Cloud billing service

An SOA-enabled billing service module for the cloud environment

Hans-Dieter Wehle
February 09, 2011

Cloud billing is the process of generating bills from the resource usage data using a set of predefined billing policies. The author defines a cloud billing service module enabled for a service-oriented architecture, covering both functional requirements — a quote service, conversion functions and policies, payment schemes, and user identification — and the non-functional, but essential, requirements such as security, scalability, standards, and fault tolerance.

The purpose of having a cloud-optimized billing component is to be able to provide an interface for generating usage bills. Cloud billing is the process of generating bills from the resource usage data using a set of predefined billing policies. The bill can be for real money or it can refer to a more abstract notion of exchange, depending on your individual cloud computing general policies — it doesn't matter because the billing service does not specify the use of a specific economic model.

This article provides a quick guide that looks at how to define a cloud billing service module designed for a service-oriented architecture environment. It addresses functional requirements such as quote services, conversion functions and policies, payment methods, and user identification concerns. It also covers the essential non-functional requirements like security, scalability, standards, and fault tolerance.

Parts of a cloud billing service module

First, let's illuminate the architecture of this billing module for the cloud. Then I'll explain the thinking behind billing policy and the language and syntax used. I'll also cover the rules/inference engine and the role it plays, then wind up this section with a quick discussion on workflow and conflict resolution.

The working module: Form follows function

The general cloud billing service module should provide port types to support all of your functional requirements (described in the next section):

- The billing service generates bills according to policies that it retrieves from a Billing Policies Repository.
• The bills are stored in the Bills Repository for later use by the manager.

Figure 1 shows the general architecture of the accounting and billing services.

**Figure 1. The architecture of the module**

The bills generation process is not automated and needs to be initiated by the manager. All the functionalities of the service is implemented in a web service.

Figure 2 shows the module from the technology point of view and illustrates the various sub-components and the interaction between these different technologies.

**Figure 2. The module from the technology point of view**

The billing service establishes the connection to the web server through a PHP interface and gets the record document from the Usage Record (UR) Repository.
Billing policies, billing syntax, language

A simple example of a billing policy is similar to the following statements:

- 1 CPU/h (hours of use) is equal to 1 token
- Discount 20 percent for CPU/h greater than 20h

However simple or complex you make your billing policies, there will be a time when the resource manager will want to modify them. Capturing every case is simply impossible and changing the source code each time you add new policies is, of course, not feasible.

It is important to make a clear distinction between the billing logic that computes the final bill and the billing policies that set guidelines on how to generate a bill. The billing policies are not part of the service; they constitute an external element that the billing service makes use of. They are written by the resource manager and stored in a repository. The billing service uses this repository to extract the necessary guidelines to generate the final bill.

Suppose you want to formalize the two policies I mentioned as examples. You can create something like this: (cpu_h(x) being \( x \) numbers of CPU hours and token(x) being \( x \) numbers of tokens, then

\[
\begin{align*}
(1) \# \text{bill} &= \text{token}(\text{cpu}_h(x)); \\
(2) \# \text{if} \\
& \quad \text{cpu}_h(x) > 20 \\
& \quad \text{then} \\
& \quad \text{bill} = \text{bill} \times 0.8;
\end{align*}
\]

This notation captures both of the policies; however it has two major problems:

- It is not a natural language.
- It cannot capture all the cases.

Remember, the person who sets the billing policies is not necessarily a programmer and writing in such notation can be a real headache. It is much easier and natural to express the billing policies in a more natural language than this notation.

The second problem is that it is very difficult to express a wide range of complex policies using this notation. Business rules can change fairly quickly and the billing rules need to follow. Anticipating the requirement of a future policy can only be done through the use of robust rules syntax.

There are also different types of languages used to describe the policies; each has its own benefits and drawbacks.

A natural language permits the expression of rules in plain English:
When the event is VM Assignment and the client's type is Platinum, then the cost per second is 0.0002 euros.

The natural language is easy to use and has a flat learning curve. This one is most appealing to non-saavy computer users.
The same policy description in a domain specific language (which uses the same syntax and semantic used in the specification document) looks like this:

EVENT = "VM Assignment", CLIENT_TYPE = "Platinum", RESOURCE_TYPE = "BLADE Type 4", RESOURCE_AGE > 240 * 60 * 60 (seconds), SERVICE_LEVEL = "Platinum", COST_PER_SECOND = 0.0002 (Euros, Pounds, etc.)

The domain specific language is more structured and is easier to make quick edits and changes in than the natural language.

More on forward chaining

Forward chaining or data-driven inference (based on the modus ponens inference rule: If P, then Q; P is true, then Q is true) is one of the two main methods of reasoning when using inference rules, syntactic rules or functions which take premises and return a conclusion. Forward chaining starts with available data, iterates forward attempting to extract more (new) data until it reaches a goal. (The other method is backwards chaining where you start with the goal and work backwards to see if the granular data supports the goal; both are based on the modus ponens rule but BC is considered goal-driven inference.)

One advantage of forward-chaining over backward-chaining is that the reception of new data can trigger new inferences; this makes the engine better suited to dynamic situations in which conditions are likely to change.

The rules/inference engine and its role

The rules/inference engine is the part of the system that evaluates the rules based on the system state or user input. Having policy syntax is not enough: You also need to have a mechanism to assert the validity of the policy and to evaluate the policy. This is what a rules engine is designed for.

The engine I'm describing is a forward-chaining rules engine using an enhanced implementation of the Rete algorithm. At this stage in the development, my colleagues and I are in the process of evaluating several rule engines and two have caught our attention:

- ACEL from IBM (Autonomic Computing Expression Language) was originally developed as a part of the Autonomic Computing Policy Language to describe conditions when a policy should be applied to a managed system.
- To use industry standards, you have to use the Java™ Rule Engine API (JSR94) as programming interface.

The Billing Rules are entered and modified through a simple web front end. Take a look at the structure of the rules/inference engine.
The Rete algorithm (pronounced RAY-tee or REE-tee) is an efficient pattern matching algorithm for implementing production rule systems and has evolved into the basis for many popular expert system shells. *Rete* is taken from the Latin word for "net."

The Rete algorithm sacrifices memory usage for increased speed; it beats the slower implementation of data-to-rules checking (check if data fits rule > toss rule if not, keep if it does > move to next rule > loop/iterate back to start of rules) by building a network (the "Rete" part) of nodes in which each node corresponds to a pattern in the conditional "if" part of a rule. What Rete does is store data-to-facts information in working memory. This in turn allows systems to share the data in the nodes to eliminate some kinds of redundancy; store partial matches when joining different fact types (which keeps the system from having to re-evaluate all of the facts whenever changes are introduced; get rid of memory elements efficiently when facts are removed from working memory.

The process of matching the new or existing facts against production rules is called *pattern matching*. There are a number of algorithms used for pattern matching by rules engines. The rules are stored in the production memory and the facts that the inference engine matches against the working memory.

Facts are inserted into the working memory where they are modified or retracted. A system with a large number of rules and facts may result in many rules being true for the same fact assertion; these rules are said to be in conflict. The agenda manages the execution order of these conflicting rules using a conflict resolution strategy.

**Workflow and conflict resolution strategy**

By workflow, I mean the rules precedence order when evaluating rules (which rule should be evaluated first?). By conflict resolution, I mean what rule to fire if more than one rule evaluates to true. Both those points are important and have to be present in a successful billing service module.

Next, let's jump right into a concise discussion of the functional requirements.

**Functional requirements of a billing module**

Now I'll succinctly detail the functional requirements to consider in a cloud billing module.

**The quote service**

The cloud brokering instance should support a quote service to permit users to ask for quotes before using a resource. For example, "what does it cost me to run such and such at that computing site?" The quote should not be binding and only be valid for a short period of time.
The conversion function and policies

The cloud brokering instance must implement a conversion function that would convert from a usage record to a currency following a basic conversion scheme — like 1CPU/hour cost = 1 token for instance — and negotiate the monetary value of the token depending on the economic model and the demand. The cloud brokering instance should permit the resource owner to provide it with custom conversion models to better serve its interests.

The customer (entity) identification process

The cloud brokering instance must permit the identification of the entity to be charged for the resource usage. A user might belong to different virtual organization and run several jobs; it is important in this case to identify who should pay for every job. It might also occur that a job is paid from different budgets depending on the money availability for instance.

The payment schemes functions and policies

The cloud brokering instance must support at least one payment scheme. Some examples are allowing a post-payment, pre-payment (at reservation time for instance), or continuous payment scheme.

The cloud service should have flexibility in terms of pricing the subject (which can change over time) or should have different tariffs like a usage fee, fixed fee, one-off fees, or allowances. A policy can look like this for example:

```
EVENT = "VM Assignment",
CLIENT_TYPE = "Platinum",
RESOURCE_TYPE = "BLADE Type 4",
RESOURCE_AGE < 240 * 60 * 60 (seconds),
SERVICE_LEVEL = "Platinum",
COST_PER_SECOND = 0.0002 (Euros)
```

```
EVENT = "ONE_OFF SERVICE 1",
RESOURCE_TYPE = "BLADE Type 4",
ONE_OFF_COST = 1
```

Non-functional, but essential, requirements

To address non-functional, essential requirements for a cloud billing module, the module should possess at minimum:

- The ability to make its transactions secure.
- Role-based authentication.
- Audit trails to permit the analysis of activity.

In particular, you must take steps to prevent unauthorized clients or impostors from sending events or from triggering the billing process without the right authentication or permission.

In conclusion

By detailing the anatomy of a cloud environment billing module, you've seen some of the considerations to ponder as you plan to develop your own. I've offered a vision of how the module
fits into accounting and billing services and how it fits into the entire cloud infrastructure. You saw examples of billing policies and syntax and language and taken a deeper dive into how the rules/inference engine works. I've also provided a checklist of sorts of functional requirements for a billing module and the non-function, but essential, requirements you'll also need to consider.

© Copyright IBM Corporation 2011
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)