appviewX

Deployment Architecture Guide

Version: 2019.3.0

Copyright AppViewX, Inc.

Trademarks

External Reference Links

This product includes software developed by the CentOS Project (www.centos.org).

This product includes software developed by Red Hat, Inc. (www.redhat.com).

This product includes software developed by VMware, Inc. (www.vmware.com).

Contact Information

AppViewX, Inc.

222 Broadway, FL 19

New York, NY 10038

Email: info@appviewx.com

Web: www.appviewx.com

Contents

Preface

Revision History

Revision	Description	Date
v1.0	Deployment Architecture	July 2019

About this Guide

The guide provides information about the deployment architecture for release 2019.3.0

Audience

The guide is for the users who want to install the AppViewX stack on managed AKS, specifically

- Platform engineers
- Implementation specialist

Text Conventions

The following text conventions are used in this document:

Convention	Description
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`codeblock`	Indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Chapter 1: Overview

AppViewX is expanding the monolithic architecture that it has currently, and is unable to cope up with the increase in addition of the vendors. So it becomes imperative to have a system that is modular and aligns along the lines of the subsystems. By this, development times are shortened and regressions/ bugs are reduced. The new architecture also gets along with the vision of the sales team where they can independently stamp and sell AppViewX offerings such as ADC+, firewall+ and so on.

Chapter 2: Salient Features

- Unified Interface for Database Communication
- Independently Testable Southbound APIs
- Promotes Modularity
- Migration to Simple Understandable Web Services
- Client Certificate Authorization
- Enabling Polyglot Development
- Load Balancing
- Dynamic VM Registration
- Swagger Document Support
- Multi-tenancy
- End to End Transaction Monitoring
- Queue Implementation
- Maven Centralized Repository

Unified Interface for Database Communication

The database abstraction layer is separated from the business logic, so that, when need arises, it enables databases to be substituted for one another. The current stack uses Mongo database. This database interface allows to seamlessly switch to another database.

Independently Testable Southbound APIs

The old architecture had an opaque tunnel which it used for communicating with devices. These meant that REST calls were testable only over the northbound API's. The device communication which was not independently testable with the framework, is now enhanced in such a way that every business logic, down to the nooks and corners of the system, are open to testing.

Promotes Modularity

The new architecture aligns the code base along the lines of subsystems, so that, all AppViewX offerings can be independently shipped.

Migration to Simple Understandable Web Services

The old architecture had OSGI, which is a very less understood and not a very commonly used technology, as its backbone. It resulted in long setup time and test enablement. The new system overcomes the above said limitations by bringing in REST web services that are simple to visualize and universally understandable.

Client Certificate Authorization

The new framework enhances security between components by the usage of client certificates. Every communication is encrypted by default to prevent eavesdropping at every level.

Enabling Polyglot Development

The components in the new framework need to conform to a standard registration process as imposed by the gateway. The components, also known as containers, can thus be written in any language, as long as it conforms to the registration process.

Load Balancing

The new framework does out of the box load balancing. The API Gateway recognizes the same REST service containers registered with different IP's and ports and does the load balancing programmatically. This alleviates the need for configuring ADC load balancers on top of these service containers, to perform load balancing.

Dynamic VM Registration

To handle extra loads, all that an admin has to do is, spin off a new container that does the same functionality, put the IP and port details into the **<LOADBALANCER>** database collection, and then call the **<GATEWAY>** endpoint. The container now becomes a container for load balancing.

```
http(s)://<gateway-ip>:<gateway-port>/avxmgr/initreg
```

Swagger Document Support

The new framework provides swagger documentation for all the registered web services out of the box. The documentation for the entire set of the documents, in AppViewX, can be seen in the following link:

```
http(s)://<gateway-ip>:<gateway-port>/avxdoc/index.html
```

Multi-tenancy

Multi-tenancy in AppViewX is enabled with the help of a tenant key which is passed in the URL as “gwkey”. A sample URL is given below

```
http(s)://<gateway-ip>:<gateway-port>/ip:port/avxapi/<apiid>?gwkey=<tenant_key>&gwsource=<request_source>
```

where,

- api-id = respective end point to which the call has to be routed
- gwkey = tenant key which is used to uniquely identify a customer
- gwsource = origin of the request

Multi-tenancy is used ideally in the demo environment. To provision a new tenant, one would:

- Request tenant addition with the Support Person/ Solution Engineer.
- Once the tenant license is generated, add it to the gateway.
- Provision a new database for the tenant and start using the architecture.

End to End Transaction Monitoring

All new requests to the gateway will be tagged with a unique transaction id. This transaction id will be logged, allowing us to track the request flow across containers.

The new framework helps the support team by enabling logging only when an error or warn occurs. not all logs are printed in log files. Only a trace of the service call will be maintained in case of successful REST service completions. This is done to alleviate the problem where an exception happens in production and by the time the support person logs in to check, the logs already would have rolled.

Queue Implementation

The new framework acts as an alternative to multithreading. The asynchronous flow supports for the call back. For async processing, there is a collection named DECORATORS in the gateway database, which in turn has a document titled ASYNC. Add a key and a value to the document, where:

- key= source action id
- value= target action id

Maven Centralized Repository

- Controlled environment for managing third party libraries.
- Simplifies the release management process for application related artifacts.
- Reduced network bandwidth.

Chapter 3: Old Architecture vs New Architecture

- Karaf OSGI containers become REST Web service containers
- Problems with OSGI
- Benefits of using REST

Karaf OSGI containers become REST Web service containers

This move is a big effort with the following considerations:

- New development on java service layer to call such rest API's.
- Node gateway will have to handle the new rest API's coming from Iagent layer.
- Node gateway will have to manage session management and RBAC.

Owing to the hardships faced with OSGI, this move becomes completely justified.

Problems with OSGI

- Testing Iagent functionality is impossible.
- Very poor knowledge in team.
- Renders very poorly to debugging and traceability.
- As multi sub system codes reside in one VM, it is impossible to find the problem causing area at any instant.
- Problems in one small area pulls everything down.
- Karaf industry adaptation is less.
- Setting up development environment will be time consuming.
- ECF communication protocol is very poorly understood.
- Any minor code change warrants an entire regression cycle.

Benefits of using REST

- Device API’s can be called by the clients, directly using REST.
- Uniform web service interface.
- API access is easily controllable and throttled.
- Improved bug fixing, debugging, and traceability.
- Testing Iagent functionality directly is now possible.
- Easy to Profile pin points to errors arising from one sub system.
- As each of the processes are loosely coupled, the unavailability of one of the module will have no effect on another.
- At least 25% time saved regards to less regression arising from the code base separation.
- Check in/out process and conflicts are resolved completely and saves time.
- Shorter times to release any kind of new additions.
- Easy to setup development environment.

Chapter 4: Implementation Details

Annotation1: AvxAction

1. Annotate the action end point class with AvxAction and provide values for the attributes available:
    - Value = Action id using which the REST will be exposed. This is a mandatory field.
    - Type = To identify whether the service exposed is a business service or a system one. This is an optional field and will have the default value set as BUSINESS.
    - Http method = HTTP Method that is supported by the service. This is a mandatory field.
    - Path parameters = Path param input that are expected. This is an optional field.
    - Description = Description about the service that is exposed. This is an optional field. If provided, it will be published in the swagger documentation for the corresponding service end point.
    - Query parameters = Input parameters passed to the service as query params. This is an optional field.
    - Permissions = Permissions required to invoke the service endpoint. This is an optional field.
    - SLA = To capture the service level agreement details. This is an optional field.
2. Extend the AvxActionExecutor class (AvxActionExecutor<O, I>) where
    - O=Input Payload
    - I=Response Payload
3. Dependency injection can be done using the AvxInject annotation.
   An example:

```
package com.appviewx.example;

@AvxAction(value = "user", httpMethod = AvxHttpMethod.GET, permissions = {

"general:acctmgmt:get" },

description = "returns hello world", path = "/{name}", queryParams = {
```

```
@AvxQueryParam("greeting") })
public class UserAction extends AvxActionExecutor<User, Void> {
@AvxInject
private IUserManagement userMgmt;
@Override
protected User execute(AvxRequestContext<Void> reqContext)
throws AvxServiceException {
User user = userMgmt.getUser(reqContext.getPathParams().get("name"));
return user;
}
}
```

Annotation2: AvxService

1. Annotate the Management class using AvxService.
2. @AvxInject can be used for injecting dependencies.

   An example:

```
package com.appviewx.example.management;
import com.appviewx.example.dao.UserDao;
import com.appviewx.plugin.framework.exception.AvxDaoException;
import com.appviewx.plugin.framework.exception.AvxServiceException;
import com.appviewx.plugin.framework.service.annotation.AvxInject;
import com.appviewx.plugin.framework.service.annotation.AvxService;
import com.payoda.commons.core.bean.User;
@AvxService
public class UserManagement implements IUserManagement {
@AvxInject
private UserDao userDao;
public User getUser(String userName) throws AvxServiceException {
User user;
try {
user = userDao.getUser(userName);
} catch (AvxDaoException e) {
throw AvxServiceException.Builder.create(e).build();
}
return user;
```

```
}

}
```

Annotation3: AvxStorage

1. Annotate the DAO class using AvxStorage.
2. Extend the DAO class with DaoImpl class.

   An example:

```
package com.appviewx.example.dao;

import com.appviewx.common.dao.DAOImpl;

import com.appviewx.plugin.framework.exception.AvxDaoException;

import com.appviewx.plugin.framework.service.annotation.AvxStorage;

import com.payoda.commons.core.bean.User;

@AvxStorage

public class UserDaoImpl extends DAOImpl<User> implements UserDao {

public UserDaoImpl() {

super("appviewx", "user");

}

public User getUser(String user) throws AvxDaoException {

User name = super.getById(user);

return name;

}

}
```

Chapter 5: Addendum 1: Setting up the Development Environment

The development environment can be setup using the following steps:

1. Clone the GIT repository in the local machine.
2. Import the projects into eclipse using **Import > Existing Maven Projects.**
3. Right-click **POM** and select run as maven clean install.
4. To start the process, right click on the project and select **Run As > Run Configurations**.
5. In the **VM arguments** section, include the following properties:

```
-Dport=<port> -Davx_property_file_path=<property_file_path> -
DCONTAINER_ID=<container_id> -DSUBSYSTEM=<subsystem>
-DDESCRIPTION=<description> -DVENDOR=<appviewx>
-DVERSION=<version> -DMINOR_VERSION=<minor_version>
-DMODULE=<module> -DCATEGORY_ID=<category_id>
```

   Example:

```
-Dport=8081 -Davx_property_file_path=D:/properties/appviewx.properties
-DCONTAINER_ID=sshkey-linux -DSUBSYSTEM=certificate
-DDESCRIPTION=SSHSouthbound -DVENDOR=appviewx -DVERSION=1.0
-DMINOR_VERSION=1 -DMODULE=appviewx -DCATEGORY_ID=SOUTHBOUND
```

Note: Port and container id should be unique.

6. In the **Run configuration** window, click the **Classpath** tab and select **Advanced > Add External folder** to specify the location of the property path.
7. In Run configuration Main menu select your project and browse the Main class of jetty server.
   Example:

   • Project : SSH_VM

   • Main : com.appviewx.common.framework.jetty.Main

8. To start the process, click **Run**.
9. To test the VMs availability, open the following URL: `http://host:port/services/reg?gwkey=test&gwsource=test`

Note: To download third party libraries from the centralized maven repository, ensure to place the `settings.xml` file in the **.m2** folder.

Sample Settings.xml

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
https://maven.apache.org/xsd/settings-1.0.0.xsd">
<localRepository>${user.home}/.m2/repository</localRepository>
<interactiveMode/>
<usePluginRegistry/>
<offline/>
<pluginGroups/>
<servers>
<server>
<id>avxrepo</id>
<username>avxreporo</username>
<password>avxreporo</password>
</server>
</servers>
<mirrors>
<mirror>
<id>internal-repository</id>
<name>Maven Repository Manager running on repo.mycompany.com</name>
<url>http://XXX.XXX.XXX.XXX/repository/avx-central/</url>
<mirrorOf>*</mirrorOf>
</mirror>
</mirrors>
<proxies/>
</settings>
```

Chapter 6: Addendum 2: Setting up the API-Gateway in the Development Environment

For gateway with license disabled

Please download the gateway executable from the below mentioned location.:

- **Windows**: https://avxarch.payoda.com/attachments/download/33119/app_windows.exe
- **Linux**: https://avxarch.payoda.com/attachments/download/33120/app_linux

After downloading the executable, please execute the file using the commands as given below.

- **Linux**: `app_linux <avx prop file path>`

  Example: `app_linux /data/appviewx.properties`

- **Windows**: `app_windows <avx prop file path>`

  Example: `app_windows D:\appviewx.properties`

For gateway with license enabled

Generate the gateway executable for the local machine by updating the following information in the build parameters of the Jenkins job.

Note: Check with the Devops team for the Jenkins job details.

Execute the following command and provide the result in the host name parameter.

1. Execute the hostname command as follows:
   - Windows: `C:\> hostname`
   - Linux: `$> hostname -f`

Note: Append the domain `.payoda.com` to the output of the `hostname` command. For example, if the output is *abc*, enter the hostname as *abc.payoda.com* in the `hostname` parameter.

2. Check all the subsystems
3. Check the build. Once the build is successful, two **gzip** packages will be generated.

   • Linux users shall use the **AppViewX_license.tar.gz file** after the build. To extract the contents of the file, execute the following command:

   ```
   tar -xvf AppViewX_license.tar.gz
   ```

   • Windows users shall perform the following steps:

   a. Install the node on the local machine \\fileserver\Software\AppviewxSoftware

   b. Download the **avxgw.tar.gz file.**

   c. To extract the contents of the file, execute the following command:

   ```
   tar -xvf avxgw.tar.gz
   ```

4. Add the following properties in the appviewx.properties file:

```
GATEWAY_PORT=7300
GATEWAY_HTTPS=FALSE
GATEWAY_LOG_TYPE=file
EXPIRY_NOTIFICATION_LIMIT=10
GATEWAY_BASE_URL=http://localhost:7300/
GATEWAY_SERVICE_URL=http://localhost:7300/avxapi
GATEWAY_KEY=<GATEWAY_KEY>
SOURCE=WEB
```

Note: The value of the GATEWAY_KEY parameter should match with the value of the TENANT_KEY attribute generated during the Appviewx gateway build.

5. Start the node gateway using the following commands:

   • Linux users: `./avxgw <property_file_path>`

   Example: `./avxgw /data/appviewx.properties`

- Windows users: `<node_exe_path> app.js <property_file_path>`

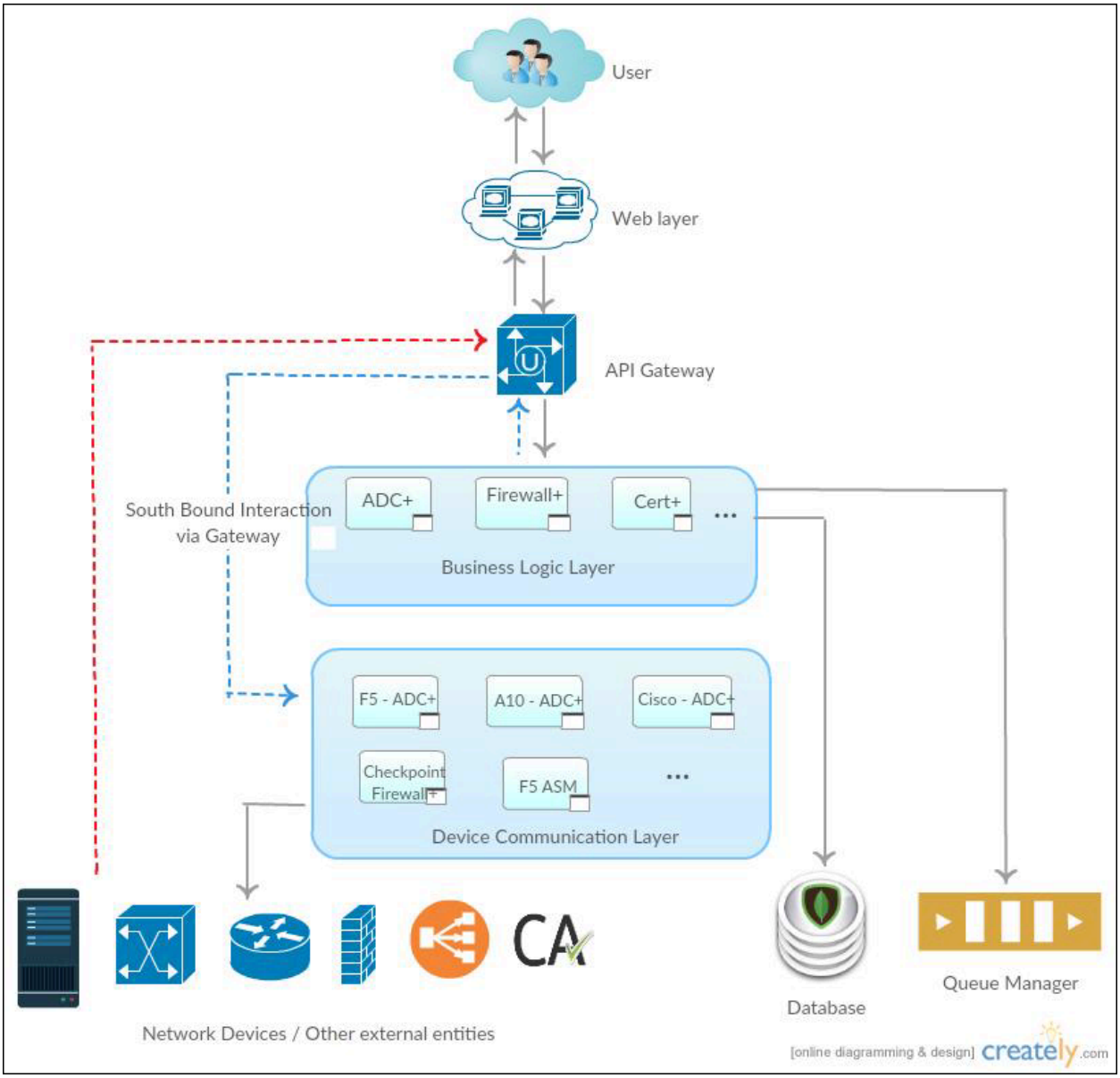
  Example: `node.exe app.js D:\appviewx.properties`

Note: For reference, please download the sample project from the following link: https://avxarch.payoda.com/attachments/download/33122/sample-action.zip

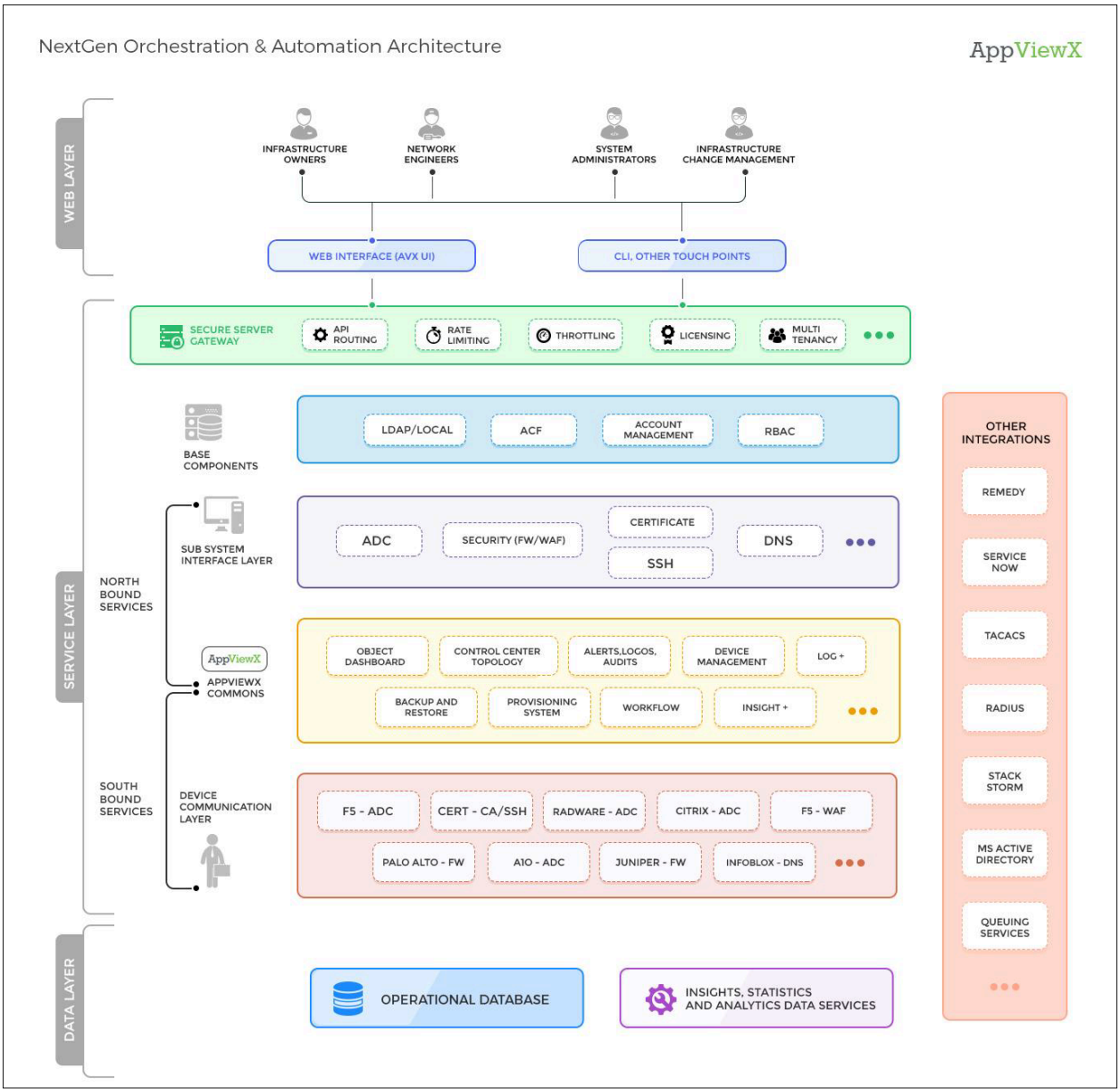
Chapter 7: Addendum 3: Diagrams

- Logical Diagram
- Conceptual Diagram
- Deployment Diagram

Logical Diagram



Conceptual Diagram



Deployment Diagram