("/{symbol}")

    @Produces("application/json")

    @Retry(maxRetries=2)

    @Timeout(value = 1, unit = ChronoUnit.SECONDS)

    @CircuitBreaker(delay = 5, delayUnit = ChronoUnit.SECONDS,
requestVolumeThreshold = 3, failureRatio = 1.0)

    /**  Get stock quote from API Connect */

    public JsonObject getStockQuote(@PathParam("symbol") String symbol)
throws IOException {
```

Bulkhead

We can also limit concurrency. For example, say we only want to allow 5 concurrent calls to our example above:

```
    @GET

    @Path("/{symbol}")

    @Produces("application/json")

    @Retry(maxRetries=2)

    @Timeout(value = 1, unit = ChronoUnit.SECONDS)

    @CircuitBreaker(delay = 5, delayUnit = ChronoUnit.SECONDS,
requestVolumeThreshold = 3, failureRatio = 1.0)

    @Bulkhead(5)

    /**  Get stock quote from API Connect */

    public JsonObject getStockQuote(@PathParam("symbol") String symbol)
throws IOException {
```

Note that this limit is per Liberty JVM. If your Liberty-based microservice was deployed to Kubernetes, and scaled up to 3 pods, the above would actually allow up to 15 concurrent calls from Portfolio to Stock Quote (5 per pod).

Fallback

All of the features described so far can also be done via Istio for non-secure calls, and generally we recommend doing so via Istio instead. Doing both will produce unexpected and undesirable results; for example, if using an mpFaultTolerance Retry of 5, and an Istio Retry of 3, you'd actually end up retrying that method 15 times before control returning to the caller. In fact, there is a special config property you can define: `MP_Fault_Tolerance_NonFallback_Enabled=false`, which you can use to disable all of the mpFaultTolerance features discussed so far if Istio is present (see https://microprofile.io/project/eclipse/microprofile-fault-tolerance/spec/src/main/asciidoc/configuration.asciidoc for details).

However, one cool feature not in Istio is the ability to define a fallback handler that should be called if the method fails. To further augment our ongoing mpFT example, let's say we want a `getStockQuoteViaIEX` method to get called if we can't get a stock quote from API Connect (the implementation of that could be to call IEX Trading directly, rather than going through our API Connect façade for it, in case APIC was having a problem at the moment):

```
    @GET

    @Path("/{symbol}")

    @Produces("application/json")

    @Retry(maxRetries=2)
```

```
    @Timeout(value = 1, unit = ChronoUnit.SECONDS)

    @CircuitBreaker(delay = 5, delayUnit = ChronoUnit.SECONDS,
requestVolumeThreshold = 3, failureRatio = 1.0)

    @Bulkhead(5)

    @Fallback(fallbackMethod = "getStockQuoteViaIEX")

    /** Get stock quote from API Connect */

    public JsonObject getStockQuote(@PathParam("symbol") String symbol)
throws IOException {
```

This would mean that if the `getStockQuote` method failed while trying to call API Connect, it would automatically fall back to calling this method instead (note you wouldn't put the JAX-RS annotations on the fallback method – you don't want it to be able to be called directly):

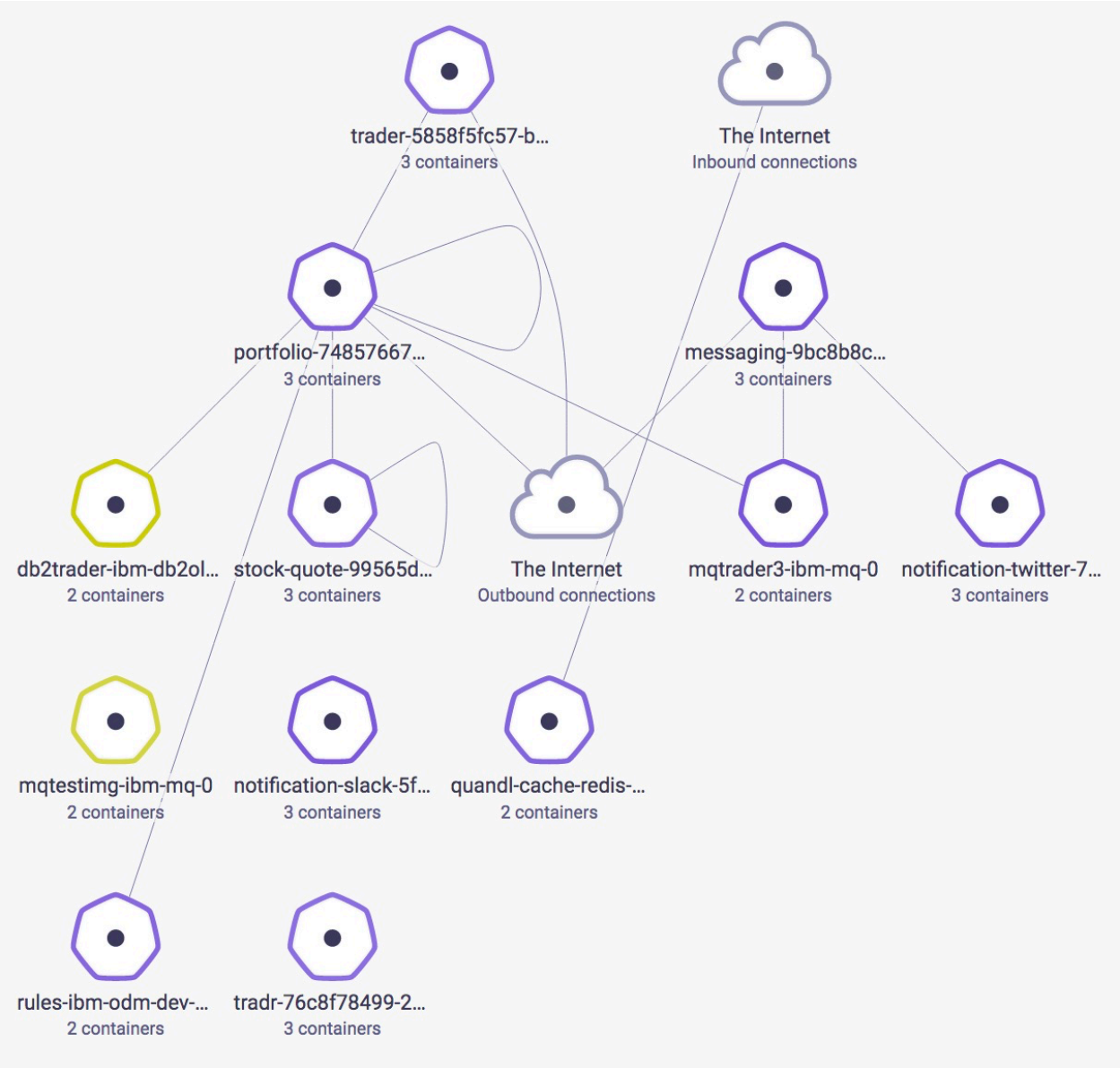
```
    /**  Get stock quote directly from IEX Trading */

    public JsonObject getStockQuoteViaIEX(String symbol) throws
IOException {
```

Istio would have no way to know what such custom logic to run if a call failed; you really need to write code, based on your domain knowledge, to say what to do in case of a failure. Note that the fallback method must have the same method signature, and will receive the same parameters as sent to the original method.

If you have to make a secure call to a service (e.g. you need to call a service via https), you will need to use MP Fault Tolerance as Istio cannot intercept secure requests.

Weave Scope

One last thought on visualizing control flow between microservices: you can use something like Weave Scope (https://www.weave.works/docs/net/latest/kubernetes/kube-addon/) to visually view such calls between pods in Kubernetes, as shown here (compare this dynamically generated diagram to the Stock Trader PowerPoint diagram from the beginning of this book):



Note that Istio has some similar technologies, as seen in its Kiali component.

Appendix C – Istio Installation

Installation

Istio 1.0.0 became generally available on July 31st, 2018. As of this release, it is now GA quality, and a full member of the ICP (and IKS) content family, with its own helm chart in the ICP catalog.

Istio Versions

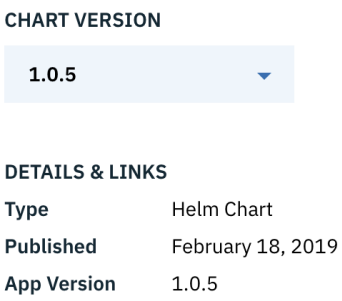
The ICP product is delivered with version of Istio that can be automatically installed as part of the ICP installation. ICP 3.1.2 is delivered with Istio 1.0.2. Alternatively, Istio can be installed in an existing deployment of ICP using the IBM curated Helm chart located in the ICP catalog. The Istio community is actively developing the framework and new patch levels are released frequently. It is possible to install Istio from the open source community if a newer version is required, however this will also require specific ICP configuration changes as well.

The installation of Istio is sensitive to what release of Helm is installed:

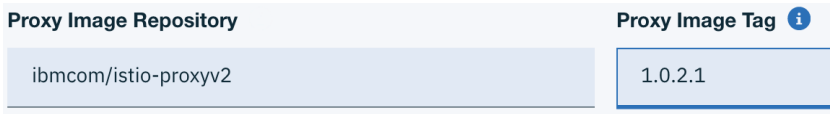
```
helm version -tls
```

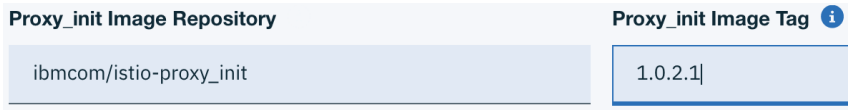
Helm 2.9 (ICP 3.1.0) will automatically install the Istio CRDs, but must be manually removed upon uninstall. *(See: https://istio.io/docs/setup/kubernetes/helm-install/ for command format.)*

Note: The IBM Helm chart version and Istio framework versions do not match. The Helm chart and open source evolve independently, and their versions are not consistent. As an example, the 1.05 Version of the Helm chart installs the 1.0.2.1 version of Istio.



The version of the Istio Docker images being downloaded are displayed on the configuration page:





Automatic Istio Install

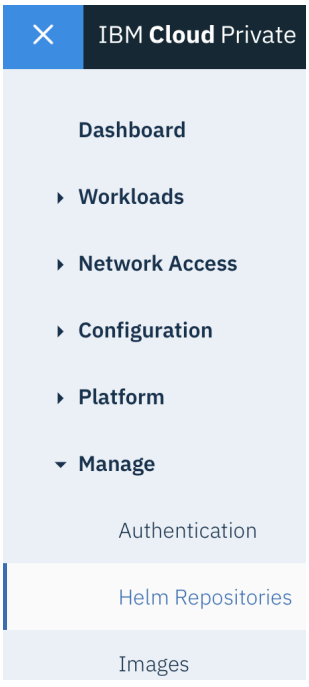
By default, Istio is ***not*** installed during ICP installation. Prior to installing ICP, changing "`istio: disabled`" to "`istio: enabled`" in the "`management services`" section of the ICP `config.yaml` file, Istio will also be installed.

(See https://www.ibm.com/support/knowledgecenter/SSBS6K_3.1.1/installing/config_yaml.html for details.)

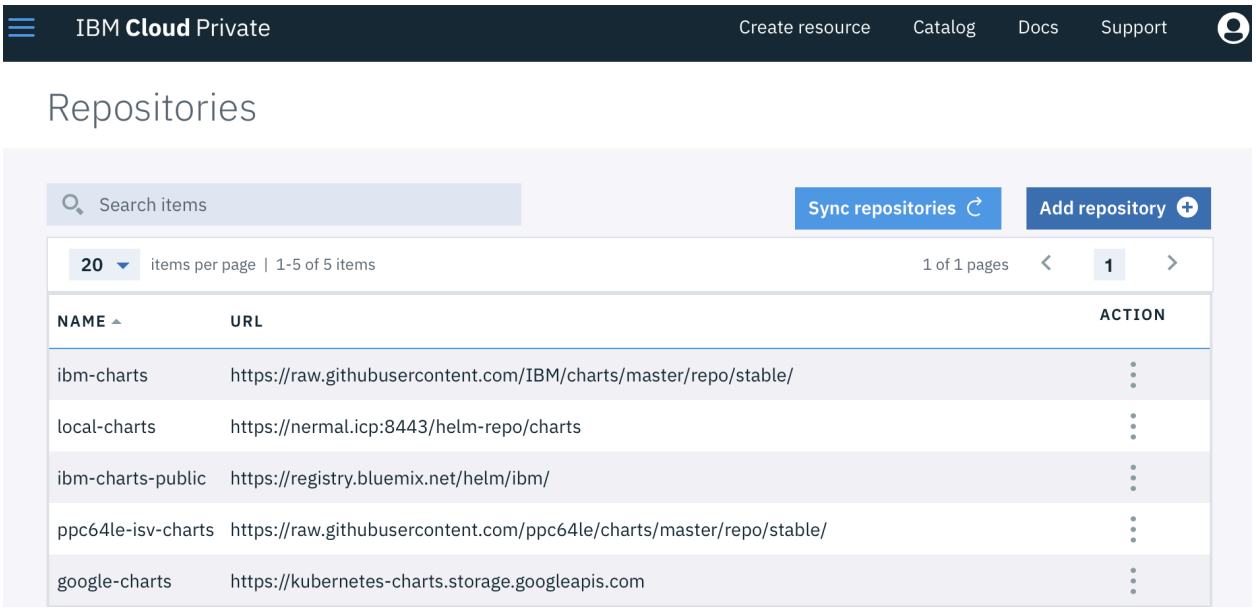
Manual Istio Install

Istio can be added manually to an existing ICP deployment. Istio is installed in ICP via an IBM curated Helm Chart. Prior to installation, ensure the latest Helm chart is available in the ICP catalog by Synching Repositories.

Go to Manage->Helm Repositories:



And click on the blue "Synch repositories" button:



The sync repositories step may take several minutes to complete.

The Istio framework is installed in its own namepace: `istio-system`. First, make sure the `istio-system` namespace exists:

```
kubectl get ns
```

If the `istio-system` namespace does not exist, create it:

```
kubectl create ns istio-system
```

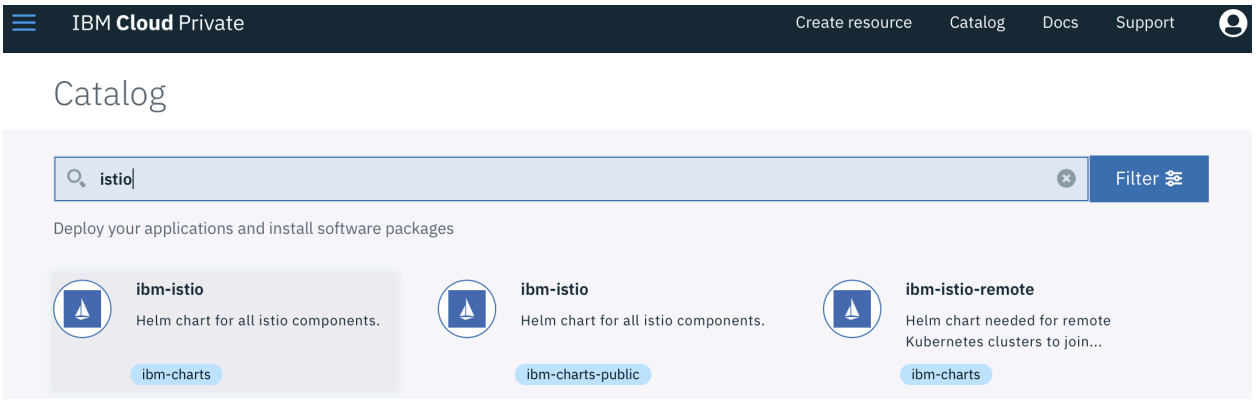
As with any other Helm chart, the CLI or UI can be used to customize your Istio installation.

To install Istio using the CLI

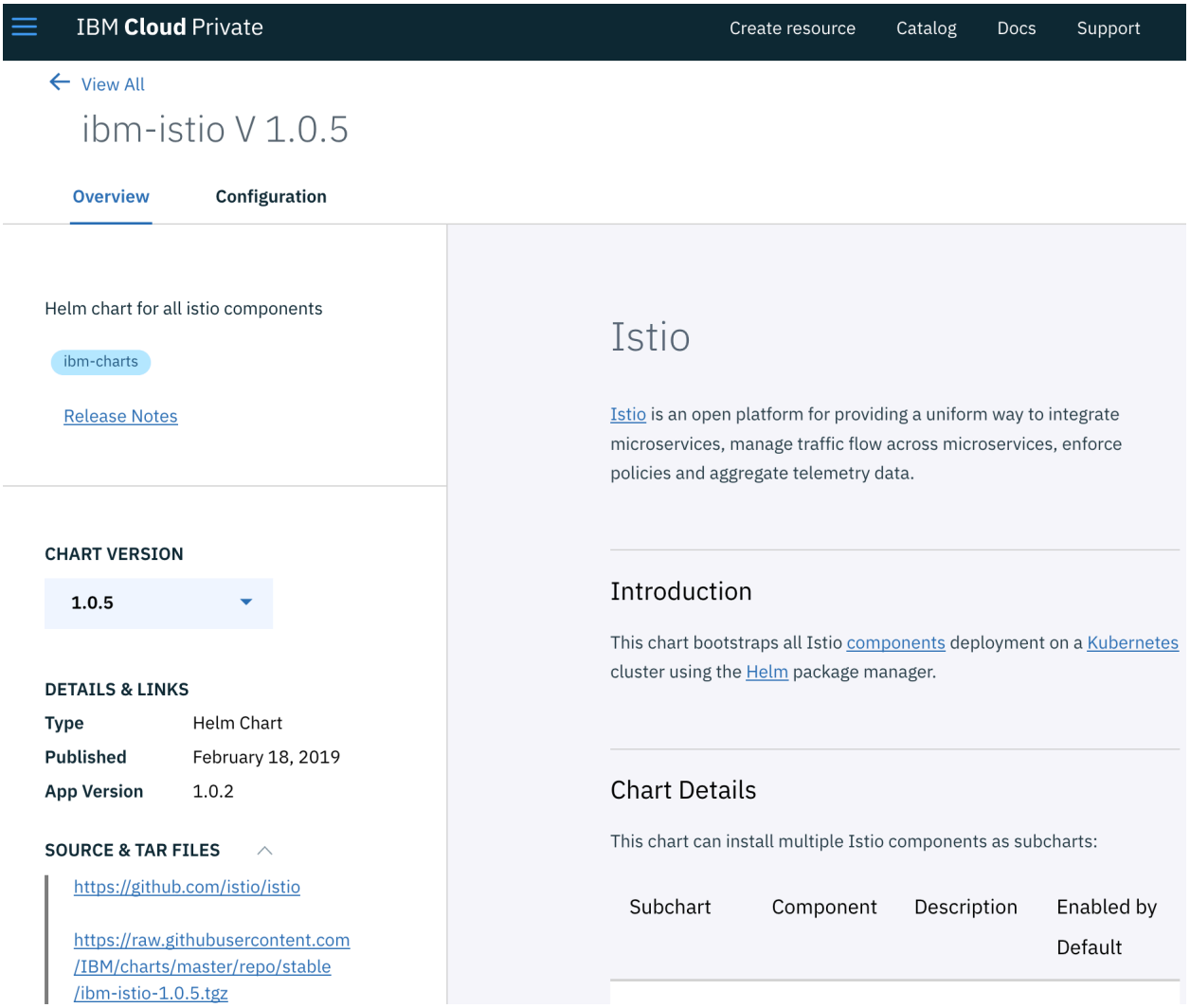
```
helm install ibm-istio --name istio --namespace istio-system
```

To install Istio using the ICP Catalog UI

Login to the ICP console and click on Catalog in the top black bar, and type "`istio`" in the search box:



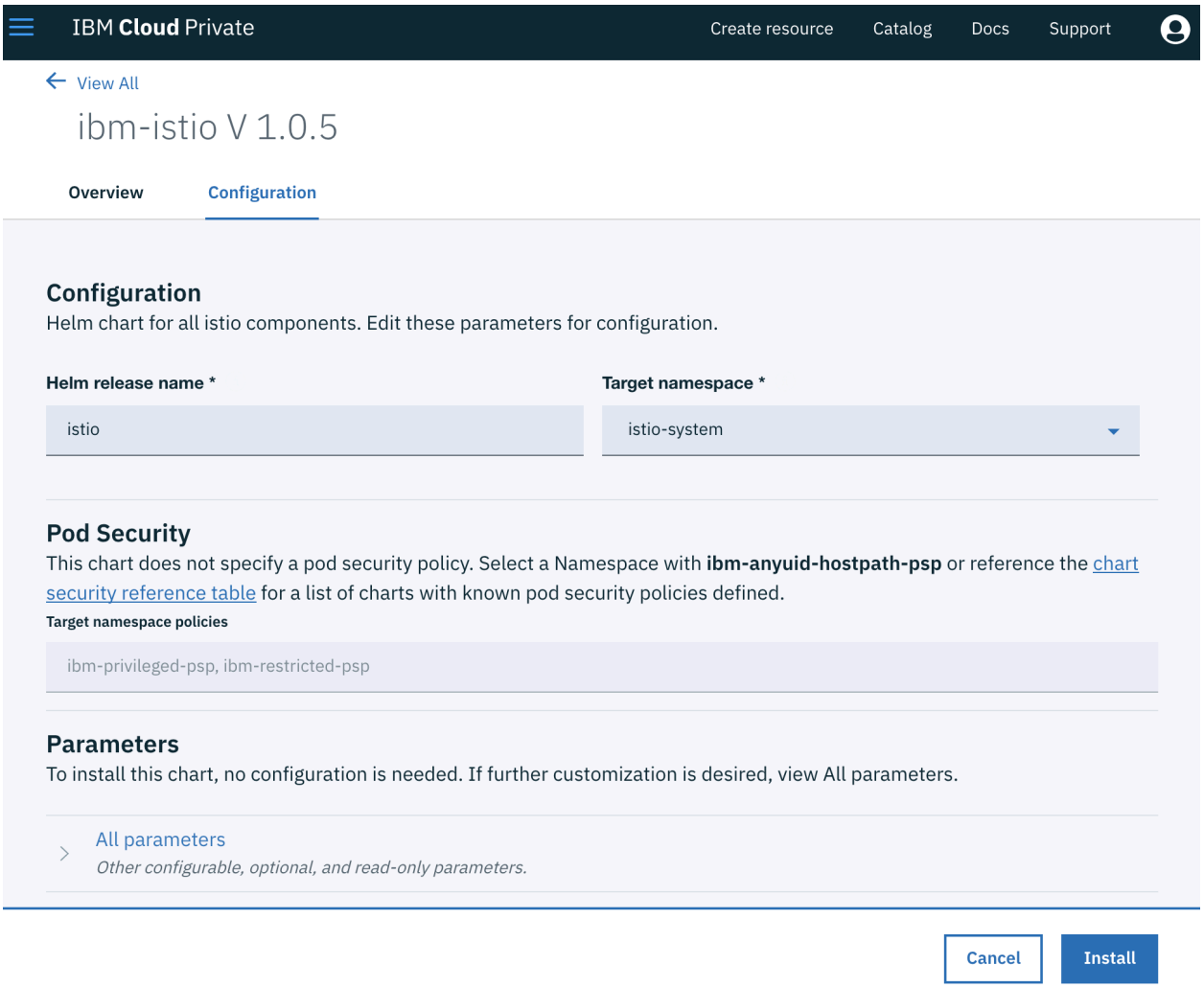
The catalog may show multiple choices for Istio. In the case shown above, there are two `ibm-istio` charts. For ICP, select the one deployed in the `ibm-charts` repository.



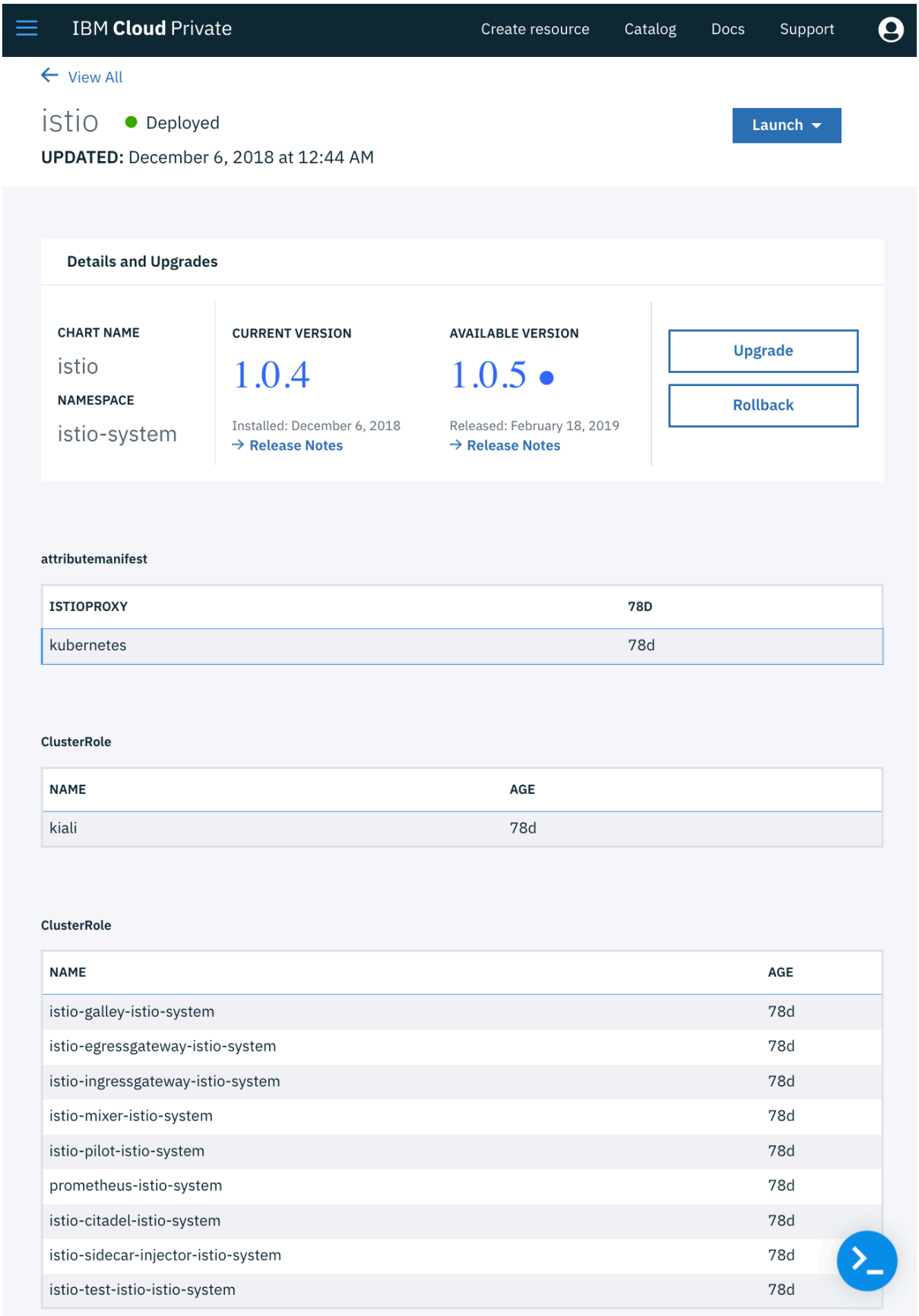
The main Istio Helm chart actually aggregates a bunch of sub-charts for each of its constituent components:

Subchart	Component	Description	Enabled by Default
ingress	Ingress	An ingress implementation with envoy proxy that allows inbound connections to reach the mesh. This deprecated component is used for legacy Kubernetes Ingress resources with Istio routing rules.	No
gateways	Gateways	A platform independent Gateway model for ingress & egress proxies that works across Kubernetes and Cloud Foundry and works seamlessly with routing.	Yes
sidecarinjectorwebhook	Automatic Sidecar Injector	A mutating webhook implementation to automatically inject an envoy sidecar container into application pods.	Yes
galley	Galley	The top-level config ingestion, processing and distribution component for Istio, responsible for insulating the rest of the Istio components from the details of obtaining user configuration from the underlying platform.	Yes
mixer	Mixer	A centralized component that is leveraged by the proxies and microservices to enforce policies such as authorization, rate limits, quotas, authentication, request tracing and telemetry collection.	Yes
pilot	Pilot	A component responsible for configuring the proxies at runtime.	Yes
security	Citadel	A centralized component responsible for certificate issuance and rotation.	Yes
telemetrygateway	Telemetry gateway	A gateway for configuring Istio telemetry addons	No
grafana	Grafana	A visualization tool for monitoring and metric analytics & dashboards for Istio	No
prometheus	Prometheus	A service monitoring system for Istio that collects metrics from configured targets at given intervals, evaluates rule expressions, displays the results, and can trigger alerts if some condition is observed to be true.	Yes
servicegraph	Service Graph	A small add-on for Istio that generates and visualizes graph representations of service mesh.	No
tracing	Jaeger	Istio uses Jaeger as a tracing system that is used for monitoring and troubleshooting Istio service mesh.	No
kiali	Kiali	Kiali works with Istio to visualise the service mesh topology, features like circuit breakers or request rates.	No
certmanager	Cert-Manager	An Istio add-on to automate the management and issuance of TLS certificates from various issuing sources.	No

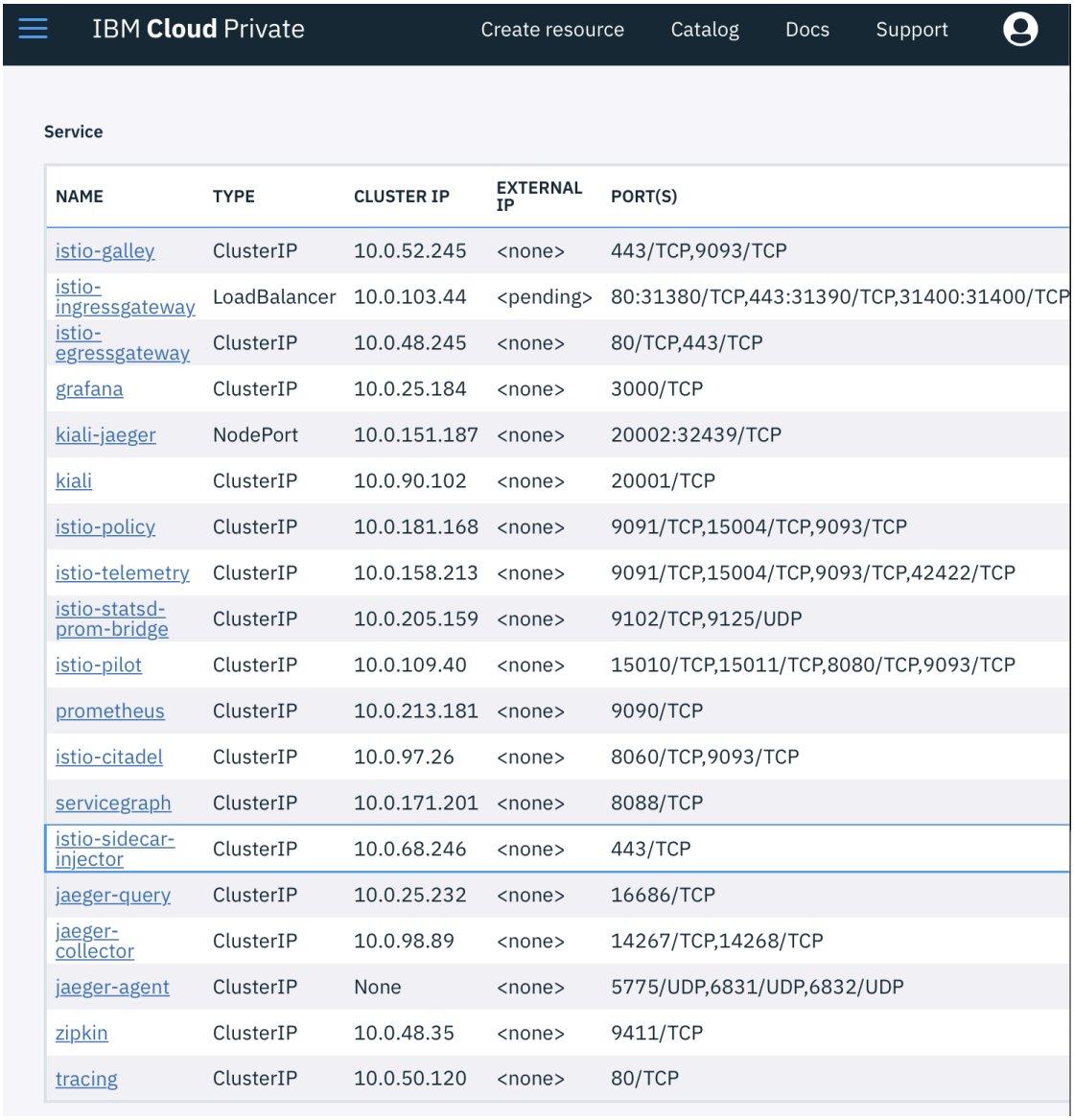
Click on the blue Configure button. Name the Helm release, in this example: `Istio`. IBM recommends using `istio-system` namespace as this is the default Istio namespace and many examples, scripts and other collateral assume this location. There are dozens of other customizable fields visible if the "All Parameters" twisty is expanded. Unless there is a specific reason to change the values however, the defaults are sufficient.



Once the Helm install completes, Istio will be installed. Navigate to "Workloads->Helm Releases" for this newly installed Istio helm chart to see the status of the installed Istio components:


IBM Cloud Private
Create resource
Catalog
Docs
Support
View All
istio
Deployed
Launch
UPDATED: December 6, 2018 at 12:44 AM
Details and Upgrades
CHART NAME
istio
NAMESPACE
istio-system
CURRENT VERSION
1.0.4
Installed: December 6, 2018
Release Notes
AVAILABLE VERSION
1.0.5
Released: February 18, 2019
Release Notes
Upgrade
Rollback
attributemanifest
ISTIOPROXY
78D
kubernetes
78d
ClusterRole
NAME
AGE
kiali
78d
ClusterRole
NAME
AGE
istio-galley-istio-system
78d
istio-egressgateway-istio-system
78d
istio-ingressgateway-istio-system
78d
istio-mixer-istio-system
78d
istio-pilot-istio-system
78d
prometheus-istio-system
78d
istio-citadel-istio-system
78d
istio-sidecar-injector-istio-system
78d
istio-test-istio-istio-system
78d

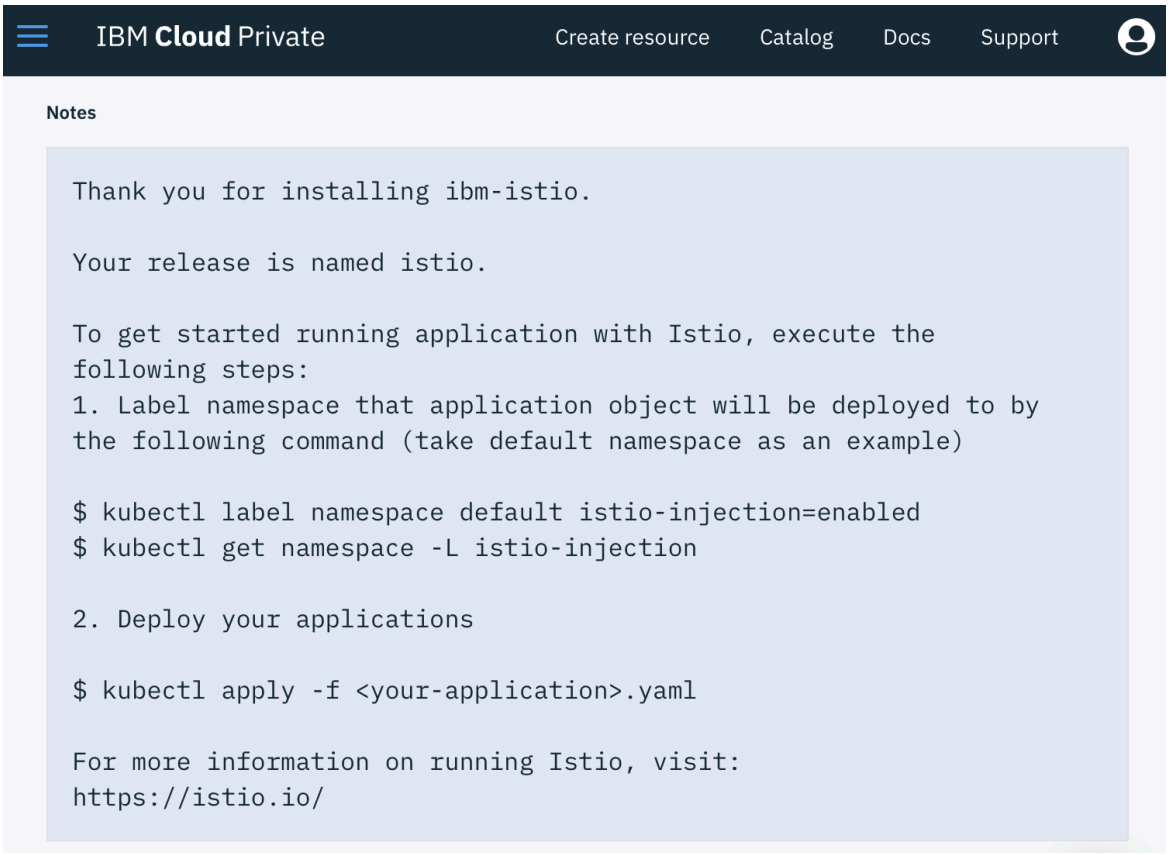
The Istio framework has a lot going on and there are many pages worth of data here. Scrolling down to `Services` displays the external endpoints available to be displayed via browser.



NAME	TYPE	CLUSTER IP	EXTERNAL IP	PORT(S)
istio-galley	ClusterIP	10.0.52.245	<none>	443/TCP,9093/TCP
istio-ingressgateway	LoadBalancer	10.0.103.44	<pending>	80:31380/TCP,443:31390/TCP,31400:31400/TCP
istio-egressgateway	ClusterIP	10.0.48.245	<none>	80/TCP,443/TCP
grafana	ClusterIP	10.0.25.184	<none>	3000/TCP
kiali-jaeger	NodePort	10.0.151.187	<none>	20002:32439/TCP
kiali	ClusterIP	10.0.90.102	<none>	20001/TCP
istio-policy	ClusterIP	10.0.181.168	<none>	9091/TCP,15004/TCP,9093/TCP
istio-telemetry	ClusterIP	10.0.158.213	<none>	9091/TCP,15004/TCP,9093/TCP,42422/TCP
istio-statsd-prom-bridge	ClusterIP	10.0.205.159	<none>	9102/TCP,9125/UDP
istio-pilot	ClusterIP	10.0.109.40	<none>	15010/TCP,15011/TCP,8080/TCP,9093/TCP
prometheus	ClusterIP	10.0.213.181	<none>	9090/TCP
istio-citadel	ClusterIP	10.0.97.26	<none>	8060/TCP,9093/TCP
servicegraph	ClusterIP	10.0.171.201	<none>	8088/TCP
istio-sidecar-injector	ClusterIP	10.0.68.246	<none>	443/TCP
jaeger-query	ClusterIP	10.0.25.232	<none>	16686/TCP
jaeger-collector	ClusterIP	10.0.98.89	<none>	14267/TCP,14268/TCP
jaeger-agent	ClusterIP	None	<none>	5775/UDP,6831/UDP,6832/UDP
zipkin	ClusterIP	10.0.48.35	<none>	9411/TCP
tracing	ClusterIP	10.0.50.120	<none>	80/TCP

A view of the pods shows:

And finally, scrolling down to the very bottom of the Helm Release to the Notes section for some final guidance:



Advanced Istio Install

At times, it may be required to install the latest version of Istio from the open source community in the absence of an existing IBM Helm chart. The following example and guidance of installing 1.0.6 using the open source community release on ICP 3.1.0.

Download Istio, add istioctl command path to your shell's path.

```
curl -L https://git.io/getLatestIstio | sh -
cd istio-1.0.6
export PATH=$PWD/bin:$PATH
```

In this example, global side-car injection is disabled by modifying the Istio values.yaml:

vi /install/kubernetes/helm/istio/values.yaml

Change: autoInject to disabled

```
    # This controls the 'policy' in the sidecar injector.
    autoInject: disabled

    # Sets the destination Statsd in envoy (the value of the "--statsdUdpAddress"
proxy argument
    # would be <host>:<port>).
    # Disabled by default.
    # The istio-statsd-prom-bridge is deprecated and should not be used moving
forward.
```

Install the helm chart via CLI into the `istio-system` namespace.

```
helm install install/kubernetes/helm/istio --name istio --namespace istio-system --tls
```

Wait for the pods to fully initialize.

Note: The default istio.io install does not install Grafana nor Kiali monitors.

Istio Client Install

The `istioctl` CLI is useful for not only manually injecting side-car containers, but also displaying various Istio configuration and state information. Like, Helm and linters, the `istioctl` CLI is sensitive to the version of Istio deployed. Choose the archive for your platform at https://github.com/istio/istio/releases/. For example, `istio-1.0.2-osx.tar.gz` on a Mac. Decompress it, and add the "`istioctl`" executable to your PATH. One can copy the `istioctl` file to `/usr/local/bin` or you can simply add the `/bin` directory of selected release to your PATH.

Note: The `istioctl` command does not detect version mismatches and one may experience strange side-car injection errors like:

```
Error: failed to convert to proto. unknown field "policyCheckFailOpen" in
v1alpha1.MeshConfig
```

Check to make sure that your PATH is still correct and validate that the `istioctl` command is the expected version:

```
istioctl version
```

Verifying Istio Install

Congratulations! Istio has been installed into your Kubernetes environment. If Istio was not installed via the automatic install process, it is generally a good idea to verify the installation.

The Istio community is developing a self-verifying command and expects to deliver this feature in release 1.1. Prior to verification capability, however use the Bookinfo application from Istio to validate your deployment.

ICP 3.1.0 Egress issue

Prior to ICP 3.1.1, the `ibm-image-enforcement` document did not conform to Istio port naming conventions. For ICP 3.1.0, correct the document:

```
kubectl delete svc ibmcloud-image-enforcement -n kube-system

cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
  labels:
    app: ibmcloud-image-enforcement
    chart: ibmcloud-image-enforcement-3.1.0
    heritage: Tiller
    release: image-security-enforcement
  name: ibmcloud-image-enforcement
  namespace: kube-system
spec:
  clusterIP: 10.0.0.45
  ports:
  - name: https-image-enforcement
    port: 443
    protocol: TCP
    targetPort: 8000
  selector:
    app: ibmcloud-image-enforcement
    release: image-security-enforcement
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
EOF
```

Create a namespace for the `bookinfo` example:

```
kubectl create ns bookinfo
cloudctl target -n bookinfo
```

ICP has image policy that will prevent images from being loaded from the sample docker containers, permit loading from the sample image locations in the `bookinfo` namespace.

```
cat <<EOF | kubectl apply -f -
apiVersion: securityenforcement.admission.cloud.ibm.com/v1beta1
kind: ImagePolicy
metadata:
    name: bookinfo-istio-image-policy
    namespace: bookinfo
spec:
    repositories:
    - name: docker.io/istio/*
      policy:
        va:
          enabled: false
    - name: docker.io/citizenstig/*
      policy:
        va:
          enabled: false
    - name: docker.io/pstauffer/curl
      policy:
        va:
          enabled: false
EOF
```

Define the `istio-user` role authorization document for the `bookinfo` namespace. *Note:* ICP 3.1.0 does not have the predefined `ibm-` roles, change `ibm-priviledged-clusterrole` to `priviledged` in the ClusterRoleBinding document below:

```
cat <<EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: istio-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: ibm-privileged-clusterrole
subjects:
- kind: ServiceAccount
  name: default
  namespace: bookinfo
EOF
```

Testing ingress, follow the directions here: https://istio.io/docs/tasks/traffic-management/ingress/

Testing egress, following the directions here: https://istio.io/docs/examples/advanced-egress/egress-tls-origination/

Note: The documentation at istio.io shows the usage of `/bin/bash` in the sleep container – the sleep container does not have the bash shell installed. Instead, execute `sh`:

```
kubectl exec -it $SOURCE_POD -c sleep sh
```