Chapter 1

Implementation/realization - cloudml-engine

The envision and design of CloudML is implemented as a proof-of-concept project *cloudml-engine*. The project is split into four different modules FIG. 1.1. Each module serves a logical task of CloudML. This chapter will go into depths of technologies and structures of the implementation.

1.1 Technologies

Cloudml-engine is based on state-of-the-art technologies that appeals to the academic community. Technologies chosen for *cloudml-engine* are not of great importance to the concept of CloudML itself, but it still important to understand which technologies were chosen, what close alternatives exists and why they were chosen.

Language *Cloudml-engine* is written in Scala, a multi-paradigm JVM based programming language. This language was chosen because JVM is a popular platform, and then especially Java. Scala is compatible with Java and Java can interact with libraries written in Scala as well. The reason not to use plain Java was because Scala is an appealing state-of-the-art language that emphasizes on functional programming which is leveraged in the implementation. Scala also has a built in system for Actors model [3] which is utilized in the implementation.

Lexical format For the lexical representation of CloudML *JavaScript Object Notation* (JSON) was chosen. JSON is a web-service friendly, human-readable data interchange format and an alternative to XML. This format was chosen because of popularity in the cloud community source and its usage area as data transmit format between servers and web applications. This means *cloudml-engine* can be extended to work as a RESTFul web-service server.

The JSON format is parsed in Scala using the lift-json parser which provides implicit mapping to Scala case-classes. This library is part of the lift framework, but can be included as an external component without additional lift-specific dependencies. GSON was considered as an alternative, but mapping to Scala case-classes was not as fluent compared to lift-json.

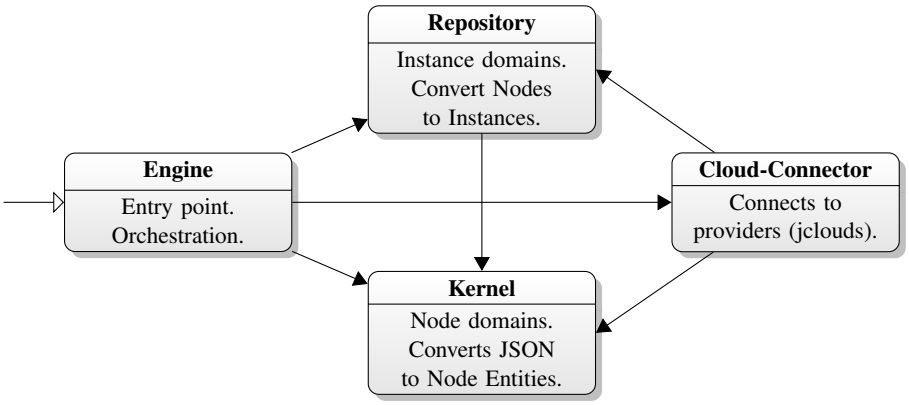

Figure 1.1: Architecture of cloudml-engine

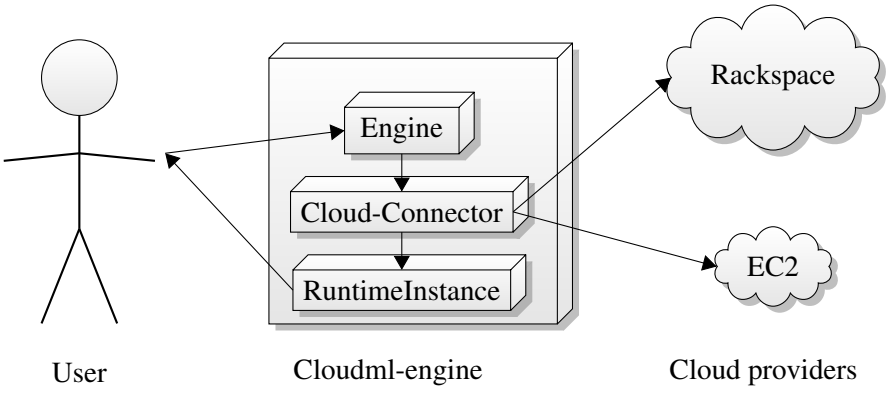

Figure 1.2: Usage flow in cloudml-engine

```
<repositories>
 <repository>
  <id>cloudml-engine</id>
  <url>
   https://repository-eirikb.forge.cloudbees.com/release
  </url>
 </repository>
</repositories>
<dependencies>
 <dependency>
  <groupId>no.sintef</groupId>
  <artifactId>engine</artifactId>
  <version>0.1</version>
 </dependency>
</dependencies>
```

Figure 1.3: Example Maven configureation section to include cloudml-engine

```
import no.sintef.cloudml.engine.Engine
...
val runtimeInstances = Engine(account, List(template))
```

Figure 1.4: Example Scala callout to *cloudml-engine*

Automatic build system There are two main methods used to build Scala programs, either using a Scala-specific tool called *Scala Build Tool* (SBT) or a more general tool called Maven. For *cloudml-engine* to have an academic appeal it were essential to choose the technology with most closeness to Java, hence Maven was chosen. Maven support modules which were used to split *cloudml-engine* into the appropriate modules as shown in FIG. 1.1. The dependency system in Maven between modules is used to match the dependencies outlined in FIG. 1.1. Parts of a dependency reference in a Maven configuration can be seen in FIG. 1.3.

Cloud connection The bridge between *cloudml-engine* and cloud providers is an important aspect of the application, and as a requirement it was important to use an existing library to achieve this connection. Some libraries have already been mentioned in the *APIs* section in CHAP. **??**, of these only *jclouds* is based on Java-technologies and therefore suites *cloudml-engine*. Jclouds uses Maven for building as well, and is part of Maven central which makes it possible to add jclouds directly as a module dependency. Jclouds contains a template system which is used through code directly, this is utilized to map CloudML templates to jclouds templates.

Distribution *Cloudml-engine* is not just a proof-of-concept for the sake of conceptual assurance, but it is also a running, functional library which can be used by anyone for testing or considerations. Beside the source repository [1] the library is deployed to a remote repository [2] as a Maven module. This repository is provided by CloudBees, how to include the library is viewable in FIG. 1.3.

Actors As mentioned earlier *cloudml-engine* utilizes the actors model through Scala, this approach is used to achieve asynchronous provisioning. This is important as provisioning can consume up to minutes for each instance. Beside the standard model provided by Scala *cloudml-engine* uses a callback-based pattern to inform users of the library when instance statues are updated and properties are added.

1.2 Modules and application flow

Cloudml-engine is divided into four main modules FIG. 1.1. This is to distribute workload and divide *cloudml-engine* into logical parts for each task.

Engine. The main entry point to the application, this is a Scala Object used to initialize provisioning. Interaction between `user` and `Engine` is visible in FIG. 1.2 where the user will initialize provisioning by calling `Engine`. `Engine` will also do orchestration between the three other modules as shown in FIG. 1.1. Since `Cloud-Connector` is managed by `Engine` other actions against instances are done through `Engine`. The first versions of *cloudml-engine* did not use `Engine` as orchestrator but instead relied on each module to be a sequential step, this proved to be harder to maintain and also introduced cyclic dependencies.

Kernel `Kernel` contains CloudML specific entities such as Node and Template. The logical task of `Kernel` is to map JSON formatted strings to `Templates` including `Nodes`. This is some of the core parts of the DSL, hence it is called *`Kernel`*. `Accounts` are separate parts that are parsed equally as `Templates`, but by another method call. All this is transparent for users as all data will be provided directly to `Engine` which will handle the task of calling `Kernel` correctly.

Repository. Has `Instance` entities, these are equivalent to `Nodes` in `Kernel`, but are specific for provisioning. Repository will do a mapping from *Nodes* (including *Template*) to *Instances*. Future versions of `Repository` will also do some logical superficial validation against *Node* properties, for instance at the writing moment it is not possible to demand LoadBalancers on Rackspace for specific geographical locations.

Cloud-Connector. is the module bridging between *cloudml-engine* and providers. It does not contain any entities, and only does logical code. It is built to support several libraries and interface these. At the moment it only implements the earlier mentioned library jclouds.

Bibliography

[1] Eirik Brandtzæg. cloudml-engine, 2012.

[2] CloudBees. Cloudbees cloudml-engine repository, 2012.

[3] Philipp Haller and Martin Odersky. Actors that unify threads and events. In *Proceedings of the 9th international conference on Coordination models and languages*, COORDINATION'07, pages 171–190, Berlin, Heidelberg, 2007. Springer-Verlag.