The Containers and Cloud-Native Roadshow Dev Track

guides-m3-labs-infra.apps.cluster-beijing-5bc2.beijing-

Your Workshop Environment

The Workshop Environment You Are Using

Your workshop environment consists of several components which have been pre-installed and are ready to use. Depending on which parts of the workshop you're doing, you will use one or more of:

- Red Hat OpenShift - You'll use one or more *projects* (Kubernetes namespaces) that are your own and are isolated from other workshop students
- Red Hat CodeReady Workspaces - based on **Eclipse Che**, it's a cloud-based, in-browser IDE (similar to IntelliJ IDEA, VSCode, Eclipse IDE). You've been provisioned your own personal workspace for use with this workshop. You'll write, test, and deploy code from here.
- Red Hat Application Migration Toolkit - You'll use this to migrate an existing application
- Red Hat Runtimes - a collection of cloud-native runtimes like Spring Boot, Node.js, and Quarkus
- Red Hat AMQ Streams - streaming data platform based on **Apache Kafka**
- Red Hat SSO - For authentication / authorization - based on **Keycloak**
- Other open source projects like Gogs (Git server that holds application source code), Knative (for serverless apps), Jenkins and Tekton (CI/CD pipelines), Prometheus and Grafana (monitoring apps), and more.

You'll be provided clickable URLs throughout the workshop to access the services that have been installed for you.

How to complete this workshop

Simply follow these instructions end-to-end. **You'll need to do quite a bit of copy/paste for Linux commands and source code modifications**, as well as clicking around on various consoles used in the labs. When you get to the end of each section, you can click the "Next >" button at the bottom to advance to the next topic. You can also use the menu on the left to move around the instructions at will.

The entire workshop is split into one or more *modules* - Look at the top of the screen in the header to see which module you are on. After you complete this module, your instructor may have additional modules to complete.

Good luck, and let's get started!

Getting Started with Service Mesh

Service Mesh and Identity

In this module, you will learn how to prevent cascading failures in a distributed environment, how to detect misbehaving services, and how to avoid having to implement resiliency and monitoring in your business logic. As we transition our applications towards a distributed architecture with microservices deployed across a distributed network, many new challenges await us.

Technologies like containers and container orchestration platforms like OpenShift solve the deployment of our distributed applications quite well, but are still catching up to addressing the service communication necessary to fully take advantage of distributed applications, such as dealing with:

- Unpredictable failure modes
- Verifying end-to-end application correctness
- Unexpected system degradation
- Continuous topology changes
- The use of elastic/ephemeral/transient resources

Today, developers are responsible for taking into account these challenges, and do things like:

- Circuit breaking and Bulkheading (e.g. with Netfix Hystrix)
- Timeouts/retries
- Service discovery (e.g. with Eureka)
- Client-side load balancing (e.g. with Netfix Ribbon)

Another challenge is each runtime and language addresses these with different libraries and frameworks, and in some cases there may be no implementation of a particular library for your chosen language or runtime.

In this scenario we'll explore how to use the OpenShift *Service Mesh* (based on the *Istio* open source project) to solve many of these challenges and result in a much more robust, reliable, and resilient application in the face of the new world of dynamic distributed applications.

What is Istio?

Istio is an open, platform-independent service mesh designed to manage communications between microservices and applications in a transparent way. It provides behavioral insights and operational control over the service mesh as a whole. It provides a number of key capabilities uniformly across a network of services:

- **Traffic Management** - Control the flow of traffic and API calls between services, make calls more reliable, and make the network more robust in the face of adverse conditions.
- **Observability** - Gain understanding of the dependencies between services and the nature and flow of traffic between them, providing the ability to quickly identify issues.
- **Policy Enforcement** - Apply organizational policy to the interaction between services, ensure access policies are enforced and resources are fairly distributed among consumers. Policy changes are made by configuring the mesh, not by changing application code.
- **Service Identity and Security** - Provide services in the mesh with a verifiable identity and provide the ability to protect service traffic as it flows over networks of varying degrees of trustability.

These capabilities greatly decrease the coupling between application code, the underlying platform, and policy. This decreased coupling not only makes services easier to implement, but also makes it simpler for operators to move application deployments between environments or to new policy schemes. Applications become inherently more portable as a result.

Sounds fun, right? Let's get started!

Examine Istio

For this module we've already installed Istio into our OpenShift platform as well as as serveral useful tools to use with it. Istio is installed in the `istio-system` project, and we've also added a few other commonly used tools like Kiali and Jaeger (more on these later)

You can also read a bit more about the Istio architecture below:

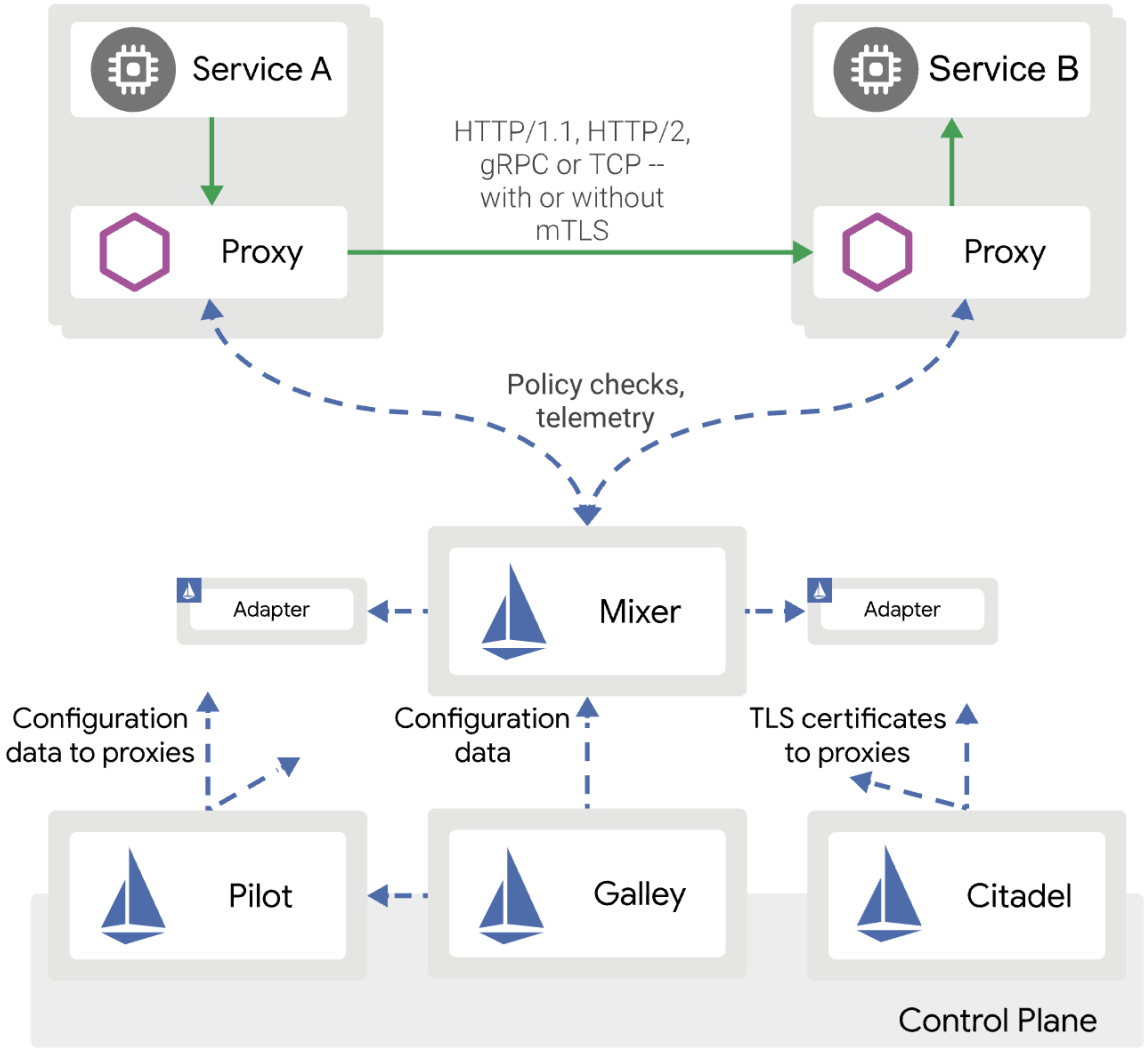
Istio Details

An Istio service mesh is logically split into a *data plane* and a *control plane*.

The **data plane** is composed of a set of intelligent proxies (*Envoy* proxies) deployed as *sidecars* to your application's pods in OpenShift that mediate and control all network communication between microservices.

The **control plane** is responsible for managing and configuring proxies to route traffic, as well as enforcing policies at runtime.

The following diagram shows the different components that make up each plane:



Istio Components

Istio uses an extended version of the Envoy proxy as a *side car* container attached to each service Pod. Envoy is a high-performance proxy developed in C++ to mediate all inbound and outbound traffic for all services in the service mesh. Istio leverages Envoy's many built-in features, for example:

- Dynamic service discovery
- Load balancing
- TLS termination
- HTTP/2 and gRPC proxies
- Circuit breakers
- Health checks
- Staged rollouts with %-based traffic split
- Fault injection
- Rich metrics

Envoy is the *data plane* component that deployed as a *sidecar* to the relevant service in the same Kubernetes pod. This deployment allows Istio to extract a wealth of signals about traffic behavior as attributes. Istio can, in turn, use these attributes in *Mixer* to enforce policy decisions, and send them to monitoring systems to provide information about the behavior of the entire mesh.

Mixer is the *control plane* component responsible for enforcing access control and usage policies across the service mesh, and collects telemetry data from the Envoy proxy and other services. The proxy extracts request level attributes, and sends them to Mixer for evaluation.

Mixer includes a flexible plugin model. This model enables Istio to interface with a variety of host environments and infrastructure backends. Thus, Istio abstracts the Envoy proxy and Istio-managed services from these details.

Pilot is the *control plane* component responsible for configuring the proxies at runtime. Pilot provides service discovery for the Envoy sidecars, traffic management capabilities for intelligent routing (for example, A/B tests or canary deployments), and resiliency (timeouts, retries, and circuit breakers).

Pilot converts high level routing rules that control traffic behavior into Envoy-specific configurations, and propagates them to the sidecars at runtime. Pilot abstracts platform-specific service discovery mechanisms and synthesizes them into a standard format that any sidecar conforming with the Envoy data plane APIs can consume. This loose coupling allows Istio to run on multiple environments such as Kubernetes, Consul, or Nomad, while maintaining the same operator interface for traffic management.

Citadel is the *control plane* component responsible for certificate issuance and rotation. Citadel provides strong service-to-service and end-user authentication with built-in identity and credential management. You can use Citadel to upgrade unencrypted traffic

in the service mesh. Using Citadel, operators can enforce policies based on service identity rather than on network controls.

Galley is Istio's configuration validation, ingestion, processing and distribution component. It is responsible for insulating the rest of the Istio components from the details of obtaining user configuration from the underlying platform (e.g. Kubernetes).

Several **Add-ons** components are used to provide additional visualizations, metrics, and tracing functions:

We will use these in future steps in this scenario!

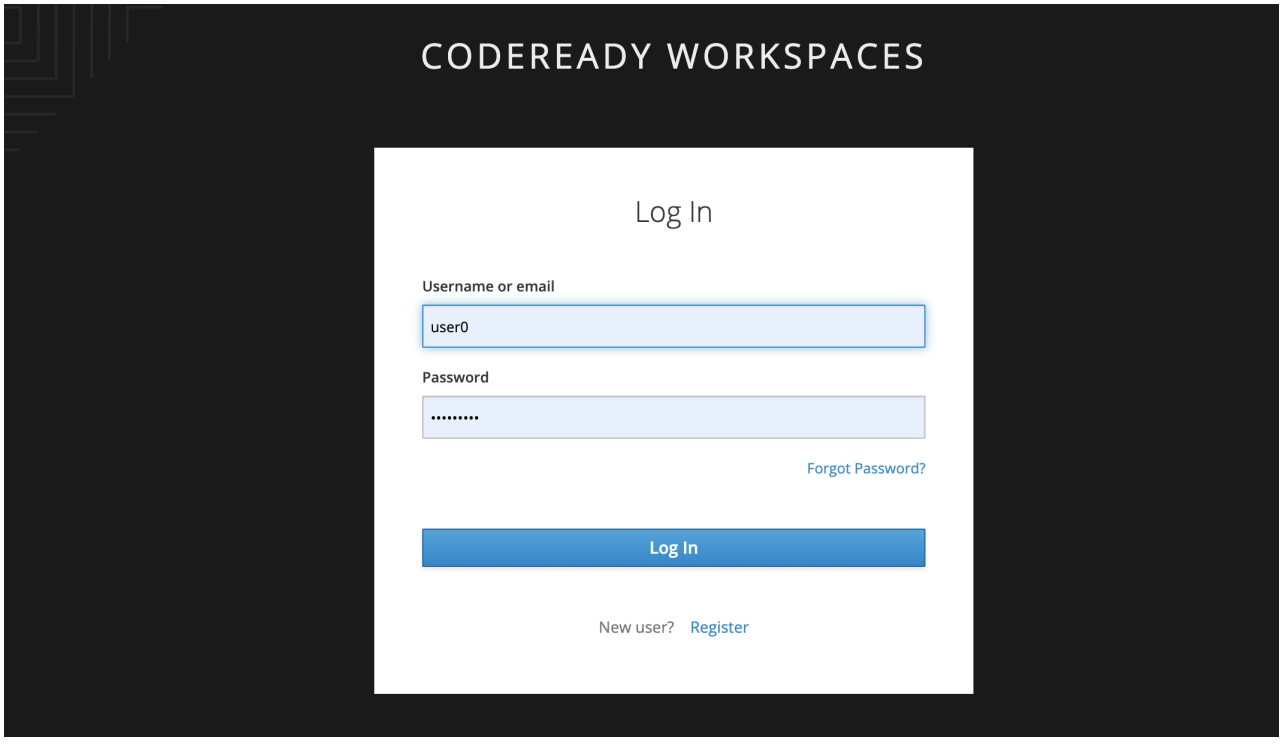
Getting Ready for the labs

> **NOTE**
>
> If you've already completed the **Optimizing Existing Applications** module then you will simply need to import the code for this module. Skip down to the **Import Projects** section.

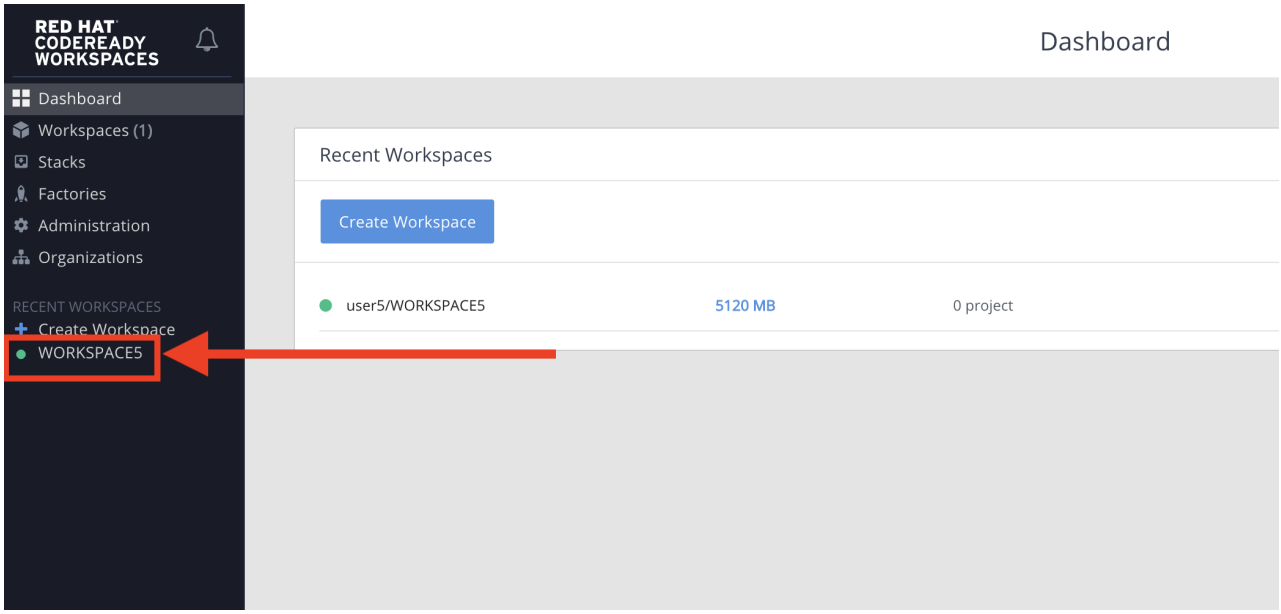
If this is the first module you are doing today

You will be using Red Hat CodeReady Workspaces, an online IDE based on Eclipe Che. **Changes to files are auto-saved every few seconds**, so you don't need to explicitly save changes.

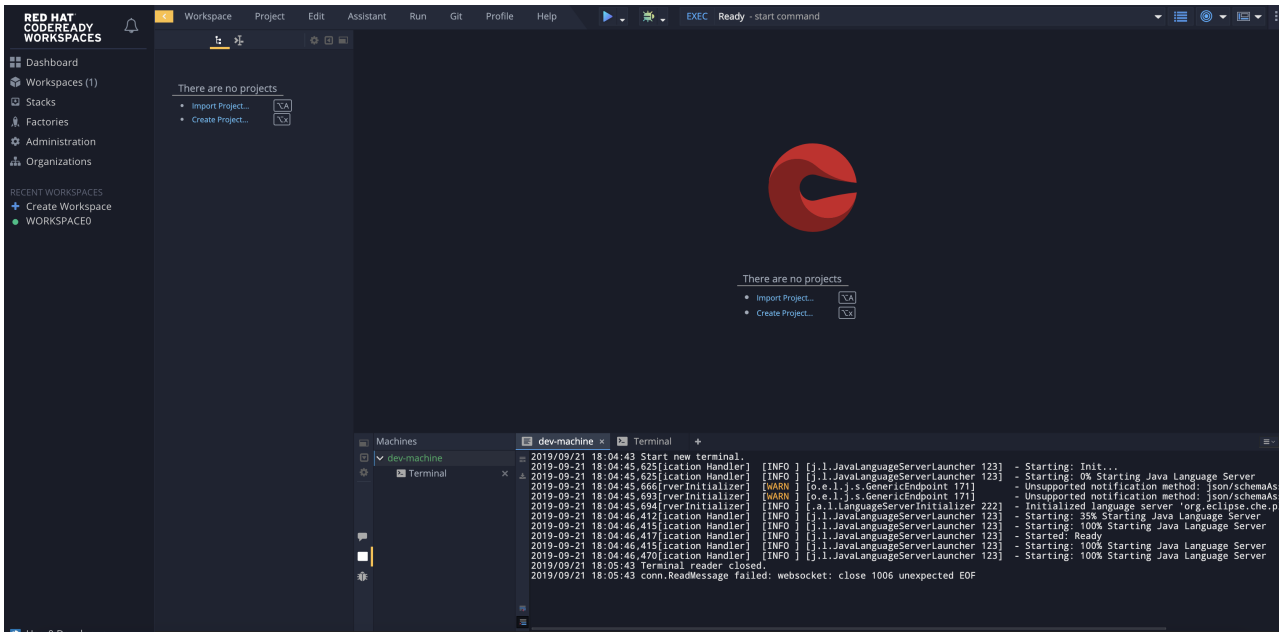
To get started, access the Che instance and log in using the username and password you've been assigned (e.g. `userXX/r3dh4t1!`):



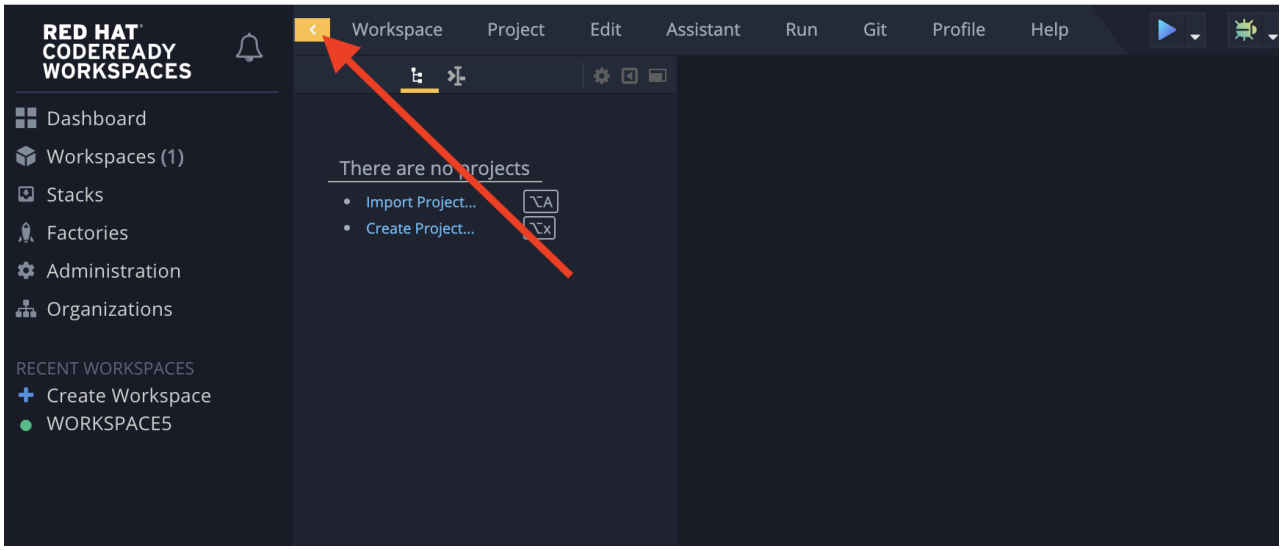
Once you log in, you'll be placed on your personal dashboard. We've pre-created workspaces for you to use. Click on the name of the pre-created workspace on the left, as shown below (the name will be different depending on your assigned number). You can also click on the name of the workspace in the center, and then click on the green button that says "OPEN" on the top right hand side of the screen:



After a minute or two, you'll be placed in the workspace:



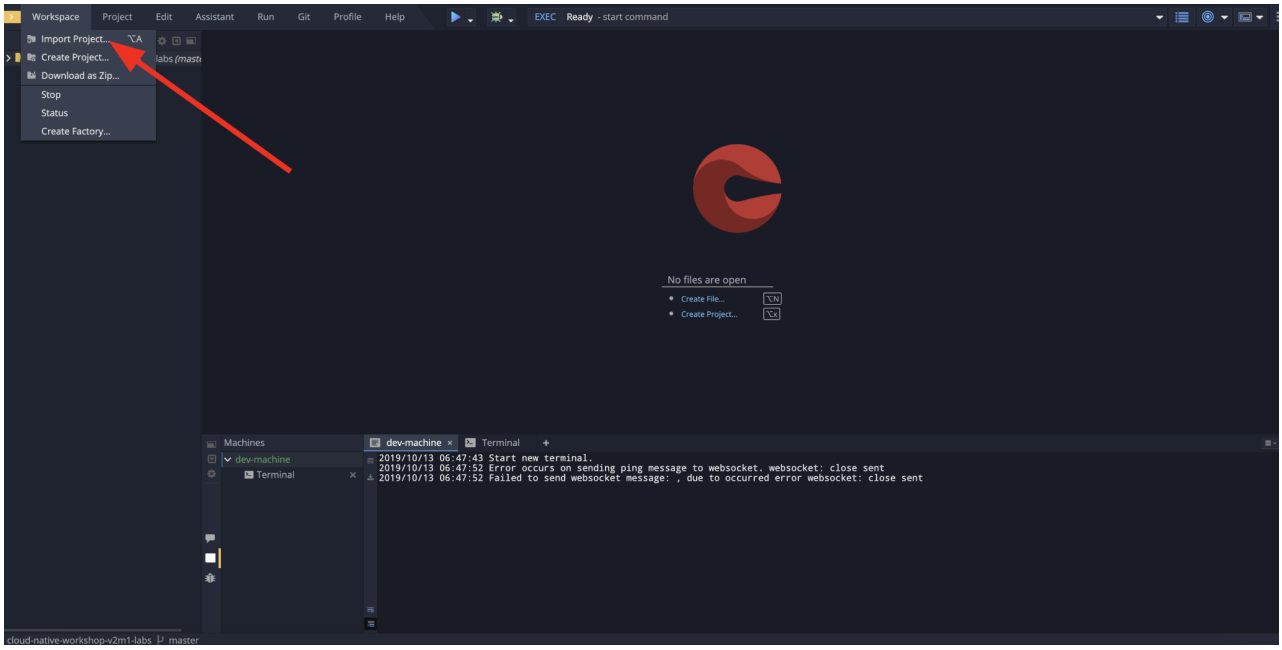
To gain extra screen space, click on the yellow arrow to hide the left menu (you won't need it):



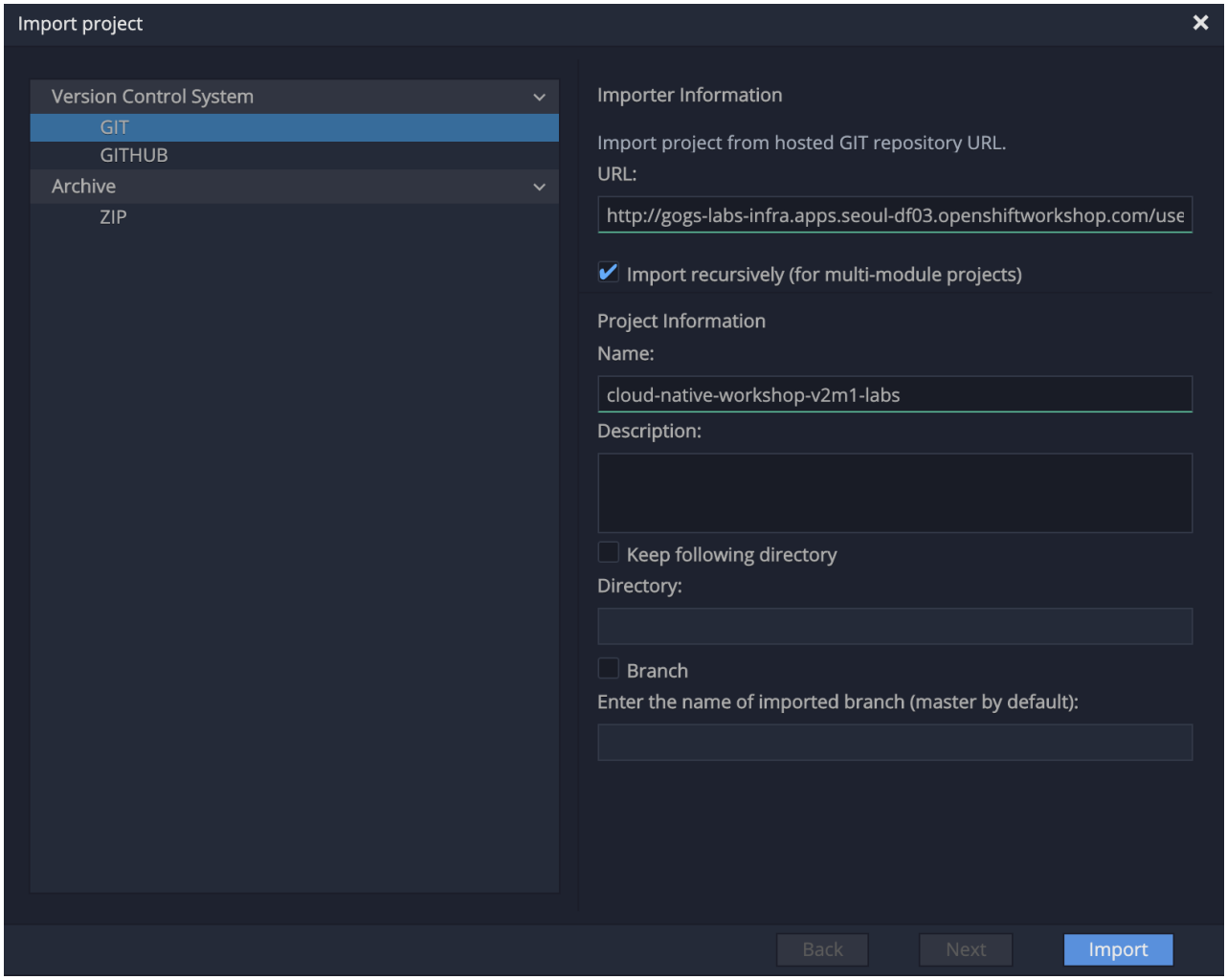
Users of Eclipse, IntelliJ IDEA or Visual Studio Code will see a familiar layout: a project/file browser on the left, a code editor on the right, and a terminal at the bottom. You'll use all of these during the course of this workshop, so keep this browser tab open throughout. **If things get weird, you can simply reload the browser tab to refresh the view.**

Import Projects

Click on the **Import Projects...** in **Workspace** menu and enter the following:

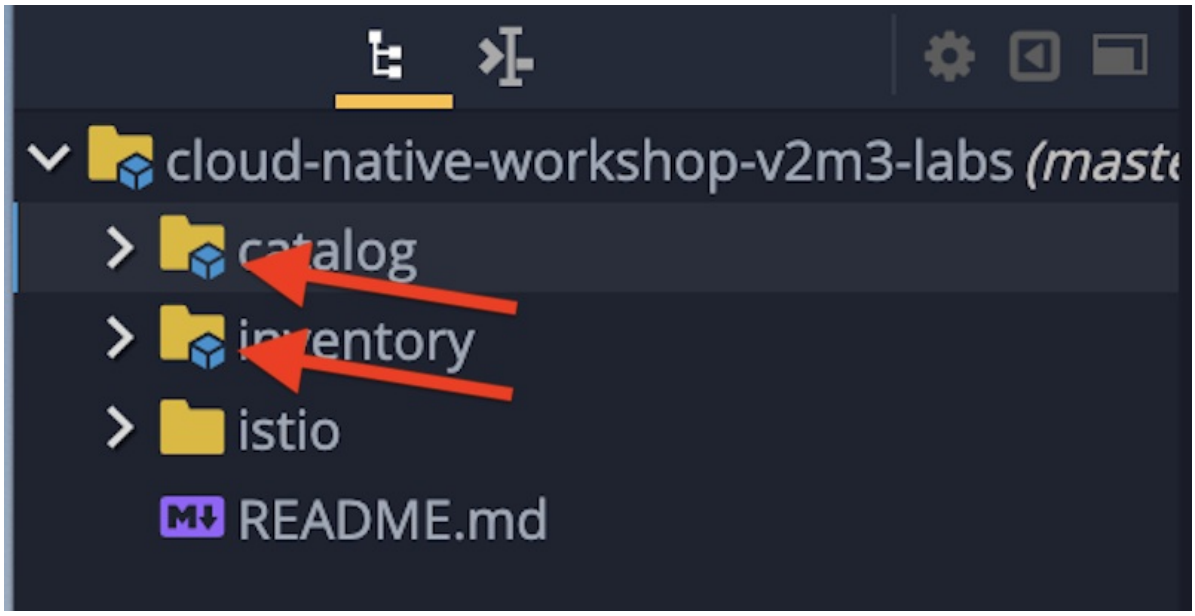


- Version Control System: `GIT`
- URL: `http://gogs-labs-infra.apps.cluster-beijing-5bc2.beijing-5bc2.open.redhat.com/userXX/cloud-native-workshop-v2m3-labs.git` (**IMPORTANT**: replace userXX with your lab user)
- Check `Import recursively (for multi-module projects)`
- Name: `cloud-native-workshop-v2m3-labs`

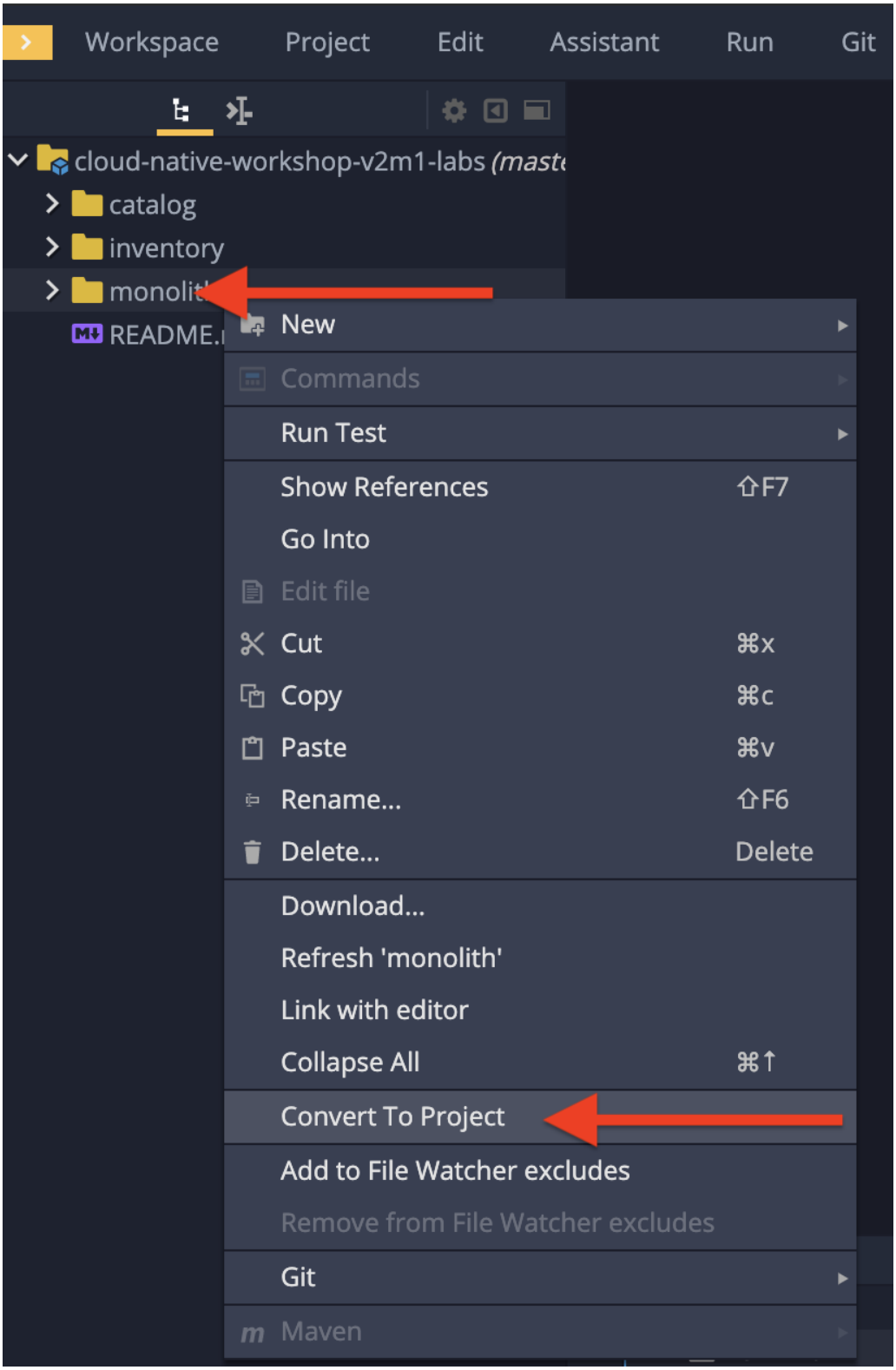


The project will be imported into your workspace and visible in the project explorer.

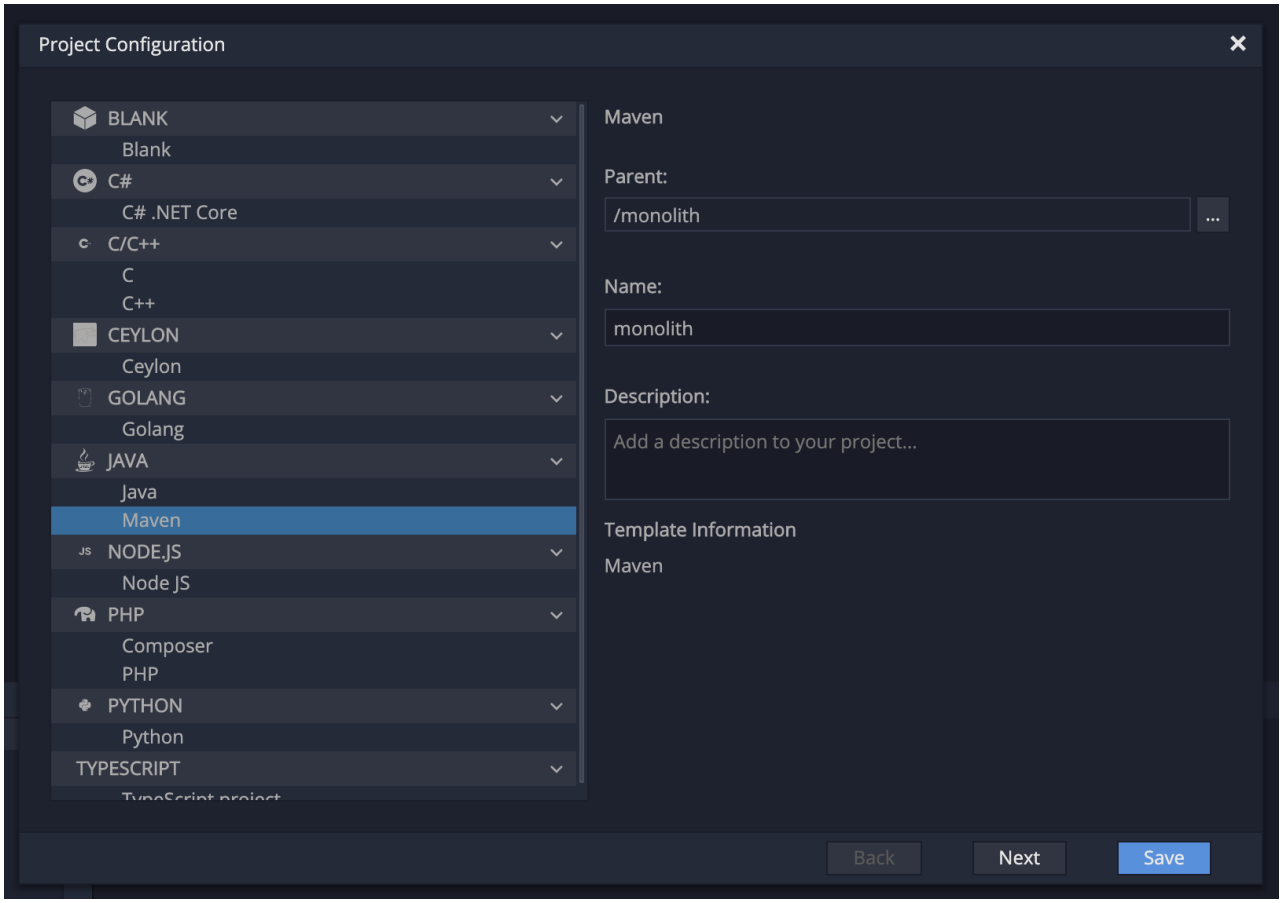
CodeReady Workspaces is a full featured IDE and provides language specific capabilities for various project types. In order to enable these capabilities for this Java-based Maven app, let's convert the imported project to a Maven project. In the project explorer, right-click on each project and then click on **Convert to Project** continuously.

> **NOTE**
>
> If you do not see the `Convert to Project` then your projects are already converted, and you should see a small icon next to each project:
>
> 
>

If not, then convert them:

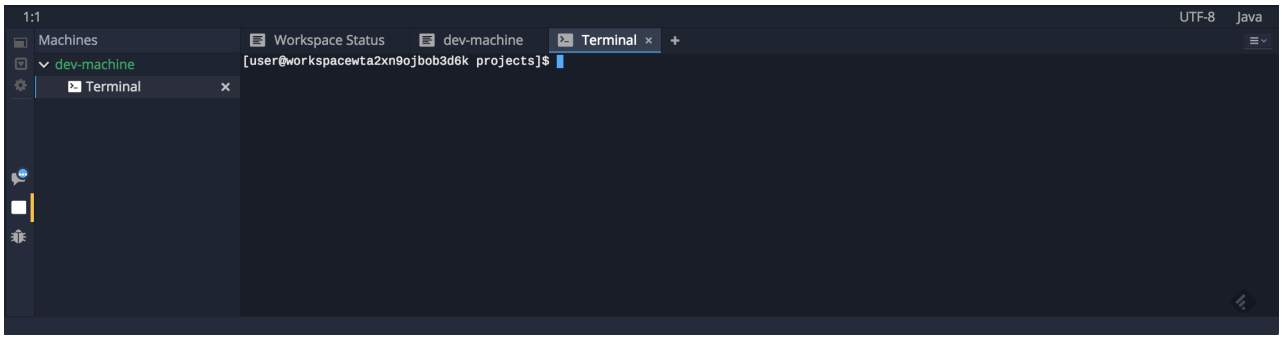


Choose **Maven** from the project configurations and then click on **Save**.



Repeat the above for `inventory` and `catalog` projects.

> NOTE: For the rest of these labs, anytime you need to run a command in a terminal, you can use the CodeReady Workspaces Terminal window. Be sure you're in the correct directory for the command(s) you wish to run!



Creating Distributed Services

Lab1 - Creating Distributed Services

In this step, we'll install a sample application into the system. This application is included in Istio itself for demonstrating various aspects of it, but the application isn't tied exclusively to Istio - it's an ordinary microservice application that could be installed to any OpenShift instance with or without Istio.

The sample application is called *Bookinfo*, a simple application that displays information about a book, similar to a single catalog entry of an online book store. Displayed on the page is a description of the book, book details (ISBN, number of pages, and so on), and a few book reviews.

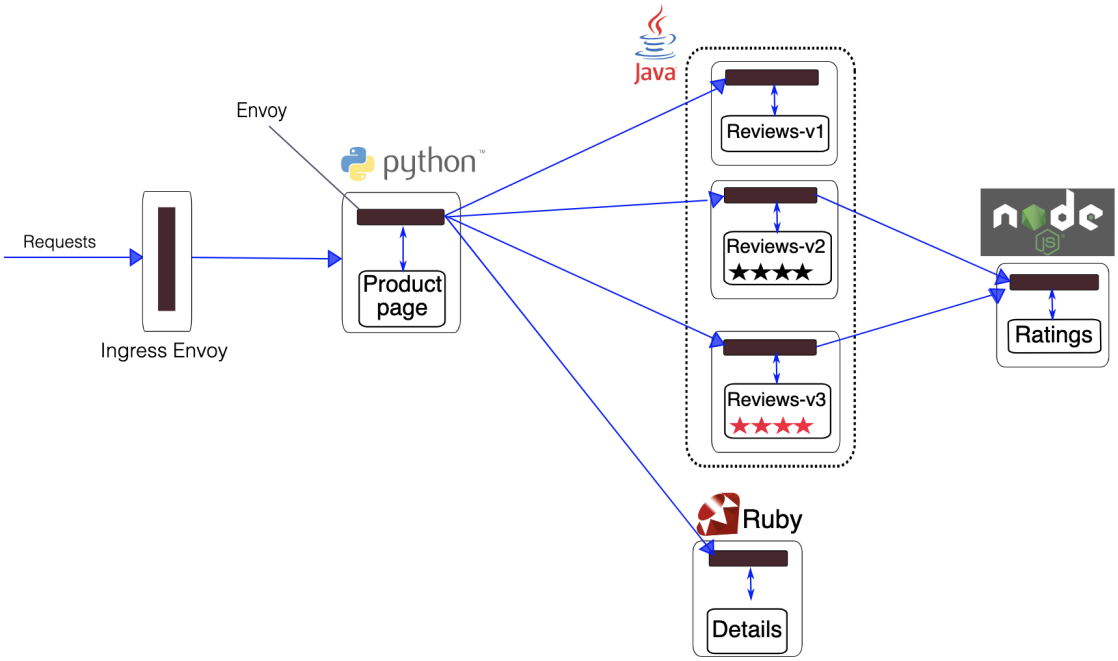
The BookInfo application is broken into four separate microservices:

- **productpage** - The `productpage` microservice calls the `details` and `reviews` microservices to populate the page.
- **details** - The `details` microservice contains book information.
- **reviews** - The `reviews` microservice contains book reviews. It also calls the `ratings` microservice to show a “star” rating for each book.
- **ratings** - The `ratings` microservice contains book rating information that accompanies a book review.

There are 3 versions of the reviews microservice:

- Version `v1` does not call the ratings service.
- Version `v2` calls the ratings service, and displays each rating as 1 to 5 **black** stars.
- Version `v3` calls the ratings service, and displays each rating as 1 to 5 **red** stars.

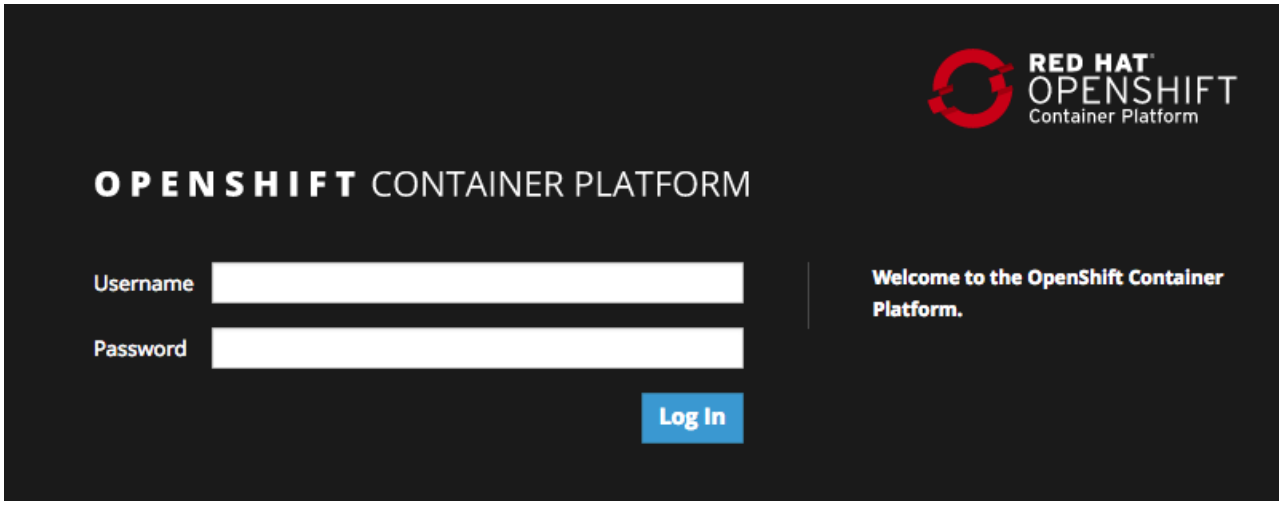
The end-to-end architecture of the application is shown below.



Bookinfo Application

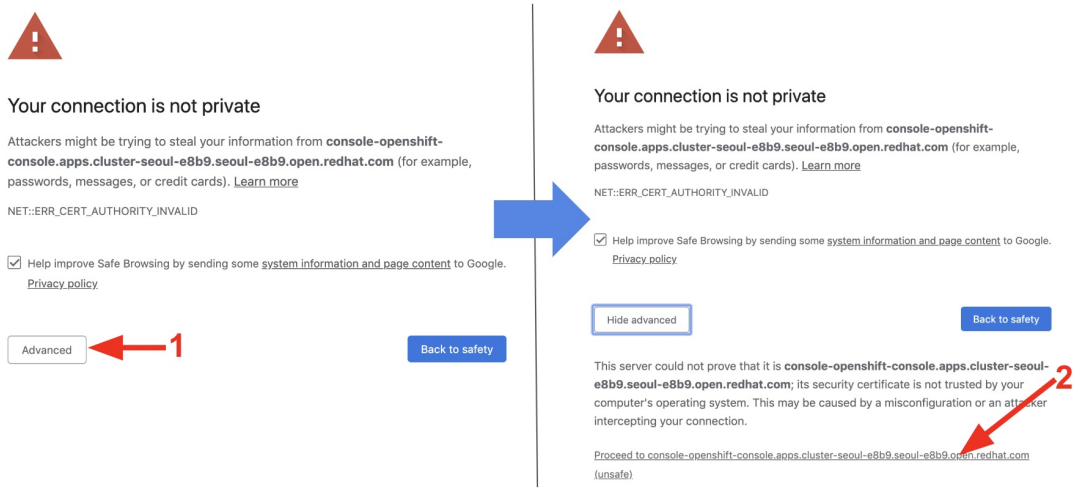
1. Deploy Bookinfo Application

First, open a new browser with the OpenShift web console

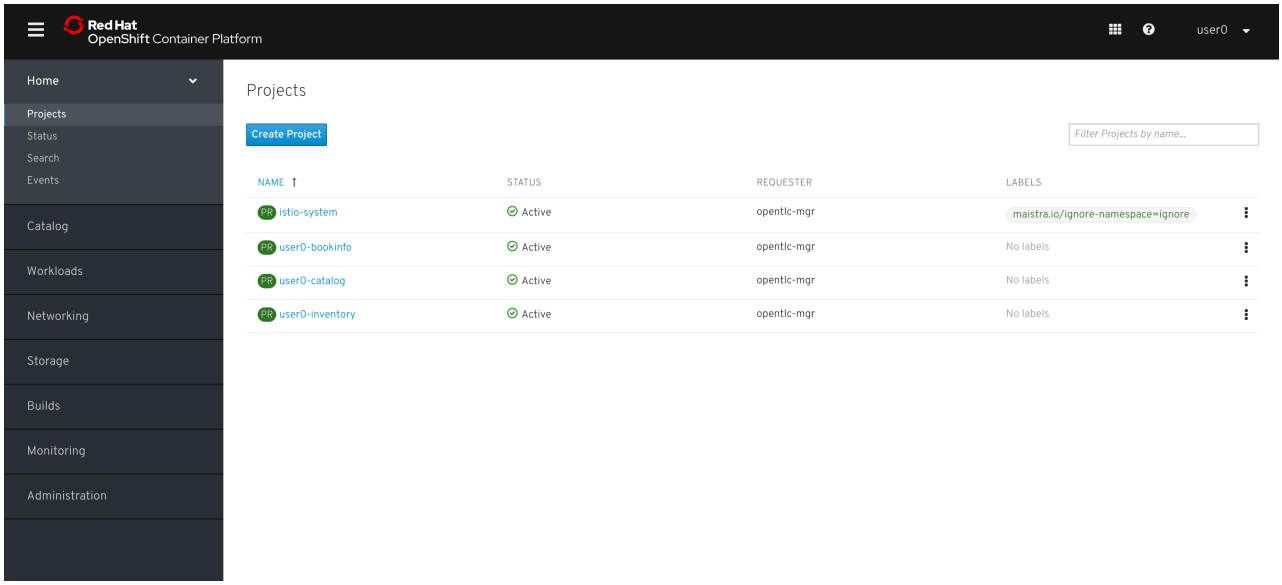


Login using:

- Username: `userXX`
- Password: `r3dh4t1!`

> **NOTE**: Use of self-signed certificates
>
> When you access the OpenShift web console or other URLs via *HTTPS* protocol, you will see browser warnings like `Your Connection is not secure` since this workshop uses self-signed certificates (which you should not do in production!). For example, if you're using **Chrome**, to accept the warning, Click on `Advanced` then `Proceed to...` to access the page.
>
> 
>
>
> Other browsers have similar procedures to accept the security exception.

Once logged in, uou will see the OpenShift landing page:



> The project displayed in the landing page depends on which labs you will run today. If you will develop `Service Mesh and Identity` then you will see pre-created projects as the above screeenshot.

Although your Eclipse Che workspace is running on the Kubernetes cluster, it's running with a default restricted *Service Account* that prevents you from creating most resource types. If you've completed other modules, you're probably already logged in, but let's login again: open a Terminal and issue the following command:

```
oc login https://$KUBERNETES_SERVICE_HOST:$KUBERNETES_SERVICE_PORT --insecure-skip-tls-verify=true
```

Enter your username and password assigned to you:

- Username: `userXX`
- Password: `r3dh4t1!`

You should see like:

```
Login successful.

You have access to the following projects and can switch between them with 'oc project <projectname>':

  * default
    istio-system
    userXX-bookinfo
    userXX-catalog
    userXX-cloudnative-pipeline
    userXX-cloudnativeapps
    userXX-inventory

Using project "default".
Welcome! See 'oc help' to get started.
```

Change to the empty **userXX-bookinfo** project via CodeReady Workspaces Terminal and this command (you should replace **userXX** with your username):

```
oc project userXX-bookinfo
```

Deploy the **Bookinfo application** in the bookinfo project:

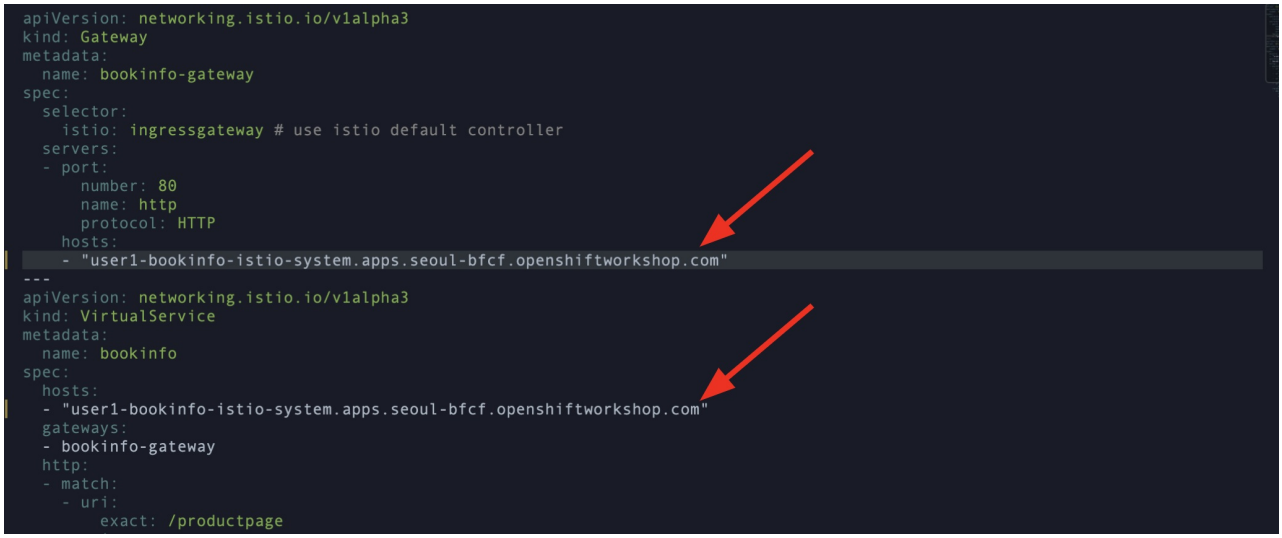
```
oc apply -f /projects/cloud-native-workshop-v2m3-labs/istio/bookinfo.yaml
```

Next, open the `istio/bookinfo-gateway.yaml` file in CodeReady.

Look for the *REPLACE WITH YOUR BOOKINFO APP URL* (there are 2 of them) and replace them with your custom url:

```
userXX-bookinfo-istio-system.apps.cluster-beijing-5bc2.beijing-5bc2.open.redhat.com
```

> Be sure to substitute your username for `userXX` !



And then create the *ingress gateway* for Bookinfo:

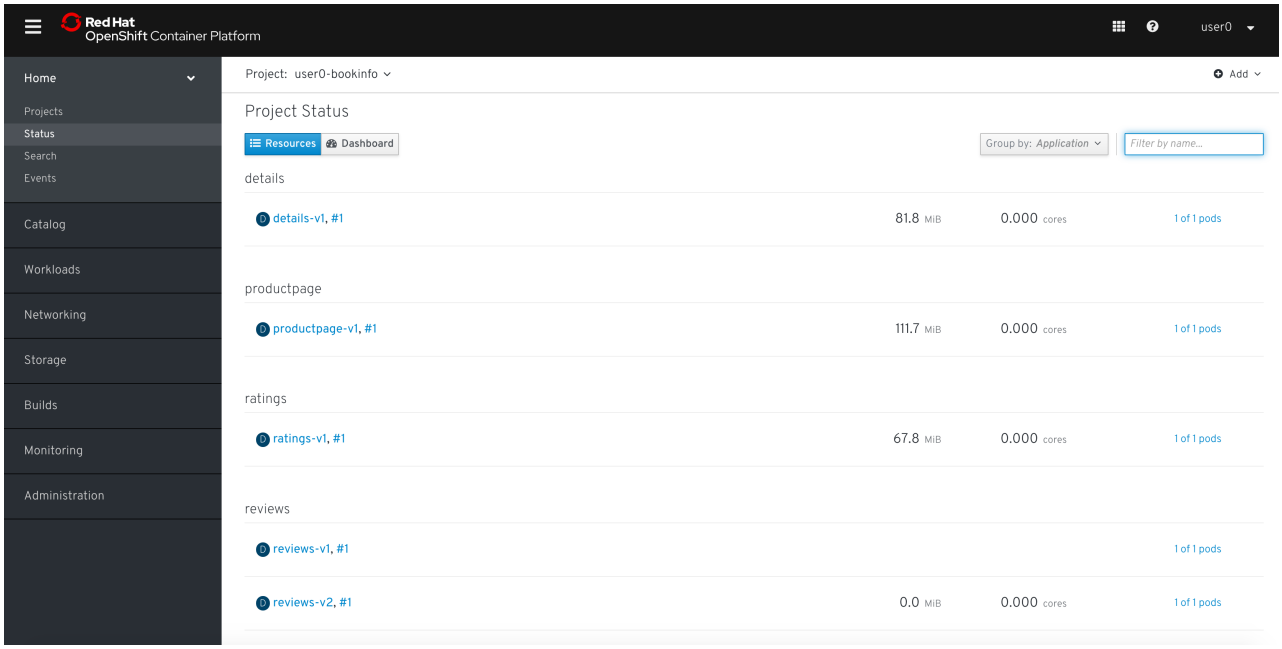
```
oc apply -f /projects/cloud-native-workshop-v2m3-labs/istio/bookinfo-gateway.yaml
```

For your conveience, set an environment variable in the CodeReady Workspaces Terminal:

`echo "export BOOK_URL=REPLACE WITH YOUR BOOKINFO APP URL" >> ~/.bashrc && source ~/.bashrc` (again, replace the same value as above).

When the app is installed, each Pod will get an additional *sidecar* container as described earlier.

Let's wait for our application to finish deploying. Go to the overview page in *userxx-bookinfo* project:



Or you can execute the following commands to wait for the deployment to complete and result `successfully rolled out` :

```
oc rollout status -w deployment/productpage-v1 && \
 oc rollout status -w deployment/reviews-v1 && \
 oc rollout status -w deployment/reviews-v2 && \
 oc rollout status -w deployment/reviews-v3 && \
 oc rollout status -w deployment/details-v1 && \
 oc rollout status -w deployment/ratings-v1
```

Confirm that Bookinfo has been **successfully** deployed via your own *Gateway URL*:

```
curl -o /dev/null -s -w "%{http_code}\n" http://$BOOK_URL/productpage
```

You should get **200** as a response.

Add default destination rules (we'll alter this later to affect routing of requests):

```
oc apply -f /projects/cloud-native-workshop-v2m3-labs/istio/destination-rule-all.yaml
```

List all available destination rules:

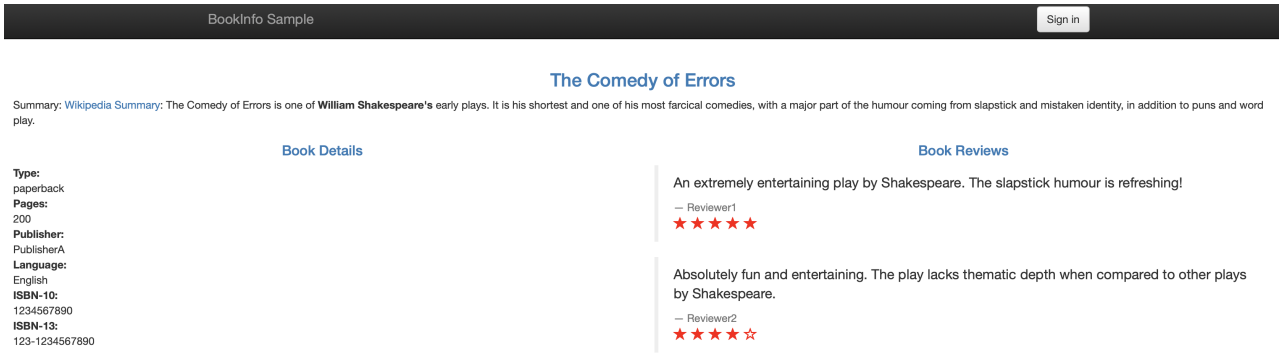
```
oc get destinationrules -o yaml
```

2. Access Bookinfo

Open the application in your web browser to make sure if it's working. You will find the URL via running the following command in CodeReady Workspaces Terminal:

```
echo http://$BOOK_URL/productpage
```

It should look something like:



Reload the page multiple times. The three different versions of the Reviews service show the star ratings differently - *v1* shows no stars at all, *v2* shows black stars, and *v3* shows red stars:

- **v1**:

> Absolutely fun and entertaining. The play lacks thematic depth when compared to other plays by Shakespeare.
>
> — Reviewer2

- **v2**:

> Absolutely fun and entertaining. The play lacks thematic depth when compared to other plays by Shakespeare.
>
> — Reviewer2
>
> ★★★★☆

- **v3**:

> Absolutely fun and entertaining. The play lacks thematic depth when compared to other plays by Shakespeare.
>
> — Reviewer2
>
> ★★★★☆

That's because there are 3 versions of reviews deployment for our reviews service. Istio's load-balancer is using a *round-robin* algorithm to iterate through the 3 instances of this service.

You should now have your OpenShift Pods running and have an Envoy sidecar in each of them alongside the microservice. The microservices are productpage, details, ratings, and reviews. Note that you'll have three versions of the reviews microservice:

```
oc get pods --selector app=reviews
```

```
NAME                       READY   STATUS    RESTARTS   AGE
reviews-v1-7754bbd88-dm4s5    2/2     Running   0          12m
reviews-v2-69fd995884-qpddl   2/2     Running   0          12m
reviews-v3-5f9d5bbd8-sz29k    2/2     Running   0          12m
```

Notice that each of the microservices shows **2/2** containers ready for each service (one for the service and one for its sidecar).

Now that we have our application deployed and linked into the Istio service mesh, let's take a look at the immediate value we can get out of it without touching the application code itself!

Congratulations!

Service Visualization and Monitoring

Lab2 - Service Visualization and Monitoring

In this lab you will visualize your service mesh using **Kiali**, **Prometheus**, **Grafana** and you will lean how to configure basic **Istio funtionalities** such as **VirtualService** and **A/B Testing**.

1. Generating application load

To get a better idea of the power of metrics, let's setup an endless loop that will continually access the application and generate load. We'll open up a separate terminal just for this purpose.

Open a new *Terminal* and execute this command with your own *Bookinfo App URL*:

```
BOOK_URL=REPLACE WITH YOUR BOOKINFO APP URL
```

```
for i in {1..1000} ; do curl -o /dev/null -s -w "%{http_code}\n" http://$BOOK_URL/productpage ; sleep 2 ; done
```

This command will endlessly access the application and report the HTTP status result in a separate terminal window.

With this application load running, metrics will become much more interesting in the next few steps.

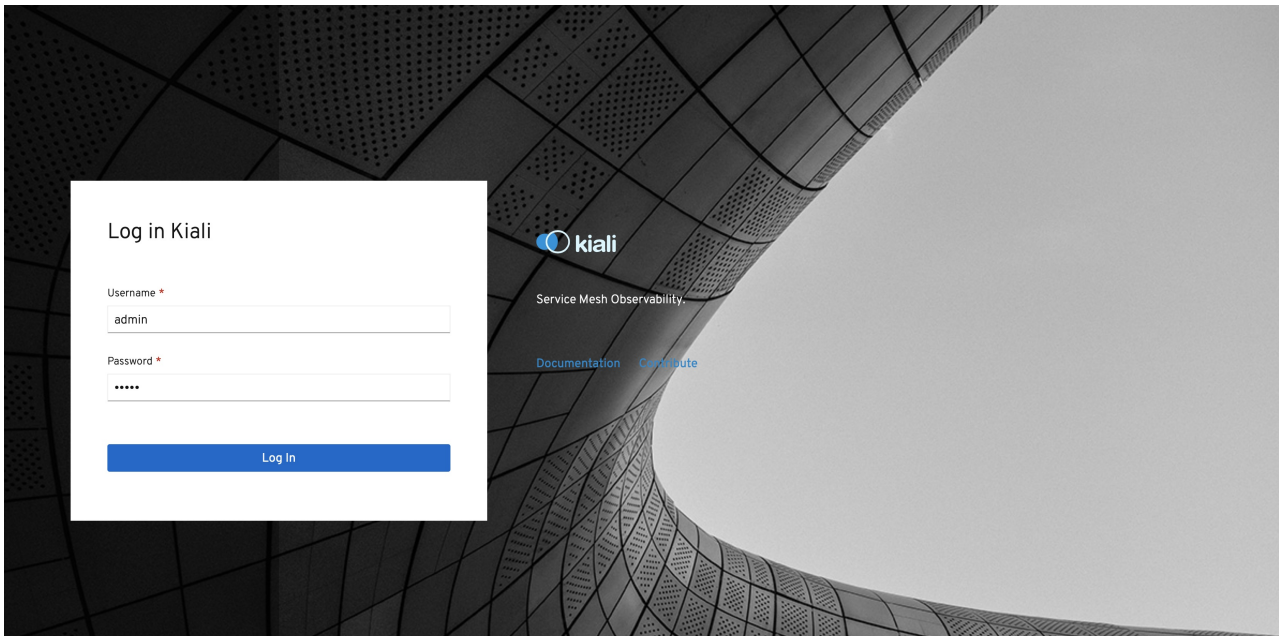
2. Examine Kiali

Kiali allows you to manage and monitor your mesh from a single UI. This UI will allow you to view configurations, monitor traffic flow and health, and analyze traces.

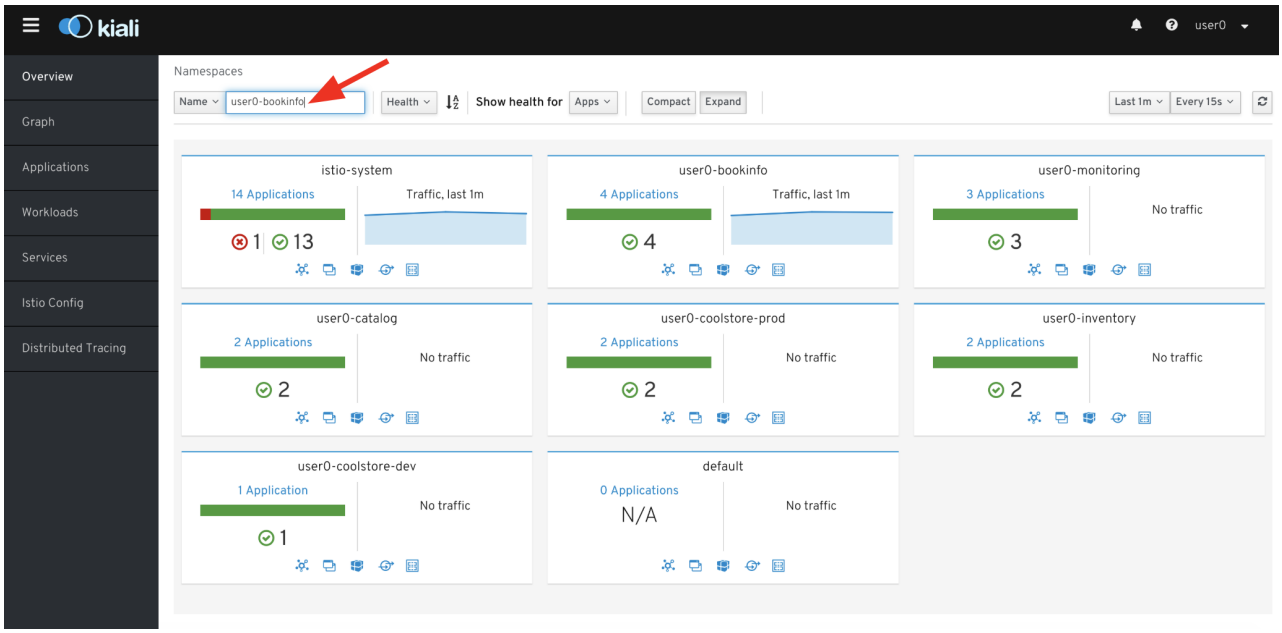
Open the Kiali console.

You should see *Kiali Login* screen. Enter the username and password as below and click *Log In*.

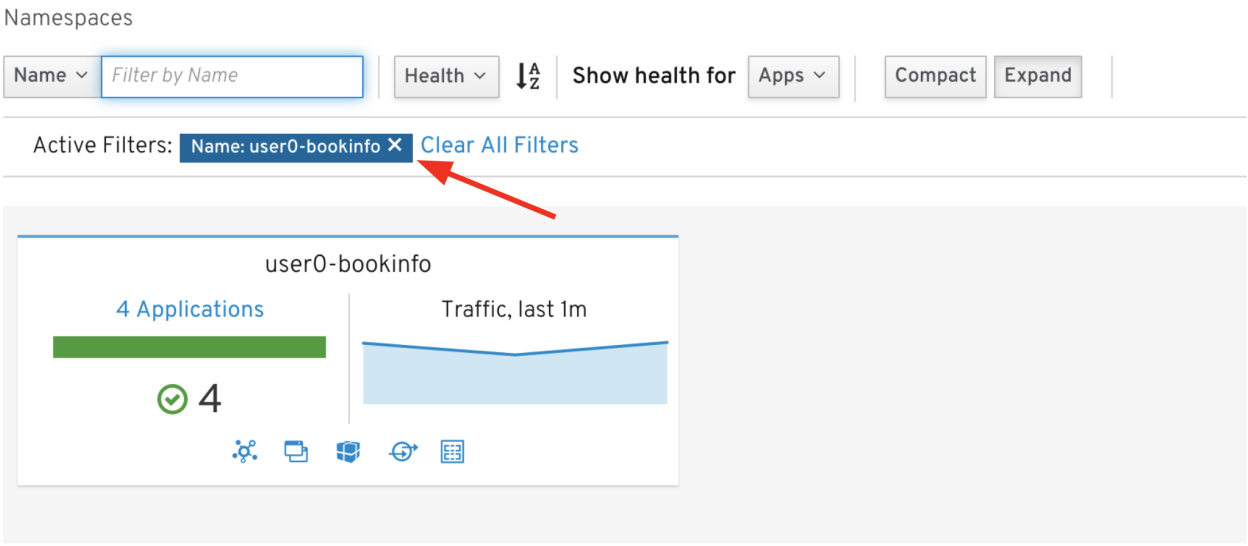
- Username: `admin`
- Password: `admin`



In the namespace selector at the top, enter your **userXX-bookinfo** application (e.g. user1-bookinfo) and press enter to filter to only show your bookinfo project.



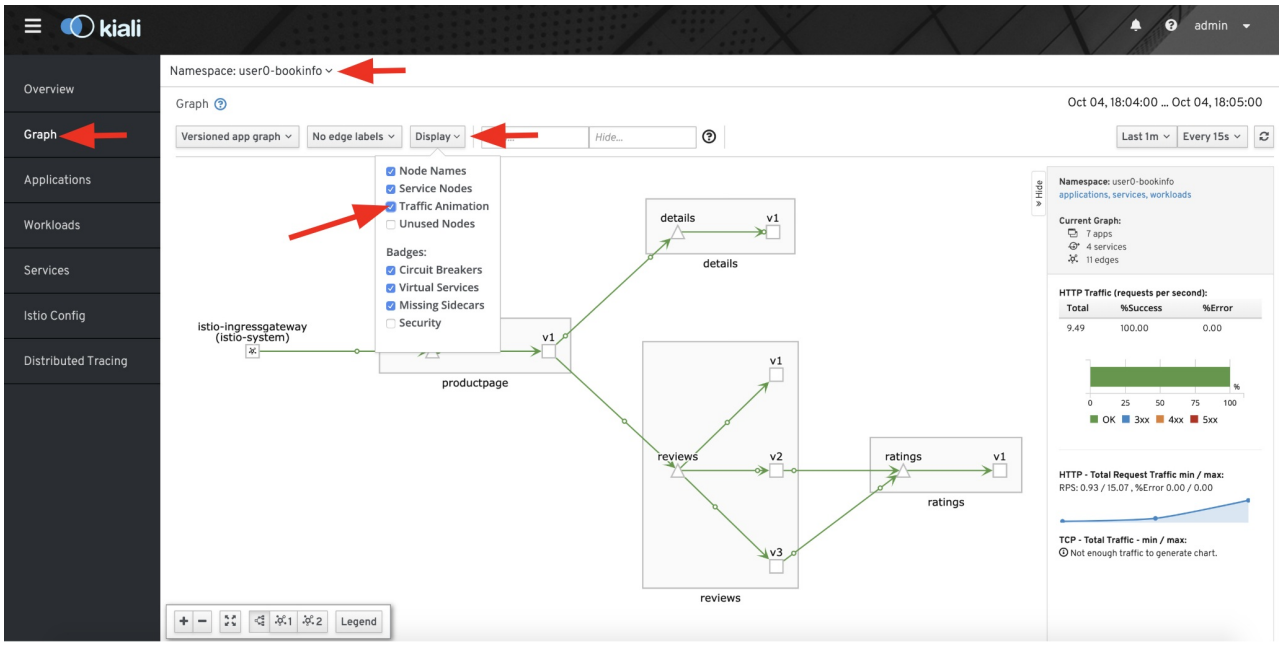
This way you will only see your personal working namespace as below:



Click on the “4 Applications” link.

Service Graph

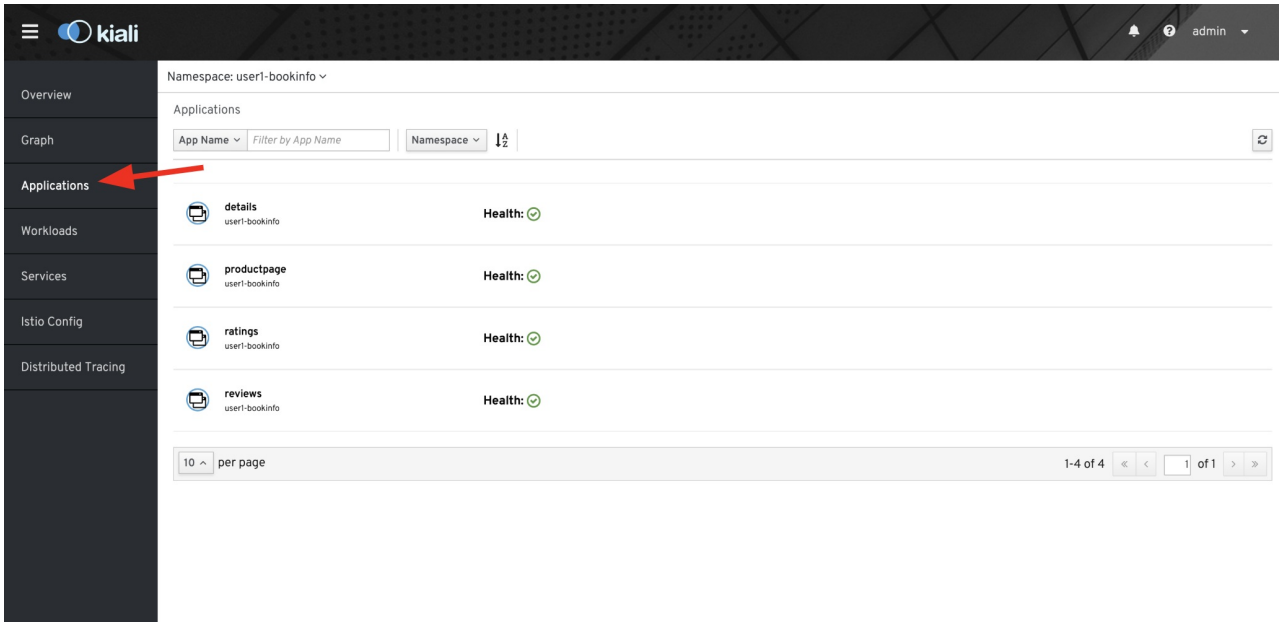
Click on the *Graph* page on the left and check **Traffic Animation** in **Display**:



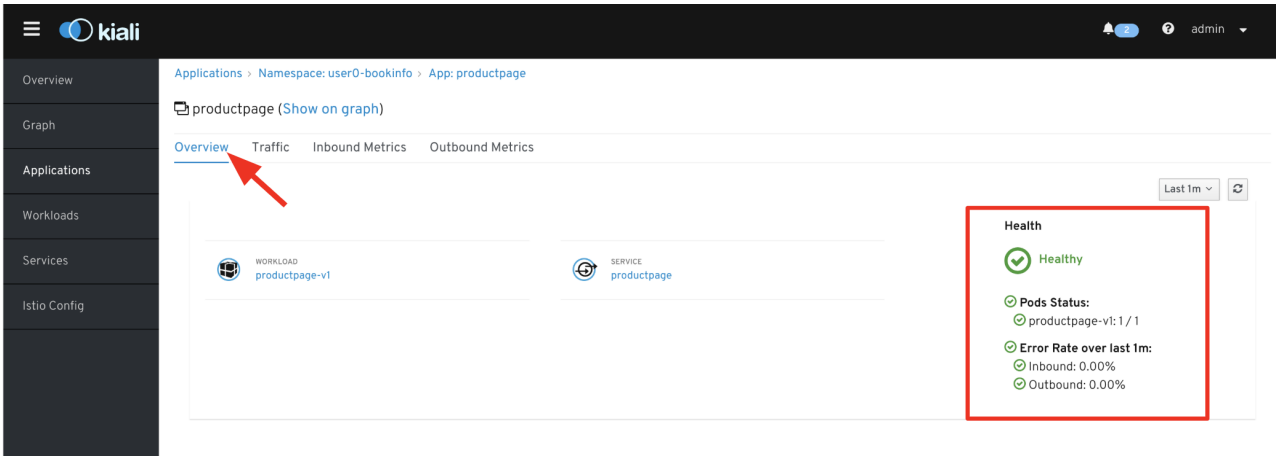
It shows a graph with all the microservices, connected by the requests going through then. On this page, you can see how services interact with each other. Observe that traffic from `productpage` to `reviews` is equally hitting all three versions of the `reviews` service, and that `v2` and `v3` are in turn hitting the `ratings` service (while `v1` does not, so therefore you get no “stars” when you get load-balanced to `v1`).

Applications

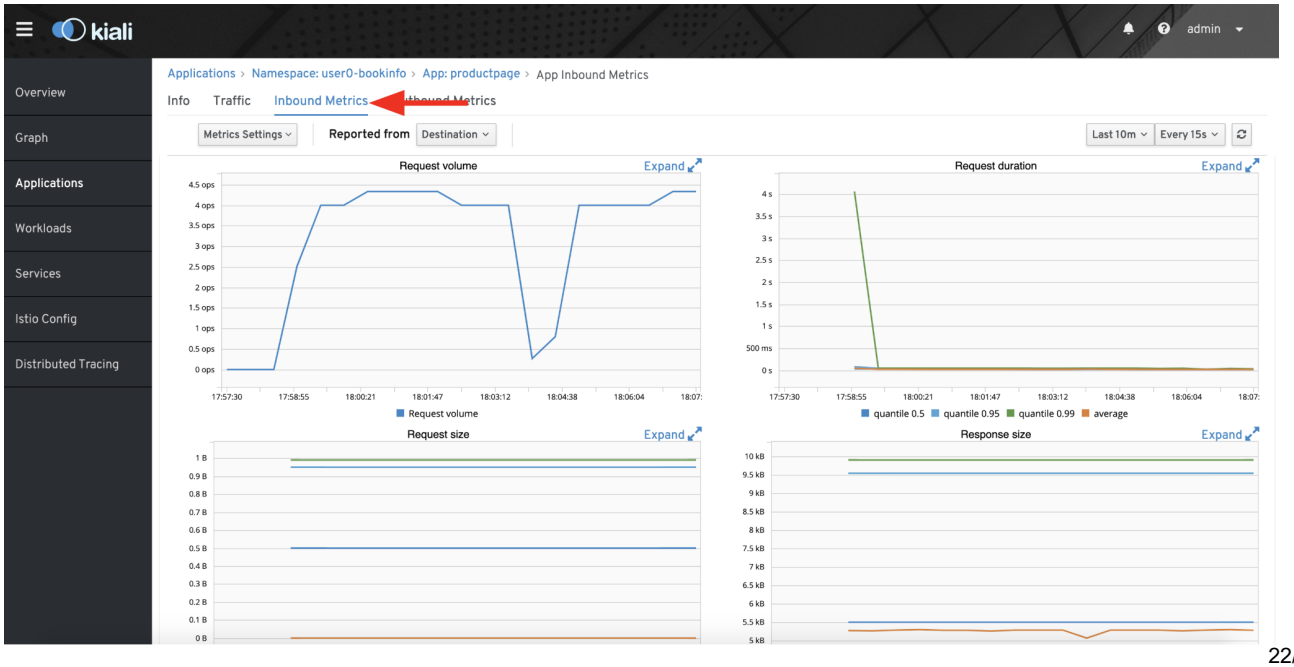
Click on **Applications** menu in the left navigation. On this page you can view a listing of all the services that are running in the cluster, and additional information about them, such as health status.



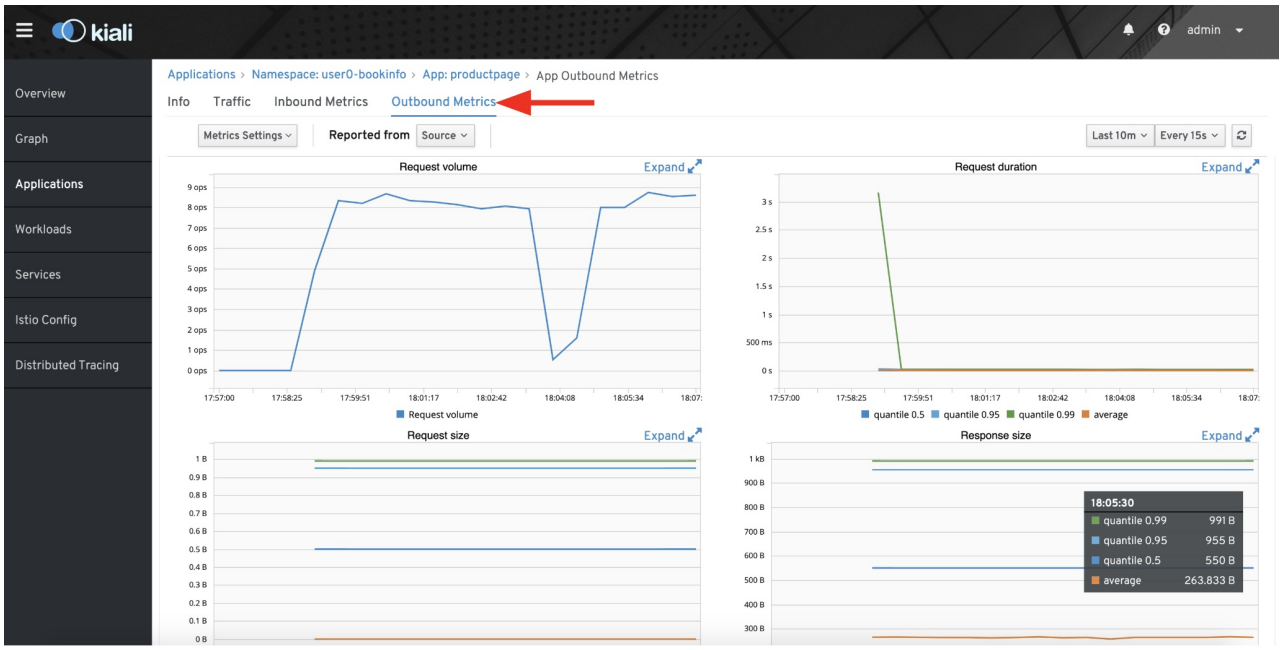
Click on the **productpage** application to see its details. You can also see the health of a service on the **Health** section when it's online and responding to requests without errors:



By clicking on **Inbound Metrics**, you can see the metrics for an application, like this:

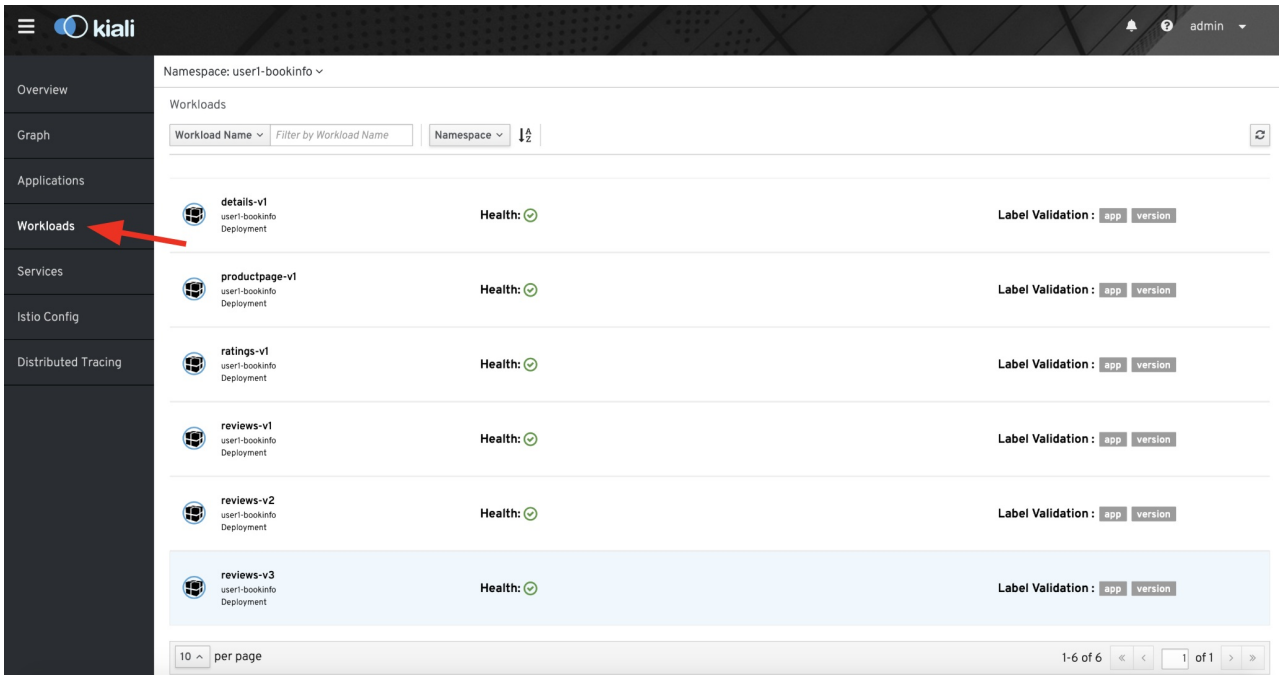


By clicking on **Outbound Metrics**, you can see the metrics for an application, like this:

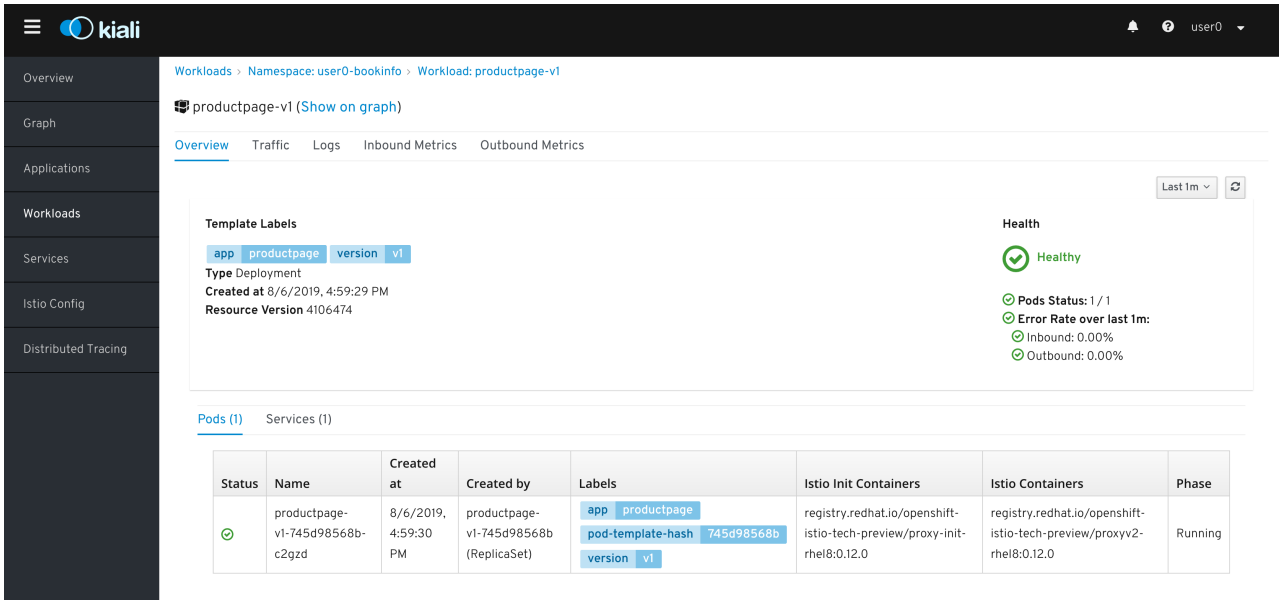


Workloads

Click on the **Workloads** menu in the left navigation. On this page you can view a listing of all the workloads that are present in your application.



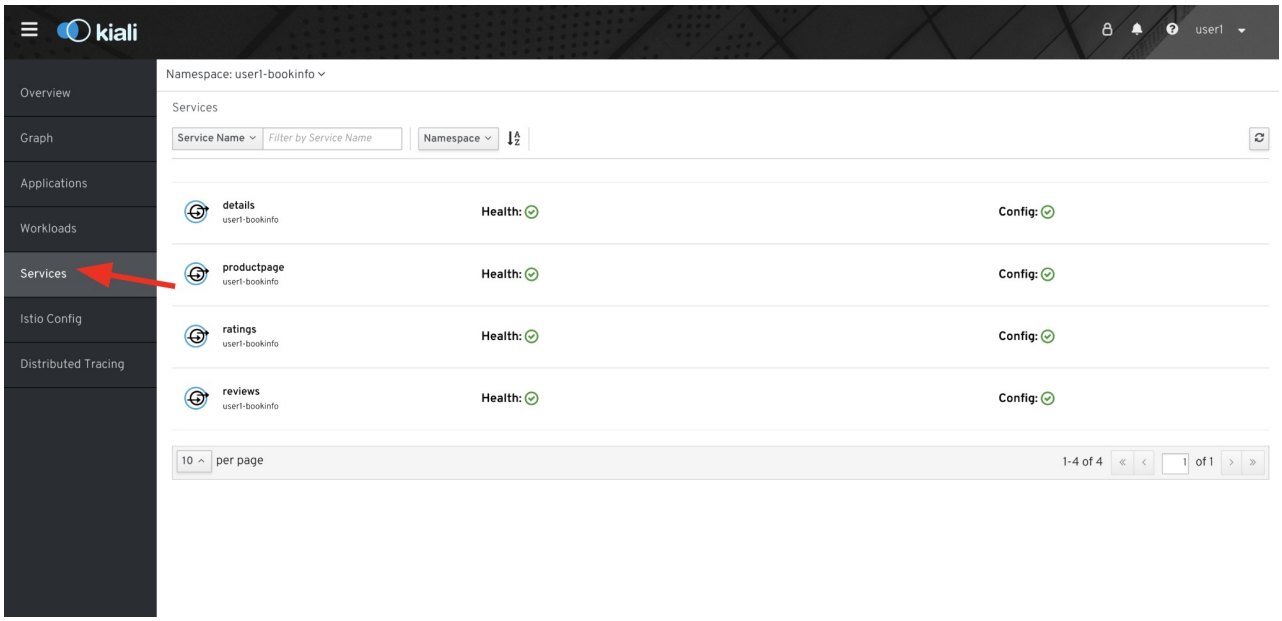
Click on the **productpage-v1** workload. Here you can see details for the workload, such as the pods and services that are included in it:



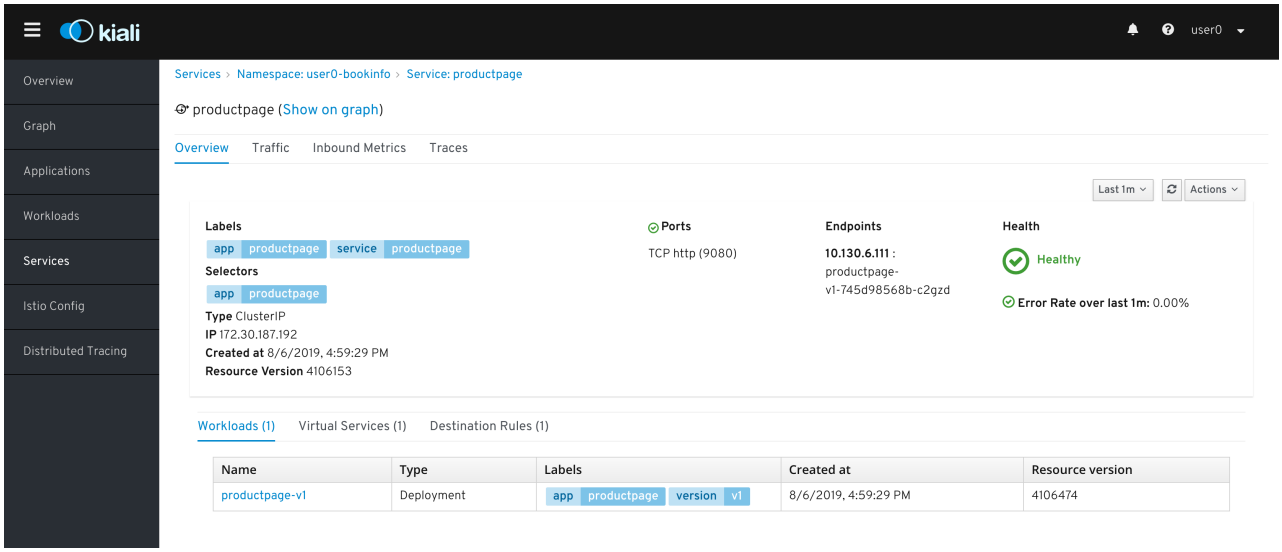
By clicking *Inbound Metrics*, you can check the metrics for the workload. The metrics are the same as the *Application* ones.

Services

Click on **Services** menu in the left navigation. Here, you can see the listing of all services.



Click on **productpage** service which will show you the details of the service, such as metrics, traces, workloads, virtual services, destination rules and more:

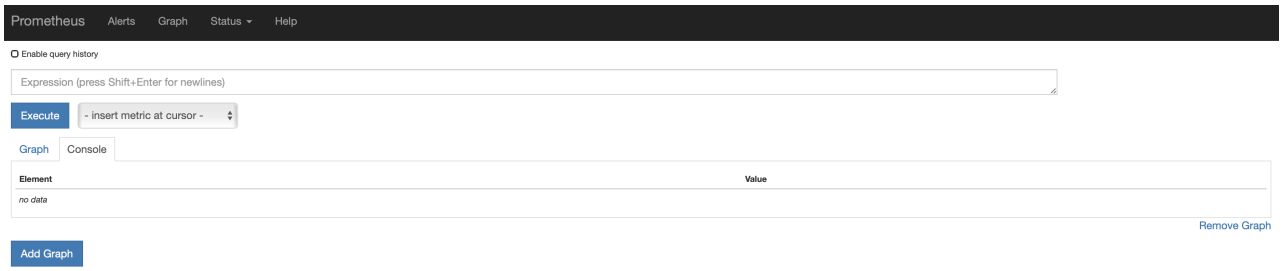


3. Querying Metrics with Prometheus

Prometheus will periodically *scrape* applications to retrieve their metrics (by default on the `/metrics` endpoint of the application). The Prometheus add-on for Istio is a Prometheus server that comes pre-configured to *scrape* Istio Mixer endpoints to collect its exposed metrics. It provides a mechanism for persistent storage and querying of those metrics metrics.

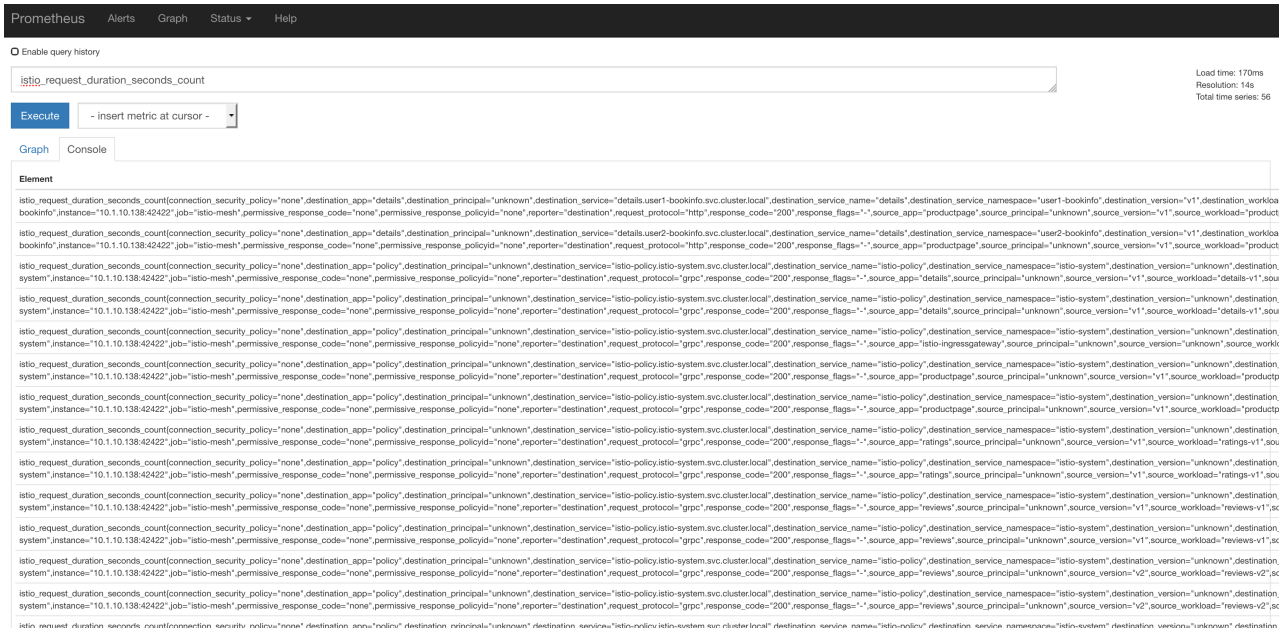
Open the Prometheus console.

You should see Prometheus home screen, similar to this:

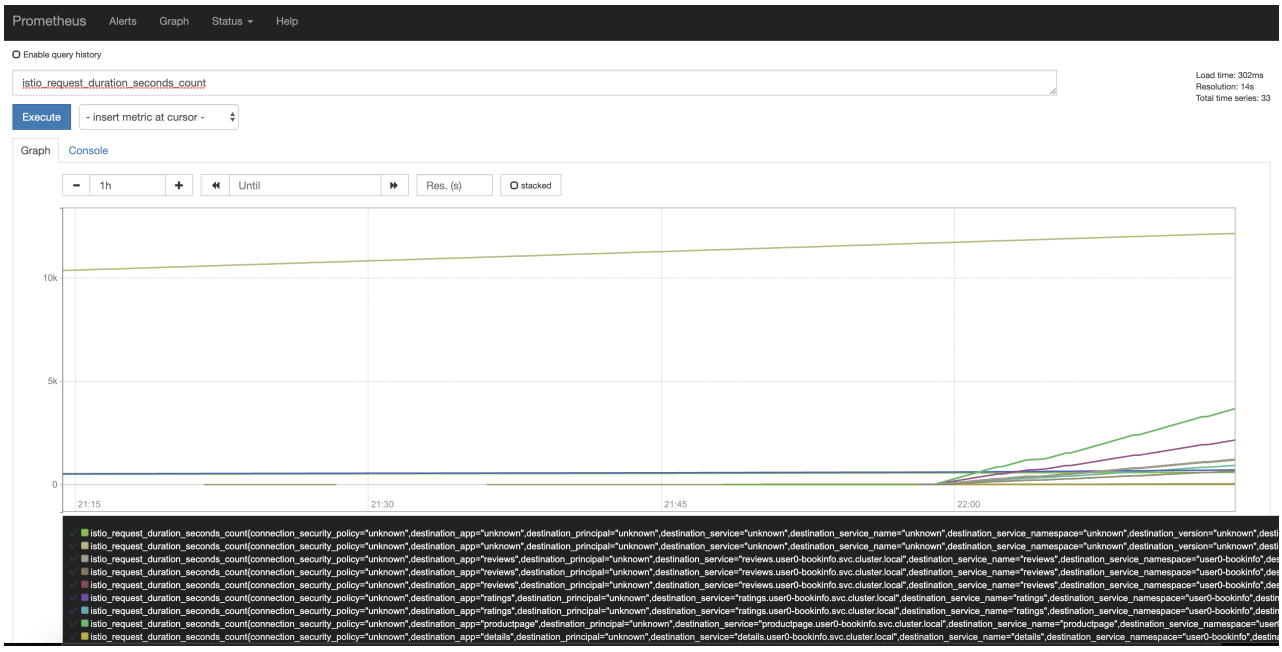


In the “Expression” input box at the top of the web page, enter the text: **istio_request_duration_seconds_count**. Then, click the **Execute** button.

You should see a listing of each of the application’s services along with a count of how many times it was accessed.



You can also graph the results over time by clicking on the *Graph* tab (adjust the timeframe from 1 hour to 1 minute for example):



Other expressions to try:

- Total count of all requests to *productpage* service:
  `istio_request_duration_seconds_count{destination_service=~"productpage.*"}`
- Total count of all requests to *v3* of the *reviews* service:
  `istio_request_duration_seconds_count{destination_service=~"reviews.*", destination_version="v3"}`
- Rate of requests over the past 5 minutes to all *productpage* services:
  `rate(istio_request_duration_seconds_count{destination_service=~"productpage.*", response_code="200"}[5m])`

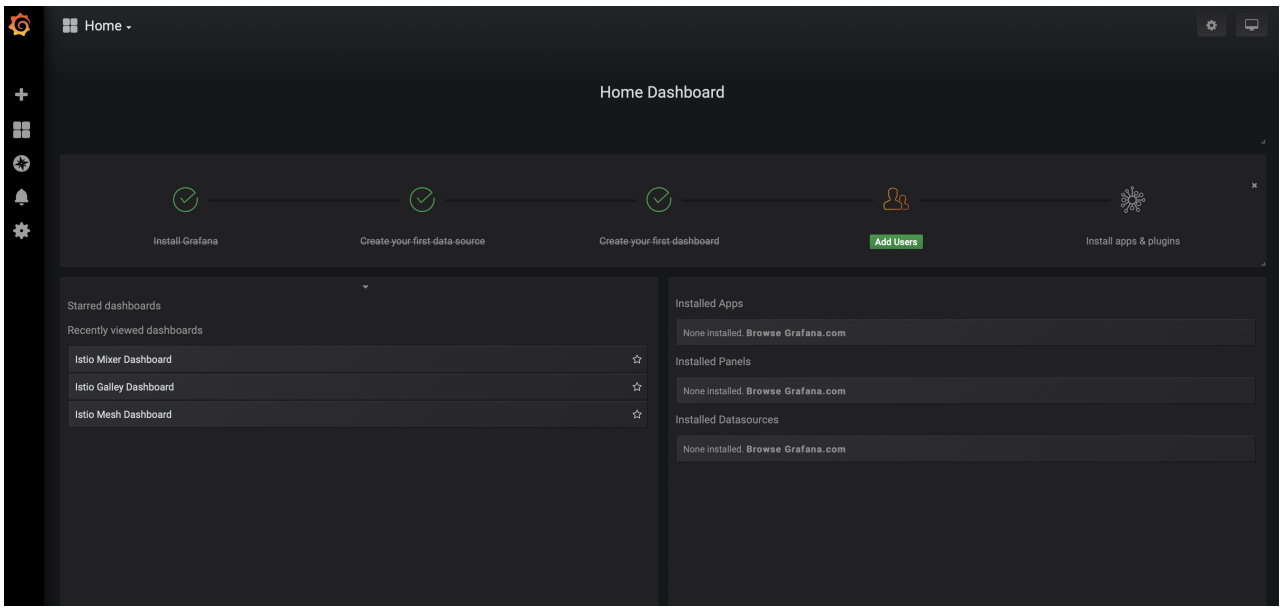
There are many, many different queries you can perform to extract the data you need. Consult the Prometheus documentation for more detail.

4. Visualizing Metrics with Grafana

As the number of services and interactions grows in your application, this style of metrics may be a bit overwhelming. Grafana provides a visual representation of many available Prometheus metrics extracted from the Istio data plane and can be used to quickly spot problems and take action.

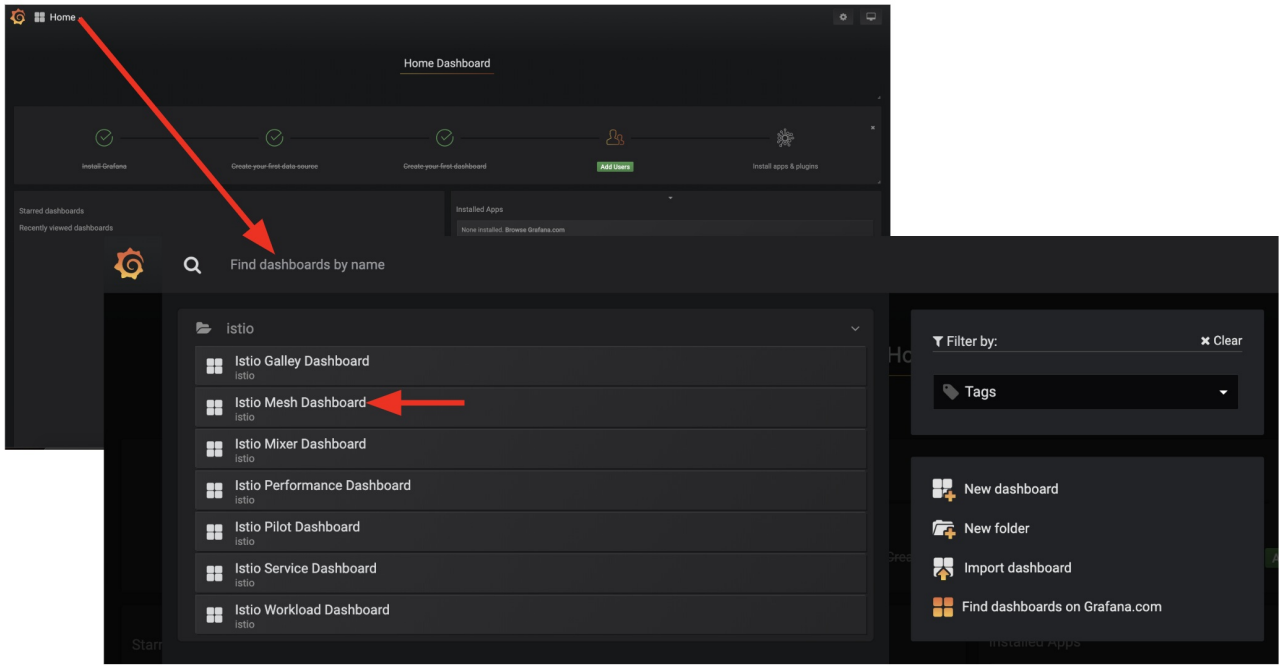
Open the Grafana console

You should see Grafana home screen, similar to this:

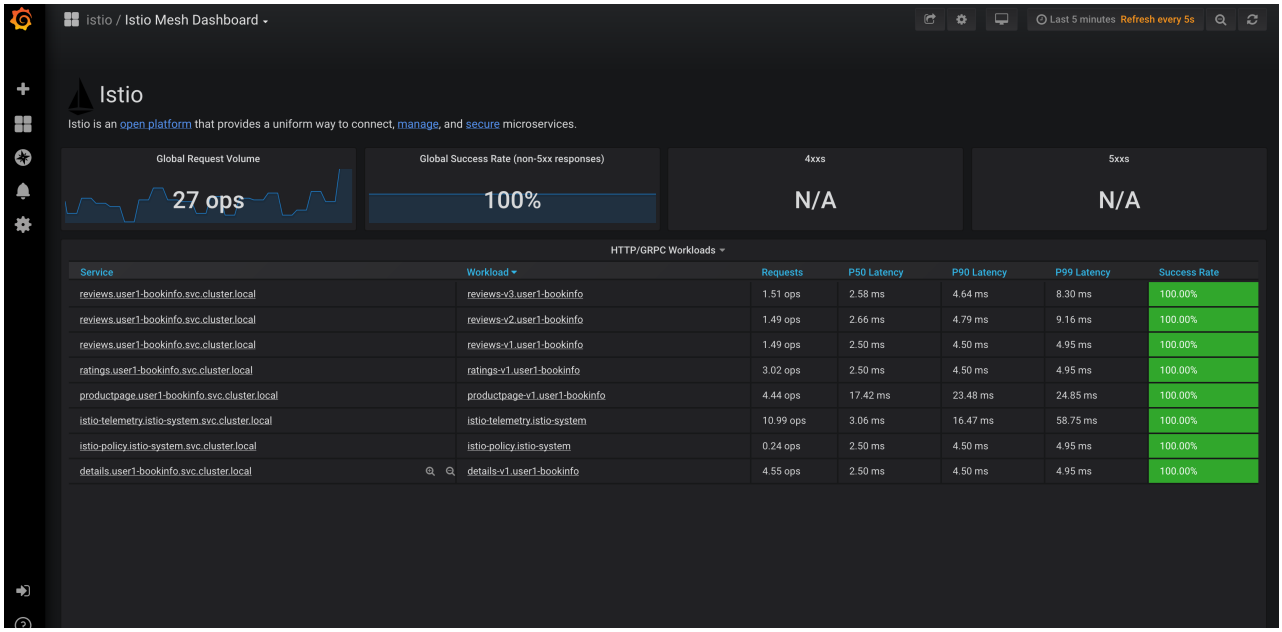


Istio Mesh Metrics

Select **Home > Istio > Istio Mesh Dashboard** to see Istio mesh metrics:

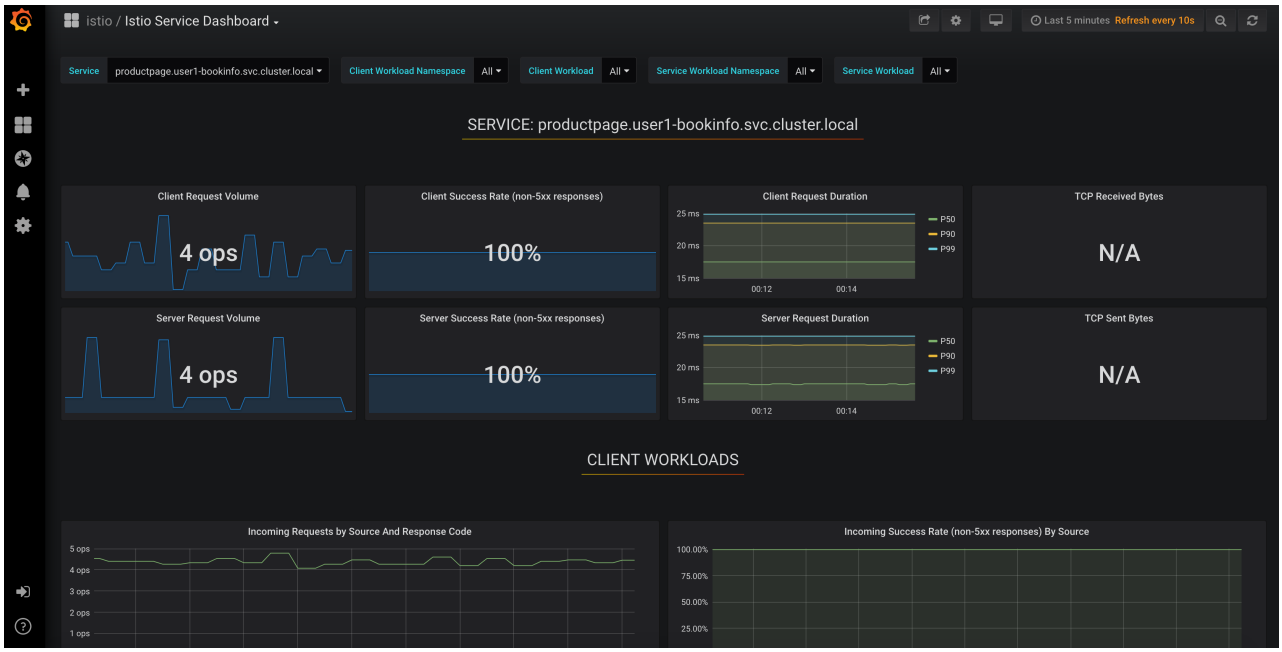


You will see the built-in Istio metrics dashboard::



Istio Service Metrics

Let's see detailed metrics of the **productpage** service. Click on **productpage.userXX-bookinfo.svc.cluster.local** and the service dashboard will look similar to this:



The Grafana Dashboard for Istio consists of three main sections:

- *A Global Summary View* provides a high-level summary of HTTP requests flowing through the service mesh.
- *A Mesh Summary View* provides slightly more detail than the Global Summary View, allowing per-service filtering and selection.
- *Individual Services View* provides metrics about requests and responses for each individual service within the mesh (HTTP and TCP).

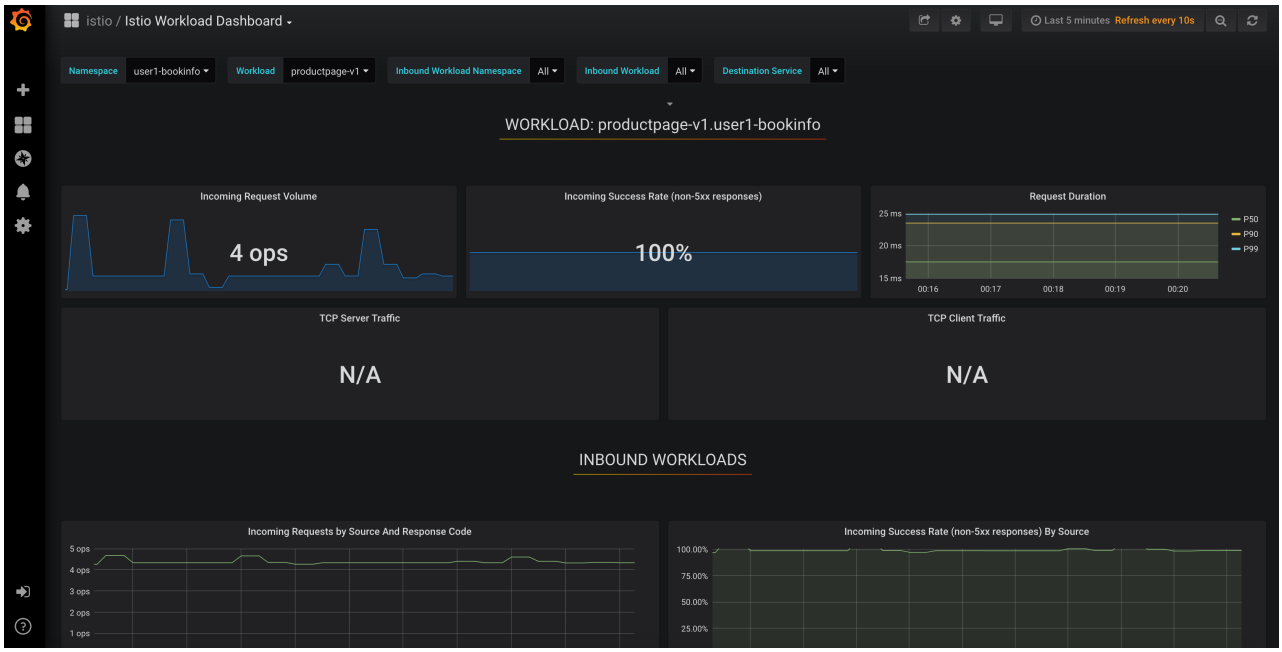
Note that *TCP Bandwidth* metrics are empty, as Bookinfo uses http-based services only. Lower down on this dashboard are metrics for workloads that call this service (labeled "Client Workloads") and for workloads that process requests from the service (labeled *Service Workloads*).

You can switch to a different service or filter metrics by *client-* and *service-workloads* by using drop-down lists at the top of the dashboard.

Istio Workload Metrics

To switch to the workloads dashboard, select **Home > Istio Workload Dashboard** from the drop-down list in the top left corner of the screen. You should see a screen similar to this:

> You should select your own userXX-bookinfo in the `Namespace` selector at the top to avoid noise from other workloads on the cluster!



This dashboard shows workload's metrics, and metrics for client- (inbound) and service (outbound) workloads. You can switch to a different workload, ot filter metrics by inbound or outbound workloads by using drop-down lists at the top of the dashboard.

For more on how to create, configure, and edit dashboards, please see the Grafana documentation.

As a developer, you can get quite a bit of information from these metrics without doing anything to the application itself. Let's use our new tools in the next section to see the real power of Istio to diagnose and fix issues in applications and make them more resilient and robust.

5. Request Routing

This task shows you how to configure dynamic request routing based on weights and HTTP headers.

Route rules control how requests are routed within an Istio service mesh. Route rules provide:

- *Timeouts*
- *Bounded retries* with timeout budgets and variable jitter between retries
- *Limits* on number of concurrent connections and requests to upstream services
- *Active (periodic) health checks* on each member of the load balancing pool
- *Fine-grained circuit breakers* (passive health checks) – applied per instance in the load balancing pool

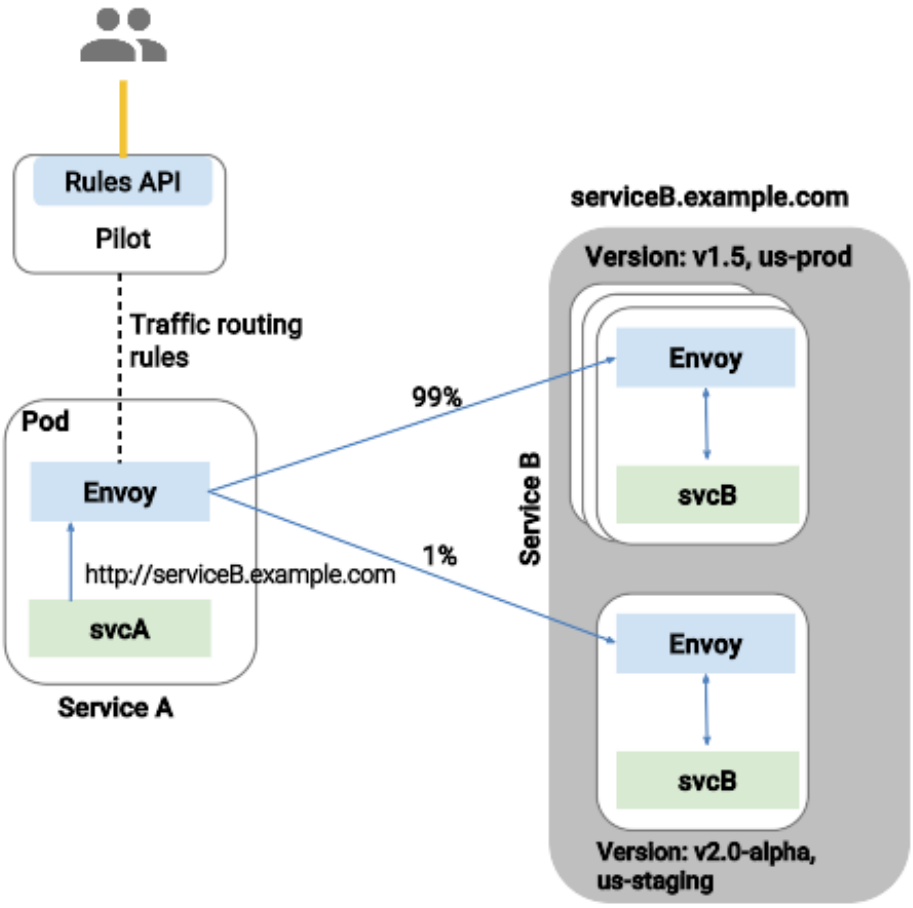
Requests can be routed based on the source and destination, HTTP header fields, and weights associated with individual service versions. For example, a route rule could route requests to different versions of a service.

Together, these features enable the service mesh to tolerate failing nodes and prevent localized failures from cascading instability to other nodes. However, applications must still be designed to deal with failures by taking appropriate fallback actions. For example, when all instances in a load balancing pool have failed, Istio will return HTTP 503. It is the responsibility of the application to implement any fallback logic that is needed to handle the HTTP 503 error code from an upstream service.

If your application already provides some defensive measures (e.g. using Netflix Hystrix), then that's OK. **Istio** is completely transparent to the application. A failure response returned by Istio would not be distinguishable from a failure response returned by the upstream service to which the call was made.

6. Service Versions

Istio introduces the concept of a service version, which is a finer-grained way to subdivide service instances by versions (*v1*, *v2*) or environment (*staging*, *prod*). These variants are not necessarily different API versions: they could be iterative changes to the same service, deployed in different environments (prod, staging, dev, etc.). Common scenarios where this is used include A/B testing or canary rollouts. Istio's traffic routing rules can refer to service versions to provide additional control over traffic between services.



As illustrated in the figure above, clients of a service have no knowledge of different versions of the service. They can continue to access the services using the hostname/IP address of the service. The Envoy sidecar/proxy intercepts and forwards all requests/responses between the client and the service.

7. VirtualService objects

In addition to the usual OpenShift object types like *BuildConfig*, *DeploymentConfig*, *Service* and *Route*, you also have new object types installed as part of Istio like *VirtualService*. Adding these objects to the running OpenShift cluster is how you configure routing rules for Istio.

For our application, without an explicit default route set, Istio will route requests to all available versions of a service in a round-robin fashion, and anytime you hit *v1* version you'll get no stars.

Let's create a default set of **virtual services** which will direct all traffic to the *reviews:v1* service version.

Open a new Terminal (while your other endless `for` loop continues to run) and execute this command to route all traffic to `v1`:

```
oc create -f /projects/cloud-native-workshop-v2m3-labs/istio/virtual-service-all-v1.yaml
```

You can see this default set of *virtual services* with:

`oc get virtualservices -o yaml`

There are default *virtual services* for each service, such as the one that forces all traffic to the *v1* version of the *reviews* service:

`oc get virtualservices/reviews -o yaml`

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  creationTimestamp: "2019-07-02T15:50:36Z"
  generation: 1
  name: reviews
  namespace: userXX-bookinfo
  resourceVersion: "2899673"
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v1
```

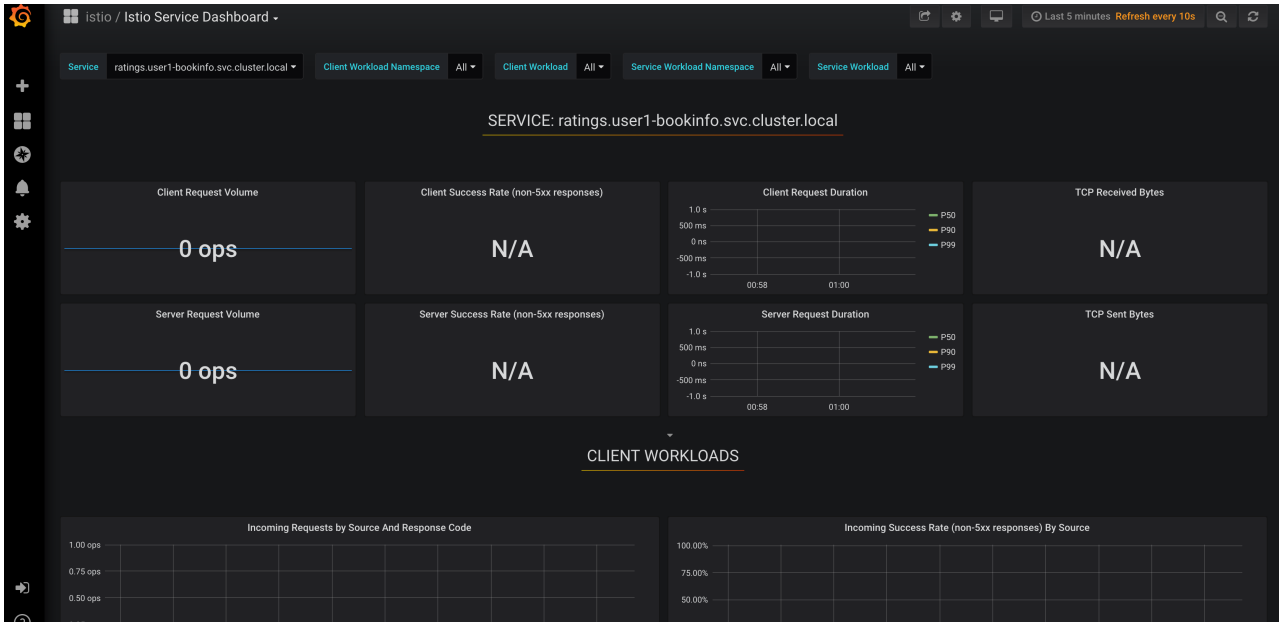
Now, access the application again in your web browser using the below link and reload the page several times - you should not see any rating stars since **reviews:v1** does not access the *ratings* service.

> **NOTE** - It may take a minute or two for the new routing to take effect. If you still see red or black stars, wait a minute and try again. Eventually it should no longer show any red/black stars.

Bookinfo Application with no rating stars at `http://$BOOK_URL/productpage`

To verify this, open the Grafana Dashboard (find this URL via *Networking* > *Routes*)

Scroll down to the **ratings** service in *Istio Service Dashboard* and notice that the requests coming from the reviews service have stopped:



8. A/B Testing with Istio

Let's enable the ratings service for a test user named *jason* by routing `productpage` traffic to *reviews:v2* and any others to *reviews:v3*. Execute:

```
oc apply -f /projects/cloud-native-workshop-v2m3-labs/istio/virtual-service-reviews-jason-v2-v3.yaml
```

> **TIP**: You can ignore warnings like *Warning: oc apply should be used on resource created by either oc create –save-config or oc apply*.

Confirm the rule is created:

```
oc get virtualservices/reviews -o yaml
```

Notice the *match* element:

```
http:
  - match:
    - headers:
        end-user:
          exact: jason
    route:
    - destination:
        host: reviews
        subset: v2
  - route:
    - destination:
        host: reviews
        subset: v3
```

This says that for any incoming HTTP request that has a cookie set to the *jason* user to direct traffic to **reviews:v2**, and others to **reviews:v3**.

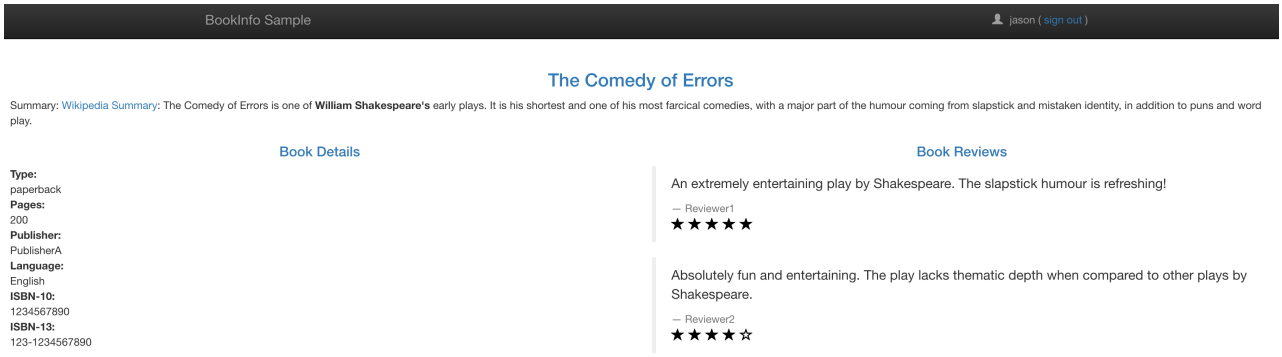
Now, access the application again via your own *Gateway URL*:

`http://YOUR_BOOK_APP_URL/productpage` and click **Sign In** (at the upper right) and sign in with:

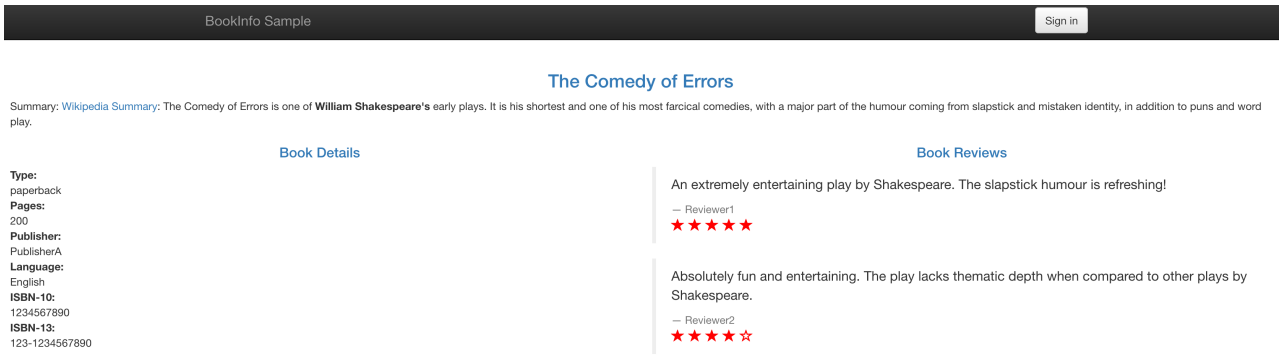
- Username: **jason**
- Password: **jason**

> If you get any certificate security exceptions, just accept them and continue. This is due to the use of self-signed certs.

Once you login, refresh a few times - you should always see the black ratings stars coming from **ratings:v2** since you're signed in as `jason`.



If you **sign out**, you'll return to the **reviews:v3** version which shows red ratings stars.



Congratulations!

In this lab, you used Istio to send 100% of the traffic to the a specific version of one of the application's services. You then set a rule to selectively send traffic to other versions of based on matching criteria (e.g. a header or user cookie) in a request.

This routing allows you to selectively send traffic to different service instances, e.g. for testing, or blue/green deployments, or dark launches, and more.

We'll explore this in the next step.

Advanced Service Mesh Development

Lab3 - Advanced Service Mesh Development

In this lab, you will use advanced service mesh features such as **Fault Injection**, **Traffic Shifting**, **Circuit Breaking**, and **Rate Limiting** with a different application - the Coolstore microservice developed in prior labs (i.e Catalog, Inventory) that you developed and deployed to OpenShift cluster in *Module 1* or/and *Module 2*.

If this is the first module you are doing today

If you have already deployed the *inventory* and *catalog* microservices from Module 1, you can skip this step!

If you haven't done Module 1 or Module 2 today, **or you didn't quite complete them**, you should deploy the coolstore application and microservices by executing the following shell scripts in CodeReady Workspaces Terminal:

> Replace `userXX` with your username before running these commands:

```
sh /projects/cloud-native-workshop-v2m3-labs/istio/scripts/deploy-inventory.sh userXX
```

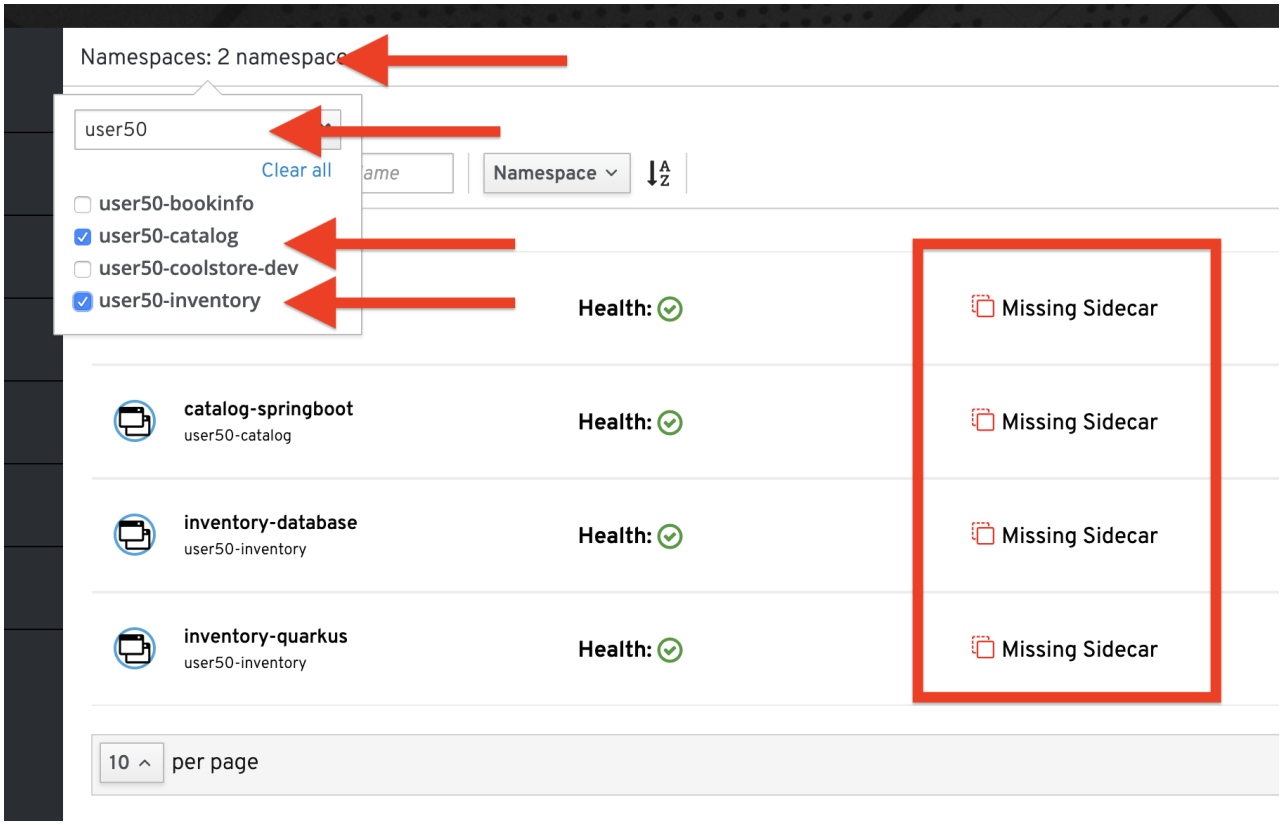
```
sh /projects/cloud-native-workshop-v2m3-labs/istio/scripts/deploy-catalog.sh userXX
```

This will build and deploy the inventory and catalog components into the `userXX-inventory` and `userXX-catalog` projects in OpenShift. They won't automatically get Istio sidecar proxy containers yet, but you'll add that in the next step!

1. Configuring Automatic Sidecar Injection in Coolstore Microservices

Let's go to the Kiali console once again to confirm that the microservices (*Catalog*, *Inventory*) are not running with a *sidecar* and are not yet visible in the service mesh.

Click on **Applications** on the left menu. Then, using the `Namespace` drop-down at the top, type in `userXX` (your username) to filter the list, and check the `userXX-catalog` and `userXX-inventory` namespaces, and de-select the `userXX-bookinfo` namespace. You will see **Missing Sidecar** in 4 applications.

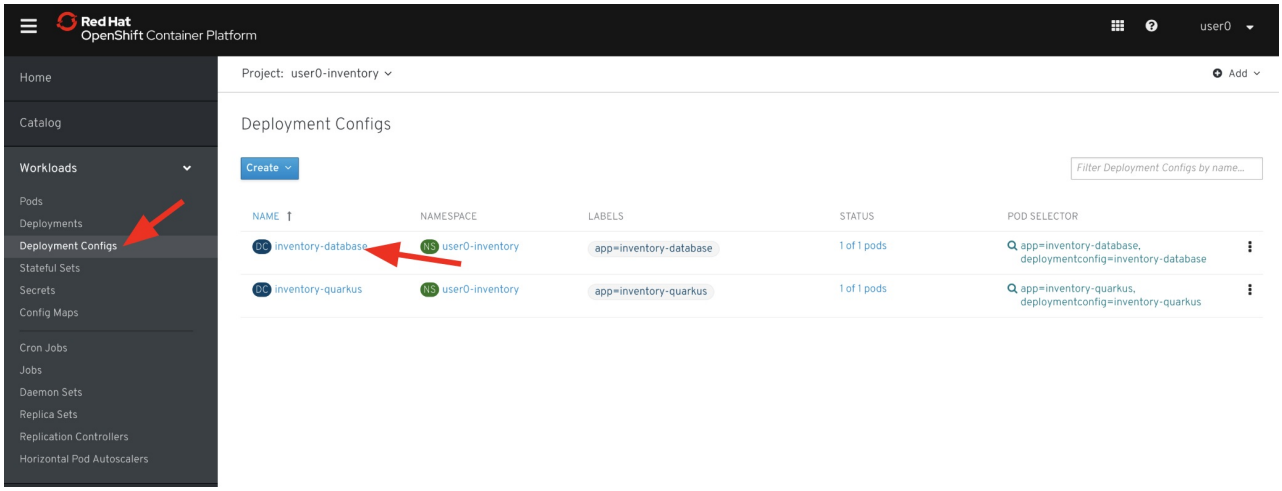


Upstream Istio community installations rely on the existence of a **proxy sidecar** within the application's pod to provide service mesh capabilities to the application. You can include the proxy sidecar by using a manual process before deployment. However, automatic injection ensures that your application contains the appropriate configuration for your service mesh at the time of its deployment.

Automatic injection of the sidecar is supported by using the *sidecar.istio.io/inject* annotation within your application yaml file. You will add this in the next step.

> Upstream Istio community installations require a specific label on the namespace after which all pods in that namespace are injected with the sidecar. The OpenShift Service Mesh approach requires you to opt in to injection using an annotation with no need to label namspaces. This method requires fewer privileges and does not conflict with other OpenShift capabilities such as builder pods.

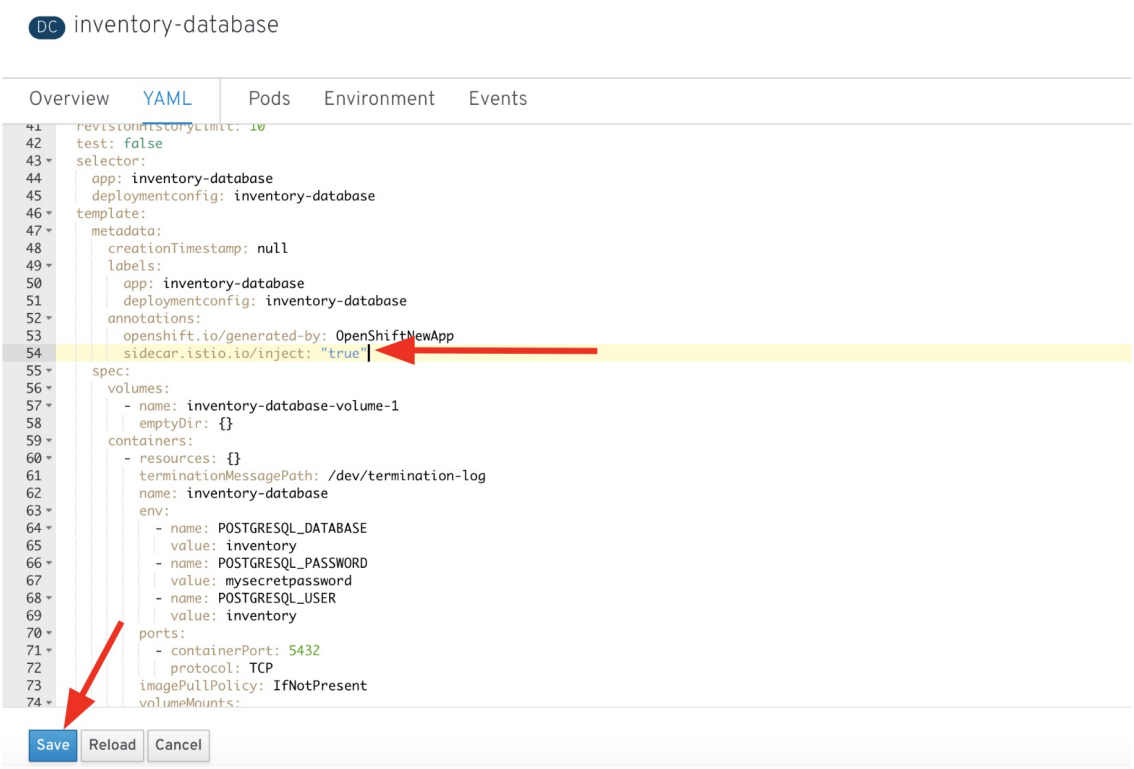
Back in the the OpenShift web console, select the *userXX-inventory* project in the project drop-down, then go to **Workloads > Deployment Configs** on the left menu, and click on *inventory-database* and then click the *YAML* tab.



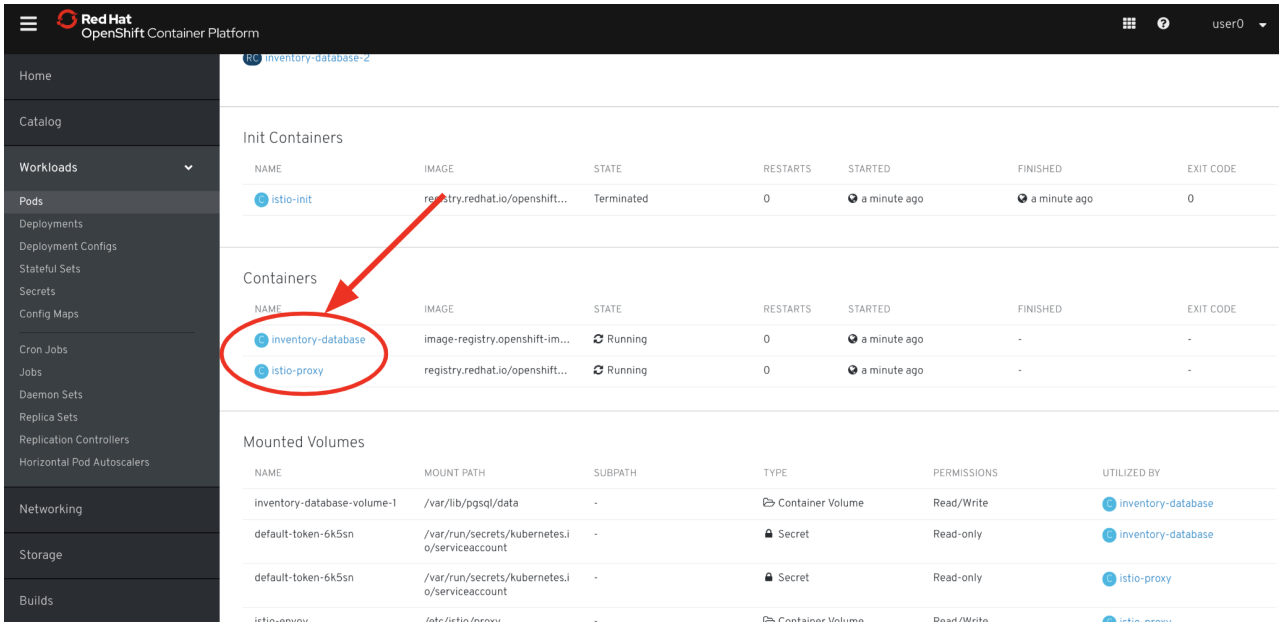
Add the following annotation in `spec.template.metadata.annotations` path in the YAML and click on **Save**.

```
sidecar.istio.io/inject: "true"
```

> **NOTE** Be sure to place this annotation in the correct `annotations` block under *spec* > *template* > *metadata* > *annotations* and match the same indentation as the annotation immediately above!



You will see a new **istio-proxy** container and *inventory-database* container in the "Pod Details" page when you navigate to *Workloads* > *Pods* > *inventory-database-xxxxx*. This new container will intercept traffic to and from the application pod and potentially alter and/or re-route it depending on service mesh configuration.



Now you will inject a sidecar container to application container (Inventory) as well. First, we need to remove the healhcheck of the inventory service as it will also be proxied and as we have no route for it, will be rejected (and the pod killed since Kubernetes cannot access the health check!). Alternative health checks involve running commands directly in the container but we'll just remove it for now. Remove it with:

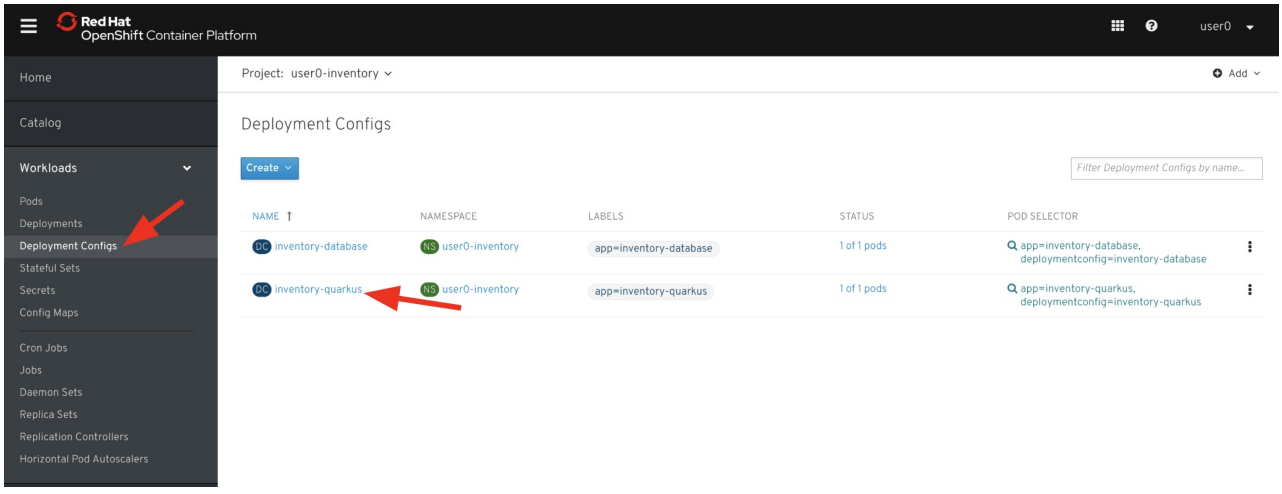
```
oc set probe dc/inventory-quarkus --remove --readiness --liveness -n userXX-inventory
```

> Replace `userXX` with your username!

Ensure the new deployemnt is successfully rolled out:

```
oc rollout status -w dc/inventory-quarkus -n userXX-inventory
```

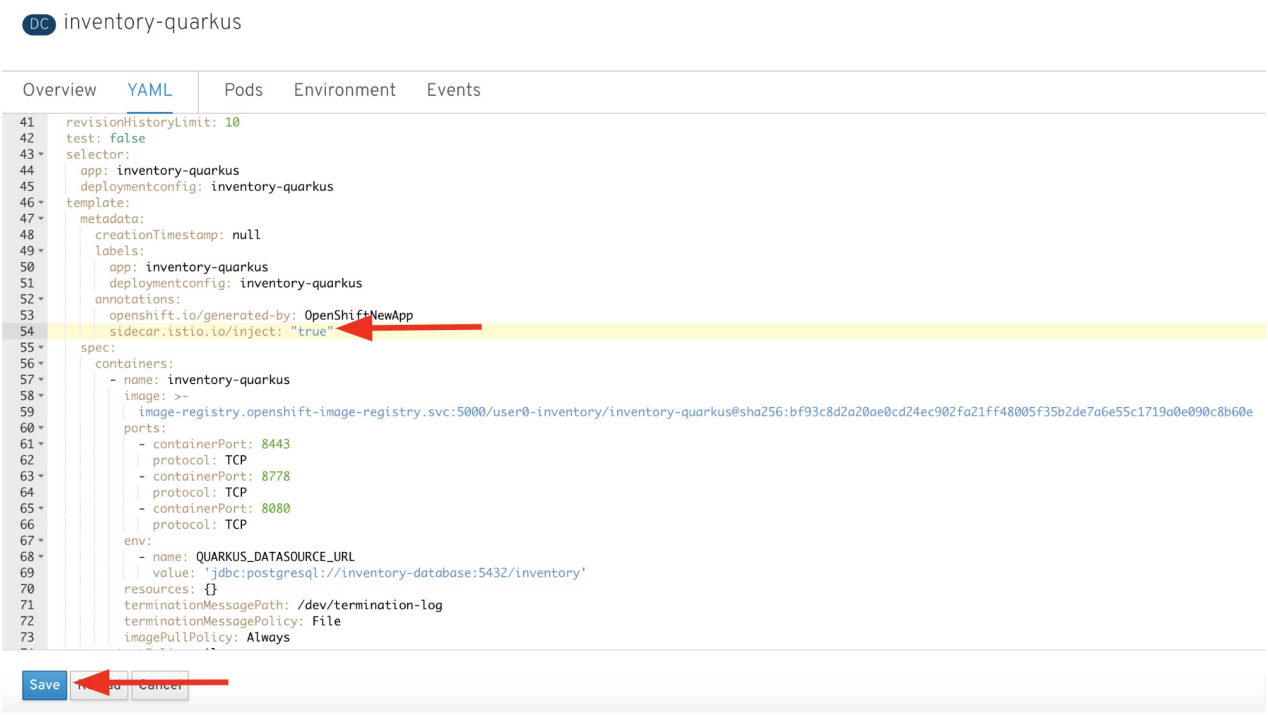
Navigate *Workloads* > *Deployment Configs* on the left menu. Select *userXX-inventory* project and click on *inventory-quarkus* and then the `YAML` tab.



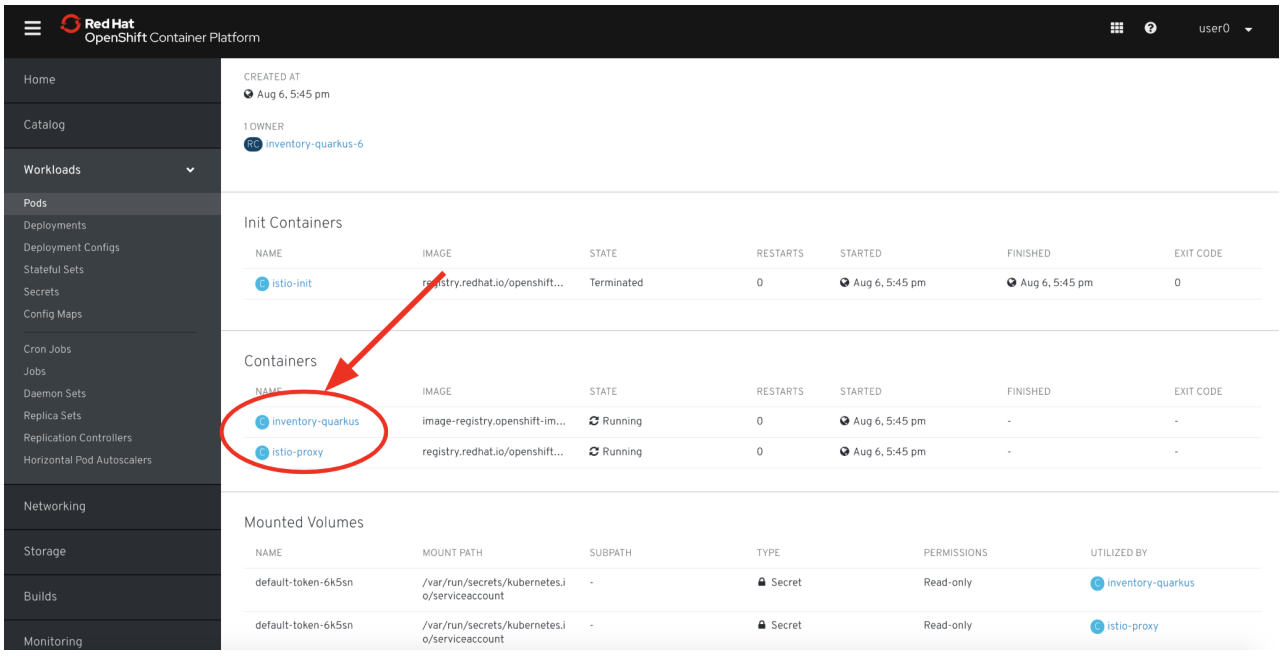
We'll add the same annotation as before:

```
sidecar.istio.io/inject: "true"
```

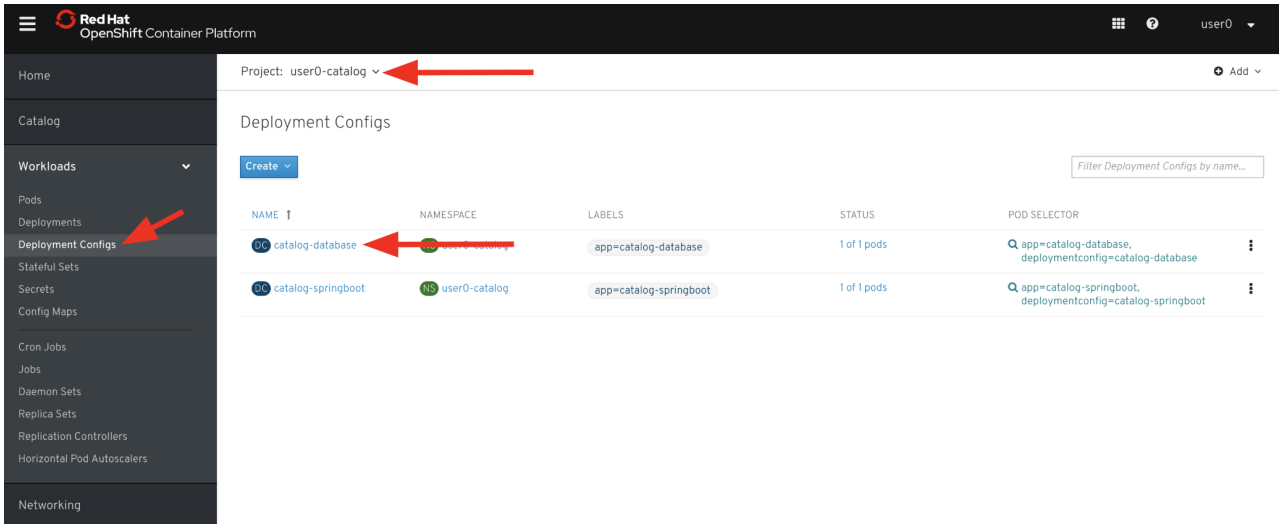
> **NOTE** Be sure to place this annotation in the correct `annotations` block under *spec* > *template* > *metadata* > *annotations* and match the same indentation as the annotation immediately above!



Again you will see **istio-proxy** container and *inventory-quarkus* container in the "Pod Details" page when you navigate *Workloads* > *Pods* > *inventory-quarkus-xxxxx*:

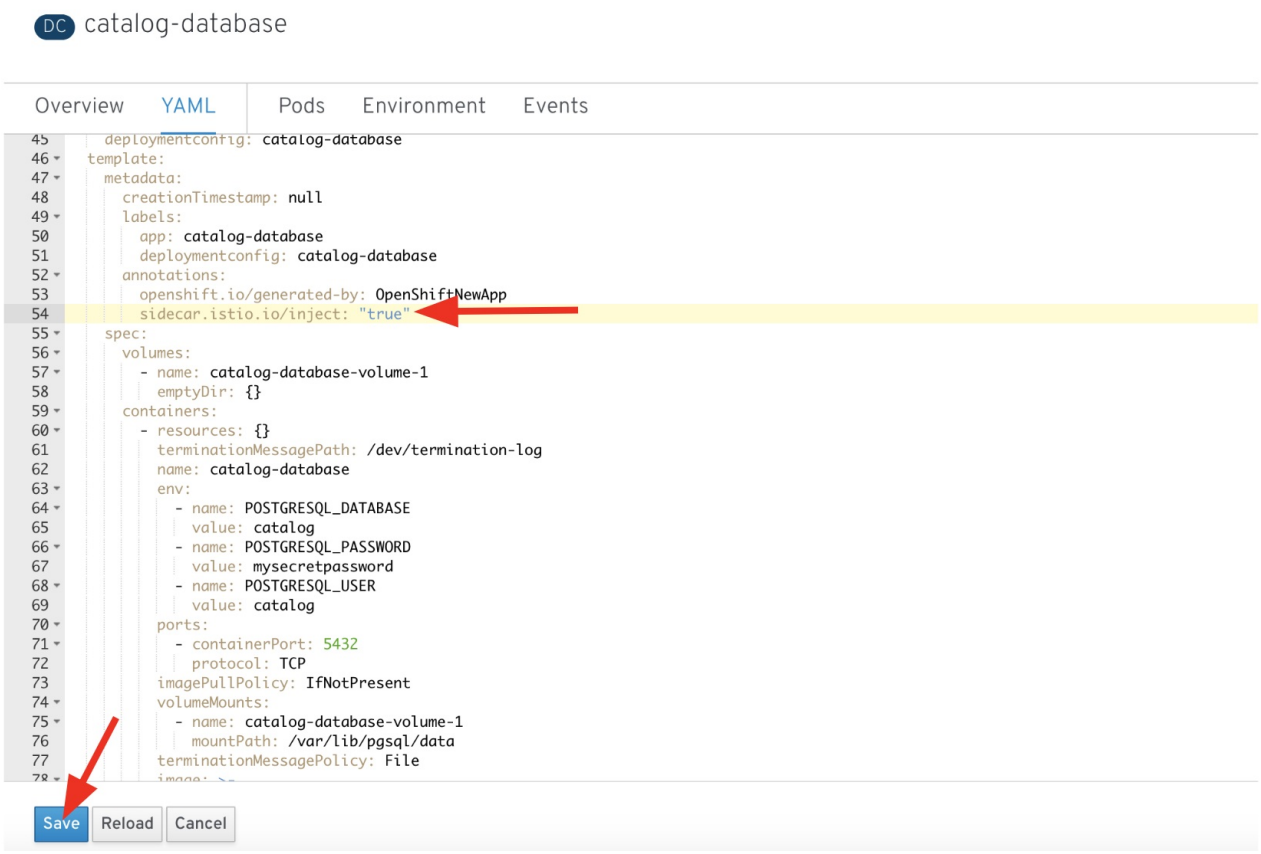


Next, do the same for the catalog and catalog's database. Go to **Workloads* > *Deployment Configs* on the left menu, select *userXX-catalog* project and click on *catalog-database*.



Then click on **YAML** tab and add the following annotation in `spec.template.metadata.annotations` path and click on **Save**.

```
sidecar.istio.io/inject: "true"
```



You will see **istio-proxy** container and *catalog-database* container in Pod Details page when you navigate *Workloads* > *Pods* > *catalog-database-xxxxx*.

Init Containers

NAME	IMAGE	STATE	RESTARTS	STARTED	FINISHED	EXIT CODE
istio-init	registry.redhat.io/openshift...	Terminated	0	10 minutes ago	10 minutes ago	0

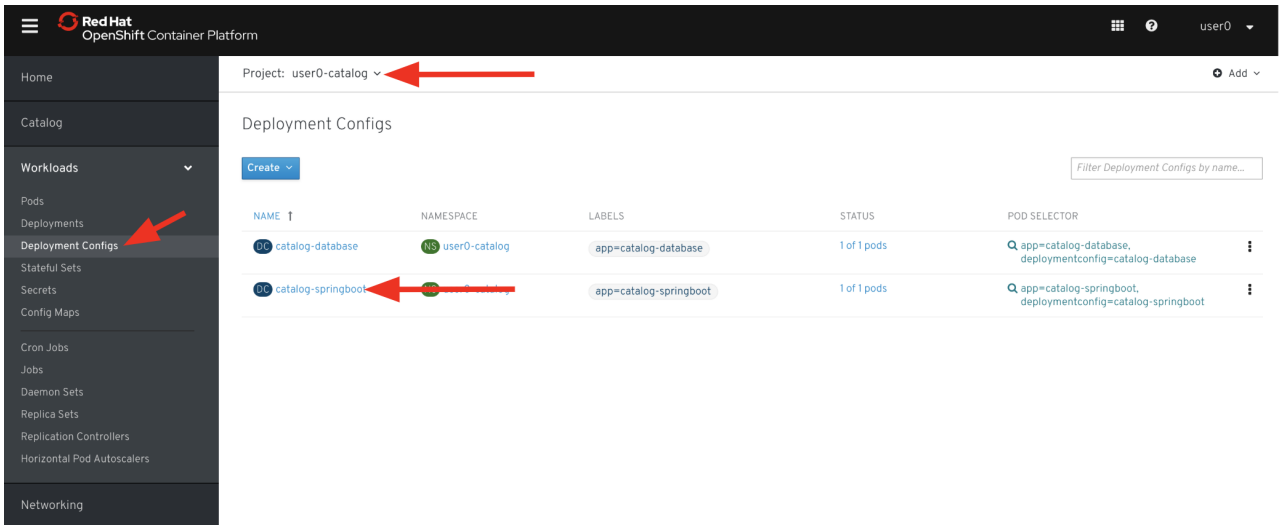
Containers

NAME	IMAGE	STATE	RESTARTS	STARTED	FINISHED	EXIT CODE
catalog-database	image-registry.openshift-im...	Running	0	10 minutes ago	-	-
istio-proxy	registry.redhat.io/openshift...	Running	0	10 minutes ago	-	-

Mounted Volumes

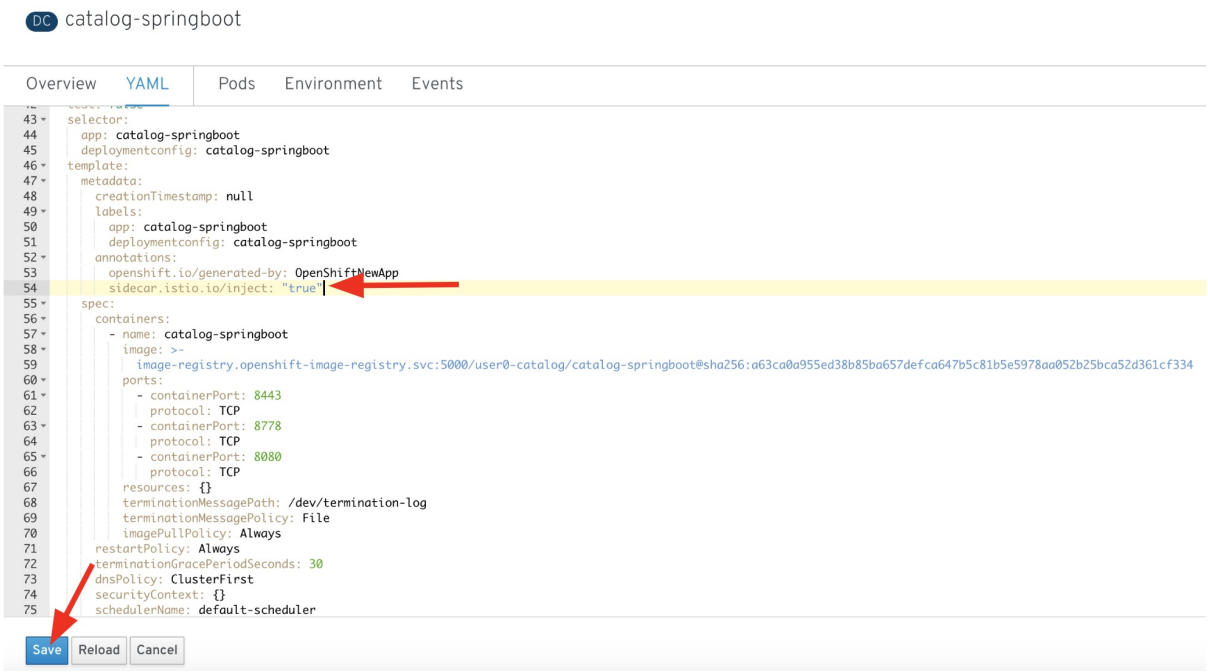
NAME	MOUNT PATH	SUBPATH	TYPE	PERMISSIONS	UTILIZED BY
catalog-database-volume-1	/var/lib/pgsql/data	-	Container Volume	Read/Write	catalog-database
default-token-7qmdq	/var/run/secrets/kubernetes.io/serviceaccount	-	Secret	Read-only	catalog-database
default-token-7qmdq	/var/run/secrets/kubernetes.io/serviceaccount	-	Secret	Read-only	istio-proxy
istio-envoy	/etc/istio/proxy	-	Container Volume	Read/Write	istio-proxy
istio-certs	/etc/certs/	-	Secret	Read-only	istio-proxy

Next, let's inject a sidecar container to application container (Catalog) as well, go to **Workloads > Deployment Configs* on the left menu, select *userXX-catalog* project and click on *catalog-springboot*.

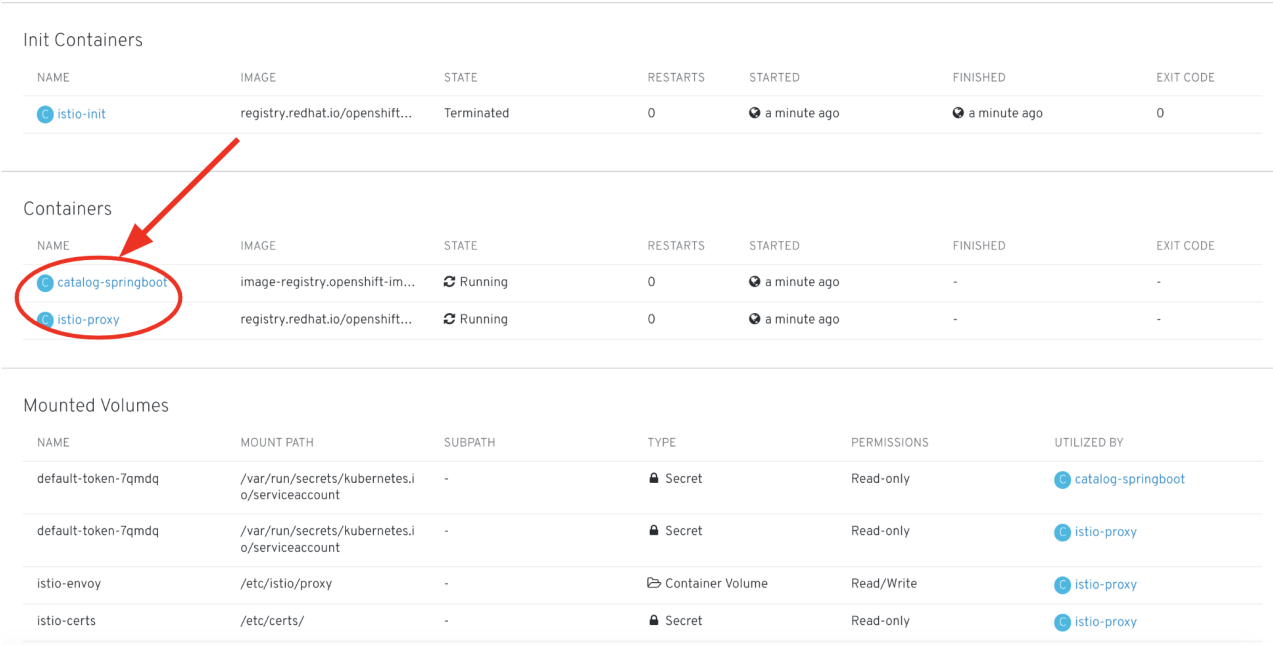


Add the same annotation (on the YAML tab):

```
sidecar.istio.io/inject: "true"
```



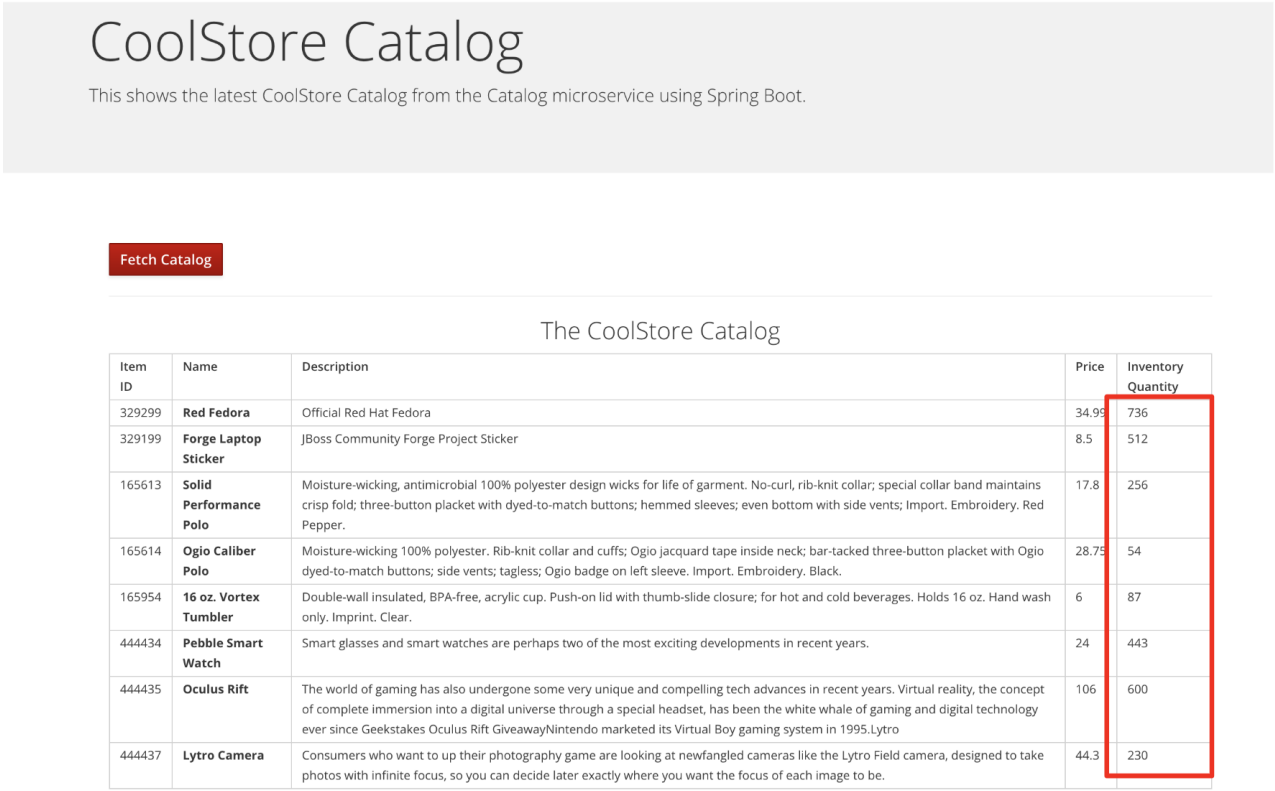
You will see **istio-proxy** container and *catalog-springboot* container in the "Pod Details" page when you navigate *Workloads* > *Pods* > *catalog-springboot-xxxxx*:



Let's make sure if inventory and catalog services are working correctly via accessing *Catalog Route URL* in your browser. You can also find the URL via *Networking* > *Routes* in OpenShift web console, after selecting the `userXX-catalog` from the *namespace* dropdown menu. Open the URL in your browser:

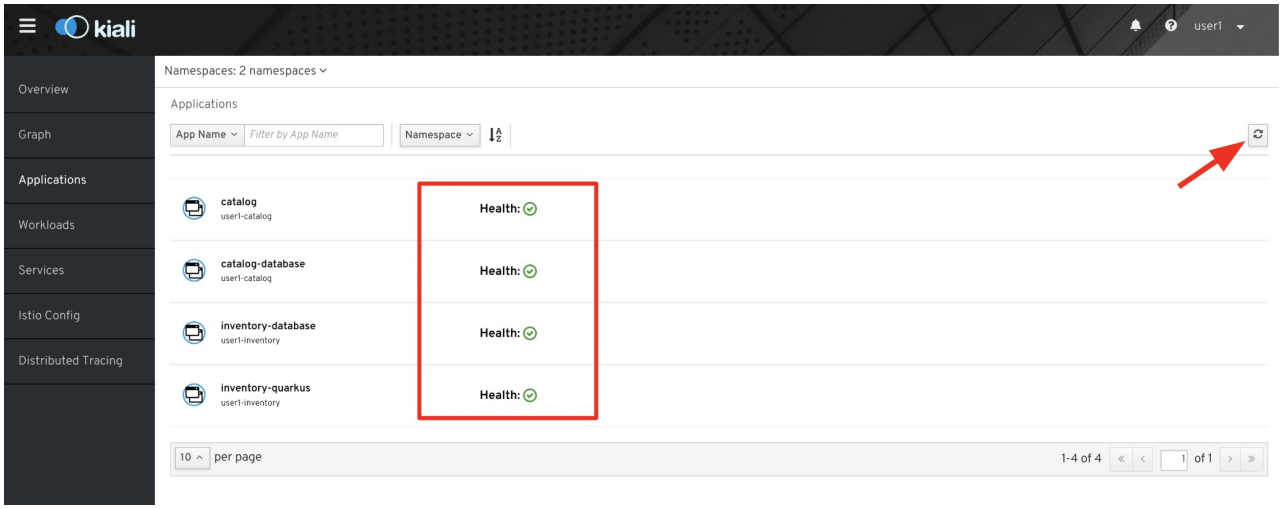
> Catalog UI (replace `userXX` with your username): http://catalog-springboot-userXX-catalog.apps.cluster-beijing-5bc2.beijing-5bc2.open.redhat.com

You will see the following web page including **Inventory Quantity** if the catalog service can access the inventory service via *Istio proxy sidecar*:

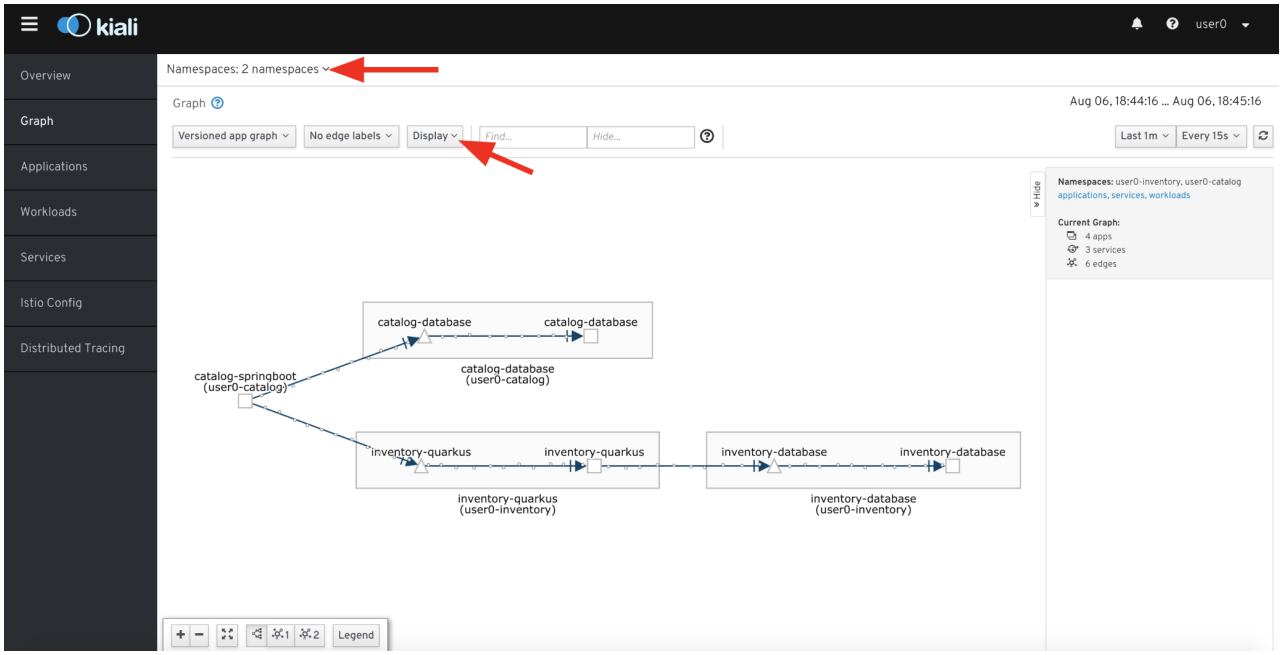


> Leave this page open as the *Catalog UI browser* creates traffic (every 2 seconds) between services, which is useful for testing.

Now, reload **Applications** in Kiali console and verify that the *Missing sidecar* warning is no longer present:



Also, go to the Service Graph page and check *userXX-inventory*, *userXX-catalog* in Namespace, check **Traffic Animation** in *Display* for understanding the traffic flow from catalog service to inventory service:



2. Fault Injection

This step will walk you through how to use **Fault Injection** to test the end-to-end failure recovery capability of the application as a whole. An incorrect configuration of the failure recovery policies could result in unavailability of critical services. Examples of incorrect configurations include incompatible or restrictive timeouts across service calls.

Istio provides a set of failure recovery features that can be taken advantage of by the services in an application. Features include:

- Timeouts
- Bounded retries with timeout budgets and variable jitter between retries
- Limits on number of concurrent connections and requests to upstream services
- Active (periodic) health checks on each member of the load balancing pool
- Fine-grained circuit breakers (passive health checks) – applied per instance in the load balancing pool

These features can be dynamically configured at runtime through Istio's traffic management rules.

A combination of active and passive health checks minimizes the chances of accessing an unhealthy service. When combined with platform-level health checks (such as readiness/liveness probes in OpenShift), applications can ensure that unhealthy pods/containers/VMs can be quickly weeded out of the service mesh, minimizing the request failures and impact on latency.

Together, these features enable the service mesh to tolerate failing nodes and prevent localized failures from cascading instability to other nodes.

Istio enables protocol-specific *fault injection* into the network (instead of killing pods) by delaying or corrupting packets at TCP layer.

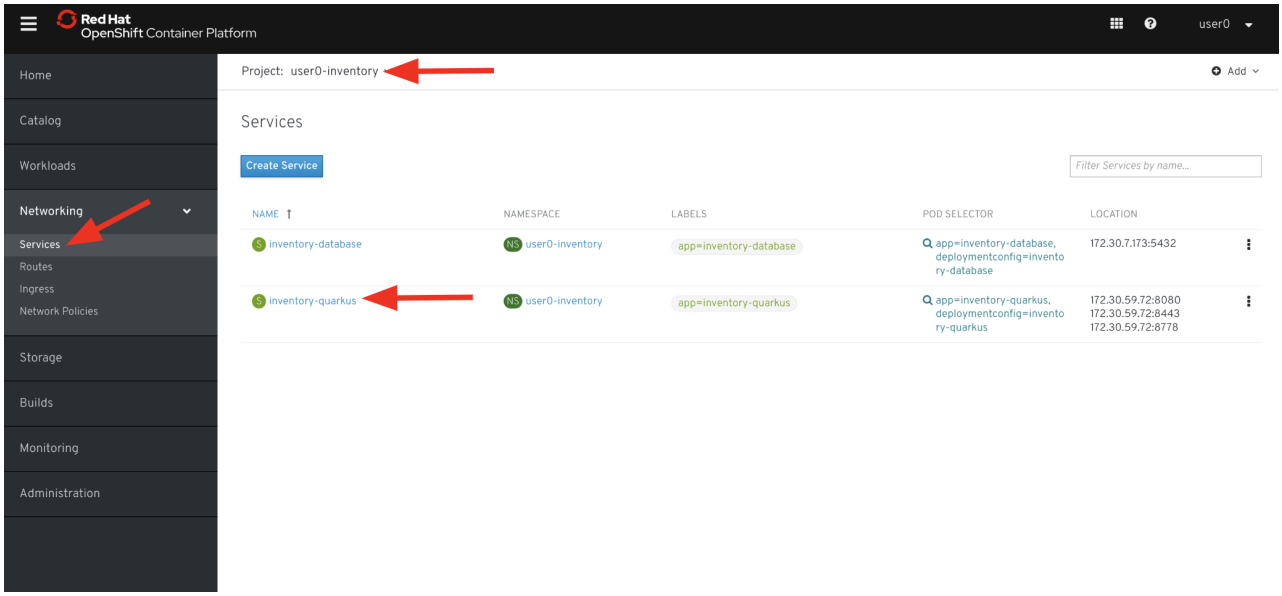
Two types of faults can be injected:

- *Delays* are timing failures. They mimic increased network latency or an overloaded upstream service.
- *Aborts* are crash failures. They mimic failures in upstream services. Aborts usually manifest in the form of HTTP error codes or TCP connection failures.

Inject a fault

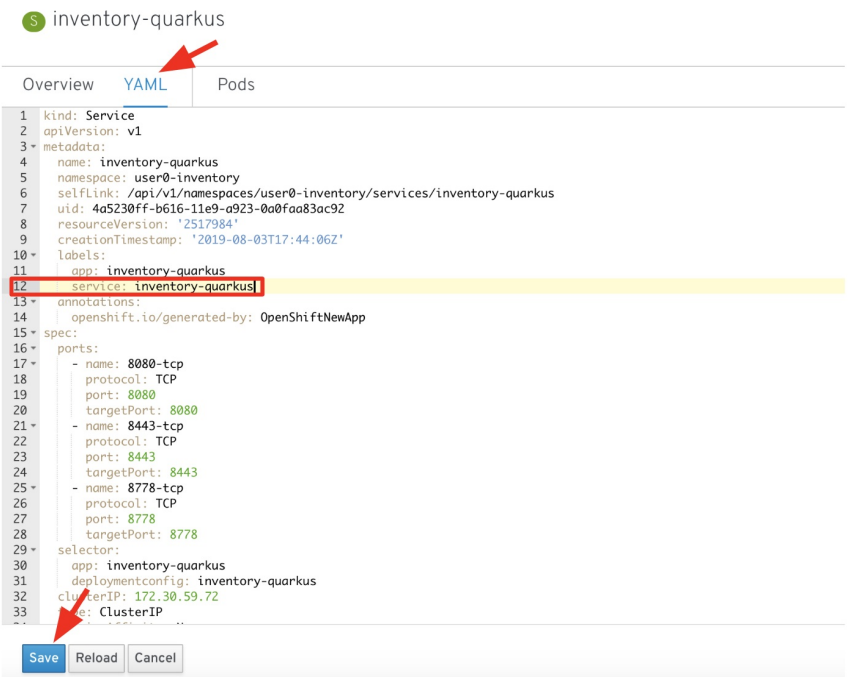
To test our application microservices for resiliency, we will inject a failure in **50%** of the requests to the *inventory* service, causing the service to appear to fail (and return `HTTP 5xx` errors).

First, add the following label in the Inventory service to use a *virtual service*. In the OpenShift Web Consle, select the *userXX-inventory* project in the project selector drop-down, then navigate to *Networking* > *Services* in the left menu, and select *inventory-quarkus*.



Click on **YAML** tab and add the following variables at the *metadata* > *labels* area of the YAML file as shown:

```
service: inventory-quarkus
```

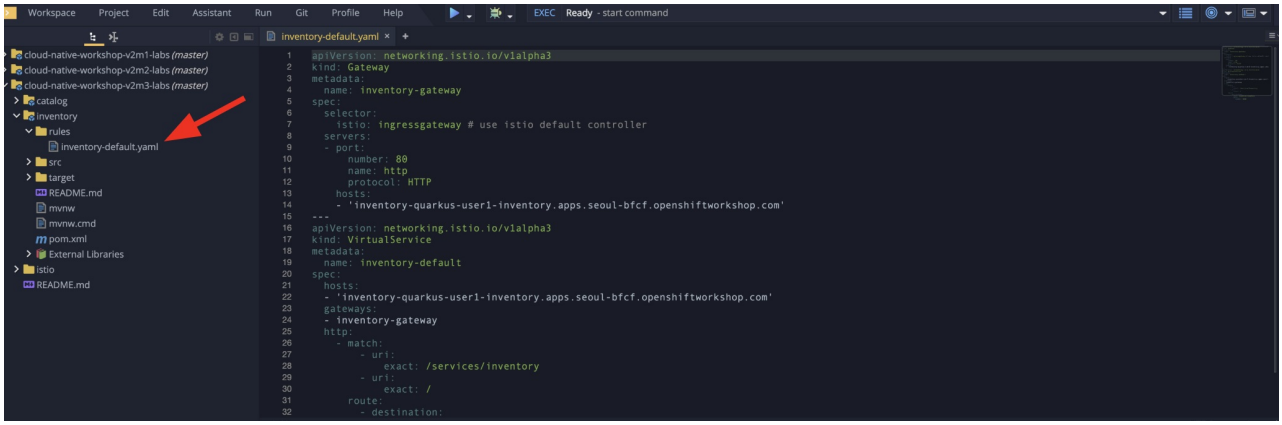


Click on **Save**.

In CodeReady, open the empty **inventory-default.yaml** file in the `/projects/cloud-native-workshop-v2m3-labs/inventory/rules/` directory. Add the below code to the file to create a gateway and virtual service:

> You'll need to replace `YOUR_INVENTORY_GATEWAY_URL` with the route URL for the inventory service, which looks like `inventory-quarkus-userXX-inventory.apps.cluster-beijing-5bc2.beijing-5bc2.open.redhat.com` (replace `userXX` with your username). There are two places to make this substitution, so do them both!

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: inventory-gateway
spec:
  selector:
    istio: ingressgateway # use istio default controller
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - 'YOUR_INVENTORY_GATEWAY_URL'
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: inventory-default
spec:
  hosts:
  - 'YOUR_INVENTORY_GATEWAY_URL'
  gateways:
  - inventory-gateway
  http:
    - match:
        - uri:
            exact: /services/inventory
        - uri:
            exact: /
      route:
        - destination:
            host: inventory-quarkus
            port:
              number: 8080
```



Delete the old direct route that was setup earlier with:

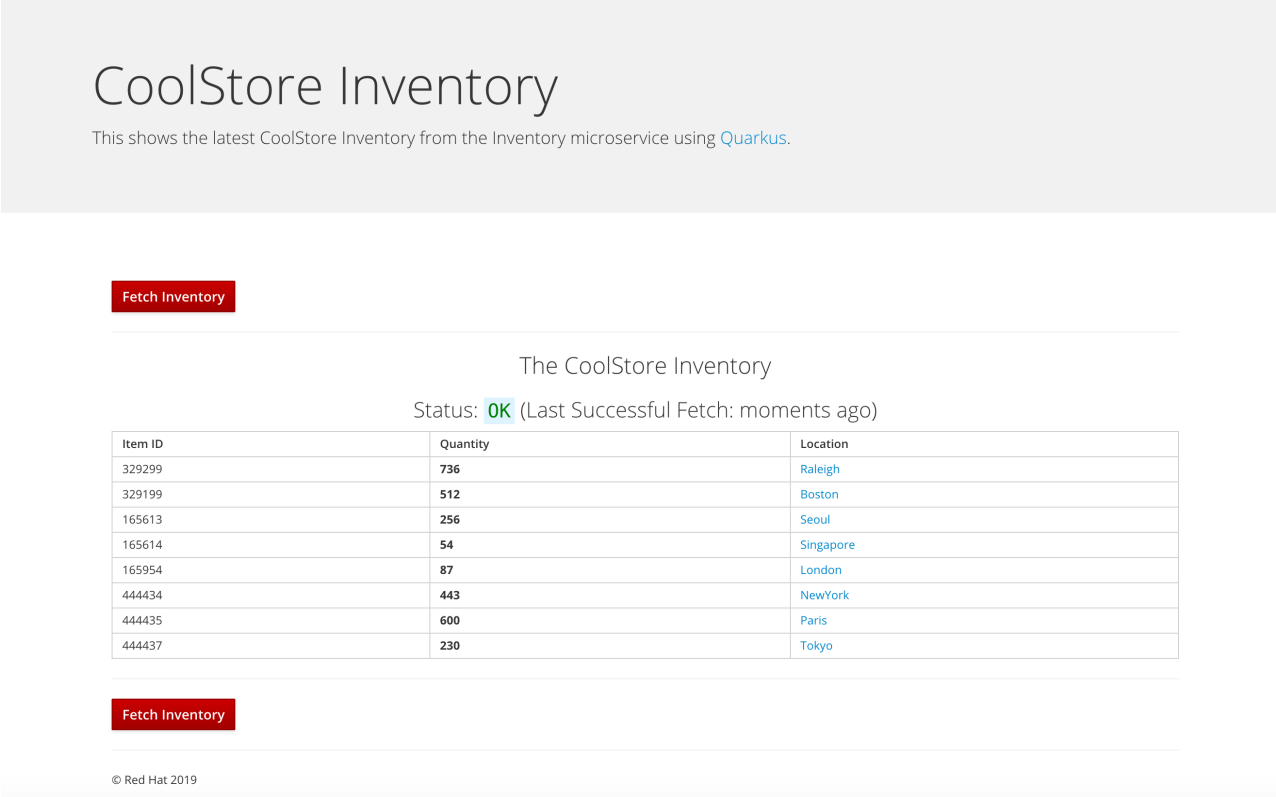
```
oc delete route/inventory-quarkus -n userXX-inventory
```

Create the new Istio-powered route by running the following command via CodeReady Workspaces Terminal to create this object in OpenShift:

`oc create -f /projects/cloud-native-workshop-v2m3-labs/inventory/rules/inventory-default.yaml -n userXX-inventory`

Now, you can test if the inventory service works correctly via accessing the **YOUR_INVENTORY_GATEWAY_URL** in your browser:

`i.e. http://inventory-quarkus-userXX-inventory.apps.cluster-beijing-5bc2.beijing-5bc2.open.redhat.com` (replace `userXX` with your username)



Let's inject a failure (*500 status*) in **50%** of requests to *inventory* microservices. Edit *inventory-default.yaml* as below.

Open **inventory-vs-fault.yaml** file in `/projects/cloud-native-workshop-v2m3-labs/inventory/rules/` and copy the following codes.

> You need to replace all `YOUR_INVENTORY_GATEWAY_URL` with the previous route URL that you copied earlier.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: inventory-fault
spec:
  hosts:
  - 'YOUR_INVENTORY_GATEWAY_URL'
  gateways:
  - inventory-gateway
  http:
    - fault:
        abort:
          httpStatus: 500
          percentage:
            value: 50
      route:
        - destination:
            host: inventory-quarkus
            port:
              number: 8080
```

Before creating a new **inventory-fault VirtualService**, we need to delete the existing inventory-default virtualService. Run the following command via CodeReady Workspaces Terminal:

`oc delete virtualservice/inventory-default -n userXX-inventory` (replace `userXX` with your username)

Then create a new virtualservice and gateway with this command:

`oc create -f /projects/cloud-native-workshop-v2m3-labs/inventory/rules/inventory-vs-fault.yaml -n userXX-inventory`

Let's find out if the fault injection works corectly via accessing the Inventory gateway once again. You will see that the **Status** of CoolStore Inventory continues to change between **DEAD** and **OK**:

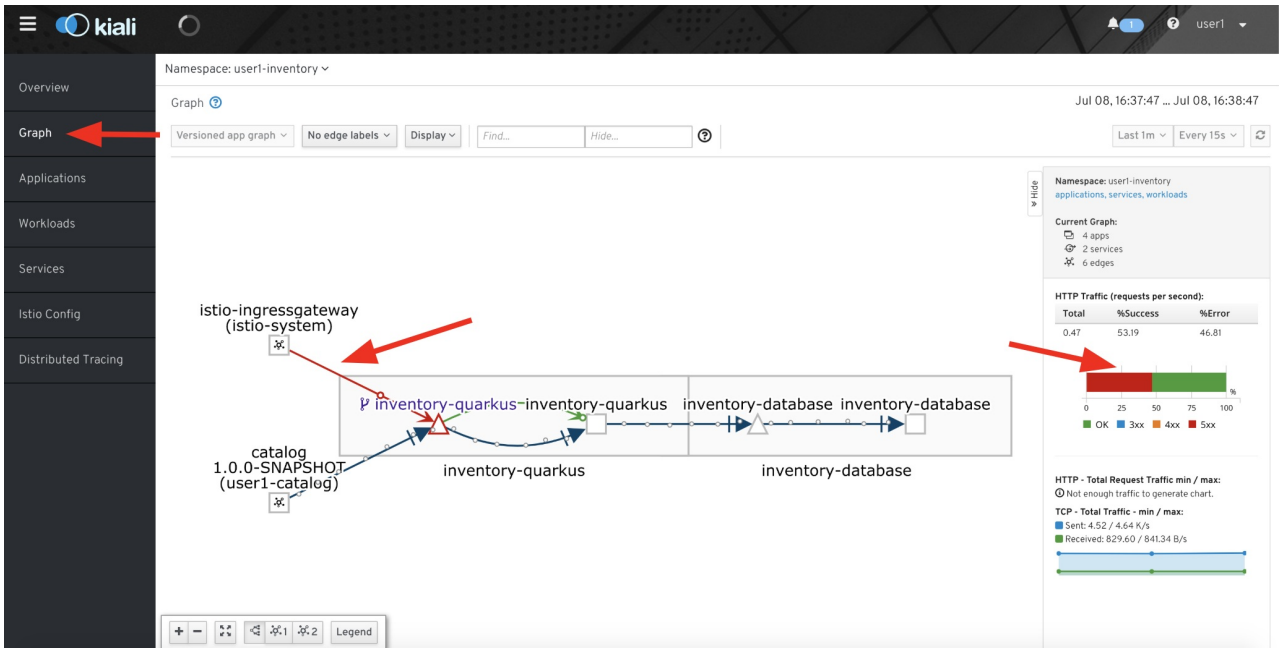
The CoolStore Inventory

Status: DEAD (Last Successful Fetch: 2seconds ago)

The CoolStore Inventory

Status: OK (Last Successful Fetch: moments ago)

In the **Kiali** console you will also see failures for 50% of traffic bound for the `inventory` service. You will see `red` traffic from *istio-ingressgateway* as well as around 50% of requests are displayed as *5xx* on the right side, *HTTP Traffic*. It may not be *exactly* 50% since some traffic is coming from the catalog and ingress gateway at the same time, but it will approach 50% over time.



Let's now add a 5 second delay for the `inventory` service.

Open **inventory-vs-fault-delay.yaml** file in `/projects/cloud-native-workshop-v2m3-labs/inventory/rules/` and copy the following code into it:

> Again, you need to replace all **YOUR_INVENTORY_GATEWAY_URL** with the previous route URL that you copied earlier.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: inventory-fault-delay
spec:
  hosts:
  - 'YOUR_INVENTORY_GATEWAY_URL'
  gateways:
  - inventory-gateway
  http:
    - fault:
        delay:
          fixedDelay: 5s
          percentage:
            value: 100
      route:
        - destination:
            host: inventory-quarkus
            port:
              number: 8080
```

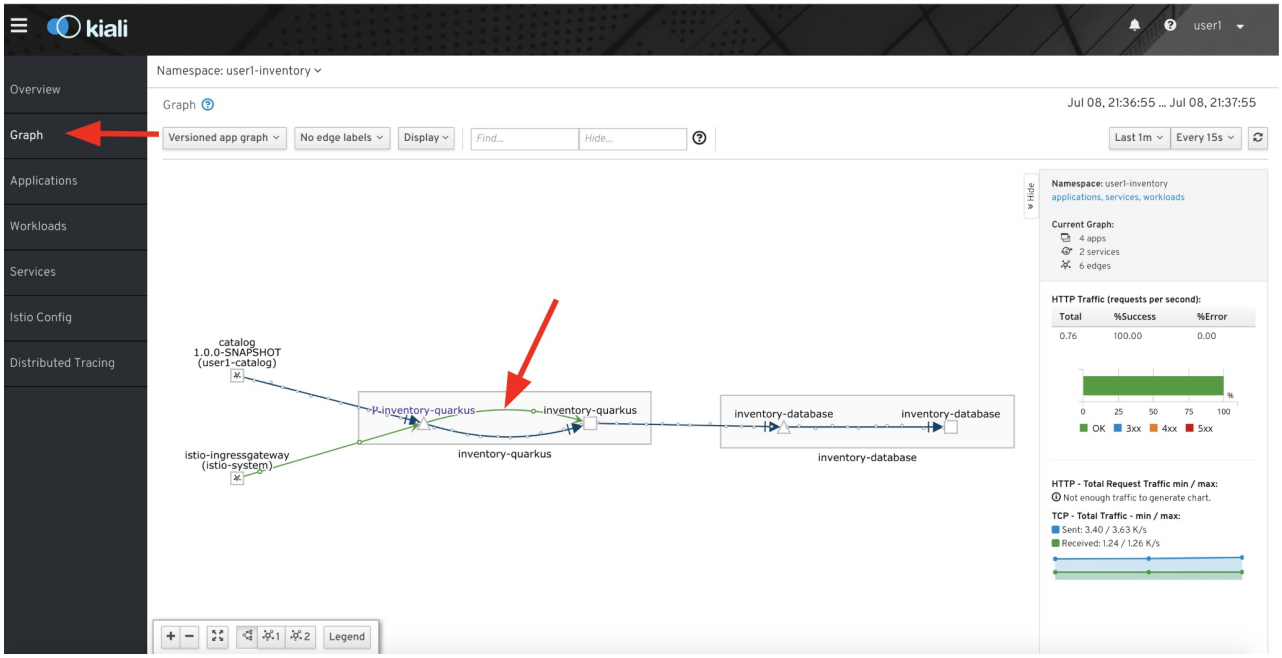
Before creating a new **inventory-fault-delay VirtualService**, we need to delete the existing inventory-fault VirtualService. Run the following command via CodeReady Workspaces Terminal:

`oc delete virtualservice/inventory-fault -n userXX-inventory`

Then create a new virtualservice and gateway.

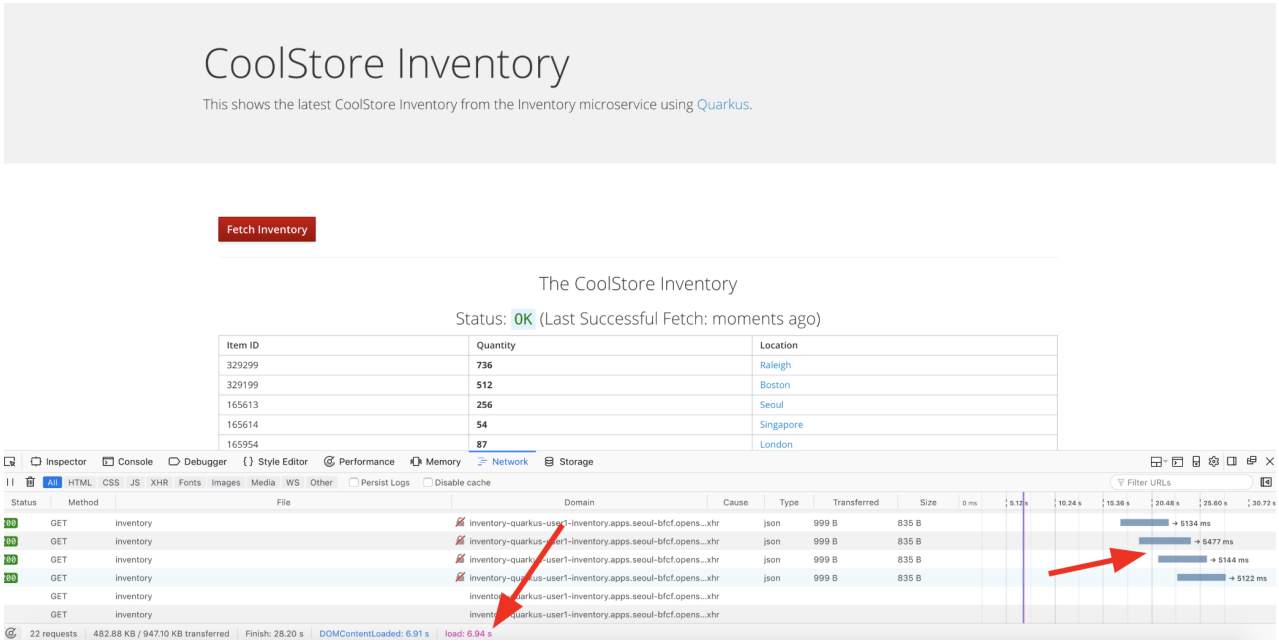
`oc create -f /projects/cloud-native-workshop-v2m3-labs/inventory/rules/inventory-vs-fault-delay.yaml -n userXX-inventory`

Go to the **Kiali Graph** you opened earlier and you will see that the `green` traffic from *istio-ingressgateway* is delayed for requests coming from catalog service. Note that you need to check **Traffic Animation** in the *Display* select box.



If the Inventory's front page was set to correctly handle delays, we expect it to load within approximately 5 seconds. To see the web page response times, open the Developer Tools menu in IE, Chrome or Firefox (typically, key combination **Ctrl**+**Shift**+**I** or **Alt**+**Cmd**+**I**), select the `Network` tab, and reload the inventory web page.

You will see and feel that the webpage loads in about 5 seconds:



Before we will move to the next step, clean up the fault injection and set the default virtual service once again using these commands in a Terminal:

> Don't forget to replace `userXX` with your username!

```
oc delete virtualservice/inventory-fault-delay -n userXX-inventory
```

```
oc delete gateway/inventory-gateway -n userXX-inventory
```

`oc create -f /projects/cloud-native-workshop-v2m3-labs/inventory/rules/inventory-default.yaml -n userXX-inventory`

Also, close the tabs in your browser for the Inventory and Catalog services to avoid unnecessary load, and stop the endless `for` loop you started in the beginning of this lab in CodeReady by closing the Terminal window that was running it.

3. Enable Circuit Breaker

In this step, you will configure a circuit Breaker to protect the calls to `Inventory` service. If the `Inventory` service gets overloaded due to call volume, Istio will limit future calls to the service instances to allow them to recover.

Circuit breaking is a critical component of distributed systems. It's nearly always better to fail quickly and apply back pressure upstream as soon as possible. Istio enforces circuit breaking limits at the network level as opposed to having to configure and code each application independently.

Istio supports various types of conditions that would trigger a circuit break:

- **Cluster maximum connections**: The maximum number of connections that Istio will establish to all hosts in a cluster.
- **Cluster maximum pending requests**: The maximum number of requests that will be queued while waiting for a ready connection pool connection.
- **Cluster maximum requests**: The maximum number of requests that can be outstanding to all hosts in a cluster at any given time. In practice this is applicable to HTTP/2 clusters since HTTP/1.1 clusters are governed by the maximum connections circuit breaker.
- **Cluster maximum active retries**: The maximum number of retries that can be outstanding to all hosts in a cluster at any given time. In general Istio recommends aggressively circuit breaking retries so that retries for sporadic failures are allowed but the overall retry volume cannot explode and cause large scale cascading failure.

> Note that **HTTP2** uses a single connection and never queues (always multiplexes), so max connections and max pending requests are not applicable.

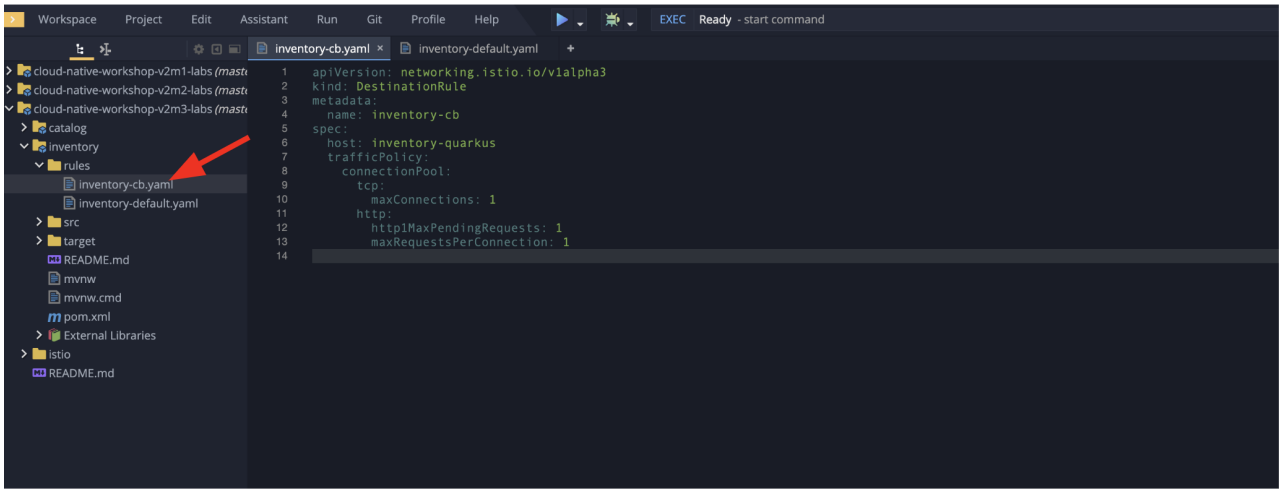
Each circuit breaking limit is configurable and tracked on a per upstream cluster and per priority basis. This allows different components of the distributed system to be tuned independently and have different limits. See the Envoy's circuit breaker for more details.

Let's add a circuit breaker to the calls to the **Inventory service**. Instead of using a *VirtualService* object, circuit breakers in Istio are defined as *DestinationRule* objects. DestinationRule defines policies that apply to traffic intended for a service after routing

has occurred. These rules specify configuration for load balancing, connection pool size from the sidecar, and outlier detection settings to detect and evict unhealthy hosts from the load balancing pool.

Open the empty **inventory-cb.yaml** file in `/projects/cloud-native-workshop-v2m3-labs/inventory/rules/` and add this code to the file to enable circuit breaking when calling the Inventory service:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: inventory-cb
spec:
  host: inventory-quarkus
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 1
      http:
        http1MaxPendingRequests: 1
        maxRequestsPerConnection: 1
```



Run the following command via CodeReady Workspaces Terminal to then create the rule:

`oc create -f /projects/cloud-native-workshop-v2m3-labs/inventory/rules/inventory-cb.yaml -n userXX-inventory`

We set the Inventory service's maximum connections to 1 and maximum pending requests to 1. Thus, if we send more than 2 requests within a short period of time to the inventory service, 1 will go through, 1 will be pending, and any additional requests will be denied until the pending request is processed. Furthermore, it will detect any hosts that return a server error (HTTP 5xx) and eject the pod out of the load balancing pool for 15 minutes. You can visit here to check the Istio spec for more details on what each configuration parameter does.

4. Overload the service

Let's use simple **curl** commands to send multiple concurrent requests to our application, and witness the circuit breaker kicking in and opening the circuit.

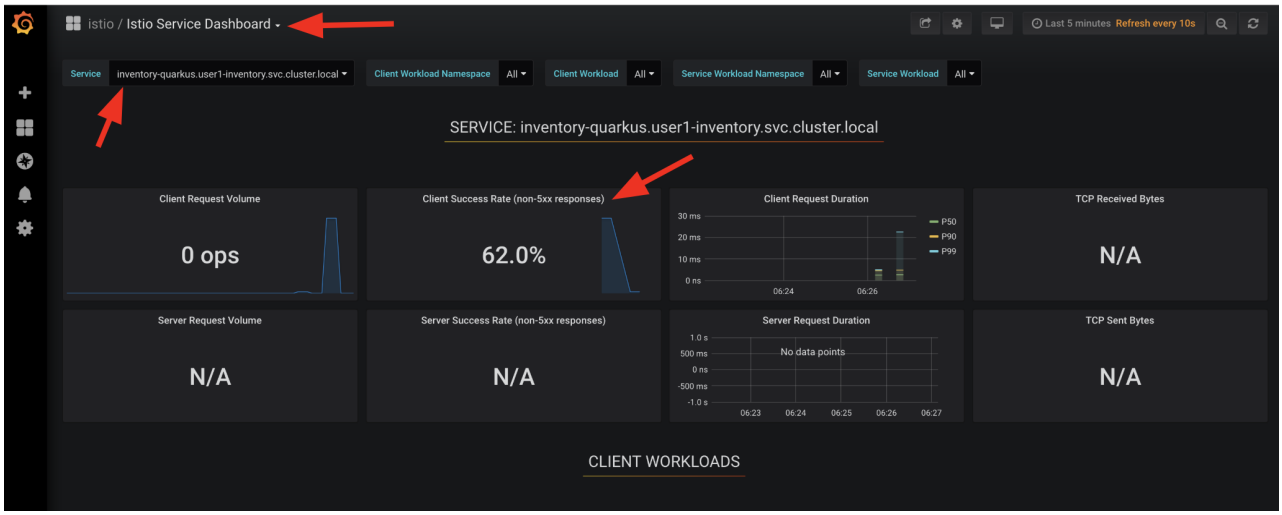
Execute this to simulate a number of users attampting to access the gateway URL simultaneously in CodeReady Workspaces Terminal.

> Replace `YOUR_INVENTORY_GATEWAY_URL` with your custom inventory URL, e.g. `http://inventory-quarkus-userXX-inventory.apps.cluster-beijing-5bc2.beijing-5bc2.open.redhat.com` .

```
for i in {1..1000} ; do
 curl 'http://YOUR_INVENTORY_GATEWAY_URL/services/inventory' >& /dev/null &
done
```

Due to the very conservative circuit breaker, many of these calls will fail with HTTP 503 (Server Unavailable). To see this, open the *Istio Service Dashboard* in the Grafana console and select `inventory-quarkus.userXX-inventory.svc.cluster.local` service:

> `NOTE` : It may take 10-20 seconds before the evidence of the circuit breaker is visible within the Grafana dashboard, due to the not-quite-realtime nature of Prometheus metrics and Grafana refresh periods and general network latency.

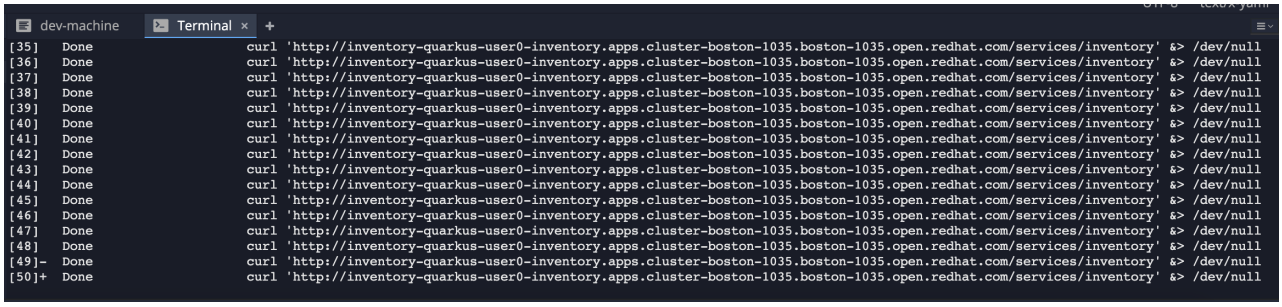


That's the circuit breaker in action, limiting the number of requests to the service. In practice your limits would be much higher.

5. Stop overloading

Before moving on, stop the traffic generator by executing the following commands in CodeReady Workspaces Terminal:

```
for i in {1..50} ; do kill %${i} ; done
```



Delete the circuit breaker of the Inventory service via the following commands. You should replace `userXX` with your namespace:

```
oc delete destinationrule/inventory-cb -n userXX-inventory
```

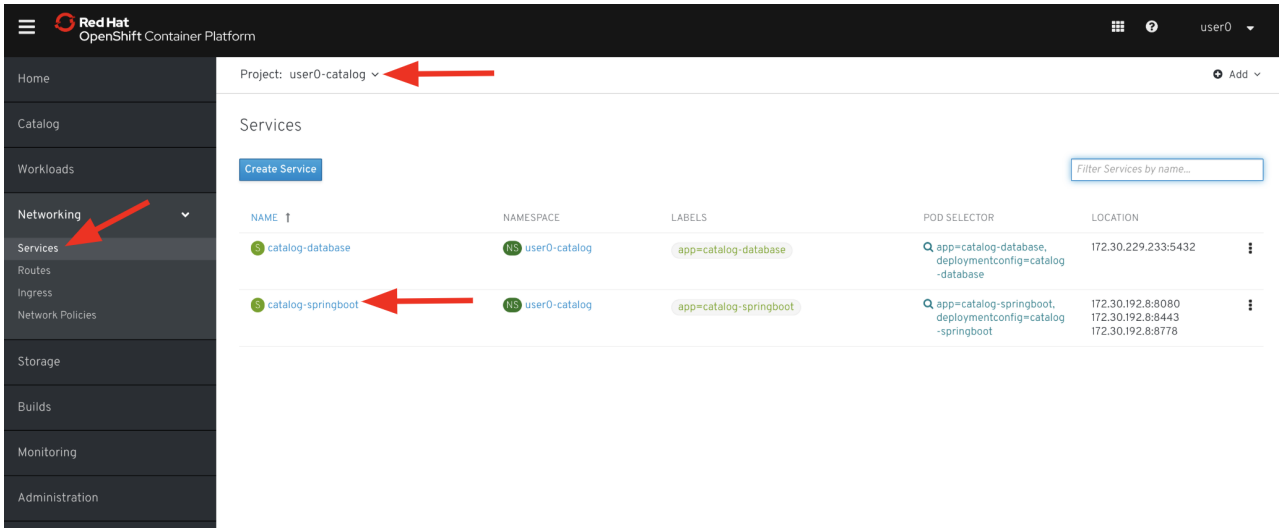
6. Enable Authentication using Single Sign-on

In this step, you will learn how to enable authenticating **catalog** microservices with Istio, JSON Web Token(JWT), and Red Hat Single Sign-On in Red Hat Runtimes.

First, let's remove the direct route to the catalog service. We want traffic to be managed by the service mesh, and not allow direct traffic. Use the following command in the CodeReady Workspaces Terminal:

```
oc delete route/catalog-springboot -n userXX-catalog
```

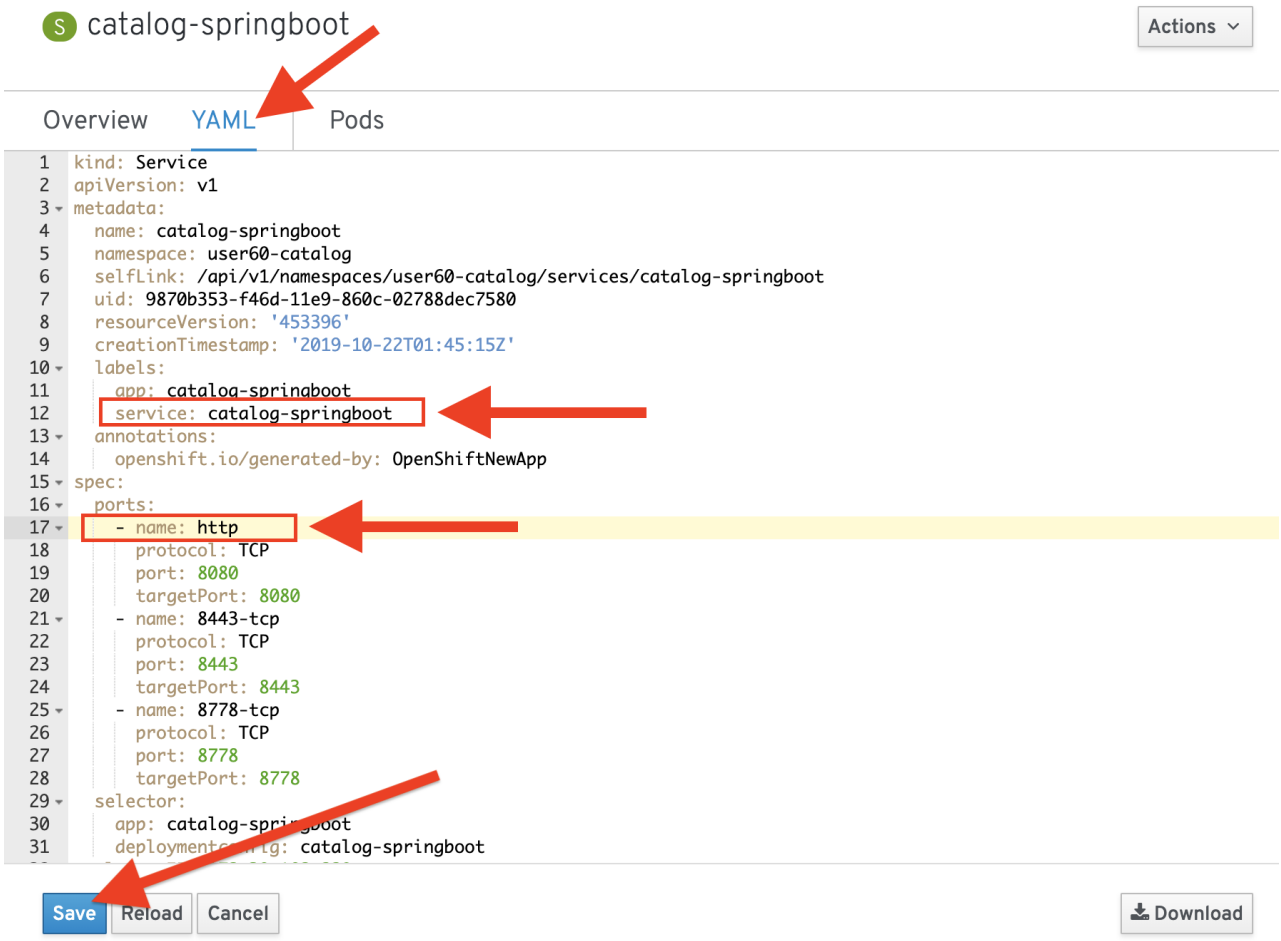
In the OpenShift web console, select the `userXX-catalog` project, then navigate to *Networking* > *Services* from the left menu, select the `catalog-springboot` service



Select the YAML tab and add the following label in the catalog service to use a **virtural service**:

```
service: catalog-springboot
```

Also, since Istio requires service names to be named with specific identifiers, change the name of the `8080-tcp` to be named `http` as shown:

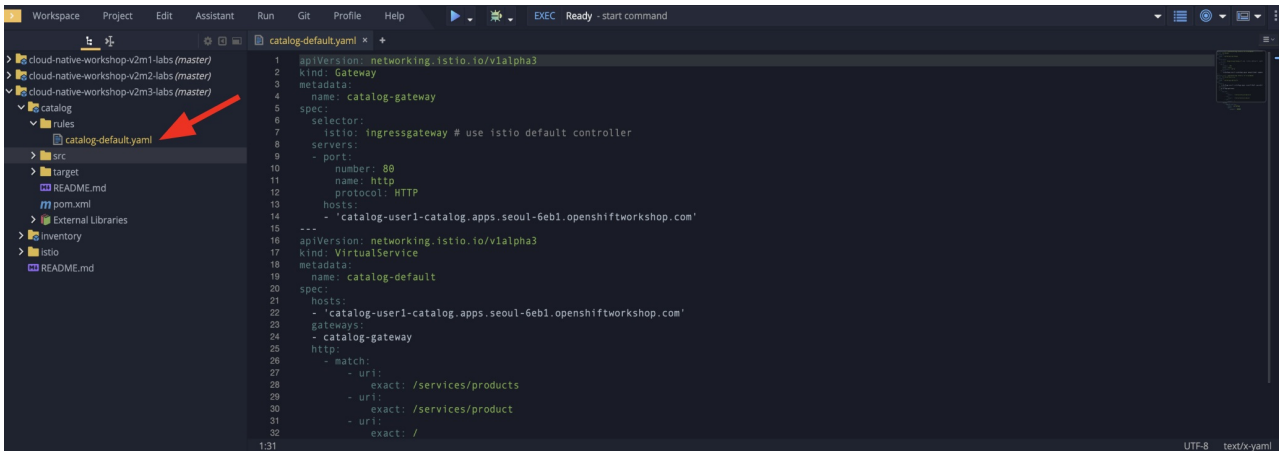


Click on **Save**.

In CodeReady, open the **catalog-default.yaml** file in `/projects/cloud-native-workshop-v2m3-labs/catalog/rules/` to make a gateway and virtual service:

> Replace all **YOUR_CATALOG_GATEWAY_URL** with the catlog route URL which will be `catalog-springboot-userXX-catalog.apps.cluster-beijing-5bc2.beijing-5bc2.open.redhat.com` but with `userXX` replaced with your username. Change the code in two places after inserting it into the `catalog-default.yaml` file:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: catalog-gateway
spec:
  selector:
    istio: ingressgateway # use istio default controller
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - 'YOUR_CATALOG_GATEWAY_URL'
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: catalog-default
spec:
  hosts:
  - 'YOUR_CATALOG_GATEWAY_URL'
  gateways:
  - catalog-gateway
  http:
    - match:
        - uri:
            exact: /services/products
        - uri:
            exact: /services/product
        - uri:
            exact: /
      route:
        - destination:
            host: catalog-springboot
            port:
              number: 8080
```

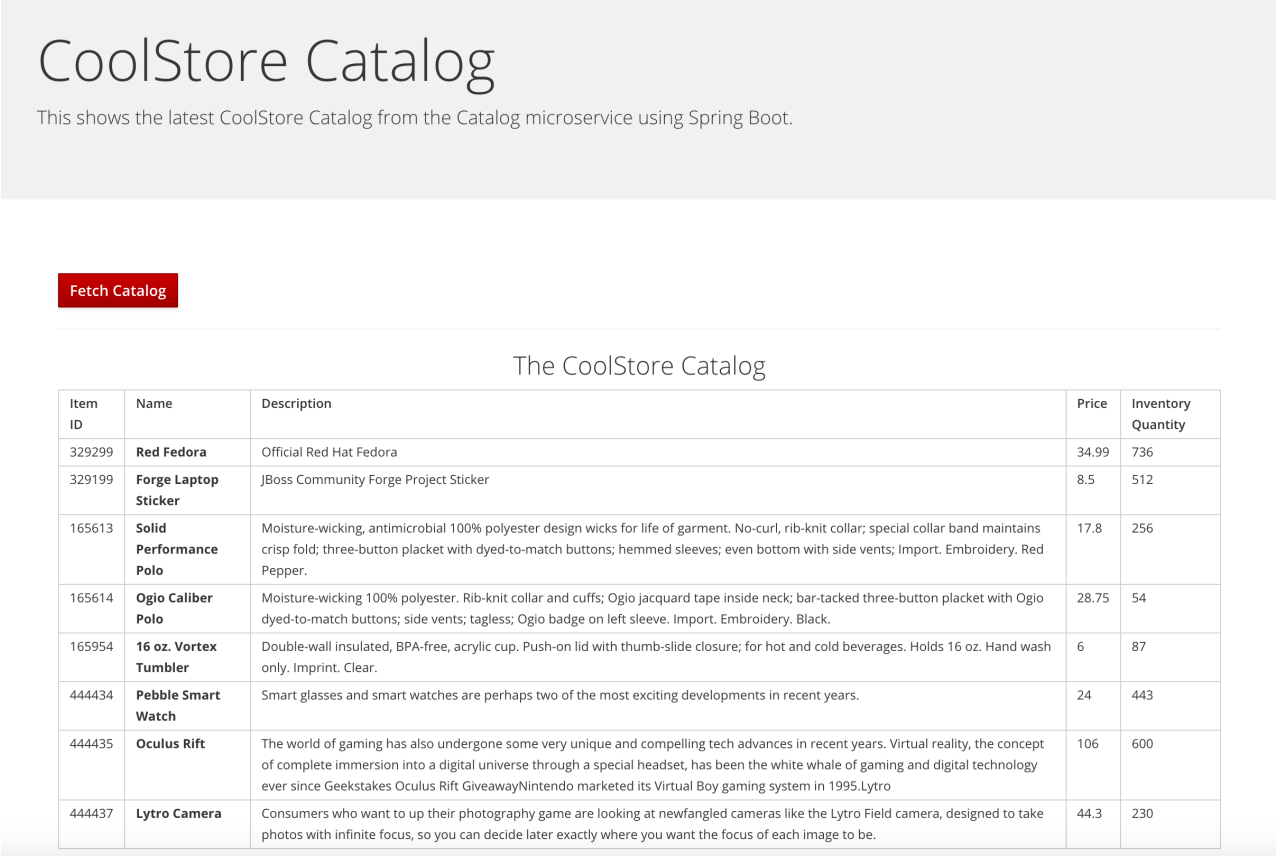


Then create this object in OpenShift by running the following command via CodeReady Workspaces Terminal:

`oc create -f /projects/cloud-native-workshop-v2m3-labs/catalog/rules/catalog-default.yaml -n userXX-catalog` (replace `userXX` with your username!)

Now, you can test if the catalog service works correctly by accessing the **YOUR_CATALOG_GATEWAY_URL** without *authentication* in your browser:

`i.e. http://catalog-springboot-userXX-catalog.apps.cluster-beijing-5bc2.beijing-5bc2.open.redhat.com`



Let's deploy **Red Hat Single Sign-On (RH-SSO)** that enables service authentication for traffic in the service mesh.

Red Hat Single Sign-On (RH-SSO) is based on the **Keycloak** project and enables you to secure your web applications by providing Web single sign-on (SSO) capabilities based on popular standards such as **SAML 2.0, OpenID Connect and OAuth 2.0**. The RH-SSO server can act as a SAML or OpenID Connect-based Identity Provider, mediating with your enterprise user directory or 3rd-party SSO provider for identity information and your applications via standards-based tokens. The major features include:

- **Authentication Server** - Acts as a standalone SAML or OpenID Connect-based Identity Provider.
- **User Federation** - Certified with LDAP servers and Microsoft Active Directory as sources for user information.
- **Identity Brokering** - Integrates with 3rd-party Identity Providers including leading social networks as identity source.

- **REST APIs and Administration GUI** - Specify user federation, role mapping, and client applications with easy-to-use Administration GUI and REST APIs.

We will deploy RH-SSO in Catalog project. Run the following commands in CodeReady Workspaces Terminal:

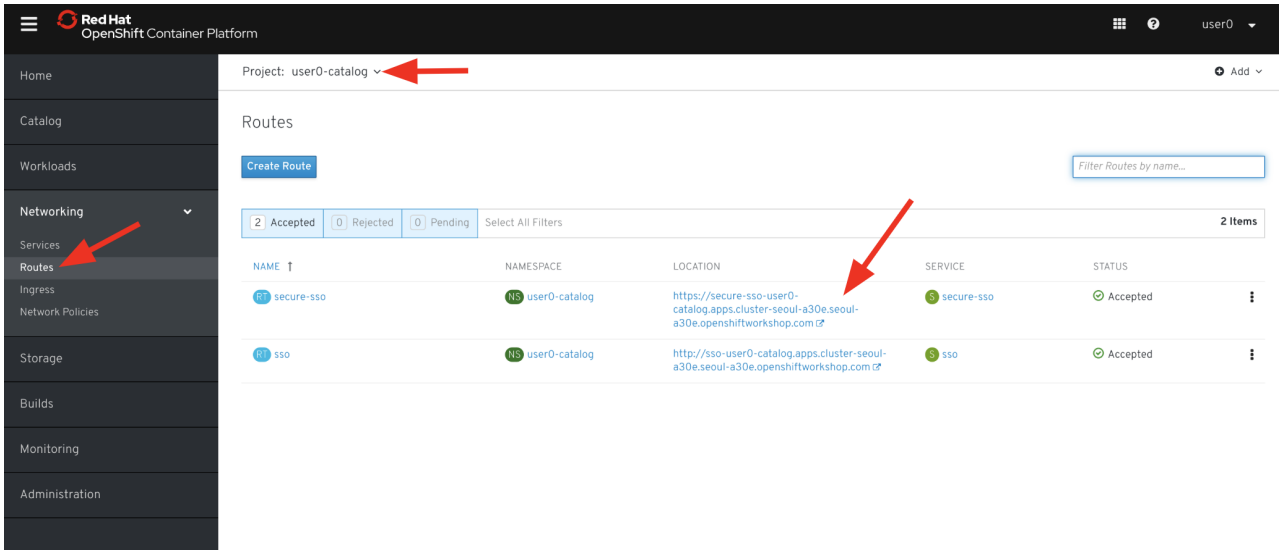
> Note: You need to replace `userXX` with your username and replace `authuserXX` below with your username plus `auth` prefix. For example, `authuser12` or `authuser2` .

```
oc -n userXX-catalog new-app ccn-sso72 \
  -p SSO_ADMIN_USERNAME=admin \
  -p SSO_ADMIN_PASSWORD=admin \
  -p SSO_REALM=istio \
  -p SSO_SERVICE_USERNAME=authuserXX \
  -p SSO_SERVICE_PASSWORD=openshift
```

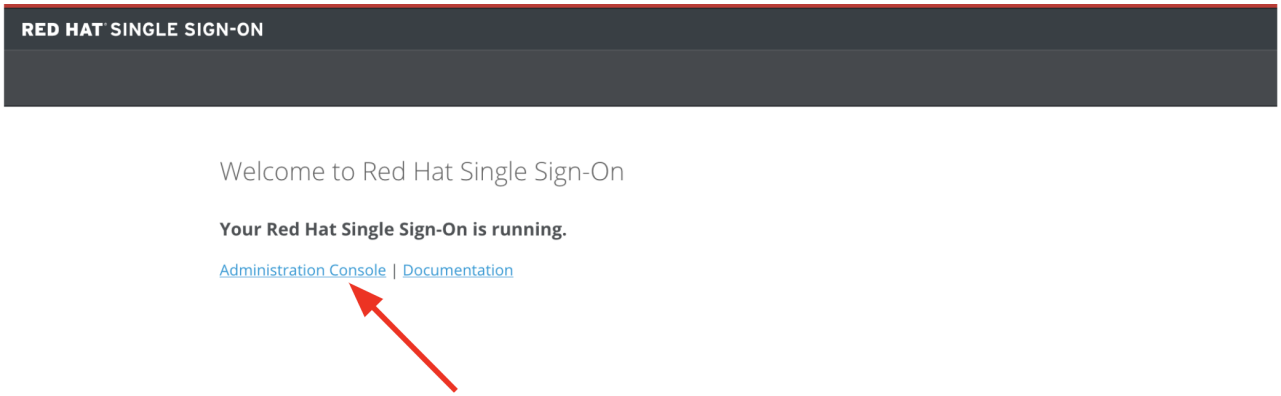
Wait for RH-SSO to be deployed using this command:

`oc rollout status -w dc/sso -n userXX-catalog` (replace `userXX` with your username)

Once this finishes (it may take a minute or two), in the OpenShift web console navigate to *Networking* > *Routes* and you will see the route URL as below (in the `userXX-catalog` project):



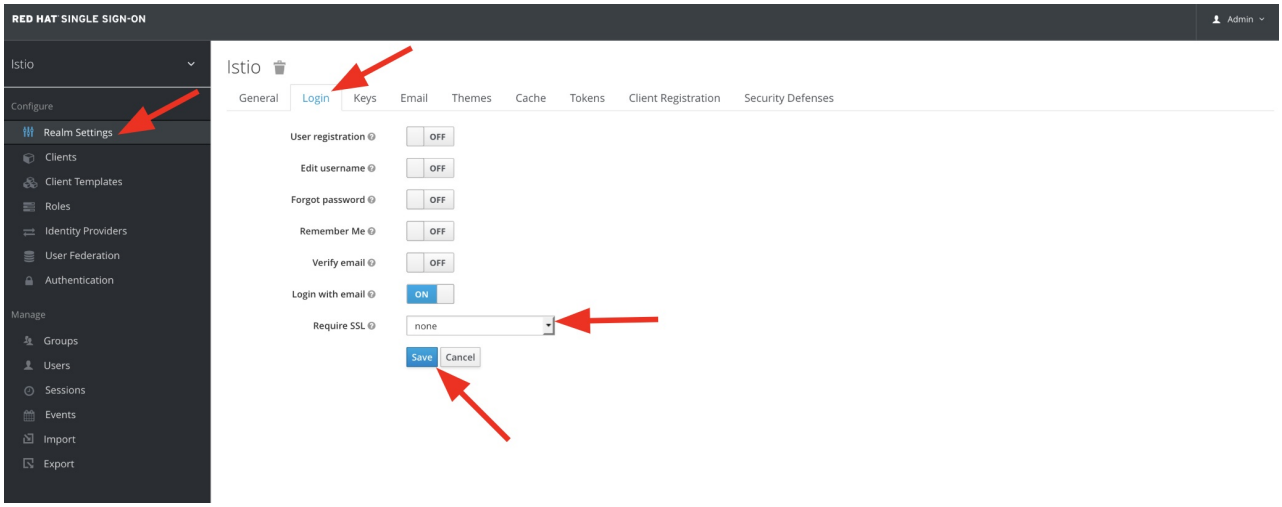
Click on **HTTPS** URL(i.e. `secure-sso-userXX-catalog.apps.cluster-beijing-5bc2.beijing-5bc2.open.redhat.com`) to access RH-SSO web console as below:



Click on *Administration Console* to configure **Istio** Ream then input the usename and password that you used earlier:

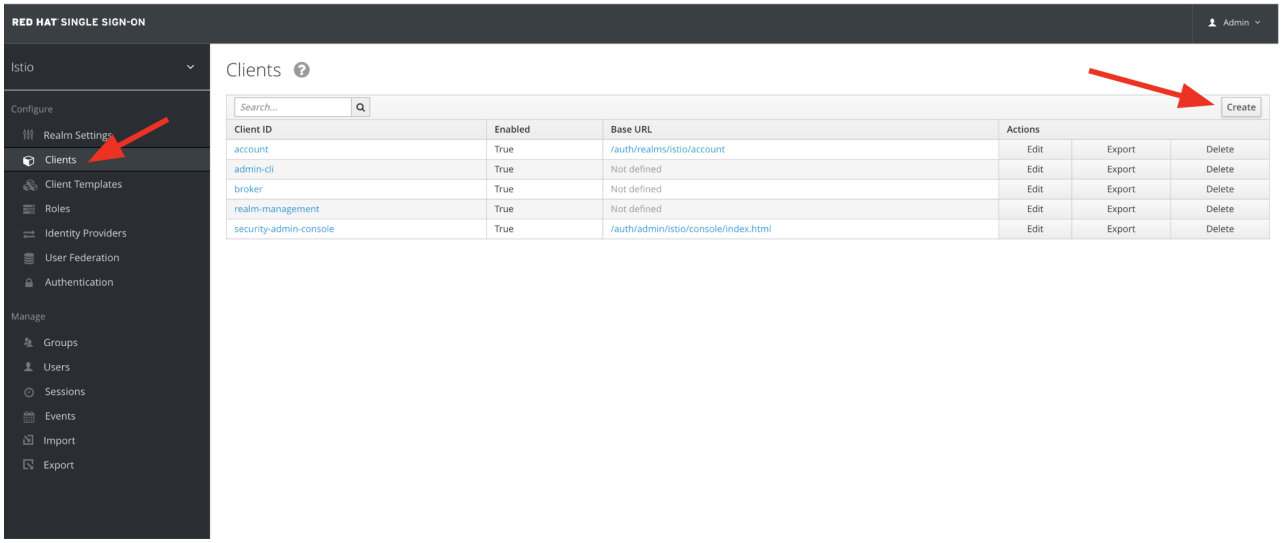
- Username or email: **admin**
- Password: **admin**

You will see general information of the *Istio Realm*. Click on **Login** tab and de-select (swich off) *Require SSL* by setting it to *none* then click on **Save**.

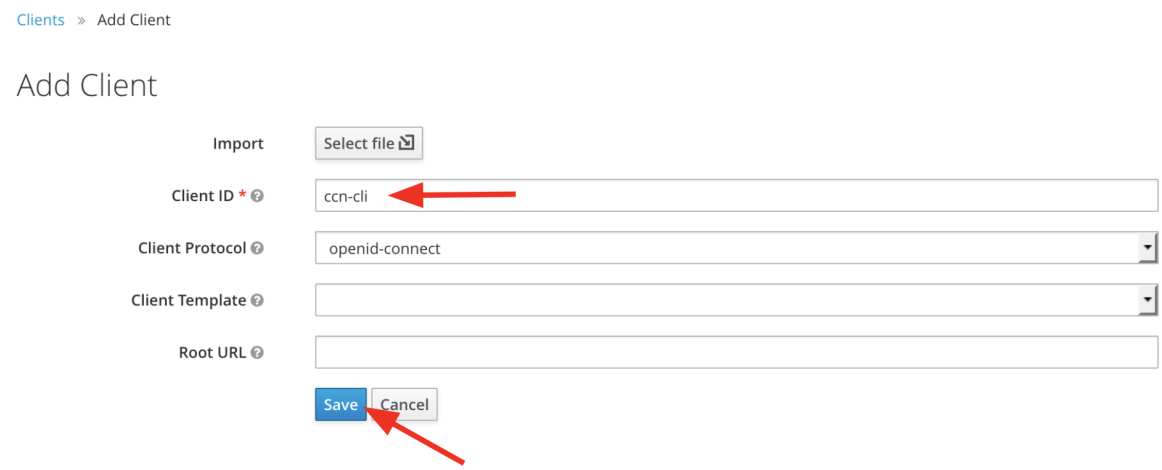


> Red Hat Single Sign-On generates a self-signed certificate the first time it runs. Please note that self-signed certificates don't work to authenticate by Istio so we will change not to use SSL for testing Istio authentication.

Next, create a new RH-SSO *client* that is for trusted browser apps and web services in our *Istio* realm. Go to **Clients** in the left menu then click on **Create**.



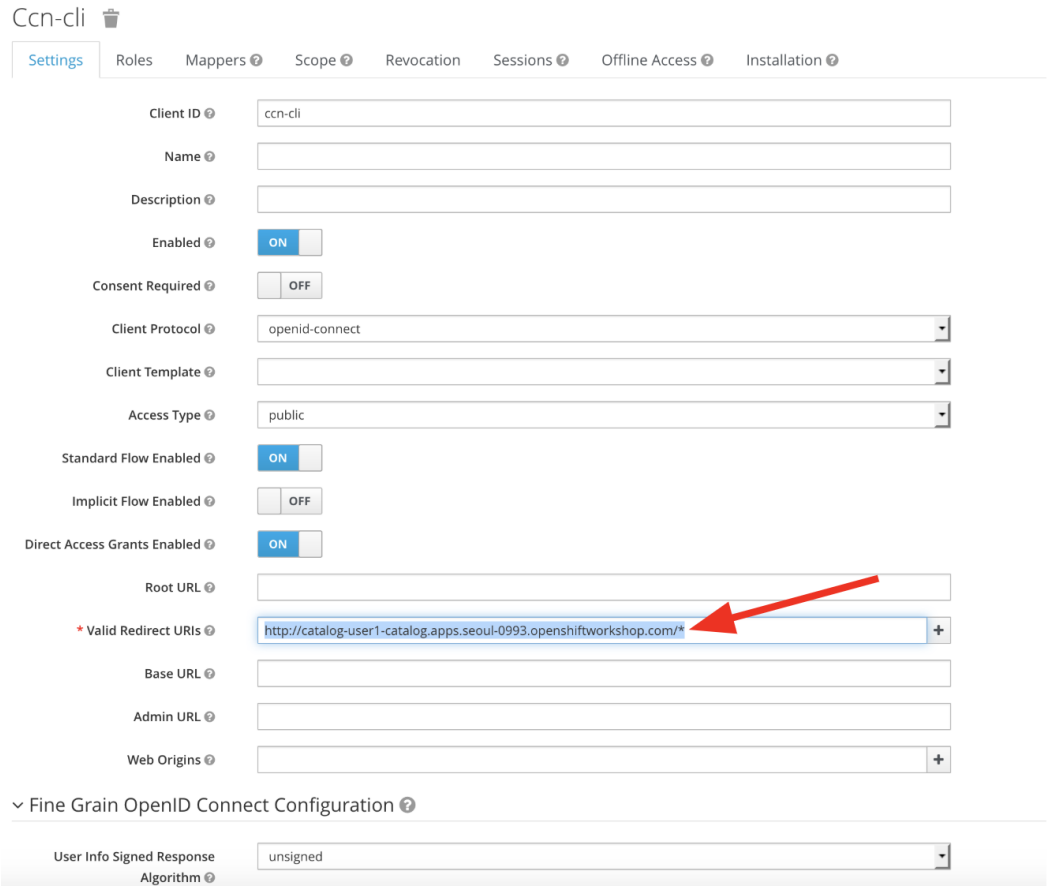
Input **ccn-cli** in *Client ID* field and click on **Save**.



On the next screen, you will see details on the **Settings** tab, the only thing you need to do is to input *Valid Redirect URIs* that can be used after successful login or logout for clients.

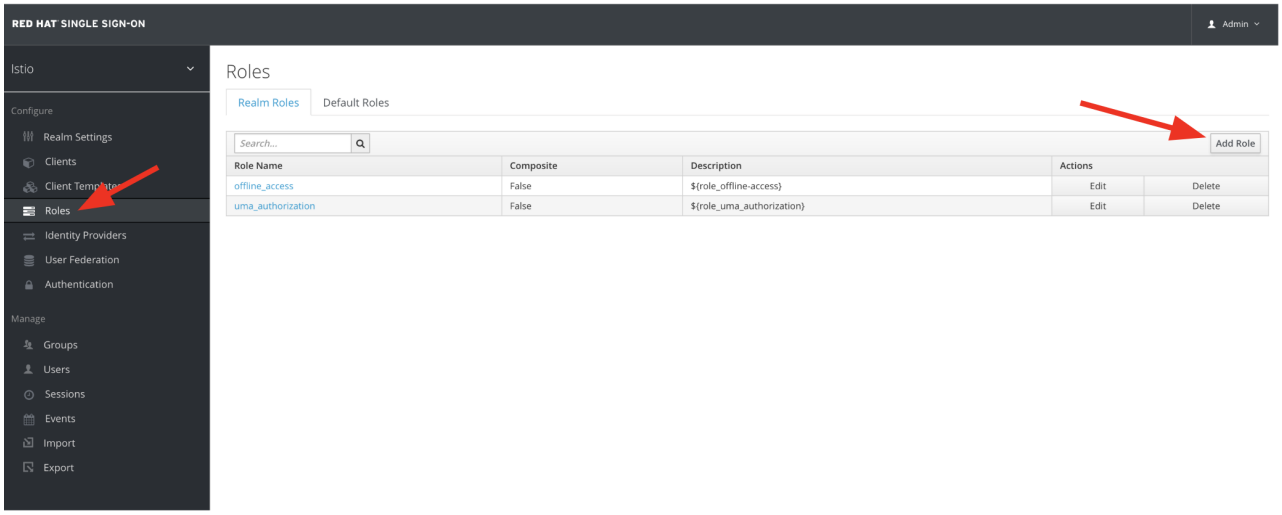
> Replace **YOUR_CATALOG_GATEWAY_URL** with your own ingress gateway URL of the catalog service and please note to add **http://** at the front as well as `/*` at the end of URL.

Valid Redirect URIs: `http://catalog-springboot-userXX-catalog.apps.cluster-beijing-5bc2.beijing-5bc2.open.redhat.com/*` (replace `userXX` with your username!)

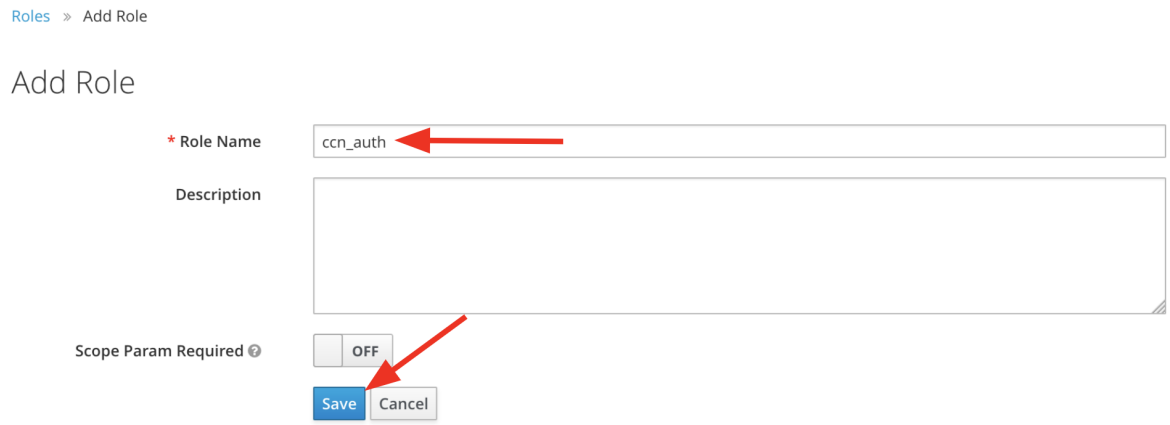


Don't forget to click **Save**!

Now, let's define a role that will be assigned to your credentials, let's create a simple role called **ccn_auth**. Go to **Roles** in the left menu then click on *Add Role*.

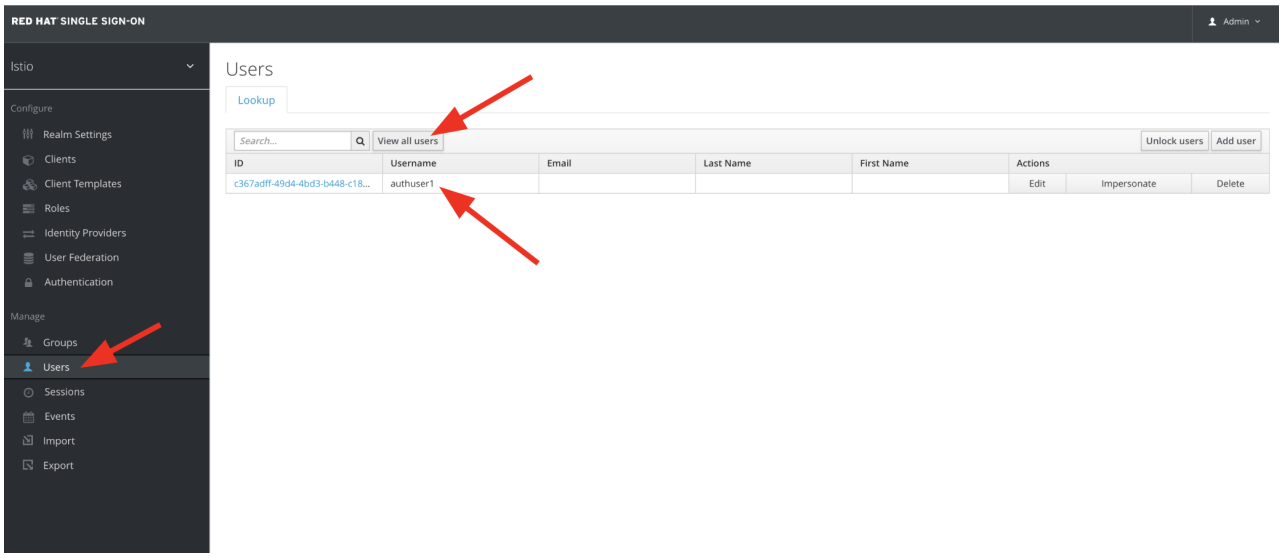


Input **ccn_auth** in *Role Name* field and click on **Save**.

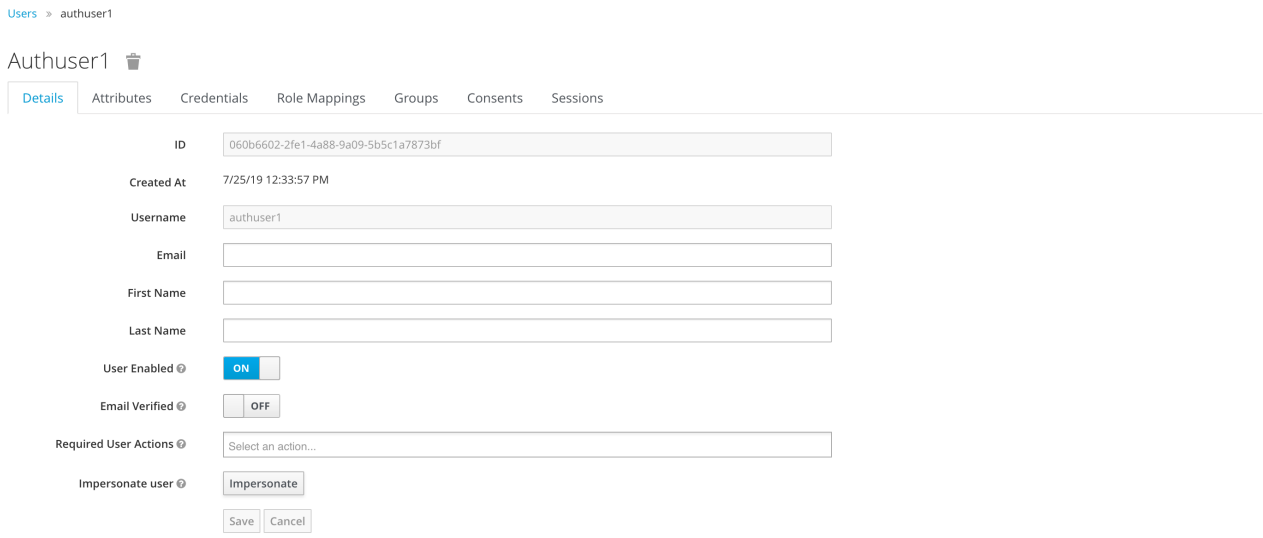


Next let's update the password policy for our *authuser*.

Go to **Users** menu on the left side menu then click on **View all users**.



If you click on the `authuserXX` ID then you will find more information such as Details, Attributes, Credentials, Role Mappings, Groups, Contents, and Sessions. You don't need to update any details in this step.

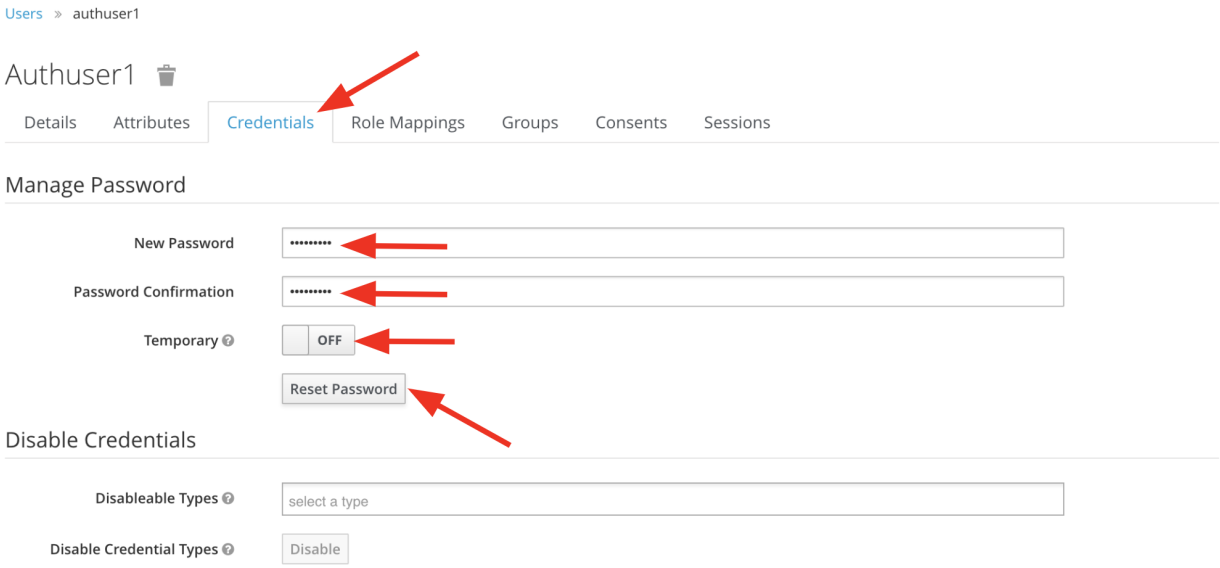


Go to **Credentials** tab and input the following variables:

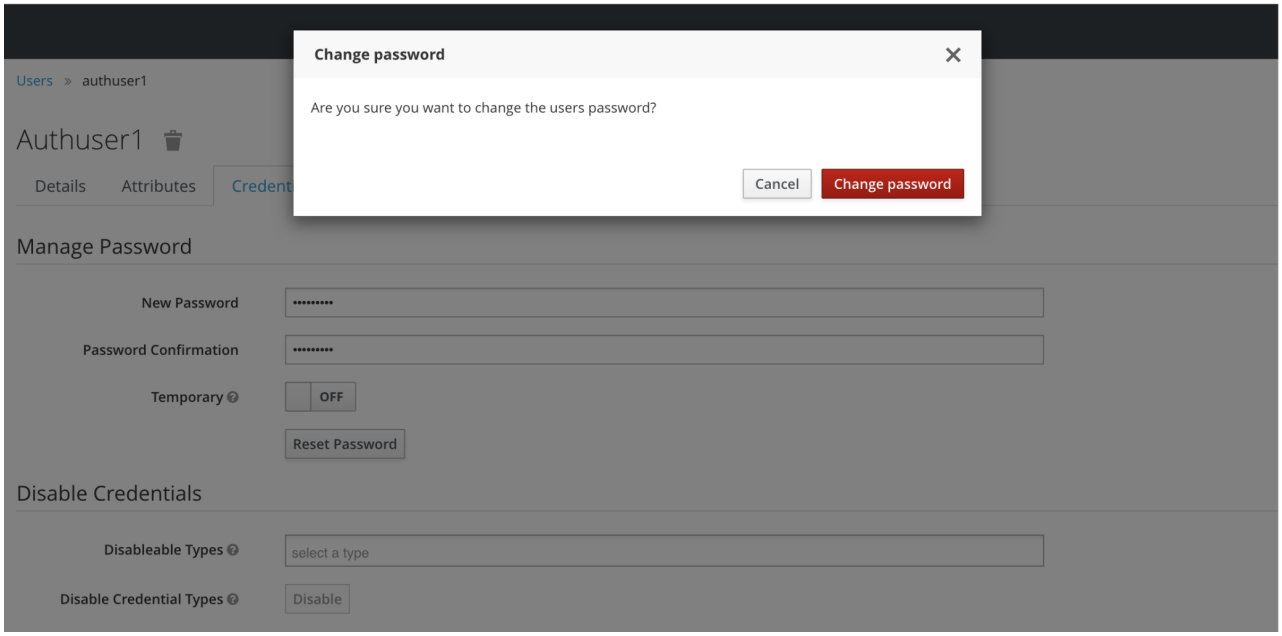
- New Password: **openshift**
- Password Confirmation: **openshift**
- Temporary: **OFF**

Make sure to turn off the "Temporary" flag unless you want the authuserXX to have to change his password the first time they authenticate.

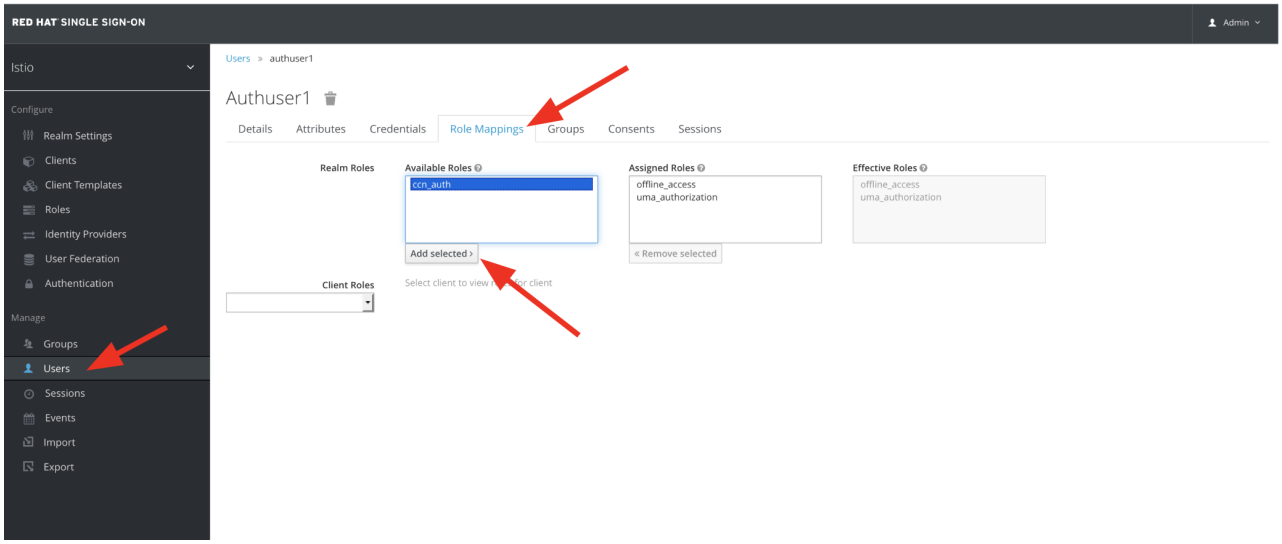
Click on **Reset Password**.



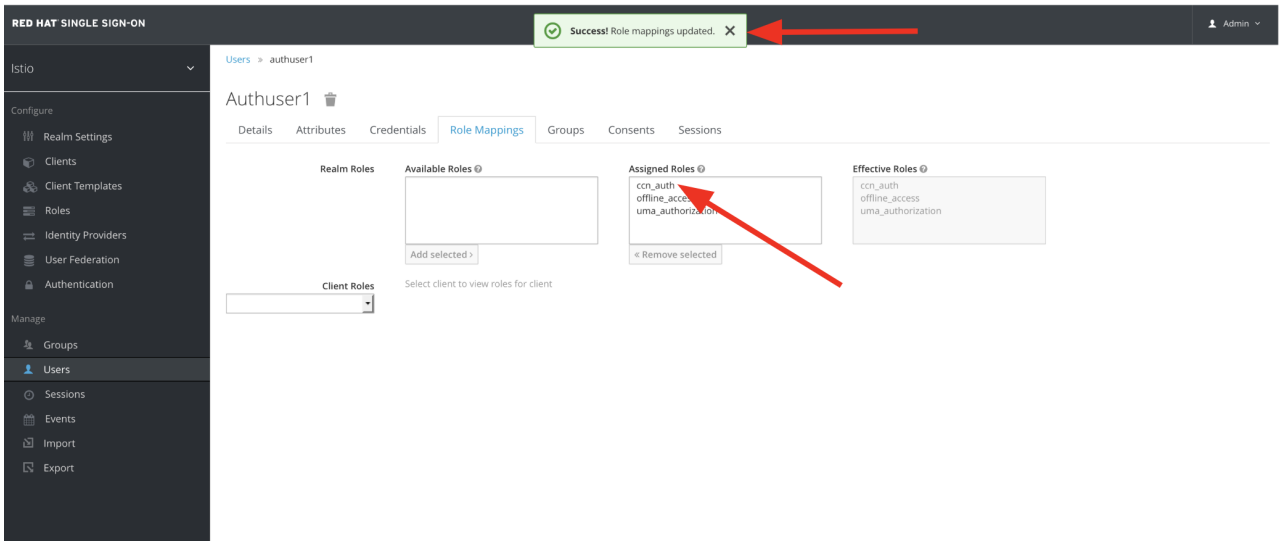
Then click on **Change password** in the popup window.



Now proceed to the **Role Mappings** tab and assign the role **ccn_auth** via clicking on *Add selected* >.



You will confirm the ccn_auth role in *Assigned Roles* box.



Well done, you have enabled RH-SSO to with a custom realm, user and role!

Turning to back to Istio, let's create a user-facing authentication policy using JSON Web Tokens (JWTs). The format is defined in RFC 7519. You can find more details how OAuth 2.0 and OIDC 1.0 work in the overall authentication flow.

In CodeReady, open the blank **ccn-auth-config.yml** file in `/projects/cloud-native-workshop-v2m3-labs/catalog/rules/` to create an authentication policy:

> Replace all **YOUR_SSO_HTTP_ROUTE_URL** with your own HTTP route url of SSO container that you created earlier and also replace **userXX** with your username.

You can also get the route url via executing the following commands in CodeReady Workspaces Terminal:

```
oc get route -n userXX-catalog secure-sso --template '\n'
```

Use this value to replace `YOUR_SSO_HTTP_ROUTE_URL`. You will also use this later!

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name: auth-policy
  namespace: userXX-catalog
spec:
  targets:
  - name: catalog-springboot
  origins:
  - jwt:
      issuer: http://YOUR_SSO_HTTP_ROUTE_URL/auth/realms/istio
      jwks_uri: http://YOUR_SSO_HTTP_ROUTE_URL/auth/realms/istio/protocol/openid-connect/certs
  principalBinding: USE_ORIGIN
```

The following fields are used above to create a Policy in Istio and are described here:

- **issuer** - Identifies the issuer that issued the JWT. See issuer usually a URL or an email address.
- **jwksUri** - URL of the provider's public key set to validate signature of the JWT.
- **audiences** - The list of JWT audiences. that are allowed to access. A JWT containing any of these audiences will be accepted.

Then execute the following oc command in CodeReady Workspaces Terminal to create this object:

`oc create -f /projects/cloud-native-workshop-v2m3-labs/catalog/rules/ccn-auth-config.yaml -n userXX-catalog` (replace `userXX` with your username!)

Now you can't access the catalog service without authentication of RH-SSO. You confirm it using a curl command (replacing `userXX` with your username) in CodeReady Workspaces Terminal:

`curl -i http://YOUR_CATALOG_GATEWAY_URL/services/products ; echo`

You should get and `HTTP 401 Unauthorized` and `Origin authentication failed.` messages.

The expected response is here because the user has not been identified with a valid JWT token in RH-SSO. It normally takes `5 ~ 10 seconds` to initialize the authentication policy in Istio Mixer. After this things go quickly as policies are cached for some period of time.

In order to generate a correct token, run next `curl` request in CodeReady Workspaces Terminal. This command will store the output Authorization token from RH-SSO in an environment variable called **TOKEN**.

> Replace `YOUR_SSO_HTTP_ROUTE_URL` with your own HTTP route url of SSO container that you created earlier.

> Also replace `authuserXX` with your authentication username, e.g. `authuser34`

```
export TOKEN=$( curl -X POST 'http://YOUR_SSO_HTTP_ROUTE_URL/auth/realms/istio/protocol/openid-connect/token' \
 -H "Content-Type: application/x-www-form-urlencoded" \
 -d "username=authuserXX" \
 -d 'password=openshift' \
 -d 'grant_type=password' \
 -d 'client_id=ccn-cli' | jq -r '.access_token')
```

Ensure you have a valid token:

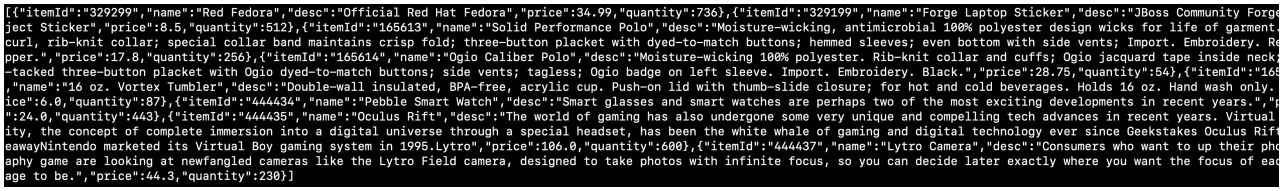
`echo; echo $TOKEN; echo`

Once you have generated the token, re-run the curl command below with the token in CodeReady Workspaces Terminal:

`curl -H "Authorization: Bearer $TOKEN" http://YOUR_CATALOG_GATEWAY_URL/services/products ; echo`

You will see the following expected output:

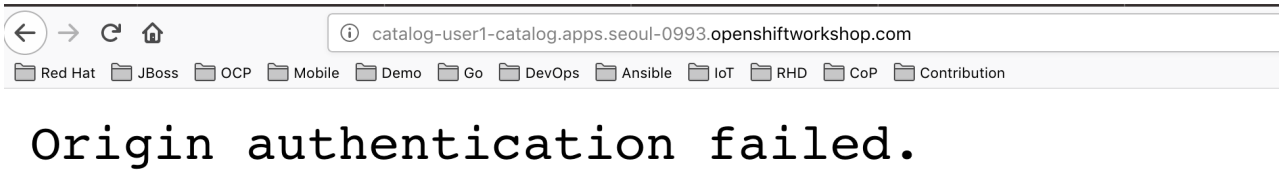
[{"itemId":"329299","name":"Red Fedora","desc":"Official Red Hat
Fedora","price":34.99,"quantity":736},{"itemId":"329199","name":
"Forge Laptop Sticker","desc":"JBoss Community Forge Project Sticker","price":8.5,"quantity":512},
{"itemId":"165613","name":"Solid
Performance Polo","desc":"Moisture-wicking, antimicrobial 100% polyester design wicks for life of garment. No-curl, rib-knit collar;
special collar band maintains crisp fold; three-button placket with dyed-to-match buttons; hemmed sleeves; even bottom with side vents;
Import. Embroidery. Red Pepper.","price":17.8,"quantity":256},{"itemId":"165614","name":"Ogio Caliber Polo","desc":"Moisture-wicking 100%
polyester. Rib-knit collar and cuffs; Ogio jacquard tape inside neck; bar-tacked three-button placket with Ogio dyed-to-match buttons;
...


[{"itemId":"329299","name":"Red Fedora","desc":"Official Red Hat Fedora","price":34.99,"quantity":736},{"itemId":"329199","name":"Forge Laptop Sticker","desc":"JBoss Community Forge
ject Sticker","price":8.5,"quantity":512},{"itemId":"165613","name":"Solid Performance Polo","desc":"Moisture-wicking, antimicrobial 100% polyester design wicks for life of garment.
curl, rib-knit collar; special collar band maintains crisp fold; three-button placket with dyed-to-match buttons; hemmed sleeves; even bottom with side vents; Import. Embroidery. Re
pper.","price":17.8,"quantity":256},{"itemId":"165614","name":"Ogio Caliber Polo","desc":"Moisture-wicking 100% polyester. Rib-knit collar and cuffs; Ogio jacquard tape inside neck;
-tacked three-button placket with Ogio dyed-to-match buttons; side vents; tagless; Ogio badge on left sleeve. Import. Embroidery. Black.","price":28.75,"quantity":54},{"itemId":"166
,"name":"16 oz. Vortex Tumbler","desc":"Double-wall insulated, BPA-free, acrylic cup. Push-on lid with thumb-slide closure; for hot and cold beverages. Holds 16 oz. Hand wash only.
ice":6.0,"quantity":87},{"itemId":"444434","name":"Pebble Smart Watch","desc":"Smart glasses and smart watches are perhaps two of the most exciting developments in recent years.","p
":24.0,"quantity":443},{"itemId":"444435","name":"Oculus Rift","desc":"The world of gaming has also undergone some very unique and compelling tech advances in recent years. Virtual
ity, the concept of complete immersion into a digital universe through a special headset, has been the white whale of gaming and digital technology ever since Geekstakes Oculus Rift
eawayNintendo marketed its Virtual Boy gaming system in 1995.Lytro","price":106.0,"quantity":600},{"itemId":"444437","name":"Lytro Camera","desc":"Consumers who want to up their pho
aphy game are looking at newfangled cameras like the Lytro Field camera, designed to take photos with infinite focus, so you can decide later exactly where you want the focus of eac
age to be.","price":44.3,"quantity":230}]

Congratulations! You've integrated RH-SSO with Istio to protect service mesh traffic to the catalog service, without having to change the application at all. Let's do it again with Spring Boot!

7. Securing Spring Boot with Red Hat Single Sing-On

Unfortunately, the catalog service still doesn't work when you access via the web page because the application has no authentication configuration yet:



Let's integrate RH-SSO authentication to the presentation layer of the catalog service. First, clean up all authentication configuration that we have tested in the previous steps. Run the following script to clean up:

`/projects/cloud-native-workshop-v2m3-labs/istio/scripts/cleanup.sh userXX` (replace `userXX` with your username!)

Next, open the **application-default.properties** in `/projects/cloud-native-workshop-v2m3-labs/catalog/src/main/resources/` and add the following settings at the bottom of the file:

Replace **YOUR_SSO_HTTP_ROUTE_URL/**

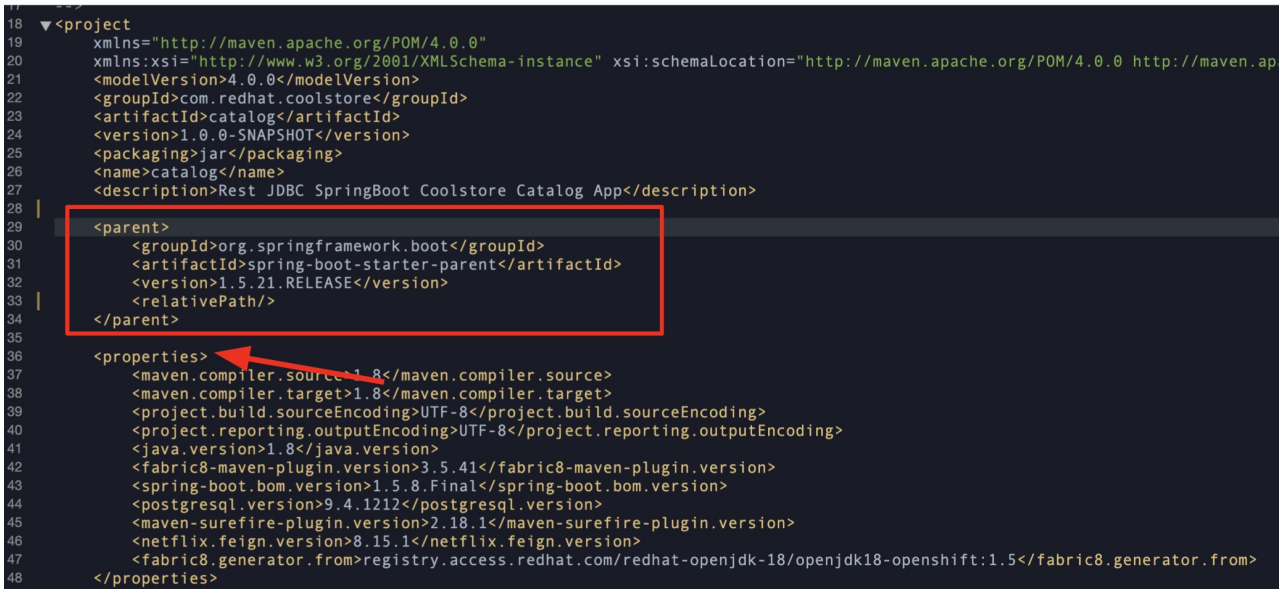
```
#TODO: Set RH-SSO authentication
keycloak.auth-server-url=http://YOUR_SSO_HTTP_ROUTE_URL/auth
keycloak.realm=istio
keycloak.resource=ccn-cli
keycloak.public-client=true

keycloak.security-constraints[0].authRoles[0]=ccn_auth
keycloak.security-constraints[0].securityCollections[0].patterns[0]=/*
```

> Also make sure to update `inventory.ribbon.listOfServers=inventory-quarkus.userXX-inventory.svc.cluster.local:8080` by replacing `userXX` with your user id

Let's update **pom.xml** in `/projects/cloud-native-workshop-v2m3-labs/catalog/` to add the needed keycloak dependency to our app:.

Add *spring-boot-starter-parent* artifact Id before *properties* element:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.21.RELEASE</version>
  <relativePath/>
</parent>
```



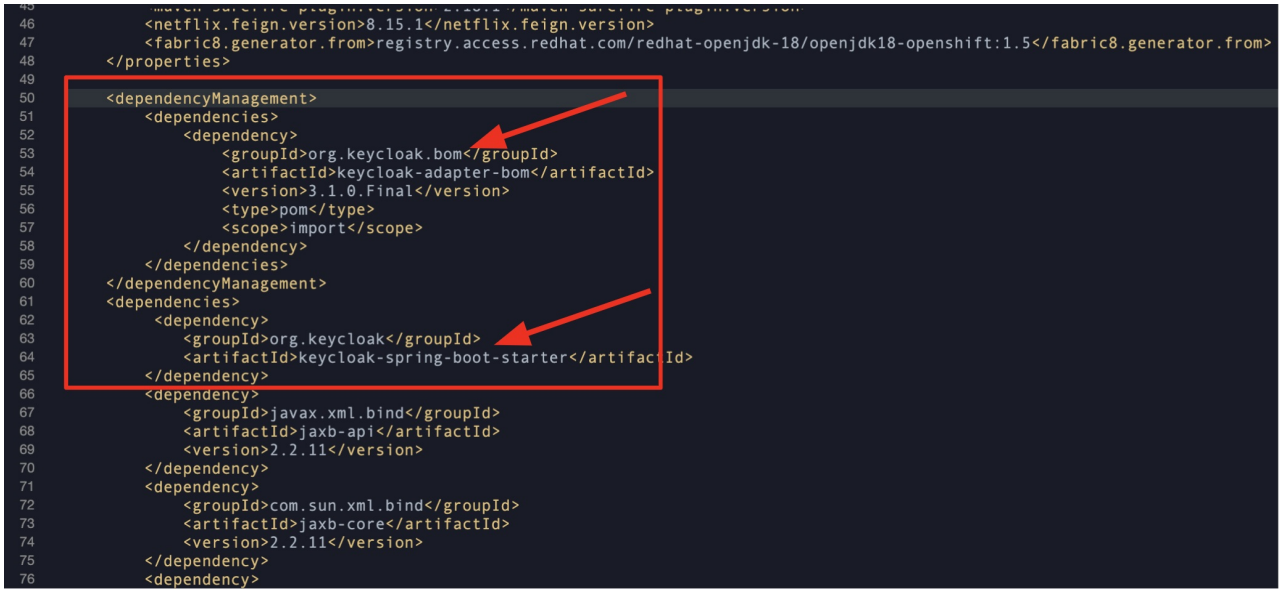
Replace **me.snowdrop** dependencyManagement and **spring-boot-starter** dependency with *keycloak* dependency.

From:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>me.snowdrop</groupId>
      <artifactId>spring-boot-bom</artifactId>
      <version>${spring-boot.bom.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>
```

To:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.keycloak.bom</groupId>
      <artifactId>keycloak-adapter-bom</artifactId>
      <version>3.1.0.Final</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
<dependencies>
  <dependency>
    <groupId>org.keycloak</groupId>
    <artifactId>keycloak-spring-boot-starter</artifactId>
  </dependency>
```



Let's re-deploy the catalog service to OpenShift by running the following maven command in CodeReady Workspaces Terminal:

```
cd /projects/cloud-native-workshop-v2m3-labs/catalog
```

```
mvn clean package spring-boot:repackage -DskipTests
```

`oc -n userXX-catalog start-build catalog-springboot --from-file=target/catalog-1.0.0-SNAPSHOT.jar --follow` (replace `userXX` with your username)

Wait for the catalog pod to restart:

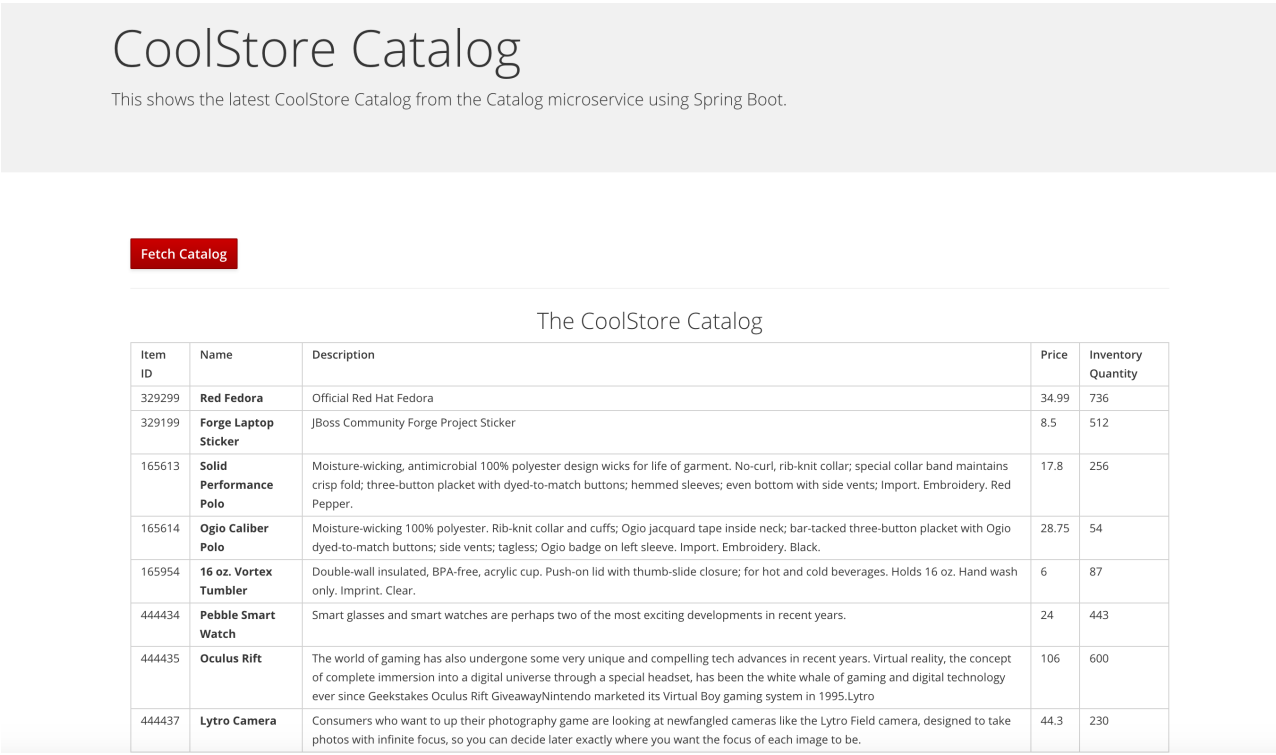
`oc rollout status -w dc/catalog-springboot -n userXX-catalog` (replace `userXX` with your username)

After the catalog pod is started, access the *catalog gateway* via a new web brower then you will redirect to the login page of **RH-SSO**.

Input the following credential that we created it in RH-SSO administration page eariler.

- Username or email: **authuserXX** (replace with your auth user, e.g. `authuser34`)
- Password: **openshift**

Finally, you can access the catalog service as below:



Summary

In this scenario you used Istio to implement many of the features needed in modern, distributed applications.

Istio provides an easy way to create a network of deployed services with load balancing, service-to-service authentication, monitoring, and more without requiring any changes in service code. You add Istio support to services by deploying a special sidecar proxy throughout your environment that intercepts all network communication between microservices, configured and managed using Istio's control plane functionality.

Technologies like containers and container orchestration platforms like OpenShift solve the deployment of our distributed applications quite well, but are still catching up to addressing the service communication necessary to fully take advantage of distributed microservice applications. With Istio you can solve many of these issues outside of your business logic, freeing you as a developer from concerns that belong in the infrastructure. **Congratulations!**