**ASSIGNMENT NO. 10**

**Title:** Install and use Cloud Foundry

**Aim:** Install and demonstrate the use of PaaS tool - Cloud Foundry to design a web application.

**Objective:** To study and demonstrate the use of Cloud Foundry for designing a web applications.

**Theory:**

Cloud Foundry is an open source cloud computing platform originally developed in-house at VMware. It is now owned by Pivotal Software, which is a joint venture made up of VMware,  EMC, and General Electric.

Cloud Foundry is optimized to deliver…

* Fast application development and deployment.
* Highly scalable and available architecture.
* DevOps-friendly workflows.
* Reduced chance of human error.
* Multi-tenant compute efficiencies.

Not only can Cloud Foundry lighten developer workloads but, since Cloud Foundry handles so much of an application’s resource management, it can also greatly reduce the overhead burden on your operations team.

Cloud Foundry’s [architectural structure](https://docs.cloudfoundry.org/concepts/architecture/) includes components and a high-enough level of interoperability to permit…

* Integration with development tools.
* Application deployment.
* Application lifecycle management.
* Integration with various cloud providers.
* Application Execution.

Although Cloud Foundry supports many languages and frameworks, including Java, js, Go, PHP, Python, and Ruby, not all applications will make a good fit. As with all modern software applications, your project should attempt to follow the [Twelve-Factor App](http://12factor.net/) standards.

Key benefits of Cloud Foundry:

* Application portability.
* Application auto-scaling.
* Centralized platform administration.
* Centralized logging.
* Dynamic routing.
* Application health management.

Applications deployed to Cloud Foundry access external resources via *Services*. In a PaaS environment, all external dependencies such as databases, messaging systems, files systems and so on are *Services*. When an application is *pushed* to Cloud Foundry, the services it should use also can be specified. Depending on the application language, auto-configuration of services is possible - for example a Java application requiring a MySQL database picks up the MySQL service on Cloud Foundry if it is the only one defined in the current *space*.

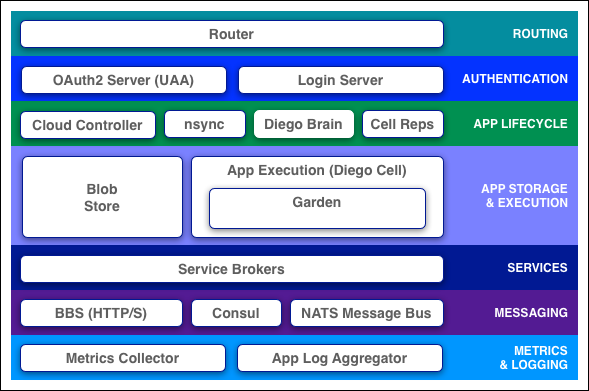
Services have to be deployed to the platform first and then are available to any application using it. Another advantage of Pivotal CF is that many pre-defined services can be deployed into it directly using the Administration Console. Users of the Open Source Cloud Foundry must make services available by writing and running BOSH scripts.

PWS provides services thanks to a partnership arrangement with App Direct.

The following services are available to Pivotal CF and on PWS:

* Data Storage: [MySQL](https://en.wikipedia.org/wiki/MySQL), [PostgreSQL](https://en.wikipedia.org/wiki/PostgreSQL), [MongoDB](https://en.wikipedia.org/wiki/MongoDB), [Redis](https://en.wikipedia.org/wiki/Redis), [Riak](https://en.wikipedia.org/wiki/Riak), [DataStax](https://en.wikipedia.org/wiki/DataStax) ([Cassandra](https://en.wikipedia.org/wiki/Apache_Cassandra)), [Neo4J](https://en.wikipedia.org/wiki/Neo4J), Pivotal HD ([Hadoop](https://en.wikipedia.org/wiki/Apache_Hadoop))
* Messaging: Pivotal [RabbitMQ](https://en.wikipedia.org/wiki/RabbitMQ)
* Development: [CloudBees](https://en.wikipedia.org/wiki/CloudBees) [Jenkins](https://en.wikipedia.org/wiki/Jenkins_(software)) ([Continuous Integration](https://en.wikipedia.org/wiki/Continuous_Integration))
* Mobile: API Gateway, Data Sync, Push Notifications (Pivotal proprietary services to support Mobile Apps).

**Architecture of Cloud Foundry:**



**Router**

The [router](https://docs.cloudfoundry.org/concepts/architecture/router.html) routes incoming traffic to the appropriate component, either a Cloud Controller component or a hosted application running on a Diego Cell.

The router periodically queries the Diego Bulletin Board System (BBS) for which cells and containers each application is currently running on. Then it recomputes new routing tables based on the IP addresses of each cell virtual machine (VM) and the host-side port numbers for the cell’s containers.

**Authentication**

**OAuth2 Server (UAA) and Login Server**

The OAuth2 server (the [UAA](https://docs.cloudfoundry.org/concepts/architecture/uaa.html)) and Login Server work together to provide identity management.

**App Lifecycle**

**Cloud Controller and Diego Brain**

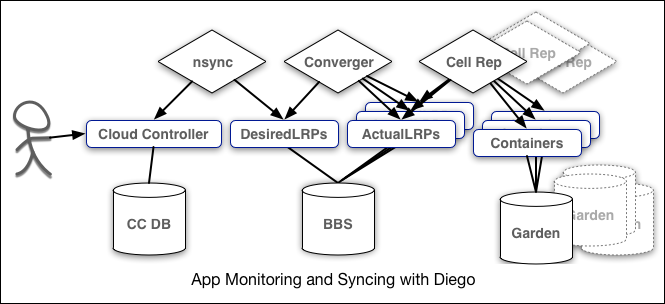
The [Cloud Controller](https://docs.cloudfoundry.org/concepts/architecture/cloud-controller.html) (CC) directs the deployment of applications. When a developer pushes an application to Cloud Foundry, she is targeting the Cloud Controller. The Cloud Controller then directs the Diego Brain through the [CC-Bridge](https://docs.cloudfoundry.org/concepts/diego/diego-components.html" \l "bridge-components) to coordinate individual [Diego cells](https://docs.cloudfoundry.org/concepts/architecture/" \l "diego-cell) to stage and run applications

In [pre-Diego architecture](https://docs.cloudfoundry.org/concepts/diego/dea-vs-diego.html" \l "design), the Cloud Controller’s Droplet Execution Agent (DEA) performed these app lifecycle tasks.

The Cloud Controller also maintain records of [orgs, spaces, user roles](https://docs.cloudfoundry.org/concepts/roles.html), [services](https://docs.cloudfoundry.org/services/overview.html), and more.

**nsync, Converger, and Cell Reps**

To keep applications available, cloud deployments must constantly monitor their states and reconcile them with their expected states, starting and stopping processes as required. In pre-Diego architecture, the [Health Manager (HM9000)](https://docs.cloudfoundry.org/concepts/diego/dea-vs-diego.html" \l "hm9k) performed this function. The nsync, Converger, and Cell Reps use a more distributed approach.



These three components work along a spectrum of representations that extends between the user and the application containers. At one end, the user sets how each application should be scaled. At the other end, instances of that application running on widely-distributed VMs may crash or become unavailable.

The nsync, Converger, and Cell Rep components work together along this spectrum to keep apps running as they should. nsync watches for changes in application scaling instructions from the Cloud Controller and writes them into DesiredLRP structures in Diego’s internal BBS database. Inside each cell, the Cell Rep watches the Garden for the state and health of the application instances running in the cell’s containers, and updates corresponding ActualLRP values in the shared BBS as they change locally.

At the center of the process, the Diego Brain’s Converger component monitors the DesiredLRPand ActualLRP values, and launches or kills application instances as appropriate to reconcile ActualLRP counts with DesiredLRP values.

**App Storage and Execution**

**Blob Store**

The blob store is a repository for large binary files, which github cannot easily manage because github is designed for code. Blob store binaries include:

* Application code packages
* Buildpacks
* Droplets

**Diego Cell**

Each application VM has a Diego Cell that executes application start and stop actions locally, manages the VM’s containers, and reports app status and other data to the BBS and [Loggregator](https://docs.cloudfoundry.org/concepts/architecture/" \l "metrics-logging).

In pre-Diego CF architecture, the [DEA node](https://docs.cloudfoundry.org/concepts/architecture/execution-agent.html) performed the task of managing the applications and containers on a VM.

**Services**

**Service Brokers**

Applications typically depend on [services](http://docs.cloudfoundry.org/services/) such as databases or third-party SaaS providers. When a developer provisions and binds a service to an application, the service broker for that service is responsible for providing the service instance.

**Messaging**

**Consul and BBS**

Cloud Foundry component VMs communicate with each other internally through http and https protocols, sharing temporary messages and data stored in two locations:

* A [Consul server](https://docs.cloudfoundry.org/concepts/diego/diego-components.html" \l "consul) stores longer-lived control data, such as component IP addresses and distributed locks that prevent components from duplicating actions.
* Diego’s [Bulletin Board System](https://docs.cloudfoundry.org/concepts/diego/diego-components.html" \l "bbs) (BBS) stores more frequently updated and disposable data such as cell and application status, unallocated work, and heartbeat messages. The BBS is currently implemented in [etcd](https://coreos.com/etcd/).

The route-emitter component uses the NATS protocol to broadcast the latest routing tables to the routers. In pre-Diego CF architecture, the [NATS Message Bus](https://docs.cloudfoundry.org/concepts/diego/dea-vs-diego.html" \l "nats) carried all internal component communications.

**Metrics and Logging**

**Metrics Collector and Loggregator**

The metrics collector gathers metrics and statistics from the components. Operators can use this information to monitor a Cloud Foundry deployment.

The Loggregator (log aggregator) system streams application logs to developers.

**Installation Steps :**

1. **Uninstall the existing version of Eclipse IDE.**

**Explanation** – The version of Eclipse installed in our labs is either Indigo or Mars. To use Eclipse as an IDE for ‘web development’, we need ‘Mars EE’ version of Eclipse. In order to avoid workspace conflicts, it is better to uninstall the previous versions.

**Commands to be executed on terminal:**

sudo apt-get autoremove eclipse\*

rm -r ~/.eclipse/

Delete the ‘workspace’ from home folder and check that eclipse does not exist anymore!

**Upgrade Java.**

To check the current version of java, type ‘java –version’ on the terminal. It would be either 1.6 or 1.7. To upgrade Java -

**Commands to be executed on terminal:**

sudo add-apt-repository ppa:webupd8team/java

sudo apt-get update

sudo apt-get install oracle-java8-installer

Type ‘java –version’ on the terminal again to check if it is now showing 1.8.

1. **Install Eclipse Mars EE.**

**Explanation** – Since Eclipse is an open source IDE, it can be freely downloaded from its official website. There are multiple ways of downloading and installing Eclipse. One way is as follows:

Open the following website in the browser –

<http://www.eclipse.org/downloads/>

Download the Eclipse Installer for Linux 64 bit OS. Once the installer is downloaded, run the executable file. It asks for the version of Eclipse to be downloaded. Select ‘Eclipse IDE for Java EE developers’ and install it.

At the end of this step, Eclipse Mars EE must be installed on the computer.

1. **Create an account on Cloud Foundry.**

**Explanation** – To be able to deploy our application on CloudFoundry, it is necessary that we must have an CloudFoundry account.

Open the following website in the browser –

https://account.run.pivotal.io/sign-up

Enter your credentials. You will be asked to verify your details through your email account.

4) Now open Eclipse EE.

5) Goto Help-> Eclipse Market place

6) Search for cloud foundry

7) Select Eclipse tools for Cloud Foundry 1.0 M6 and click install now

8) Complete the installation.

9)Now Goto New-> Project

10) Select Dynamic Web Project.

11) Give the project name and set target runtime as Cloud Foundry (Runtime) v1.0

12) Click on finish

13) In project explorer window right click on the project you created and select New-> HTML file.

14) Give a name to html file and click on finish

15) Code the html file

16) Now goto Window-> Show View -> Server.

17) In server tab right click and select new server.

18) Search for cloud foundry.

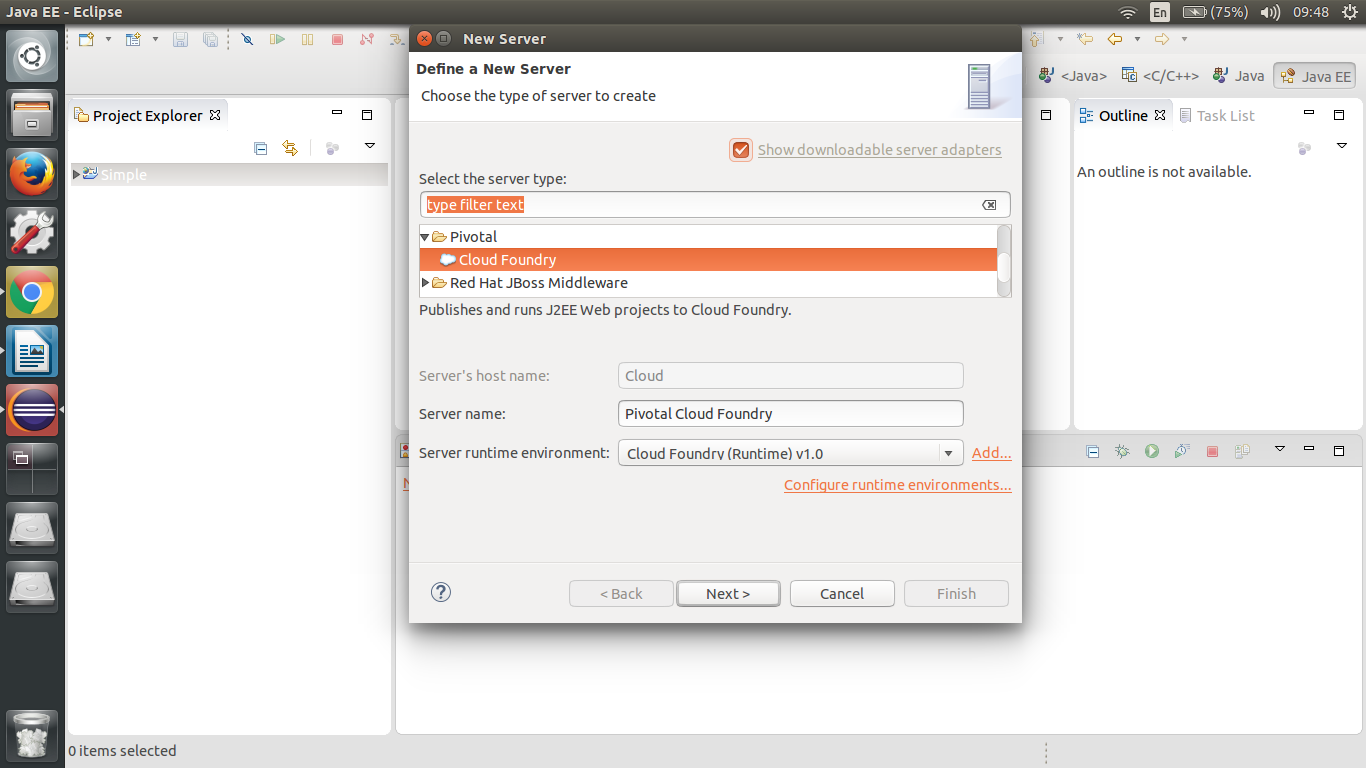
19) Click on next and enter the email id and password you used to create account on cloud foundry.

20) Click on validate account and then click on finish.

21) Right click on the project, select Run as -> Run on Server.

22) Click on finish.

23) Congrats!!! Your Application is published.

****

**Mathematical Modelling**:

**Input**

**Output**

Let ‘S’ be the system such that,

S = {I, O, Fn, Sc, Fc}

Where,

I -> {I1, I2, . . . , In} : set of inputs

O -> {O1, O2, . . . , On} : set of outputs

Fn -> {Fn1, Fn2, . . . ,Fnn} : set of functions

Sc -> {Sc1, Sc2, . . . ,Scn} : set of success cases

Fc -> {Fc1, Fc2, . . . ,Fcn} : set of failure cases

**I: Set of Inputs**

I1: Pivotal credentials.

I2: Active internet connection.

I3: numbers for sorting

**O: Set of Outputs**

O1: sorted list

O2: Deployed web app

**Fn: Set of Functions**

Fn1: merge sort (recursive)

1. partitioning the list
2. comparing values
3. appropriate swapping

Fn2: Display

**Sc: Success Cases**

Sc1: valid credentials entered.

Sc2: internet connection is active

Sc3: sorted array displayed.

**Fc: Failure Cases**

Fc1: Internet connection lost

Fc2: Array not sorted properly due to wrong logic.

Fc3:Not a valid subscription for the cloud foundry space .

**Input** : Set of numbers to be sorted

**Output**: Sorted List .

**Platform :** Ubuntu 14.04 , Eclipse EE Neon , jdk 1.8+

**Conclusion** : Thus a webapp is created and deployed on cloud using cloud foundry.