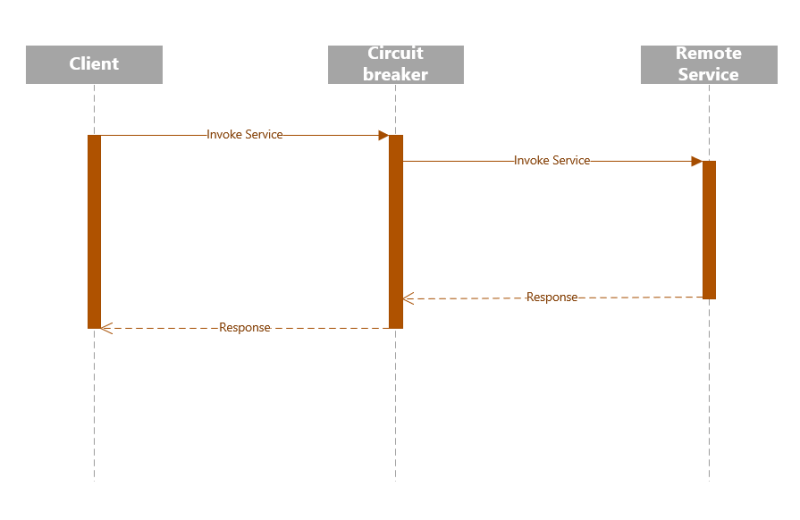
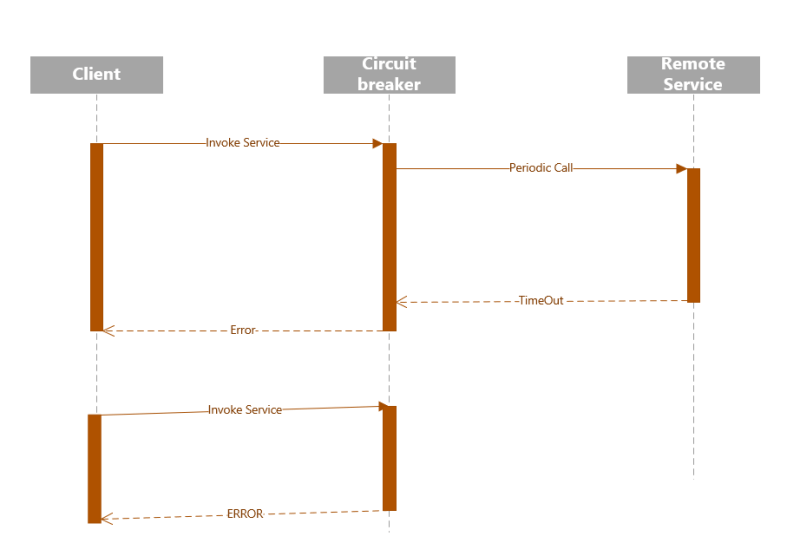
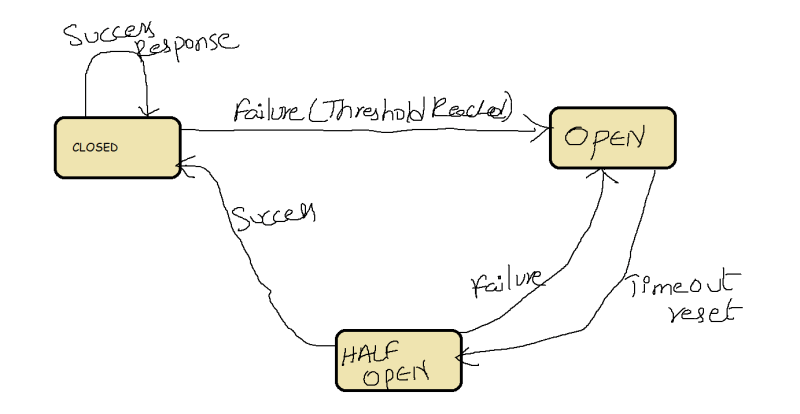
**Circuit Breaker for Microservices**

* As with an electricity circuit breaker (mcb), when the service encounters number of failures, the circuit breaker trips down for a particular duration. In that period, all the attempts made to invoke the failing service will fail immediately
* Once the configured duration is over, the circuit break will allow certain number of requests to pass through and monitor whether the requests succeed. If there are failures then circuit breaker continues to be tripped for a given period
* Circuit breaker has the following states

Closed: The circuit breaker being in a CLOSED state means everything is working fine and all calls pass through remote services



Open: Once the number of timeouts reaches a predetermined threshold in the circuit break, it trips the circuit breaker to the OPEN state. In the OPEN state, the circuit breaker returns an error for all call to the service without making calls to the remote service 

Half Open: After a certain duration, the circuit switches to a HALF-OPEN state to test if the underlying problem still exists. The circuit breaker uses a mechanism to make a trail call to the remote service periodically to check if it has recovered. If the call to the Remote Service failed, then circuit breaker remains in the OPEN state. IF the call returns success then the circuit breaker switches to the CLOSED state. The circuit breaker then returns all external calles to the service with an error during HALF-OPEN state 

**Do I really need a service mesh?**

* A service mesh provides a consistent way to connect, secure and observe microservices
* If you are deploying your first or second microservice, you donot have to use service mesh but rather focus on k8s
* If you have an existing application architecture that provides the observability, security and resilience you need, then you are already in good place
* The exact point at which service mesh clearly outweigh cost varies from organization to organization.
* Generally, teams often realize they need a consistent approach once they have five or six microservices, many users push to a dozen or microservices, before they notice increasing cost of utility code and the increasing complexity of differences across the applications. Best thing observed is try using a service when your application has more or less half a dozen microservices.
* Here are some states that may reduce or eliminate the urgency to use a service
  + All of your microservices are written in one language ("monoglot") by developers in your organization, building from common framework
  + You have partially or totally monolithic architecture where application logic is built into one or two containers instead of several
  + You use application protocols that not served by existing service mesh (so usually NOT HTTP, HTTP2, gRPC)

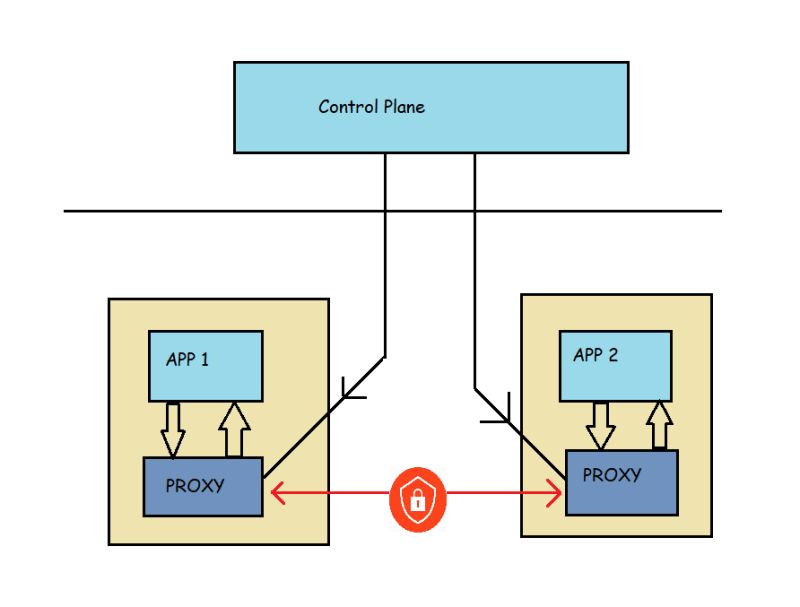
Do we really need service mesh? Yes, if we are managing more than 6 microservice.

Example of service mesh? Istio and consul

**What is Service Mesh Anyway?**

* A service mesh is a programmable framework that allows you to observe, secure and connect microservices.
* It doesn’t establish connectivity b/w microservices, but instead has policies and controls that are applied on top of existing network to govern how microservice’s interact.
* A service mesh ultimately, shifts implementation responsibilities out of the application and moves them to the network. This is accomplished by injecting behaviour and controls with int application that are applied to the network
* This is how you can accomplish things such as metrics collection, communication tracing and secure communication without changing applications themselves

**How will Service Mesh Work?**

* Basic workflow by injecting a sidecar container 
* Following are the list of behaviours commonly found in a service mesh implementation
  + Traffic shaping with dynamic routing controls b/w services
  + Resiliency support for service communications such as circuit breakers, timeouts and retries
  + Observability of traffic b/w services
  + Tracing of communication flows
  + Secure communication b/w services

**The Service Mesh Ecosystem**

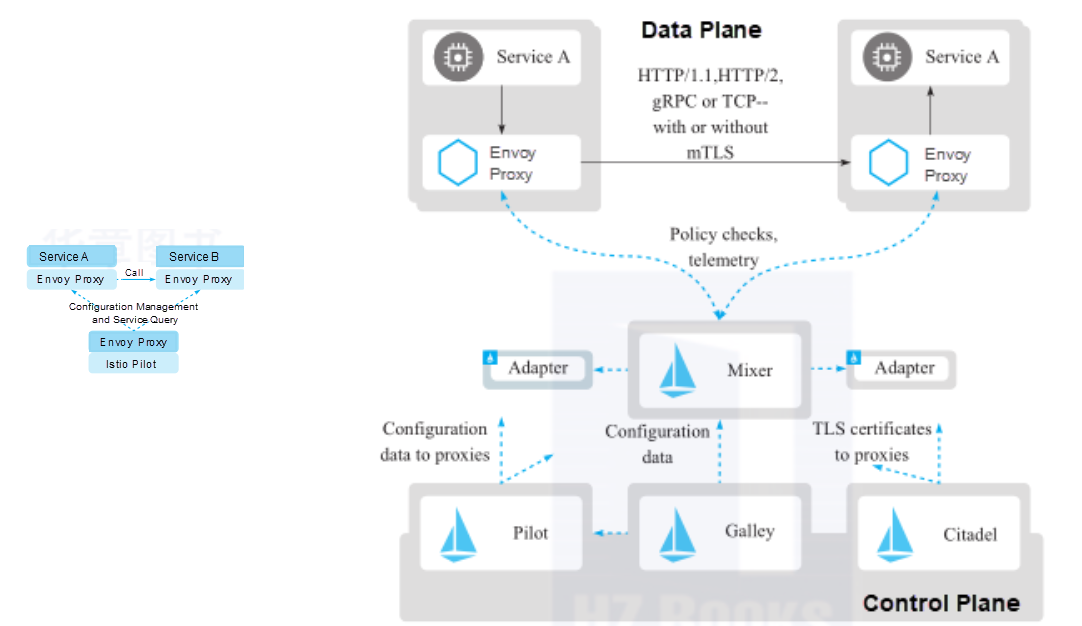
* Envoy:
  + This Envoy proxy is an open-source project create at Lyft.
  + This is an edge and service proxy that was custom built with the complexities and challenges of cloud native applications.
  + While Envoy itself doesn’t constitute a service mesh, it is a key component of service mesh eco system
* Istio:
  + This is an opensource project co-founded by IBM, Google and Lyft in 2017.
  + Istio makes it possible to connect, secure and observe your microservices
  + Istio itself builds upon many other open-source projects such as
    - Envoy
    - Kubernetes
    - Jaeger
    - Prometheus
  + Istio control plane extends the k8s API Server and utilizes Envoy proxy for client side proxies
  + Istio supports mutual TLS (mTLS) authentication communication b/w services, traffic shifting, mesh gateways, monitoring and metrics with Prometheus & Grafana, As well as custom policy injection
* Consul Connect:
  + This is service mesh developed by HashiCorp
  + This extends existing product offering Consul, which has service discovery as primary feature & built in features such as key value store, health checking & service segmentation (for secure TLS communication b/w services)
  + Consul Connect is an open source project with Hashicorp as predominant contributor
  + Consul Connect uses Envoy as sidecar proxy and Consul Server and control plane for programming side cards
  + Consul Connect includes secure mTLS support b/w microservices and observability with Prometheus and Grafana Projects.
* Linkerd:
  + The Linkerd service mesh project is an opensource as well as CNCF project with focus on providing an ultra-lightweight mesh implementation.
  + Linkerd has key capabilities of a service mesh including observability using Prometheus and Grafana, secure mTLS communication and support for traffic shift.
  + The client-side proxy used with Linkerd was developed specifically within the Linkerd project & was written in Rust.
* App Mesh: This is a cloud service hosted by AWS to provide service mesh with application-level networking support for compute services within AWS such as ECS, Fargate, EC2 and EKS

**Why Istio?**

* Istio is a mature service mesh implementation that allows you break down the complexity of distributed cloud native deployments by taking complex functionality out of the application code & moving it into network
* Istio is most feature rich and it’s also built to serve the enterprise use cases
* Istio has following features
  + Automatic metrics and network tracing collected b/w services within the mesh, as well as inbound and outbound network communication with external clients and services
  + Advanced rule-based traffic routing & control with automatic load balancing for HTTP, gRPC, WebSocket and TCP traffic across service versions
  + Automatic mTLS management for secure service-to-service communication using strong identity base authentication & authorization
  + Built-in service communication resiliency with retries, circuit breaking failover and fault injection

**Istio Architecture**

* Architecture
* The components of the control plane
  + Pilot:
    - This is essential component that programs Envoy sidecars. This converts Istio defined APIs into Envoy-Specific configurations
    - Responsible for service discovery with in service mesh
  + Citadel: This provides critical security capabilities within the Istio service by managing certificates and provide strong service identities
  + Galley: Galley manages istio’s configuration, It validates, ingests, processes and distributes istio configuration to other control-plane services
  + Envoy: Istio use Envoy proxy for sidecars as well as gateways



Understanding Bookinfo Application, Istio Traffic Management, Virtual Service, Destination Rules, Gateways, Service Entries and Sidecars

**Istio Installation**

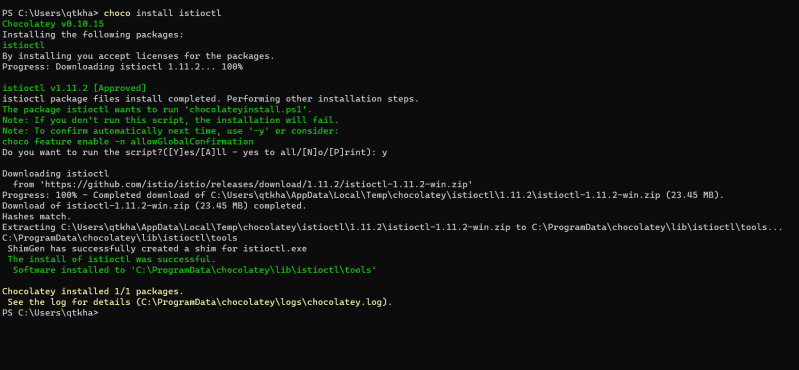
Istio Setup:

Method selection <https://istio.io/latest/about/faq/#install-method-selection>

* Download & install istio:

<https://istio.io/latest/docs/setup/getting-started/#download>

* Windows instance:

Ensure kubectl is installed and then install istioctl 

Linux Instance (Azure)

Install azure cli <https://learn.microsoft.com/en-us/cli/azure/install-azure-cli-linux?pivots=apt>

sudo apt update

curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash

az login

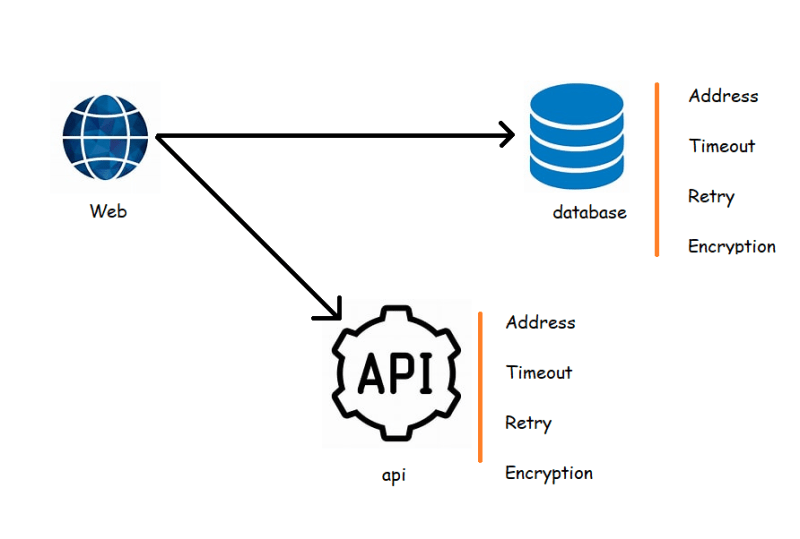
Now lets create a aks cluster <https://learn.microsoft.com/en-us/azure/aks/learn/quick-kubernetes-deploy-cli>

Now download istio by following steps https://istio.io/latest/docs/setup/getting-started/#download

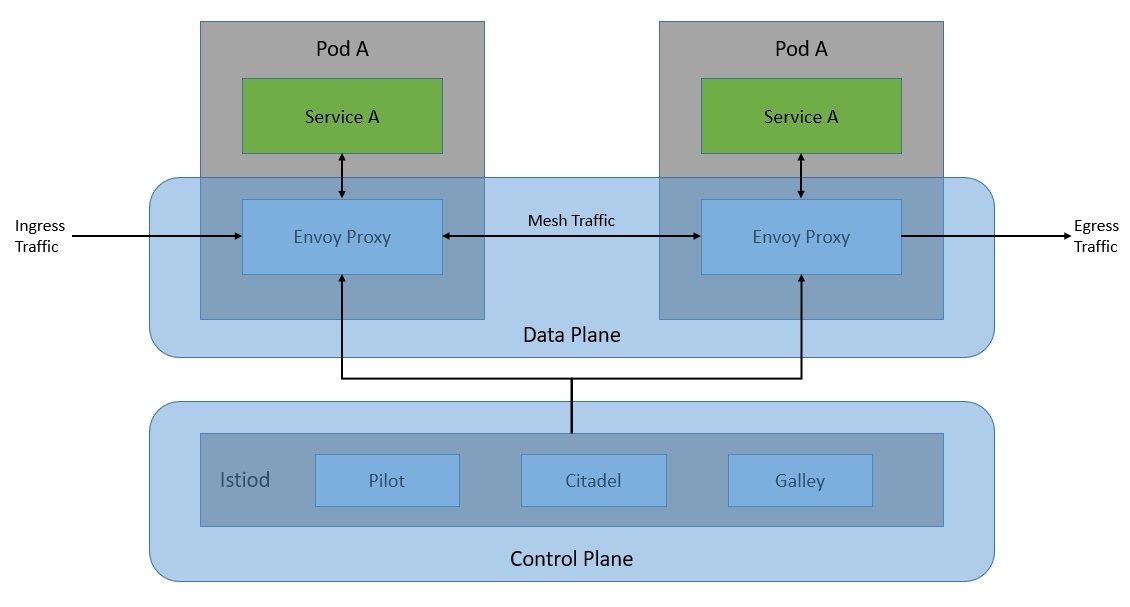
Now let’s create istio <https://istio.io/latest/docs/setup/getting-started/#install>

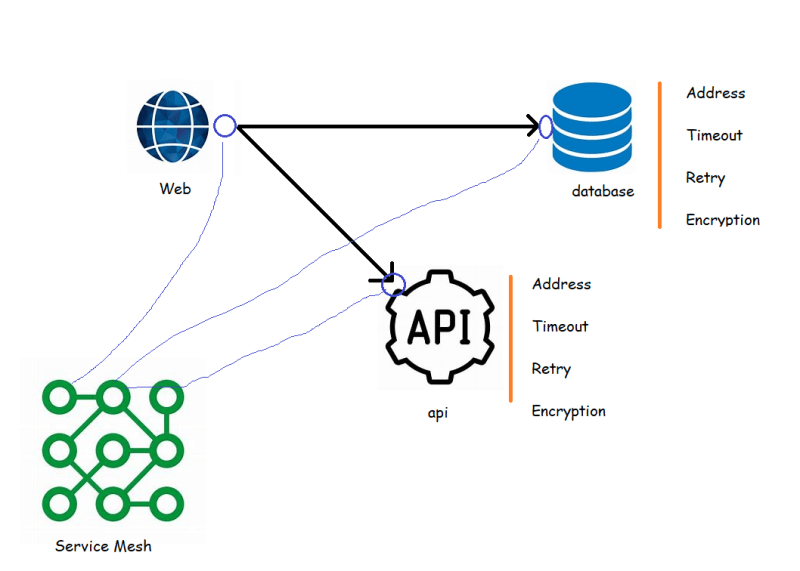
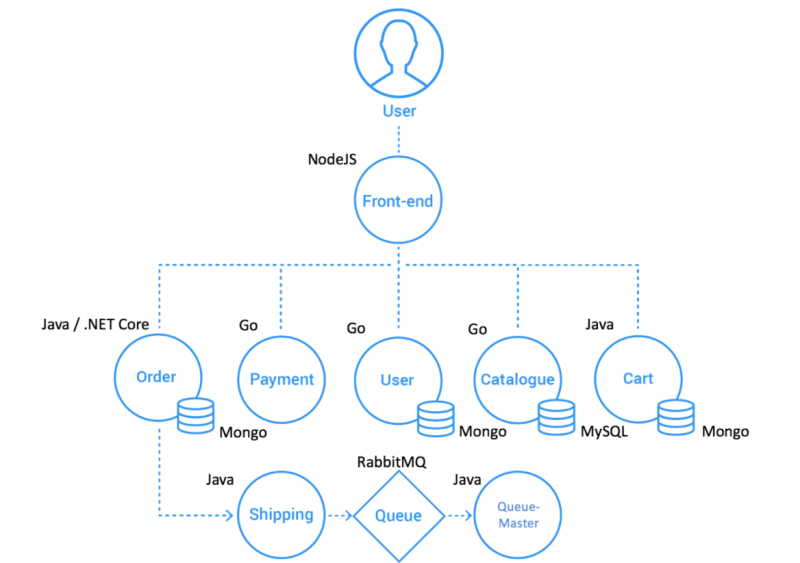
**Virtual Service**

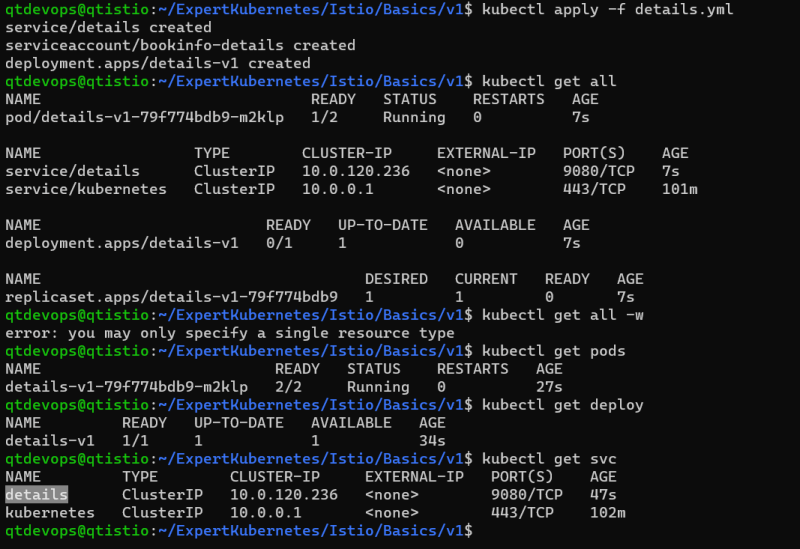
<https://istio.io/latest/docs/reference/config/networking/virtual-service/>

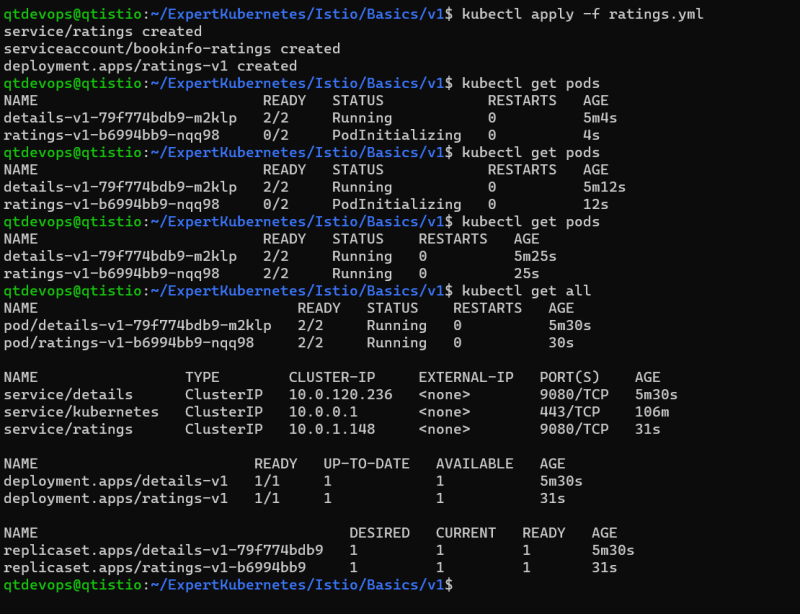
Communication between software components in an application 

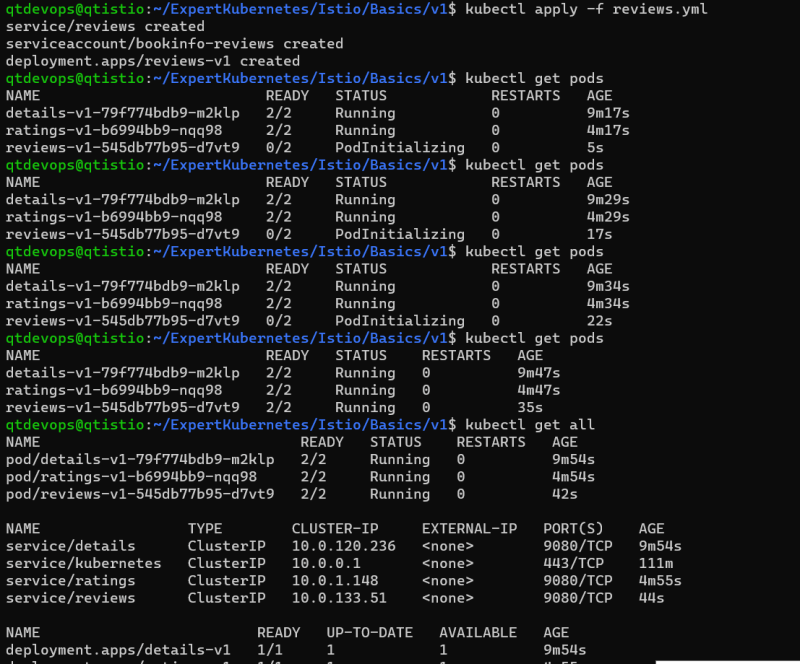
Service to service communication by ISTIO

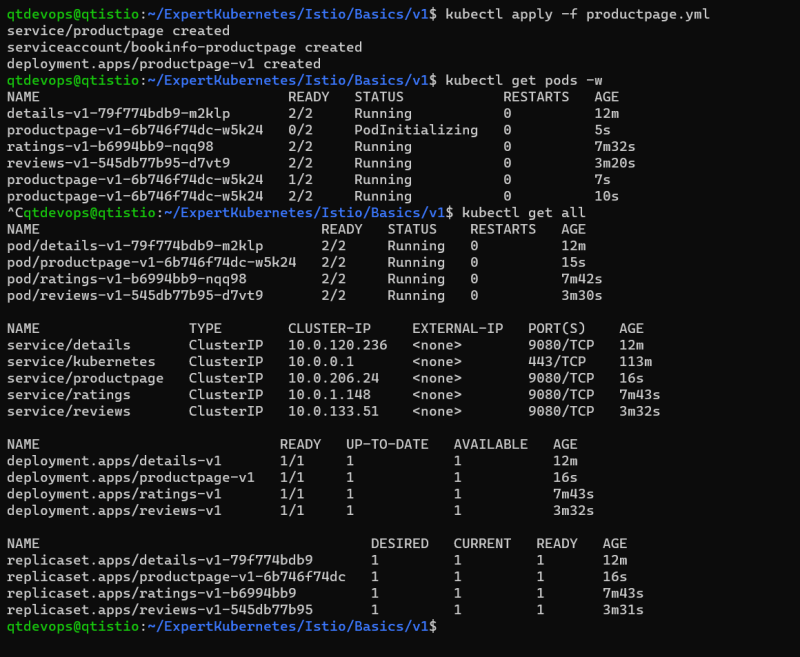


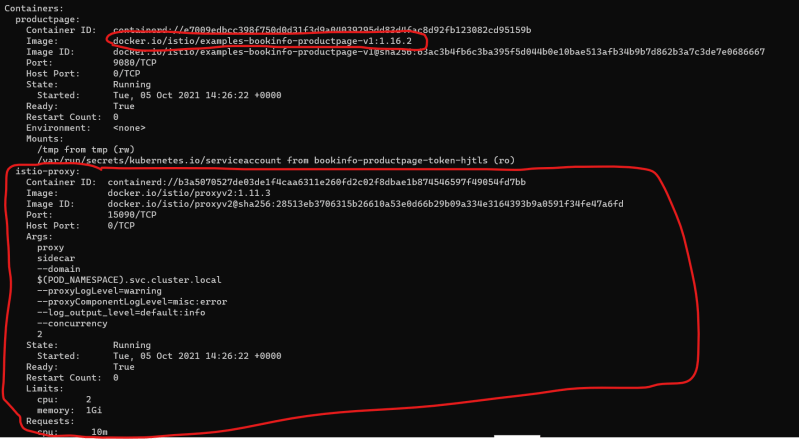
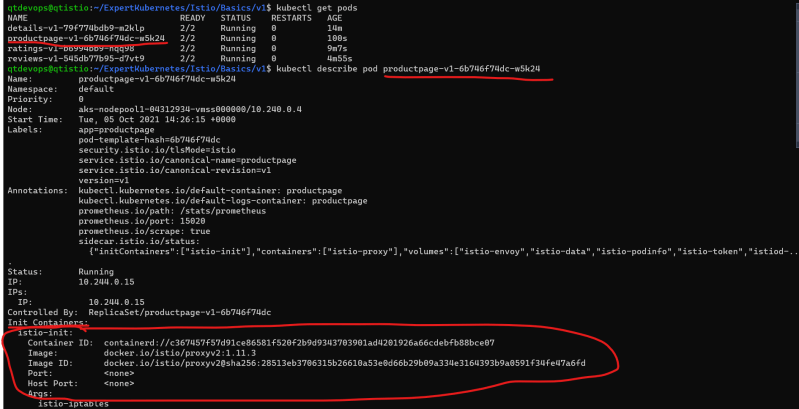
Microservice demo application architecture 

The changeset for details service <https://github.com/asquarezone/ExpertKubernetes/commit/f5cf65f8a2a1316e25ad6b576ab0ce6917dee4b5> and apply this 

Add ratings service changeset <https://github.com/asquarezone/ExpertKubernetes/commit/40ec099b26f747a88296a766bbecb9781f9ba8e3> and apply these changes 

Created a review application with only v1. <https://github.com/asquarezone/ExpertKubernetes/commit/ec79e917835cfbbfce04308b63126486aa154158> for the changes and apply the changes 

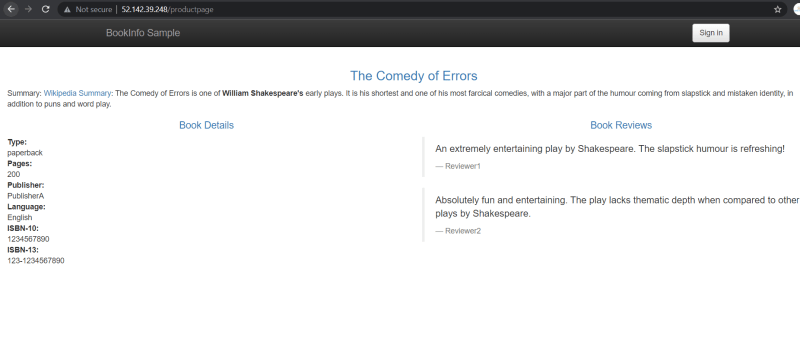
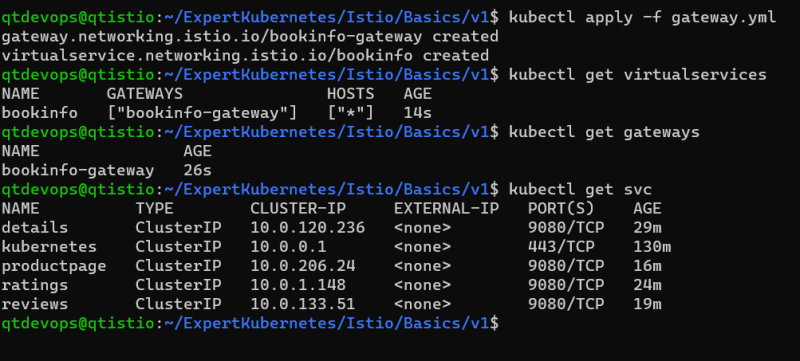
<https://github.com/asquarezone/ExpertKubernetes/commit/d418459ea5705e1681e78e328765370524bd79e1> for the product page changes and apply the product page 

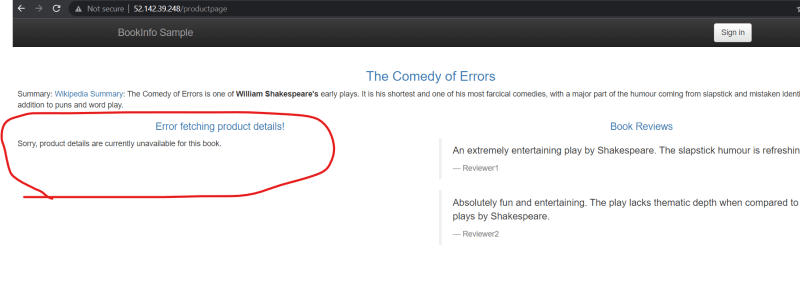
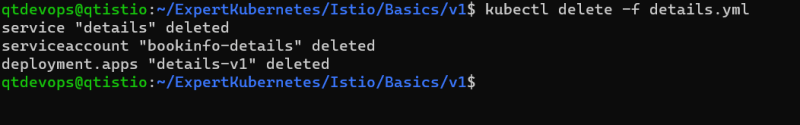
Now lets try to observe any pod 

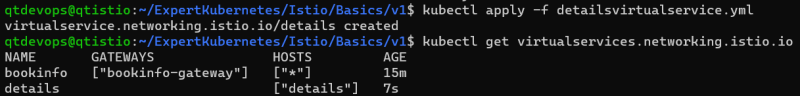
* The envoy proxy is automatically injected into the pod, this is because while we were setting up istio we have executed the following command, which enables automatic istio injection in default namespace

kubectl label namespace default istio-injection=enabled

* We need to access the application from outside k8s cluster, for this we would be using the gateway
* Add the gateway for external access <https://github.com/asquarezone/ExpertKubernetes/commit/49657328d08ca358fe2c59e2da06b9888bc2e99b> for the changeset.

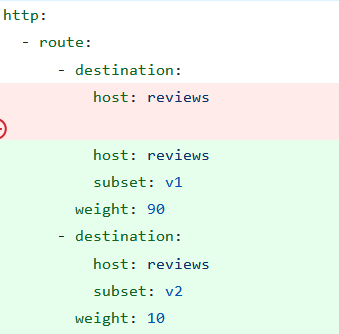
In the above changeset we have custom resources defined by istio, lets apply the changes 

Now lets delete the details service and try to use the products page 

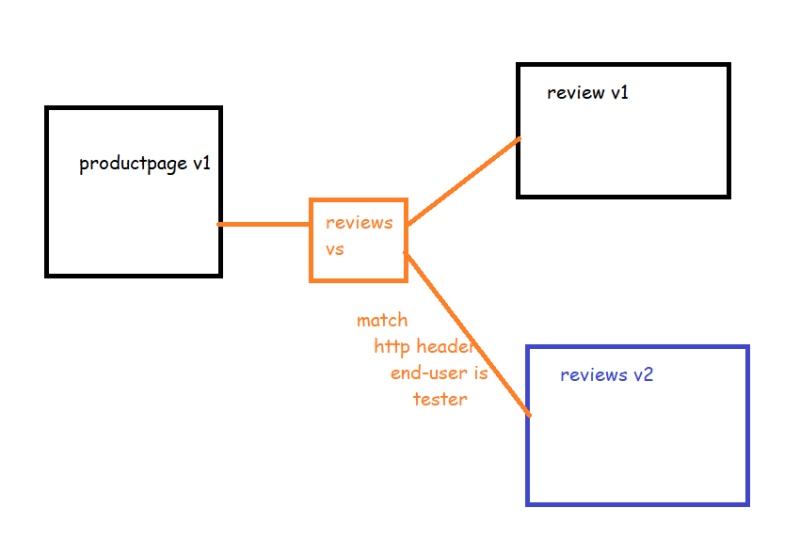
Lets create a virtual service for details with retries <https://github.com/asquarezone/ExpertKubernetes/commit/28d970175b9cb4326e5e488f067fce5106e37ab2> for the changes 

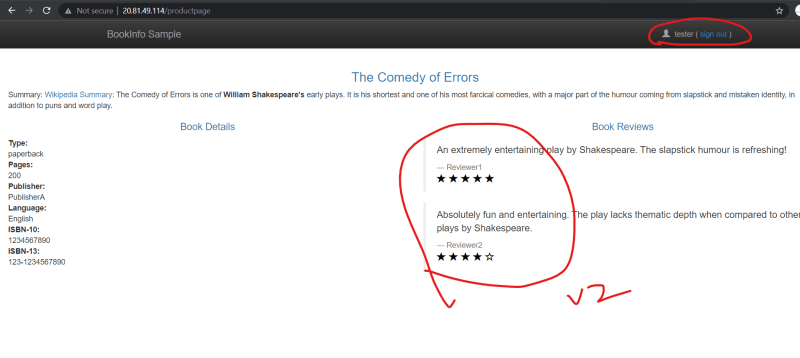
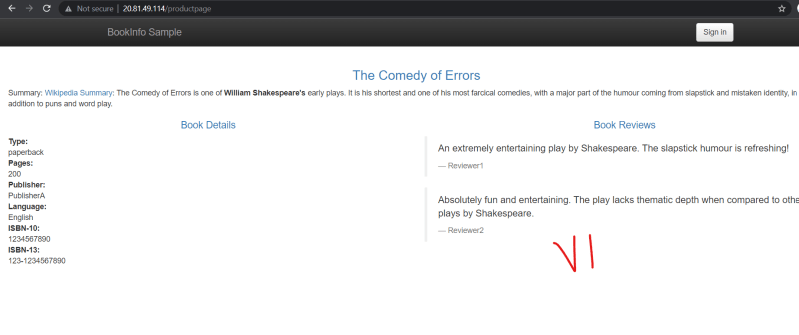
**Virtual Service Continued**

* Now we have created a new details api with container image pushed to docker hub <https://hub.docker.com/repository/docker/shaikkhajaibrahim/details-api> When env SERVICE\_VERSION is set to
  + v-unavailable: It would return 503 response
  + v-timeout: The request would time out with 500 response
  + v-timeout-first-call: The first request would time out
  + <nothing>: application would work normally
* For the manifests <https://github.com/asquarezone/ExpertKubernetes/commit/831d0a300892d1d74f7eebc4d450f971d9d3bea0> for the changeset
* Now deploy v2
* Added reviews virtual service <https://github.com/asquarezone/ExpertKubernetes/commit/8ff6db5a6f429b6f35ede7b78c4cf591f812d5c1> and apply the changes
* Now let’s add version 2 of reviews. Apply the changes <https://github.com/asquarezone/ExpertKubernetes/commit/51fc1c7dbdb76301e2117f0328c1871b131c66bd>
* Made changes in the virtual service to use the destination rules and send 90% traffic to v1 of reviews and 10% to v2 of reviews <https://github.com/asquarezone/ExpertKubernetes/commit/0358e0394a295e27a1e654f87acba5393f02fea3> for the changeset

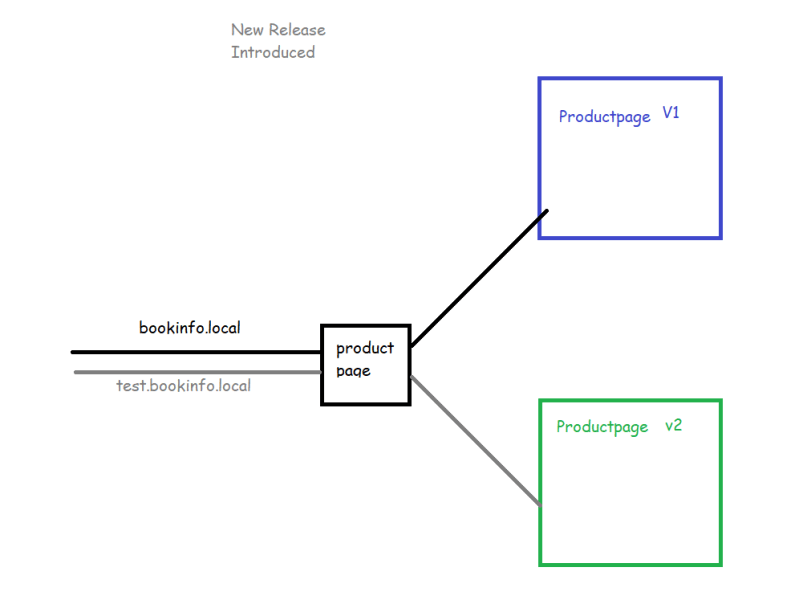


**Dark Launch with Istio**

* Let’s assume we would want to release the new version of application/component and only certain users have to use this new release
* To demonstrate this let’s use two versions of v1, v2 of reviews application 
* <https://github.com/asquarezone/ExpertKubernetes/commit/962ef8494304497e38ed1f110f470d35bb399ebf> for the changeset
* Deploy the application from v3 folder

Normal user view 

**Blue Green Deployment**

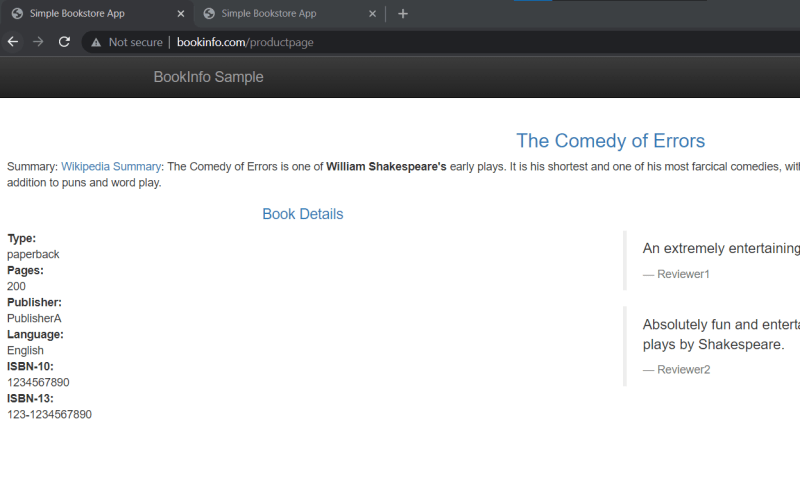
* Basic steps involed 

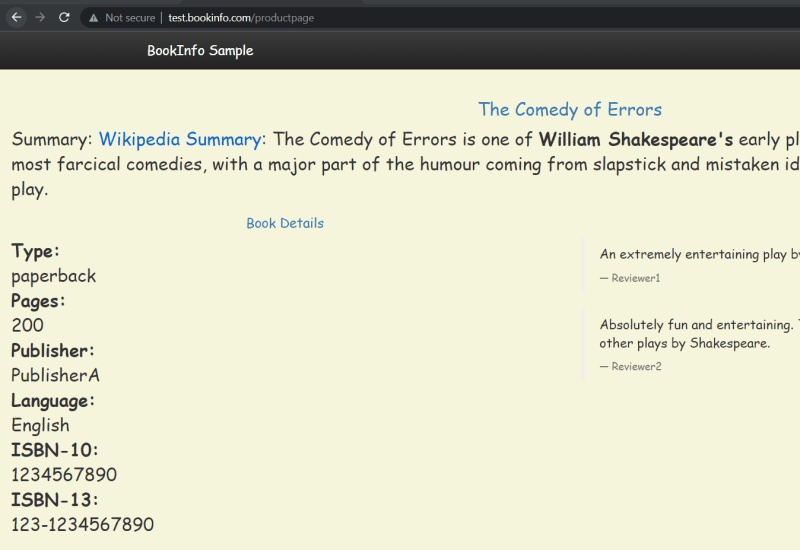
To demonstrate this lets try to create a new version for product page <https://hub.docker.com/r/shaikkhajaibrahim/productpage>

shaikkhajaibrahim/productpage:v2

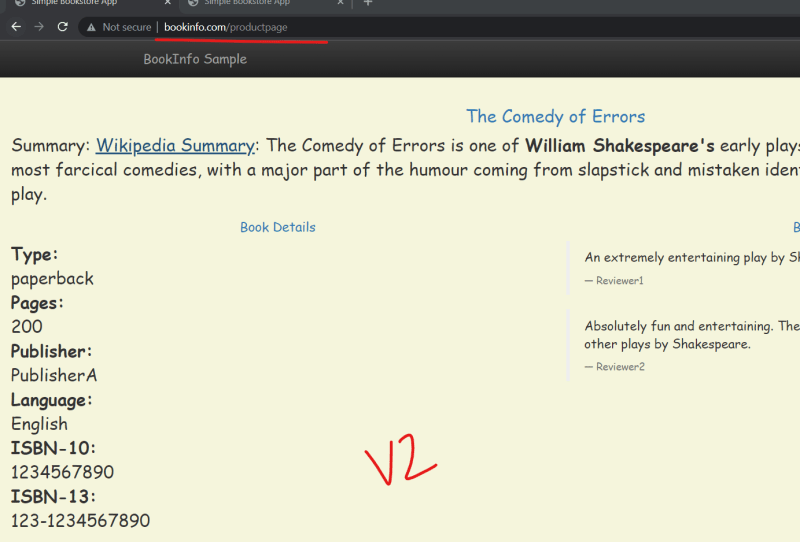
<https://github.com/asquarezone/ExpertKubernetes/commit/722bccd02a99927f8367730a43cf03706568fe89> for the changes

Now apply the deployment.

Try to access using <http://bookinfo.com> (after adding entries into dns) 

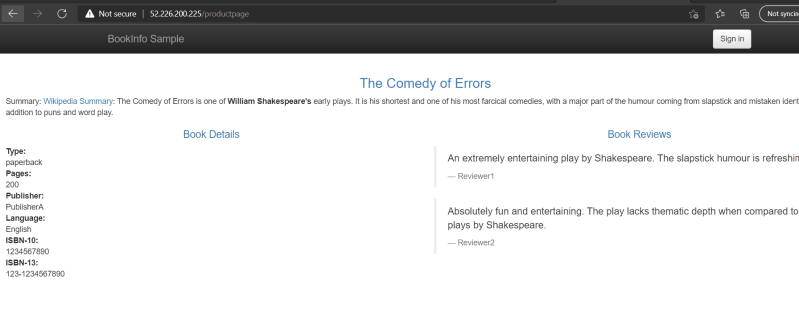
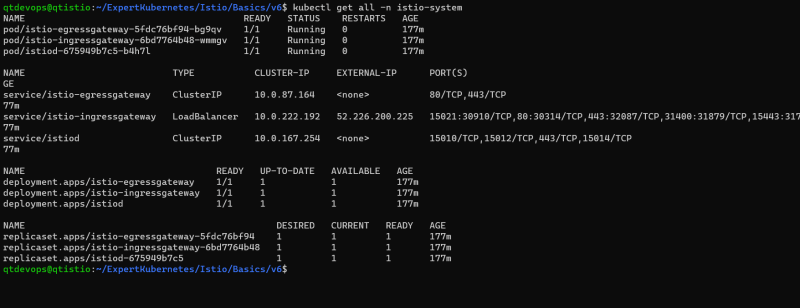
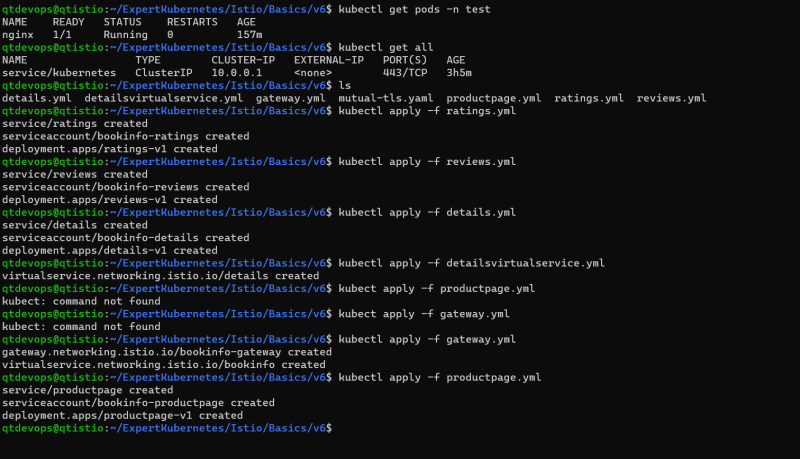
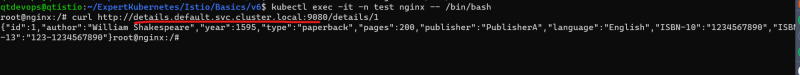
Try to access using <http://test.bookinfo.com> (after adding entries into dns) 

Now we have added simple files to move to new version for all the users and roll back to previous version in case of issues <https://github.com/asquarezone/ExpertKubernetes/commit/8361174c39ce472bc6bf878619411f5ac529212d>

Moving v2 to all the users 

Exercise: Create a Canary deployment of the product page using the two versions.

**Mutual TLS**

* Create a nginx pod in test namespace
* Created a book info application in default namespace 
* Now lets try to login into nginx pod in the test namespace and send the curl request to details page 
* Now lets enable mutual tls across all the resources in the default namespace

---

apiVersion: security.istio.io/v1beta1

kind: PeerAuthentication

metadata:

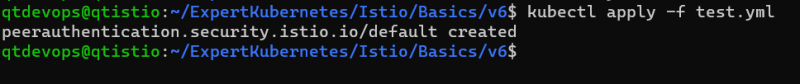
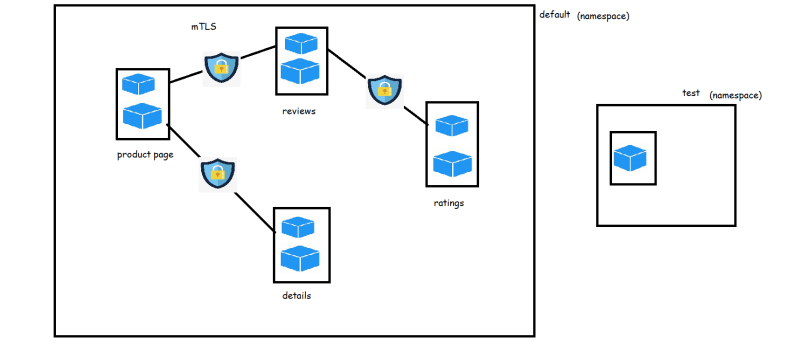
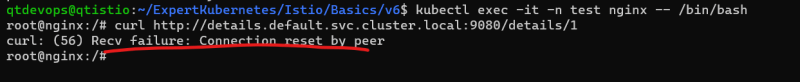
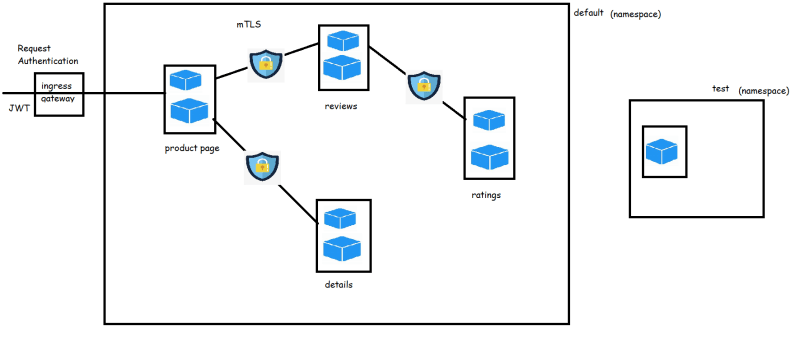
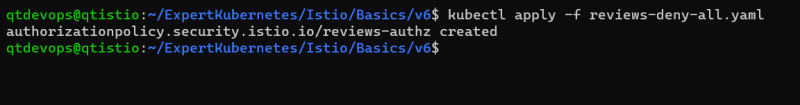
name: default

namespace: default

spec:

mtls:

mode: STRICT

* Apply the above yaml 
* Now try to login into nginx and send the curl request 
* For securing external request use the combination of RequestAuthentication & JWT (Json Web Tokens) 
* If we need to setup only selected services to access our service we can setup Authorization policy by making use of service accounts
* We have create a review authorization policy to deny all the requests [refer](https://github.com/asquarezone/ExpertKubernetes/commit/c41588b69c35f280662e0213070ba54730145272) 
* The yaml so far is

---

apiVersion: security.istio.io/v1beta1

kind: AuthorizationPolicy

metadata:

name: reviews-authz

namespace: default

spec:

selector:

matchLabels:

app: reviews

* Now if you access product page, it will not be able to show reviews
* Now lets change yaml to allow access from product page. <https://github.com/asquarezone/ExpertKubernetes/commit/885c18a1eefc5abbbad527583f3b9a00a8bef9ba> for the changes

---

apiVersion: security.istio.io/v1beta1

kind: AuthorizationPolicy

metadata:

name: reviews-authz

namespace: default

spec:

selector:

matchLabels:

app: reviews

rules:

- from:

- source:

principals: ["cluster.local/ns/default/sa/bookinfo-productpage"]

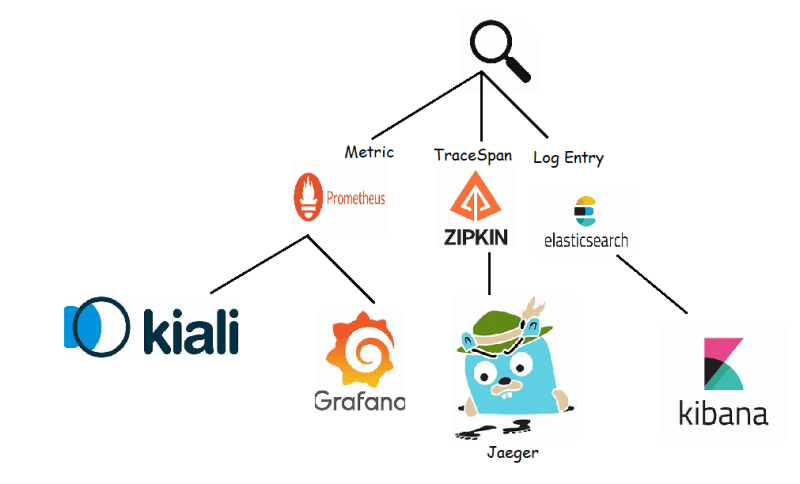
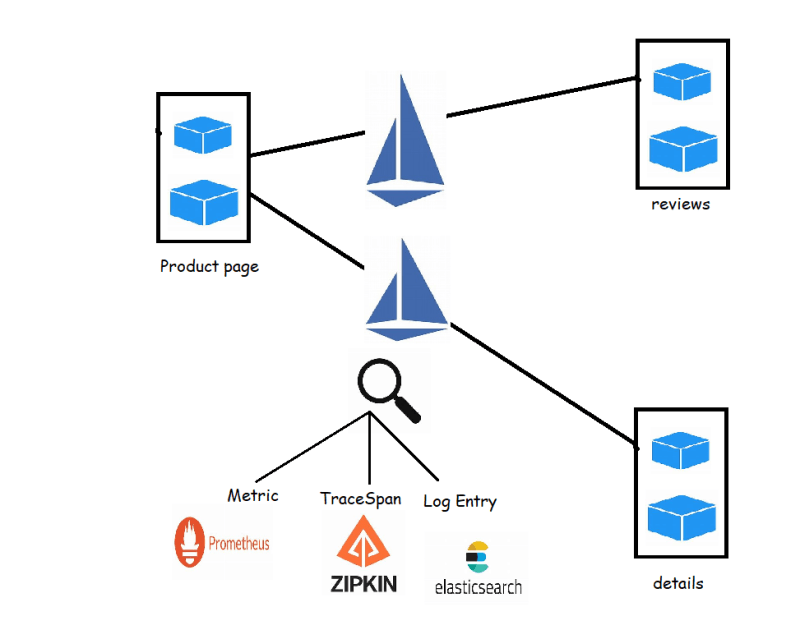
to:

- operation:

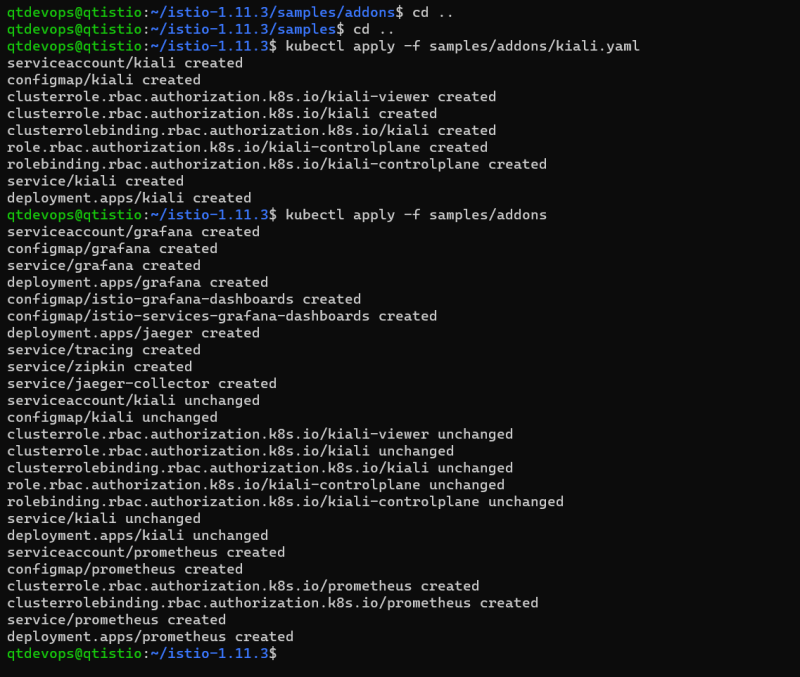
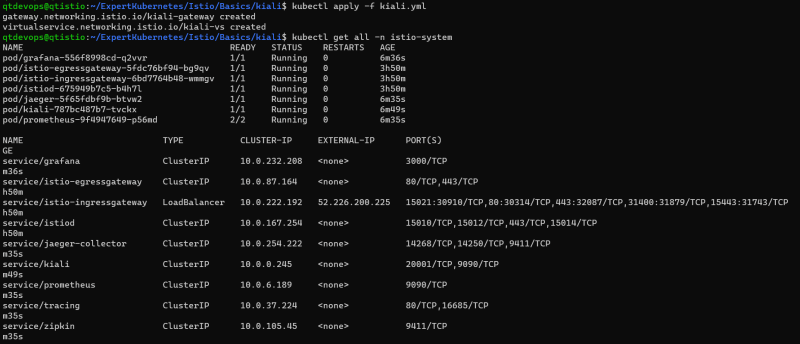
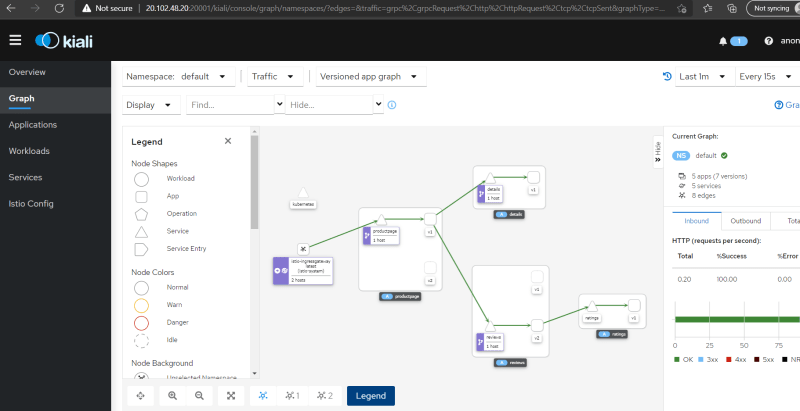
methods: ["GET"]

* Now if you access product page, it will be able to show reviews

**Observability Using Service Mesh**

* From Istio Service mesh we can view logs, metrics and visualizations as istio supports metric adapters which can be connected by multiple popular opensource tools like Prometheus, Kiali, Jaeger 

**Visualize the Service Mesh using Kiali**

* Lets try to deploy the Blue Green Deployment <https://github.com/asquarezone/ExpertKubernetes/tree/master/Istio/Basics/v4>
* Now Navigate to istio directory and execute the command <https://istio.io/latest/docs/setup/getting-started/#dashboard> 
* <https://github.com/asquarezone/ExpertKubernetes/commit/a7e657da8aa0bd6cd5f862207cadacd7205e61ff> for the kiali gateway 
* From your node where you are running istio try to use istioctl dashboard kiali --address 0.0.0.0
* Now try to use your application and look at graph of kiali 
* Lets use fortio docker container to add the artificial load to our application
* Now try to create a docker container anywhere with the following command to generate artificial load

docker container run `

--add-host "bookinfo.com:52.226.200.225" `

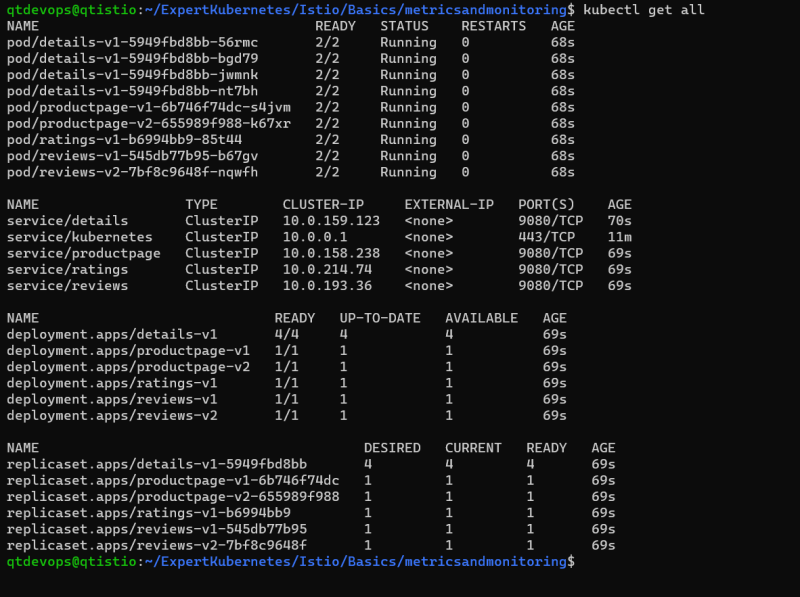
fortio/fortio load -c 32 -qps 25 -t 5m http://bookinfo.com/productpage

docker container run `

--add-host "test.bookinfo.com:52.226.200.225" `

fortio/fortio load -c 32 -qps 25 -t 5m http://test.bookinfo.com/productpage

**Using Other tools for observability**

* <https://github.com/asquarezone/ExpertKubernetes/commit/0aa4efb98c53fe911ce87e1ea3f52a9ef52cc3b5> for the sample application
* Now deploy the application 
* Lets enable mtls <https://github.com/asquarezone/ExpertKubernetes/commit/e1e593f2ae485dfb6a0e8843f4f41a5475dc0d9f>
* Now lets try to run the application fortio generating load for 30 m

docker container run `

--add-host "bookinfo.com:20.84.9.83" `

fortio/fortio load -c 32 -qps 25 -t 30m http://bookinfo.com/productpage

docker container run `

--add-host "test.bookinfo.com:20.84.9.83" `

fortio/fortio load -c 32 -qps 25 -t 30m http://test.bookinfo.com/productpage

Tracing using Jaeger requires some change in the application, To do this Application needs to have the following HTTP Headers

* + x-request-id
  + x-b3-traceid
  + x-b3-spanid
  + x-b3-parentspanid
  + x-b3-sampled
  + x-b3-flags
  + x-ot-span-context

Please watch the recording of this session to trace, log and view metrics to make whole k8s with istio observable

Use the following configuration to setup EFK <https://github.com/asquarezone/ExpertKubernetes/tree/master/Istio/Basics/metricsandmonitoring/EFK> and also <https://logz.io/blog/logging-istio-with-elk-and-logz-io/>