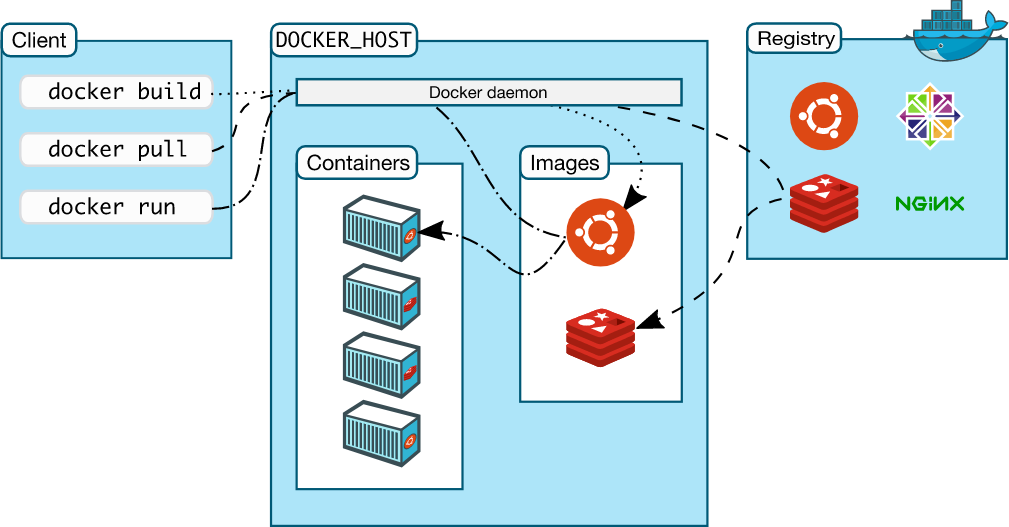
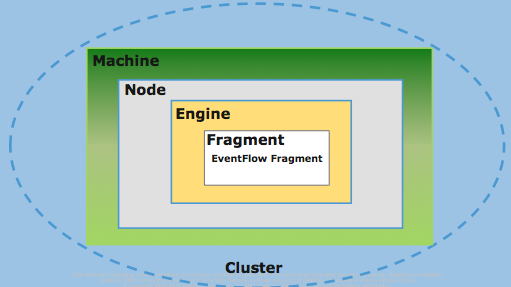
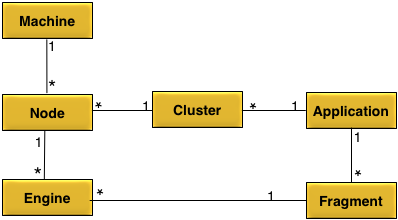
**DOCKER Architecture**



* Here containers are like nodes an executable component, Images are the image of our application which resides in a container and Registry is like a GitHub from where our required images will be pulled.

**Streambase Architecture**



**Conceptual Model**

* **Machine** — an execution context for a node.
* **Application** — business specific functionality.
* **Fragment** — an executable part of an application.
* **Cluster** — a logical grouping of nodes that communicate to support an application.
* **Node** — a container for engines.
* **Engine** — executable context for a fragment.

Docker Containers and StreamBase Nodes:

A screenshot of a cell phone

Description automatically generated

Openshift and StreamBase Nodes:

A picture containing screenshot

Description automatically generated

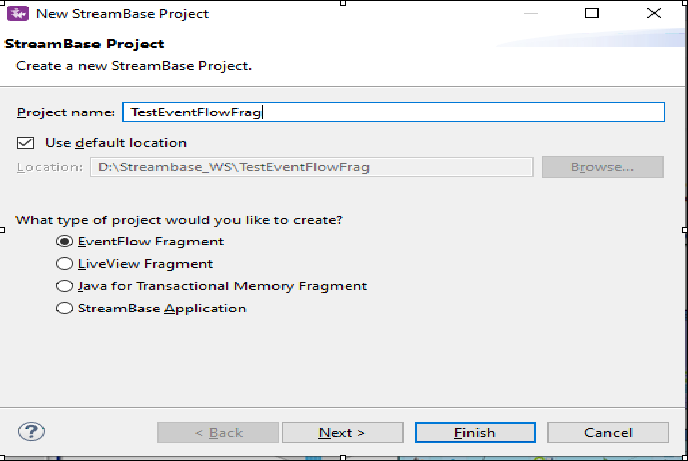
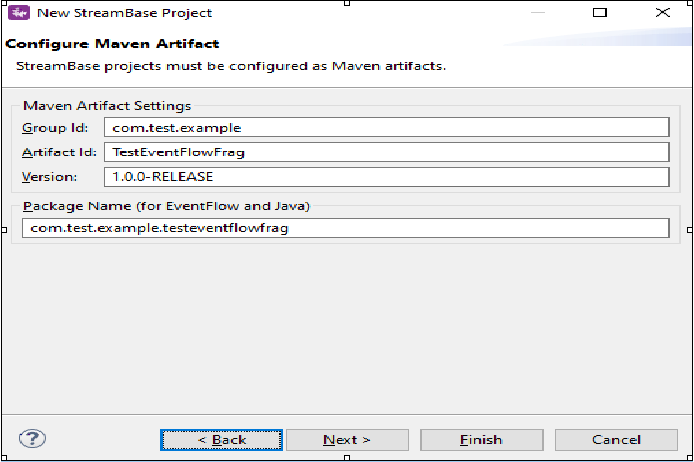
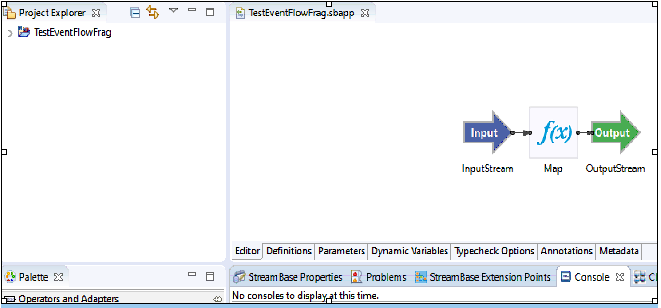
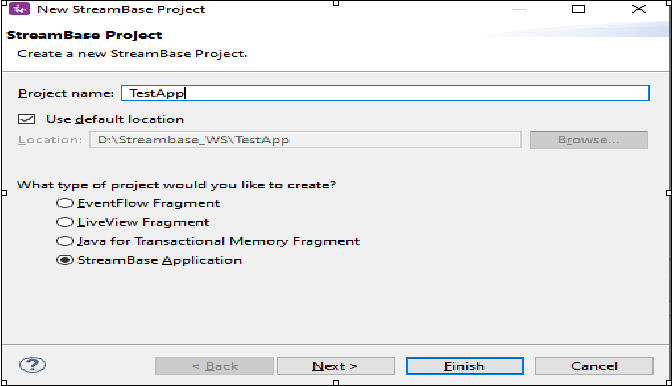
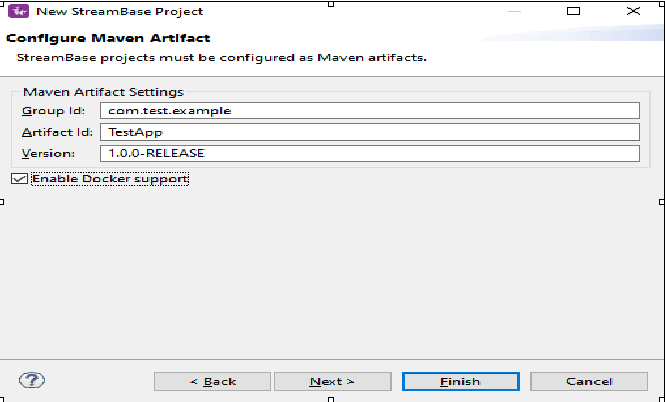
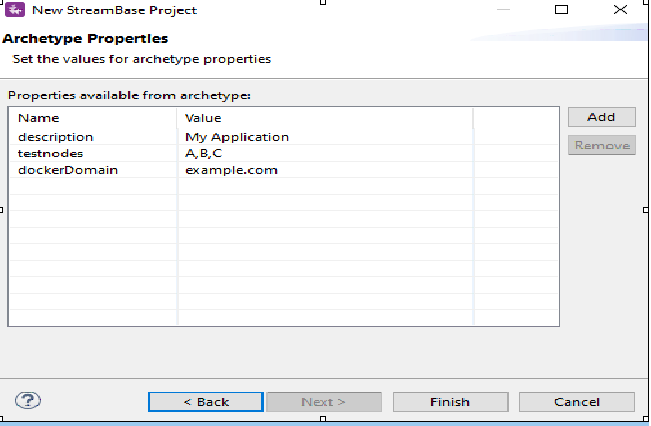
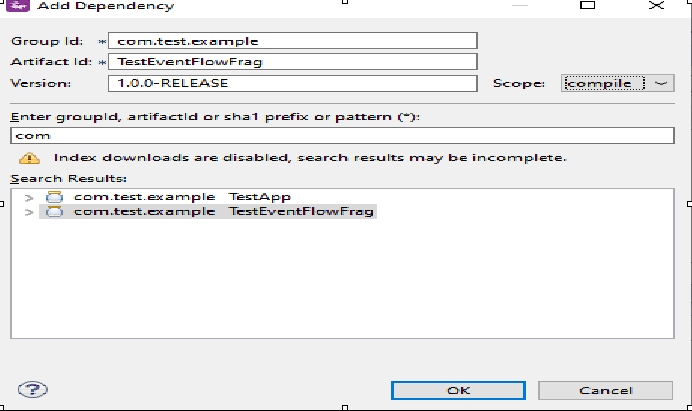
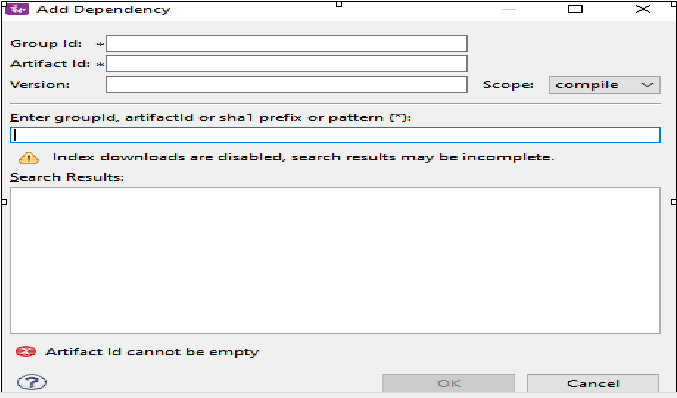
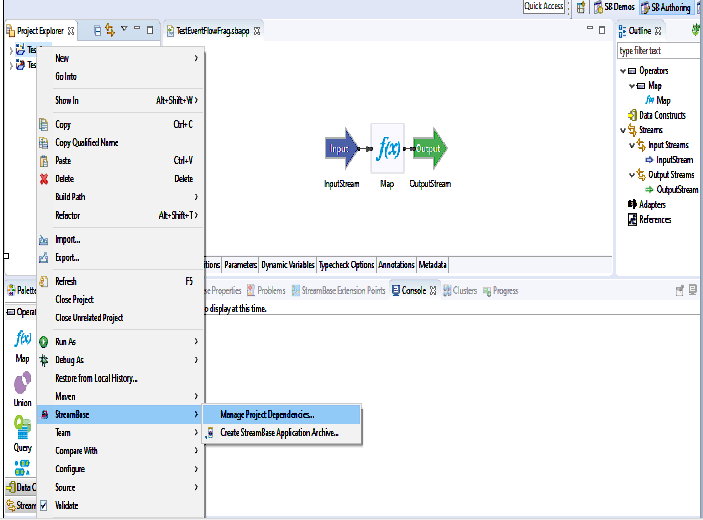
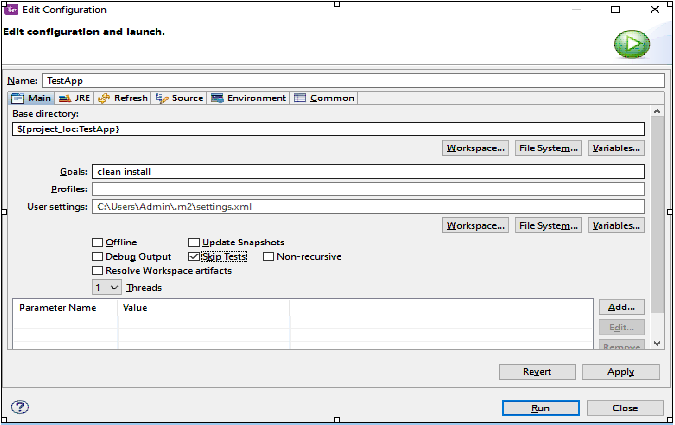
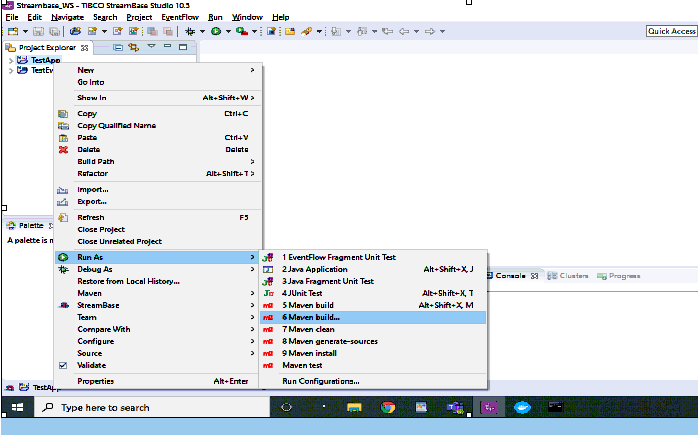
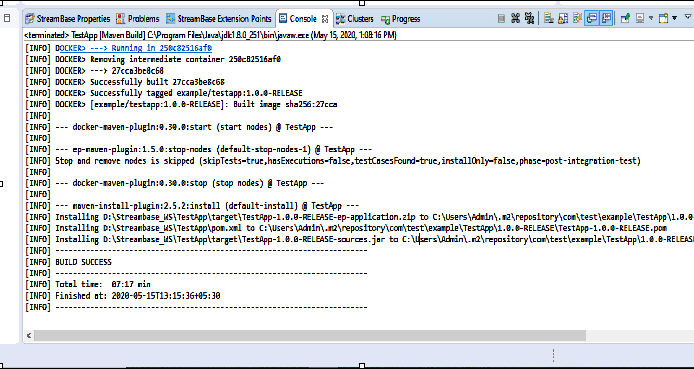
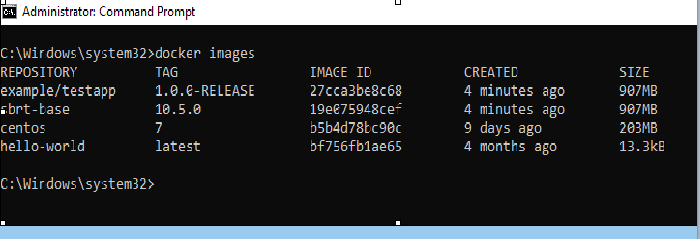
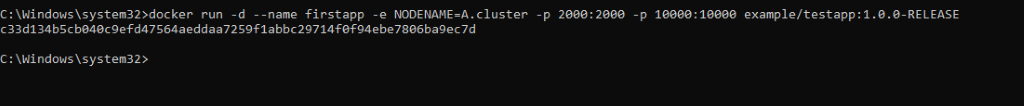
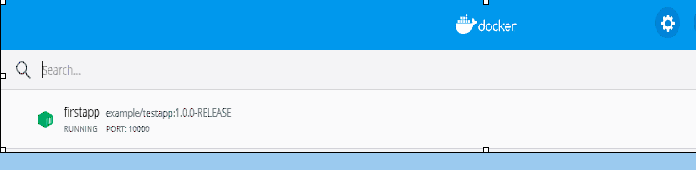
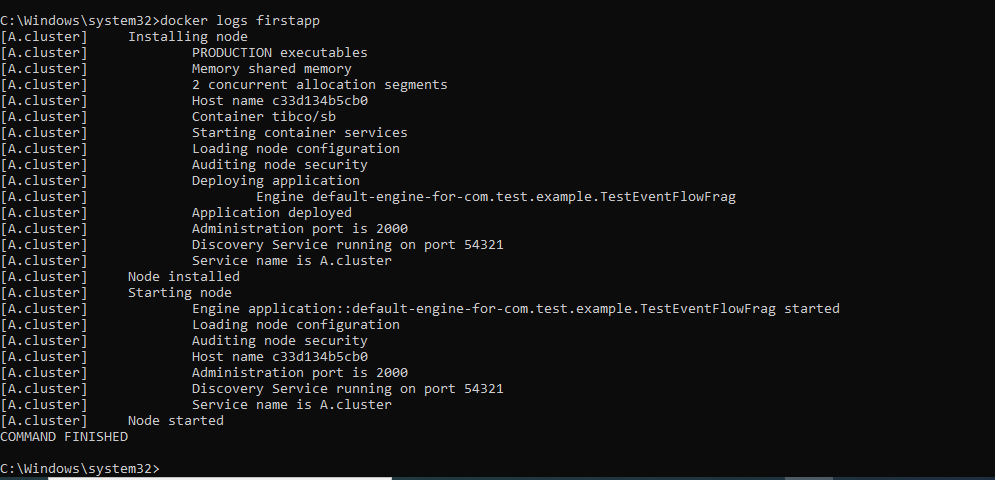
Supported Docker Environments:

A screenshot of a cell phone

Description automatically generated

* The **StreamBase Runtime** for build or staging servers and for production deployment is supported on the following platforms:
  + Red Hat Enterprise Linux Server 7\*\*
  + Red Hat Enterprise Linux Server 6 (deprecated)
  + CentOS Linux 7\*\*
  + Oracle Linux 7\*\*
  + Amazon Linux 2 on Amazon EC2
  + Microsoft Windows Server 2012 R2
  + Microsoft Windows Server 2016
* On Red Hat platforms, TIBCO supports and recommends Red Hat Enterprise MRG Realtime for deployments that require low-latency processing.
* For all Linux systems, see the Perl, Zip, and compatibility library requirements below.
* The **StreamBase Runtime** is explicitly not supported for build, staging, or production deployment on:
  + macOS
  + Windows client platforms (as listed above)
* In all cases, TIBCO Streaming supports the latest service packs and operating system updates.

**Steps for** **Dockerizing Streambase Application :**

* Create the EventFlow fragment and place all your logic inside this fragment. Select the Radio button as **EventFlow Fragment** as shown in below Screenshot.
* Click Next and provide the value for **Group Id**, **Artifact Id** and **Version**. Provide the packagename as <<Group Id>>.<<any name>> as shown in below screenshot. Please note down these three parameters and click finish.
* Sample Eventflow Fragment:
* Now create a new Streambase Project. Select Radio button as **StreamBase Application**.
* Now click next and make sure you provide the same **Group Id** and **Version** that you specified for the EventFlow Fragment created Earlier. Also click on **Enable Docker Support** checkbox and click next.
* Now you can specify **description**, **testnodes**, **dockerDomain** and click finish.
* **testnodes**: Specify one or more node names for test nodes started when you run Maven Test or another Maven goal that requires Test.
* **dockerDomain**: Specify a domain name that will allow your Docker containers to communicate with each other. The domain name you specify here is configured into the TrustedHosts section of this application's security.conf file to allow all StreamBase Runtime node to participate in the same domain as the Docker containers.
* Add the EventFlow Fragment Project depedency to our main Streambase Application. Right click on Streambase Application Project **(TestApp in my case) --> StreamBase>Manage Project Dependencies**. Add the depedency for the EventFlow Fragment and click ok.
* Now In the Project Explorer view, select and right-click the Streambase Application Project **(TestApp in my case)--> Run As>Maven Build...**
* After that In the **Goals** field, enter **clean install** and Select the **Skip Tests** check box. Click Apply and Run.
* After the above step check the console and it should be showing BUILD SUCCESS message.
* Get a list of your currently installed Docker images using either the docker images or docker image ls command:
* Now in docker you have run the following command:
* **docker run -d --name firstapp -e NODENAME=A.cluster -p 2000:2000 -p 10000:10000 example/testapp:1.0.0-RELEASE** (Use the image name shown in the output of your own docker images command.)
  + **-p** options map your host computer's ports 2000 and 10000 to the same ports on the node to be run in Docker.
  + **--name** is used to name the Docker container
  + **-e** is the environment variable for docker
  + **NODENAME** Environment variable name for Docker
  + **1.0.0-RELEASE** is the TAG that you will see when you will run docker images command.
* You must set the NODENAME environment variable for the Docker image with the -e or --env option, as shown. You can specify the node and cluster names you prefer, as long as you specify the same cluster name for Docker-hosted images that will run together as a cluster.
* After running the above command, you will see as shown in below screenshot:
* Also, in docker you can see your Application is running.
* You can check logs using command **docker logs firstapp**
* To communicate with the containerized node from the Docker host, you must set a new password for the default username.
* **docker exec firstapp epadmin --servicename=A.cluster change password --username=tibco --password=tibco**
* Now with the above image details you can deploy your application, to any of the containerized application like aws, kubernetes, etc...