The Java EE 7 Platform and Support for the PaaS Model

0. Introduction

NIST [1] defines cloud computing as follows:

Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

It calls out the following five essential characteristics of cloud computing: *on demand self-service, broad network access, resource pooling, rapid elasticity, measured service.*

It offers the following definition of Platform-as-a-Service (PaaS):

The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly application hosting environment configurations.

Additionally, it offers the following definition of Software-as-a-Service (SaaS):

The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through a thin client interface such as a web browser (e.g., web-based email). The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings.

It is a goal of Java EE 7 to add to the platform support for the PaaS model as well as a limited form of the SaaS model while preserving as much as possible the established Java EE programming model and the considerable investments made by customers, vendors, and system integrators into the Java EE ecosystem.

1. Roles

The Java EE 6 specification defines the following roles:

- Java EE Product Provider
- Application Component Provider
- Application Assembler
- Deployer
- System Administrator
- Tool Provider
- System Component Provider

To support the proposed PaaS and Saas use cases, Java EE 7 will define the following additional roles:

- PaaS Product Vendor
- PaaS Provider
- PaaS Account Manager
- PaaS Customer
- Application Submitter
- Application Administrator (temporary name, final name TBD)
- End-user

The following traditional Java EE roles acquire new or expanded definitions:

- Java EE Product Provider
- System Administrator
- Deployer

Here is a short description of each new and modified role.

Modified Roles:

Java EE Product Provider: A creator/distributor of software that provides the provisioning, deployment, and management infrastructure and APIs for Java EE applications to run in a PaaS environment.

System Administrator: The individual(s) responsible for installing, configuring and managing the PaaS environment, including the resources that are made available to applications running in the environment. When any resource in the system needs to be maintained or upgraded the System Administrator is tasked with performing the maintenance.

Deployer: The individual(s) responsible for configuring the application to run in the PaaS cloud environment. The Deployer installs the application into the PaaS environment, handles its provisioning, and configures its external dependencies.

New Roles:

PaaS Provider: A company that stages a PaaS product and offers a PaaS environment to internal or external customers who will typically be charged based on their resource consumption.

PaaS Account Manager: Each PaaS Customer wishing to deploy applications to the PaaS environment must first establish an account that they can use to log into the system, and that gives them permissions to deploy to and access PaaS resources. The PaaS Account Manager works for the PaaS Provider and is charged with the maintenance of all of the PaaS Customer accounts. The Account Manager may adjust permissions of a given Customer account according to the SLA with that Customer.

Application Submitter: The Application Submitter takes an application and submits it to the PaaS environment on behalf of the PaaS Customer. If the PaaS Customer is an individual then he or she will be the same person as the Submitter. If the PaaS Customer is a company then the Submitter will be an employee of that company and will use the company Customer credentials to log into the PaaS System in order to deploy the application. In cases where the application is intended only for use by the Customer submitting the application (i.e., non-SaaS use), the Submitter and the Deployer may be the same person.

PaaS Customer: An individual or company account holder with permissions to deploy and run applications in the PaaS environment and to access resources available there. Each Customer may be considered a separate " tenant " of the PaaS system in that there is no sharing of runtime state or data between Customers. Customer resource access is also typically isolated from other Customers, although specific configurations may allow general resources to be shared by multiple Customers with each Customer having permissions to access its own Customer-specific segments of the resource data.

Application Administrator: The Application Administrator monitors and manages applications for the PaaS Customer. Unlike the System Administrator, who is employed by the Paas Provider, the Application Administrator works for the PaaS Customer and is only responsible for the applications and resources immediately available to that Customer.

End-user: In some cases the PaaS Customer is running an application that is offering a service accessible to public consumers. End-users access a URL to make use of the service and invoke the PaaS Application through the URL to achieve some desired result. End-users may know nothing about the PaaS system that is hosting the application, or the circumstances in which the application is running.

Other terminology:

We adopt the following definition for an application in the context of PaaS:

PaaS Application: A discrete software artifact containing domain-specific code that can be uploaded to and deployed on the PaaS environment by a PaaS Customer. The artifact may consume PaaS resources and be distributed across multiple JVM instances according to QoS settings and/or an SLA. Depending on its terms of use, a PaaS application may subsequently be deployed on the PaaS environment by potentially any number of other PaaS Customers.

Tenant: Since in the model described here a PaaS Customer corresponds to an isolation domain, we will use the term "Tenant" to avoid misunderstandings with other uses of the word "customer" in the business context.

Application Developer: We will use term Application Developer to denote an application developer in the common sense. In the traditional Java EE terminology, this role is split between Application Component Provider and Application Assembler.

2. Examples

2.1 Example 1

Acme has been building and selling Java EE containers for years but has decided to become a PaaS Product Provider and build and sell a PaaS product that makes use of their Java EE container. They release their AcmePaaS product and sell it to Cloudify Yourself Systems. Cloudify Yourself is a PaaS Provider that has purchased a roomful of hardware and plans to sell resource usage to customers. Cloudify Yourself hires Cindy Comguru to act as their System Administrator and Bob Neatly to act as their PaaS Account Manager. Cindy sets up the PaaS software on their hardware and installs some databases that can be accessed from within the PaaS software.

MomAndPop Widgets approaches Cloudify Yourself wanting to run their public web application in the cloud. They sign up as the first PaaS Customer to use Cloudify Yourself and are immediately given account credentials. They have an in-house Application Developer who takes their existing web application and makes any necessary changes to it to prepare it to be deployed to the Cloudify Yourself PaaS environment as a PaaS Application. The developer then logs into the Cloudify Yourself portal, using the credentials given to MomAndPop Widgets when they first signed up, and acts as the Application Submitter by taking his created application and uploading it to the Cloudify Yourself system. Immediately afterwards, he acts as the Deployer in provisioning and deploying the application to a set of instances in the PaaS environment.

Soon after, an End-user surfing the MomAndPop web site decides to buy a widget and places an order that goes through the PaaS Application and gets filled by MomAndPop.

2.2 Example 2

Nile Systems has acquired a lot of hardware to handle their peak period loads, but which sits dormant the rest of the year. They already had a software layer that used a Java EE server and found that with a little effort they were able to turn it into a full-fledged PaaS layer. As a PaaS Provider (using a product they developed internally) they devised a business around allowing external consumers to use their excess hardware through a PaaS environment. One of the system administrators in their IT division was given the task of System Administrator and a previous sales engineer was assigned to be the PaaS Account Manager to maintain the customer accounts.

Joe Programmer wanted to explore a new venture that he was looking at so he signed up with Nile for an account. Once approved by the Account Manager and given login permissions, he was able to write a simple app, upload it and deploy it to the Nile PaaS environment. With PaaS Customer credentials Joe was able to be the Application Developer, Application Submitter, and Deployer all in one and to get his application up and running very quickly to see what it looked like.

2.3 Example 3

SimplyCRM, a vendor of a CRM product, decides to enter the software-on-demand space. Steve, an Application Developer at SimplyCRM, creates a version of the company's flagship CRM product packaged as a PaaS Application.

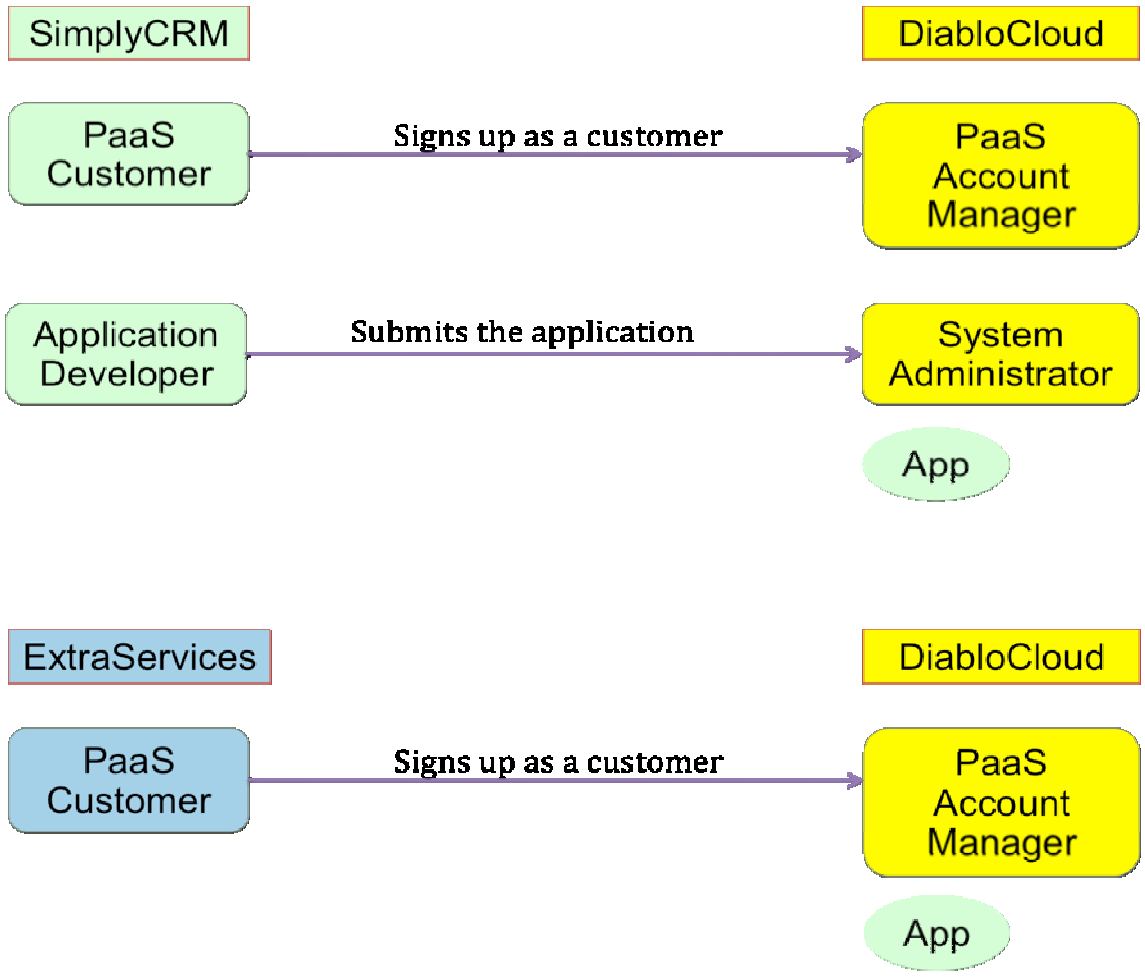
Once the application has been tested and is cleared for production use, Steve hands off the application to Sandra, an employee at SimplyCRM who is the point of contact with the PaaS Providers the company intends to partner with. Sandra signs up SimplyCRM as a PaaS Customer of DiabloCloud, a PaaS Provider. As part of the deal, the two companies agree on the terms under which usage of the SimplyCRM application will be billed to DiabloCloud Customers. Sandra then submits the application on the DiabloCloud infrastructure, acting as the Application Submitter. She then interacts with Dean, DiabloCloud's System Administrator, to publish the application so that other PaaS Customers can find it. As the Application Submitter, Sandra is responsible for insuring that the necessary artifacts and metadata for customization and configuration are available so that other Customers can reconfigure and customize the application when they deploy it for their own use (and/or use by their own end users).

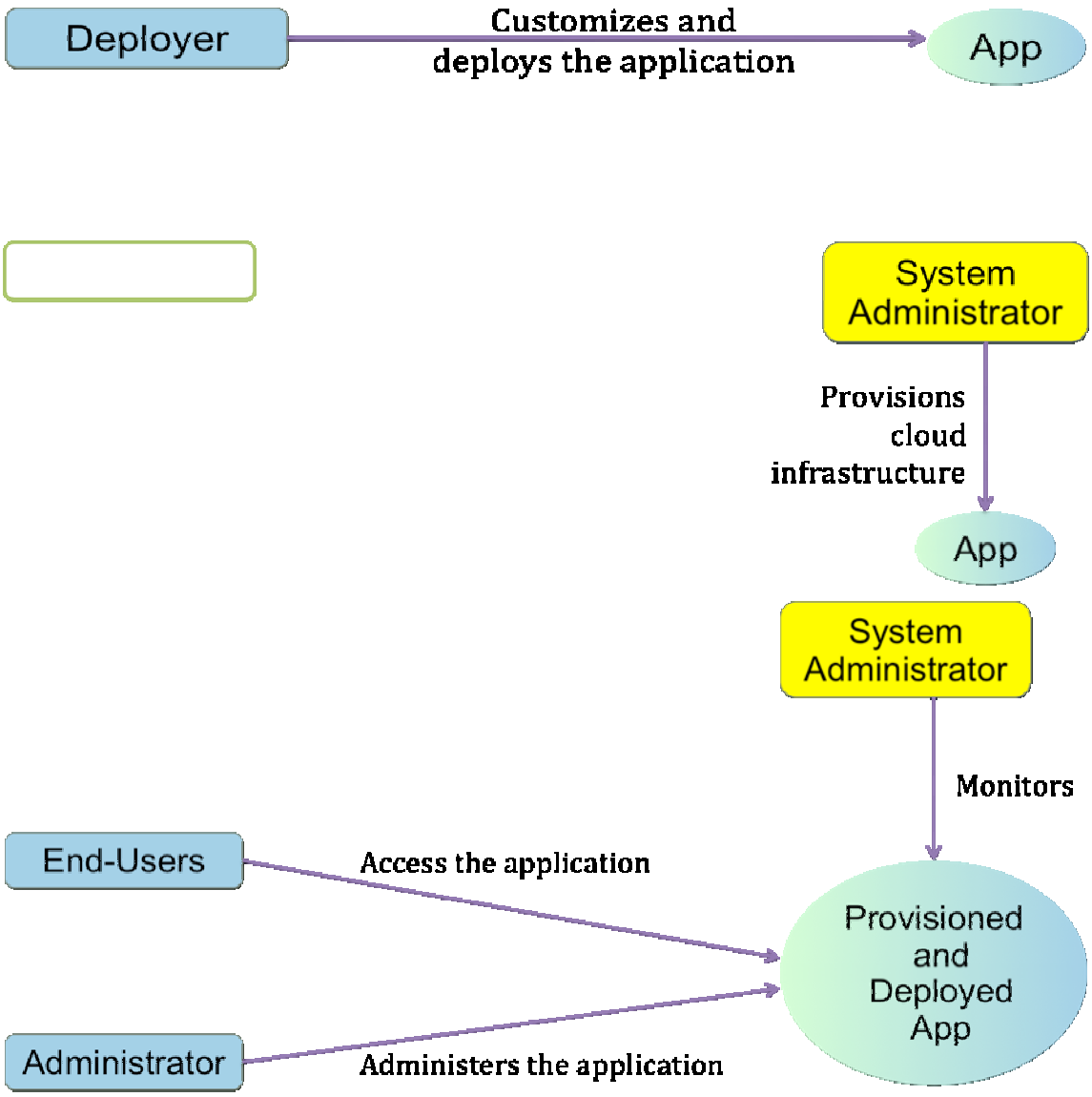
Some time later, ExtraServices Inc., a consultancy, discovers SimplyCRM 's PaaS Application offering on top of DiabloCloud. Wishing to move to an on-demand CRM system, ExtraServices Inc. signs up as a PaaS Customer of DiabloCloud, interacting

with their PaaS Account Manager. Ed, ExtraServices' Deployer, works with DiabloCloud's System Administrator Danny to ensure that the SimplyCRM application is customized and deployed to address the needs of ExtraServices. This includes making sure that the expected ExtraServices Inc. End-Users of the application have proper credentials to access it. Danny (System Administrator) verifies the application has been provisioned and configured to meet ExtraServices Inc.'s needs.

Finally, Erika, a End-User at ExtraServices Inc., starts using the SimplyCRM application running on the DiabloCloud infrastructure. DiabloCloud will meter ExtraServices Inc.' use of cloud resources in general and of the SimplyCRM application. It will then bill ExtraServices Inc. for their usage and, if properly authorized by SimplyCRM, for any costs associated with the ExtraServices Inc.'s usage of the SimplyCRM application.

Walkthrough of Example 3





3. Description of the Model

The model, illustrated by the examples above, and which we propose for Java EE 7, can be summarized as follows: single application submission, one or more Tenants, one or more application instances per Tenant (for example, in a cluster or other HA configuration).

In this model, an Application Developer creates a PaaS Application. The Application Submitter then uploads the application to a PaaS environment on behalf of a PaaS Customer (a Tenant). The necessary per-Tenant configuration is created, possibly including some customizations provided by the Deployer. Then any number of

application instances belonging to the Tenant are provisioned and started, as determined by the Deployer in accordance with the Customer's SLA. At this point, End-Users associated with or otherwise authorized by the Tenant can access the application. An application instance (to reiterate, identified here by an application classloader within a JVM) will only ever receive requests addressed to the Tenant for which it was deployed.

Note that "submission" here refers to an Application Submitter uploading the artifacts that comprise an application to a PaaS Provider. Application submission should not be confused with the act of deploying such artifacts on an instance (possibly clustered) of an application server. In the more advanced case—illustrated by Example 3 and described below—in which a limited form of SaaS is supported, application submission additionally involves the specification of information related to ways in which the application may be (re)configured for additional tenets and options provided for further customization. Deployment is the second step in the workflow for configuring the application on behalf of its tenant in preparation for running it in the cloud.

"Instance" should be taken to mean an instance of the application, as exemplified by an application class loader managed by some container code inside a Java Virtual Machine.

Multi-tenant Use of the Application:

The scenario illustrated by Example 3 is a limited form of SaaS. An important distinction, however, is that while the application has been deployed for multiple tenants, the runtime *instances* of the application are never themselves multi-tenant.

In this more advanced scenario, an Application Developer creates an application, the Application Submitter then submits the application, provides metadata and/or other artifacts that provide for tenant-specific (re)configuration and customization, and the Deployer then deploys it to a PaaS environment. Subsequently, a different PaaS Customer (a different Tenant) signs up with the PaaS Provider to access the application, interacting with the PaaS Account Manager. A Deployer for that particular Tenant works with the System Administrator to configure the application to provide all the desired customizations for that Tenant. Then any number of application instances belonging to that Tenant are provisioned and started. At this point, End-Users associated with or otherwise authorized by that Tenant can access the application. At any time that Tenant's Application Administrator can monitor the application and, if needed, obtain more resources from the PaaS Provider's System Administrator.

An Application may declare itself to be enabled for use by multiple tenants ("Multi-tenant Enabled") by setting a flag (name TBD) in the cloud deployment descriptor (provisionally named "cloud.xml").

ISSUE: We will need to scope out across the platform JSRs what is entailed by being Multi-tenant Enabled. Assuming that the application source is not available to

the tenants, this will entail externalization of configuration information beyond what we currently support. For example, it is desirable that configuration information be provided for customization of the branding of the application for the tenant, for customization of the UI, and for customization of resources to be used by the application.

The tool used to submit an application must allow specifying cloud deployment information for applications that do not contain a cloud deployment descriptor. The tool must also allow Application Submitters to make a submitted application private to a Tenant or to designate that it is enabled for use by multiple tenants and to specify the options that are available for tenant-specific (re)configuration.

While in the multi-tenant scenario, instances of the application proper continue to be isolated from each other, the configuration of the application for the Tenant must guarantee that any resources accessed by the application are isolated as well, i.e., that any Tenant can only access resources that are identified with that Tenant's identity. (This may included shared resources, as described below.)

Note that it must be possible to add Tenants to an already submitted application without resubmitting the application or otherwise affecting the operations of any existing Tenants of the application.

In general, Tenant identity is managed by a combination of the Java EE container(s) and any number of resource managers. For example, a servlet container may use a virtual server facility to map incoming requests to a Tenant. Application code cannot modify the Tenant identity, nor can it establish or relinquish the association of an instance or a thread within the JVM with a specific Tenant. At runtime, the container will make an identifier representing the identity of the current Tenant available to any Multi-tenant Enabled application in JNDI under the name "java:comp/tenantId". Applications that are not enabled for use by multiple tenants will not have access to a Tenant identifier. Containers must pass the same Tenant identifier to all application instances deployed for a given Tenant. Furthermore, Tenant identifiers must be unique across all tenants of a PaaS Provider.

ISSUE: We need to place some restrictions on allowed Tenant identifiers, since application code may use them to name e.g. files or other resources. A couple of options: (a) Java identifiers of bounded length, (b) UUIDs.

In order to remain compatible with previous versions of Java EE, Java EE compatible PaaS Provider products must be capable of running all Java EE applications, including ones that are not Multi-Tenant Enabled. Such applications include, for example, CTS test applications. To support the use of applications that are not Multi-tenant Enabled, products must include a special default Tenant (or "tenant zero") capable of running all Java EE applications. In production deployments, access to the default Tenant may, and likely will, be restricted for security reasons.

For correct operation in multi-tenant environments, Multi-tenant Enabled Applications may be restricted in their use of connection APIs that require a username/password combination, or, possibly more generally, of any connection factory API class. As a general rule, the container should be in charge of establishing connections to resource managers using the appropriate authenticators, as well as any other connection parameters it deems necessary. Applications may mark certain resources as being shared and/or under application control in the cloud descriptor, then manually control the connections. Security or service-level considerations may lead PaaS Providers to reject such applications. In order to enable the compatibility testing of products, Tenant zero must be able to deploy any Multi-tenant enabled Applications.

There may be additional restrictions on Multi-tenant Enabled Applications. Nevertheless, we expect the vast majority of the Java EE programming model to be unchanged from prior releases, the differences being mostly in the resource acquisition area and possibly security. The introduction of connection-less versions of existing APIs may further reduce the perceived restrictions on application code.

Resource managers:

We distinguish resource managers according to tenant awareness:

- Tenant-aware resource managers work with the container to obtain a Tenant's identity and use it appropriately;
- Tenant-unaware resource managers require the container to configure them at provisioning time with Tenant-specific information.

For example:

- A JPA provider for a single persistence unit, with a shared database and shared schema could be implemented as a Tenant-aware resource manager;
- A JDBC provider whose underlying relational database does not support functionality similar to the Oracle-specific Virtual Private Database will typically be Tenant-unaware.

From a container perspective, a single Tenant-aware resource manager configuration may be bound to application instances allocated to different Tenants. On the other hand, in the case of a Tenant-unaware resource manager, the container will need to create multiple resource definitions (presumably one per Tenant) out of a single resource reference contained in the application, then provision the appropriate definition to application instances for that Tenant. Instances of resource manager classes are not shared across tenants.

Other issues:

The Java EE 7 specification will not require that application instances for different Tenants be realized as separate processes (JVMs). While this will be the most likely implementation choice given the current state of application isolation in Java SE, it is

possible that new developments in SE 8 or later will enable products to isolate application instances more effectively.

ISSUE: The management APIs, and in particular JSR-77, may be required only to work for Tenant zero, in which case they must show information about all applications deployed for Tenant zero. Alternatively, each Tenant could have its own instances of the JSR-77 interfaces.

ISSUE: We need to determine what characteristics, if any, determine an application to be ” PaaS-Enabled ”. (Conversely, should we expect a Java EE application to “just work” if it is not Multi-tenant Enabled?)

ISSUE: For the purpose of compatibility, we may declare support for applications that are PaaS-Enabled or for applications that are Multi-tenant Enabled as being optional in the specification, so that products will only have to comply with the additional requirements imposed for the support of such applications if they support such PaaS usages.

ISSUE: should we require products that are not Multi-tenant Enabled to reject applications that are Multi-tenant Enabled, or should they be allowed to ignore deployment information they do not understand, such as the cloud descriptor?

ISSUE: we need to define what SPIs can be expected to work and how they behave in the presence of multitenancy. Currently, Java EE defines the following SPIs: Connectors, JAAS, JASPIC, JTA, JPA, JavaMail.

4. References

[1] A NIST Notional Definition of Cloud Computing, available at http://csrc.**nist**.gov/groups/SNS/cloud-computing/cloud-def-v15.doc