About Dasein Cloud

Dasein Cloud is a Java library that provides a wide abstraction for Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) cloud computing APIs. The sole objective of Dasein Cloud is to enable a Java programmer to write code once, compile it once, and have it be able to operate against many different clouds—to even add new clouds into the mix post-deployment. Furthermore, this write-once, run against any cloud philosophy must not be accomplished through a least-common denominator approach to abstraction.

In this document:

- Structure
- Dasein Cloud Applications
- Cloud Provider Implementations

Structure

Dasein Cloud is organized into a core project (dasein-cloud-core), a suite of unit tests (dasein-cloud-test), and any number of cloud-specific implementations. As a developer writing applications that leverage Dasein Cloud, the only project you should ever care about is dasein-cloud-core. Example 1 shows how to list servers in any cloud.

In this example, I've hidden the basic configuration information. You can pull the configuration information from command-line arguments, configuration files, a database, or even a JNDI directory. The actual Java code, however, will not vary from cloud to cloud.

```
import org.dasein.cloud.CloudException;
import org.dasein.cloud.CloudProvider;
import org.dasein.cloud.InternalException;
import org.dasein.cloud.compute.ComputeServices;
import org.dasein.cloud.compute.VirtualMachine;
import org.dasein.cloud.compute.VirtualMachineSupport;

public class ServerLister {
    static public void main(String ... args) {
        CloudProvider provider = getCloudProvider(args);

        // does this cloud provider support general compute services?
        ComputeServices compute = provider.getComputeServices();

        if( compute == null ) {
            System.out.println(provider.getName() + ": no compute
support");
            return;
        }
        // of the supported compute services, are VMs supported?
        if( !compute.hasVirtualMachineSupport() ) {
            System.out.println(provider.getName() + ": no VM support");
            return;
        }
        // list the virtual machines
        VirtualMachineSupport vms = compute.getVirtualMachineSupport();

        for( VirtualMachine vm : vms.list() ) {
            System.out.println(vm.getName());
        }
    }

    static public CloudProvider getCloudProvider(String ... args) {
      // more on constructing cloud providers later
    }
}
```

Example 1. Basic code for listing servers in any cloud with Dasein Cloud

This code will work against any cloud, even clouds that don't provide virtual machine services. You don't need to know ahead of time what services a given cloud supports. You don't even need to know what clouds you intend to support when you write your software.

The first part of the code is logic to make sure that the cloud is a compute cloud with virtual machine support. Depending on the type of application you are building, you can use this meta-data exposure of cloud capabilities to ignore non-compute clouds or provide different menus for different clouds based on real-time discovery of cloud capabilities.

The next part of the code fetches VM objects from your cloud provider and prints out user-friendly names. Though not all clouds have the concept of user-friendly names, your VirtualMachine object representation is guaranteed to have something it represents and the virtual machine name. You don't need to worry about the nuances of how each cloud represents a VM.

Versioning

Dasein Cloud is versioned in the form of [lexical year-month of major release].[point release]. The current major release, for example, is 2012.04. While every component of Dasein Cloud has independent point releases, they are all binary compatible with the all of the other components of the same major release. Non-compatible changes to dasein-cloud-core drive a new major release.

For example, the current release version of dasein-cloud-core is 2012.04.1. It does not introduce any changes over 2012.04 that break existing code. You can safely plug it into your production deployments without worrying about other dependencies—not your code on Dasein Cloud, not other Dasein Cloud components on dasein-cloud-core, and not dasein-cloud-core's dependencies on other third-party libraries.

Similarly, the project team may release point releases for different clouds at different points in time during the life of a major build. Reasons for point releases include not only bug fixes, but also the addition of new functionality in existing clouds. For example, dasein-cloud-nova 2012.04.1 introduced relational database support in the HP cloud in accordance with the dasein-cloud-core 2012.04 model for relational databases. If your code is written properly, you can replace dasein-cloud-nova 2012.04 with dasein-cloud-nova 2012.04.1 and your code will immediately and automatically discover that HP now supports relational databases.

Because of this number schema, you may have core on 2012.04.1, AWS on 2012.04.5, OpenStack on 2012.04.1, and vSphere on 2012.04. As long as they all have the same major version, you don't have to worry any further about compatibility or upgrading within a point release.

Major releases break compatibility. Sometimes upgrading to a new major release is nothing more than a re-compile of your existing code. Sometimes it doesn't even require that much, since often a new major release will introduce new concepts on which your code is

not yet dependent. In general, however, a new major release will address incompatibilities that your code will have to address.

Building Against Dasein Cloud

The first step to writing an application against Dasein Cloud is including it in your project dependencies. Dasein Cloud publishes to Maven Central, so you can include Dasein Cloud in your Maven or Gradle project dependencies without manually downloading anything. Example 2 shows how to include Dasein Cloud in a Maven pom.xml.

```
<dependency>
    <groupId>org.dasein</groupId>
    <artifactId>dasein-cloud-core</artifactId>
    <version>2012.04.1</version>
    <scope>compile</scope>
    <optional>false</optional>
</dependency>
```

Example 2: Maven dependency declaration for Dasein Cloud

That's the only entry you need for compile-time dependency management. However, you are almost certainly going to need one or more runtime entries for the clouds you wish to target. Example 3 shows a runtime dependency on the AWS implementation.

```
<dependency>
    <groupId>org.dasein</groupId>
    <artifactId>dasein-cloud-aws</artifactId>
    <version>2012.04.1</version>
    <scope>runtime</scope>
    <optional>false</optional>
</dependency>
```

Example 3: A runtime dependency on AWS makes sure maven packages the AWS support for deployment

We currently do a terrible job documenting what the latest release is. The best way currently is to navigate to *http://repo1.maven.org/maven2/org/dasein/* and check out the versions in Maven central.

Once you have defined your dependencies, you should be able to write code against Dasein Cloud, compile it, and deploy it to your application environments.

Implementing For Dasein Cloud

When you want to implement Dasein Cloud to support a specific cloud, you start out including the dependency on dasein-cloud-core just as you do when writing an application to run against Dasein Cloud. The similarities end there. The full list of steps are:

- Include the dasein-cloud-core compile time dependency
- Include the dasein-cloud-test test time dependency
- Added the testing profile to your Maven pom.xml and your settings.xml
- Implement whatever classes you need to support your target cloud
- Run the Dasein Cloud test suite
- (Optional) Submit to the Dasein Cloud team for Open Source publication

You won't include other implementations as runtime dependencies. They won't be relevant to you (unless, for some reason, you are extending the behavior of an existing implementation).

On the other hand, you will include a dependency on the Dasein Cloud test suite. The Dasein Cloud test suite will ensure that your implementation of the Dasein Cloud API conforms to the expected behavior for Dasein Cloud implementations. You will also need to add testing properties to your pom.xml like Example 4.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.6</version>
    <configuration>
        <systemProperties>
            <property>
                <name>endpoint</name>
                <value>${endpoint}</value>
             </property>
             <property>
                <name>accountNumber</name>
                <value>${accountNumber}</value>
             </property>
             <property>
                <name>apiSharedKey</name>
                <value>${apiSharedKey}</value>
             </property>
             <property>
                <name>x509Cert</name>
                <value>${x509Cert}</value>
             </property>
             <property>
                <name>x509Key</name>
                <value>${x509Key}</value>
             </property>
             <property>
                <name>apiSecretKey</name>
                <value>${apiSecretKey}</value>
             </property>
             <property>
                <name>cloudName</name>
                <value>${cloudName}</value>
             </property>
             <property>
                <name>providerName</name>
                <value>${providerName}</value>
             </property>
             <property>
                <name>regionId</name>
                <value>${regionId}</value>
             </property>
             <property>
                <name>test.region</name>
                <value>${test.region}</value>
             </property>
             <property>
                <name>test.dataCenter</name>
                <value>${test.dataCenter}</value>
             </property>
             <property>
                <name>test.machineImage</name>
                <value>${test.machineImage}</value>
```

```
                </property>
                <property>
                   <name>test.product</name>
                   <value>${test.product}</value>
                </property>
            </systemProperties>
            <includes>
                <include>**/MyTestSuite.java</include>
            </includes>
        </configuration>
    </plugin>
```

Example 4: Test suite configuration properties

The last line is important because it specifically references a class you must write. This is your test suite implementation class, and it's very simple as shown in Example 5.

```
package org.dasein.cloud.mycloud;

import junit.framework.Test;
import org.dasein.cloud.test.ComprehensiveTestSuite;
import org.dasein.cloud.test.TestConfigurationException;

public class MyTestSuite {
    static public Test suite() throws TestConfigurationException {
        return new ComprehensiveTestSuite(MyCloud.class);
    }
}
```

Example 5: A test suite implementation that binds the Dasein Cloud test suite to your cloud

The referenced `MyCloud.class` class is a class you write to implement the Dasein Cloud `CloudProvider` interface. We'll cover that in more detail in a bit.

The best way to get started is simply to copy the pom.xml from an existing cloud provider. To run the tests, you will need to create a profile in your ~/.m2/settings.xml (or whatever the Windows equivalent is). Example 6 shows a sample profile.

```
<profile>
  <id>mycloud</id>
  <properties>
    <endpoint>https://api.endpoint.mycloud.com/</endpoint>
    <accountNumber>accountNumber</accountNumber>
    <apiSharedKey>apiSharedKey</apiSharedKey>
    <apiSecretKey>apiSecretKey</apiSecretKey>
    <cloudName>MyCloud</cloudName>
    <providerName>PlatformName</providerName>
    <regionId>regionId</regionId>
    <test.dataCenter>dataCenterId</test.dataCenter>
    <test.region>regionId (again, don't ask why)</test.region>
    <test.machineImage>machineImageId</test.machineImage>
    <test.product>product size, flavor, or whatever</test.product>
  </properties>
</profile>
```

Example 6: A sample profile entry for your settings.xml so the tests can connect to the cloud

The final step is the implementation of the actual Dasein Cloud interfaces for your cloud and then running the tests. To run the tests, execute the maven command:

```
mvn -Pmycloud clean test
```

In this example, the "mycloud" is the ID of the profile you specified in your Maven settings.xml file as described in Example 6. These tests take a long, long time to run.

The Object Model

The core of the Dasein Cloud object model is a `CloudProvider` implementation that provides access to various kinds of optional services, each of which will support one or more cloud resources. The current list of services are:

- Data center services
- Compute services
- Network services
- Platform services
- Admin Services
- Identity Services
- Storage Services

Any given cloud may or may not implement the above services with the exception of data center services. Every cloud is required to implement data center services.

You can check if a cloud supports any particular service through the hasXXXServices() method. For example, the following code determines if your cloud has network services:

```
if( provider.hasNetworkServices() ) { /* perform some networking action */ }
```

Similarly, each service may provide support for none, any, or all resources supported by Dasein Cloud. Just as you can programmatically discover what services are supported, so can you discover what resources are supported:

```
if( provider.hasNetworkServices() ) {
  NetworkServices svc = provider.getNetworkServices();

  if( svc.hasFirewallSupport() ) {
    // execute firewall interaction
  }
}
```

If you are writing code that is meant to be cross-cloud, you should always wrap any logic interacting with specific resources with meta-data checks and handle the lack of support for a specific resource appropriately. In the above code, for example, we might be attempting to set up networking so that our testing tools can talk to a monitoring tool we have installed on a target VM. If the cloud has firewall (aka security group) support, we need to establish the proper firewall rules to allow that kind of ingress traffic. We probably also want to check for IP address support so that we can validate that there's a routable IP address to the VM.

Understanding Data Center Services

The Data center services form the one set of services that are guaranteed to be present in every cloud. The `DataCenterServices` interface defines the contract, and that contract divides a cloud into regions and data centers. Most clouds do not actually have concepts of both data centers and regions, but a Dasein Cloud implementation will craft those divisions for you so that you don't have to worry about the nuances of each cloud.

A region is exactly like an Amazon region. It represents a single jurisdiction and a high-level single point of failure. If a cloud supports more than one region, it is claiming that resources may not be shared across the regions and there are zero common points of failure among the regions. Every cloud has at least one region. Many clouds have more than one.

A data center is exactly like an Amazon availability zone. It represents a common local geography and a medium-level single point of failure similar to that of a physical data center. Across data centers, you may achieve some level of redundancy—though nowhere near the redundancy you see across regions. Within a data center, resources may have privileged communication with each other and may share private network topologies. Every region is guaranteed to have at least one data center. Most clouds ultimately have exactly one data center.

Some cloud resources are regionally bound and some are data center bound. In some cases, Dasein Cloud enforces that binding. In other cases, Dasein Cloud provides meta-data that enables you to determine how the cloud binds different kinds of resources (for example, load balancers are regionally bound in some clouds and data center bound in others).

Dasein Cloud Applications

At this point, you should be fully configured to begin writing code. The starting place for all Dasein Cloud programming is the configuration of a `ProviderContext` for a specific cloud. In other words, to interact with Dasein Cloud, you will need to know the class name of a class that implementations the `CloudProvider` Java interface for that cloud and configure it using a `ProviderContext` object that you assemble.

Configuration and Connecting

I glossed over the configuration details earlier in Example 1:

```
static public CloudProvider getCloudProvider(String ... args) {
   // more on constructing cloud providers later
 }
```

Example 7 shows what reallly should happen in that method.

```
static public CloudProvider getCloudProvider(String ... args)
throws Exception { // this can actually throw a number of exceptions
    CloudProvider provider = Class.forName(args[0]).newInstance();
    ProviderContext ctx = new ProviderContext();
    Properties props = new Properties();

    props.load(this.getClass().getResourceAsStream(args[1]));

    publicKey = props.getProperty("apiSharedKey");
    privateKey = props.getProperty("apiSecretKey");
    if( publicKey != null && privateKey != null ) {
        ctx.setAccessKeys(publicKey.getBytes(), privateKey.getBytes());
    }

    ctx.setAccountNumber(props.getProperty("accountNumber"));
    ctx.setCloudName(props.getProperty("cloudName"));
    ctx.setEndpoint(props.getProperty("endpoint"));
    ctx.setProviderName(props.getProperty("providerName"));
    ctx.setRegionId(props.getProperty("regionId"));

    publicKey = props.getProperty("x509Cert");
    privateKey = props.getProperty("x509Key");
    if( publicKey != null && privateKey != null ) {
        try {
            BufferedInputStream input = new BufferedInputStream(new
FileInputStream(publicKey));
            ByteArrayOutputStream output = new ByteArrayOutputStream();
            byte[] tmp = new byte[10240];
            int count;

            while( (count = input.read(tmp, 0, 10240)) > -1 ) {
                output.write(tmp, 0, count);
                output.flush();
            }
            ctx.setX509Cert(output.toByteArray());
            input.close();
            output.close();
            input = new BufferedInputStream(new
FileInputStream(privateKey));
            output = new ByteArrayOutputStream();
            while( (count = input.read(tmp, 0, 10240)) > -1 ) {
                output.write(tmp, 0, count);
                output.flush();
            }
            ctx.setX509Key(output.toByteArray());
            input.close();
            output.close();
        }
        catch( IOException e ) {
            fail(e.getMessage());
        }
    }
    Properties custom = new Properties();
```

```
        Enumeration<?> names = props.propertyNames();

        while( names.hasMoreElements() ) {
            String name = (String)names.nextElement();

            custom.setProperty(name, props.getProperty(name));
        }
        ctx.setCustomProperties(custom);
        provider.connect(ctx);
        return provider;
    }
```

Example 7: Configuration of a cloud provider implementation

Example 7 illustrates a generic configuration that supports any potential set of Dasein Cloud providers. Something like this is often the safest way to go. Of course, you can manually construct cloud provider references, pull them from JNDI, or handle them in other ways. The important thing about Example 7 is that, again, there is absolutely no code specific to any cloud. It will work with any cloud.

Once you call `provider.connect(ProviderContext)`, you are ready to interact with the cloud. You will to interact with one or more of the services to access support objects that describe the resources in the cloud provider. In Example 1, we saw an example of accessing the compute services to get VM support and fetch the VMs currently running in the target cloud.

The best way to see examples of Dasein Cloud in action is to have a look at the source code for the test suite from dasein-cloud-test.

Cloud Provider Implementations

Implementing Dasein Cloud to support applications that want to interact with a specific cloud involves providing an extension of the `CloudProvider` class and whatever services and resources that cloud supports. The most basic implementation must provide a `CloudProvider` extension and an implementation of the `DataCenterService` interface. Everything else is dependent on the features underlying cloud.

The `CloudProvider` Implementation

Any implementation starts with an extension of the `CloudProvider` abstract class. The best way to do this is to actually extend the `AbstractCloud` abstract class which provides some common functions that may be override in specific contexts. A very basic CloudProvider looks like Example 8.

```
public class NovaOpenStack extends AbstractCloud {

    public NovaOpenStack() { }

    @Override
    public @Nonnull String getCloudName() {
        ProviderContext ctx = getContext();
        String name = (ctx == null ? null : ctx.getCloudName());

        return (name != null ? name : "OpenStack");
    }

    @Override
    public @Nonnull NovaLocationServices getDataCenterServices() {
        return new NovaLocationServices(this);
    }

    @Override
    public @Nonnull String getProviderName() {
        ProviderContext ctx = getContext();
        String name = (ctx == null ? null : ctx.getProviderName());

        return (name != null ? name : "OpenStack");
    }

    @Override
    public @Nullable String testContext() {
        try {
            NovaMethod method = new NovaMethod(this);
            AuthenticationContext ctx = method.authenticate();

            return (ctx == null ? null : ctx.getTenantId());
        }
        catch( Throwable t ) {
            t.printStackTrace();
            return null;
        }
    }
}
```

Example 8: The most basic `CloudProvider` implementation possible

Of the above methods, only the `testContext()` method should require much in the way of explanation. It's a check to verify that the specified `ProviderContext` has valid API credentials for talking to the cloud. If valid, it returns a proper account number. If not, it returns `null`. What constitutes an account number is different for different clouds. In OpenStack, it's your tenant ID; in EC2 it's your AWS account number.

In practice, a `CloudProvider` implementation is likely to be much more complex. For one, it will provide access to many more services than simply the data center services. In

addition, it will certainly have a number of helper methods that support context-specific functions you may use throughout your code.

Services Implementations

Early, we discussed the various services you may implement with the data center services being the only required service. There is no common base class for all services, but instead a unique interface for each service with an abstract implementation that empty resource support. You will almost always want to extend the abstract implementation and not directly implement the interface. The one exception is `DataCenterServices` which has no abstract implementation.

The Service Objects

Other than data center services, all services provide access to one or more kinds of resources through support interfaces for each service. A typical service looks like the OpenStack compute service in Example 9.

```
public class NovaComputeServices extends AbstractComputeServices {
    private NovaOpenStack provider;

    public NovaComputeServices(@Nonnull NovaOpenStack provider) {
        this.provider = provider;
    }

    @Override
    public @Nonnull NovaImage getImageSupport() {
        return new NovaImage(provider);
    }

    @Override
    public @Nonnull NovaServer getVirtualMachineSupport() {
        return new NovaServer(provider);
    }

    @Override
    public @Nullable SnapshotSupport getSnapshotSupport() {
        if( provider.getProviderName().equals("HP") ) {
            return new HPVolumeSnapshot(provider);
        }
        return null;
    }

    @Override
    public @Nullable VolumeSupport getVolumeSupport() {
        if( provider.getProviderName().equals("HP") ) {
            return new HPBlockStorage(provider);
        }
        return null;
    }
}
```

Example 9: A ComputeServices implementation for OpenStack.

This service provides access to support objects for images, servers, snapshots, and block volumes. The abstract parent class provides null support implementations for unsupported resources so you can focus only on the things your cloud supports. In short, every services class you implement is likely to have only a bunch of these support getter functions and nothing more.

The Support Interfaces

The support interfaces prescribe the actual interaction with the cloud. These interfaces generally focus on a single resource (like a virtual machine) and support all the create, read, update, and delete interactions common with those resources. In addition, the support

interface will prescribe meta-data that describes the differences among the ways in which various clouds represent a given resource.

Each interface starts with an `isSubscribed()` method. The `isSubscribed()` method will return true if the credentials in the current context authenticate properly with the cloud provider and those credentials have access to the resource behind a support interface. Credentials may not have access either because no subscription has been set up to a particular cloud service or the API keys have access control restrictions placed on them. Either way, an application knows whether or not to even bother attempting to interact with a support interface implementation based on the response from `isSubscribed()`.

Each interface then often prescribes:

- A getXXX() method to fetch individual instances of the supported resource
- Other getXXX() methods to fetch individual dependent objects behind this resource
- One or more list() or search() methods for listing or searching against a supported resource and any dependent objects
- One or more ways of provisioning new instances of a resource
- One or more ways of disposing of instances of a resource
- Meta-data for describing cloud-specific behaviors

Dasein Cloud prescribes a special exception called OperationNotSupportedException that should be thrown when a client triggers a method is not supported by your cloud and cannot safely be a “no-op”. If a method can safely be a no-op, it’s best to leave it as such. But if a client really should have expected something to happen, then you should throw the exception.