

Enterprise Drools: Multiple Deployments of Business Rules

What could be more robust and scalable than a Drools Deployment? Two Drools deployments! This white paper outlines 3 options to deploy multiple instances of Drools Business Rules as part of your Enterprise Java application.

The problem that we are trying to solve.

Often Drools is deployed as a 'Decision Service' on it's own server. Other parts of the application make a call to the Drools service as required to evaluate business logic. This business logic is contained in Drools business rules. The non-rules parts of the application don't care how the business rules are implemented; all they know is that they call an API and get the correct result back.

We could be making a local call to the Drools Server, or a remote call via EJB / web services. If it is a remote call (running on a different server) we have the possibility of having multiple Rules (Drools) servers without the rest of the application knowing (or caring) which Drools server they are talking to.

These multiple Drools servers give an additional level of robustness. If one server fails, another is immediately available. It also allows the Drools servers to 'share the load' and handle high volumes of calls.

The problem is *how do we ensure a call to **any** Drools Server (using the same base data) yields the same result, at any given moment in time.*

Background to the Options

We've made a couple of assumptions for all the options:

- No matter how many instances of Drools we have, that they are all running the same set of Rules. Using the Rule Agent to ensure automatic deployment of rules makes this assumption realistic.
- With any synchronisation, we can quickly get into race conditions. These are hard to find bugs that might only appear if event X happens milliseconds after event Y. We'll assume that synchronisation happens in a reasonable period of time (e.g. milliseconds rather than seconds). There are techniques to guarantee even better synchronisation, but for most applications 'a couple of milliseconds' is good enough.
- It's always useful to have some 'master' record of information (e.g. the original data stored on a database). This means that you can roll back to and re-evaluate your rules from a given moment in time. Given that most Enterprise Java apps have store information in a database like Oracle or

Microsoft SQL Server, you're likely to have this option available.

- Stateful Sessions are where Drools remembers the state (for each client) between calls. In Stateless sessions, this information is deliberately forgotten by Drools once it has been returned to the caller.

Option 1 – Avoid Synchronisation by using Stateless Sessions

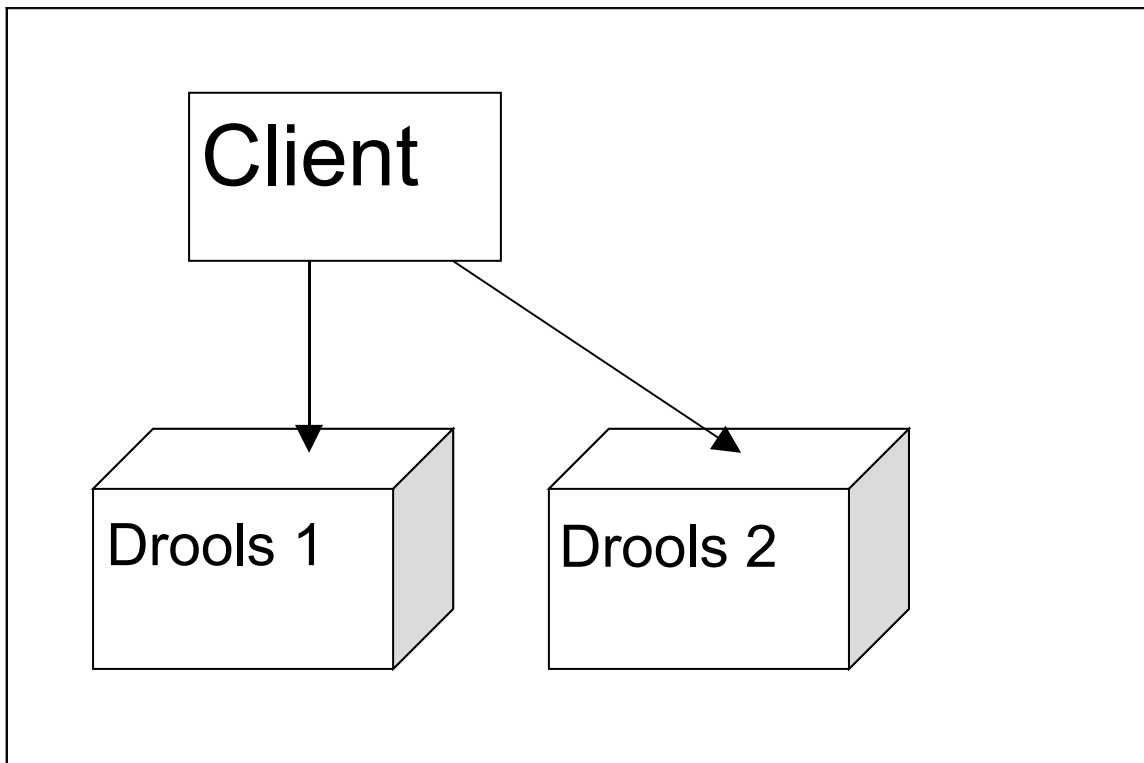
A major lesson that people have learned from using Enterprise Java Beans (EJB) is that stateless sessions work best and should be seen as the 'default option'. In a stateless scenario, the server (be it EJB or Drools) 'forgets' all session information between calls (but remembers the common business rules). **In effect we pass all the information the business rules need each and every time we call Drools.**

Unless we are talking about extremely large sets of data, this is easier than it sounds. After all, we need pass the information at some stage from the application (client) to drools at some point; either we can do it bit-by-bit, or we can do it all at once.

The (small) extra overhead of passing all information every time in this stateless option is justified by the advantages:

- Because we pass all data in every call we are not 'tied' to any particular Drools server. In the diagram below, we can call Drools 1, Drools 2 (or even Drools 10) and get the same response.
- We don't have to worry about synchronisation, as Drools 'forgets' any previous information once it has finished with the current call. The client always has access to the master copy of the information.

Conceptually it is most simple; our client has the master information, and chooses when to call Drools.

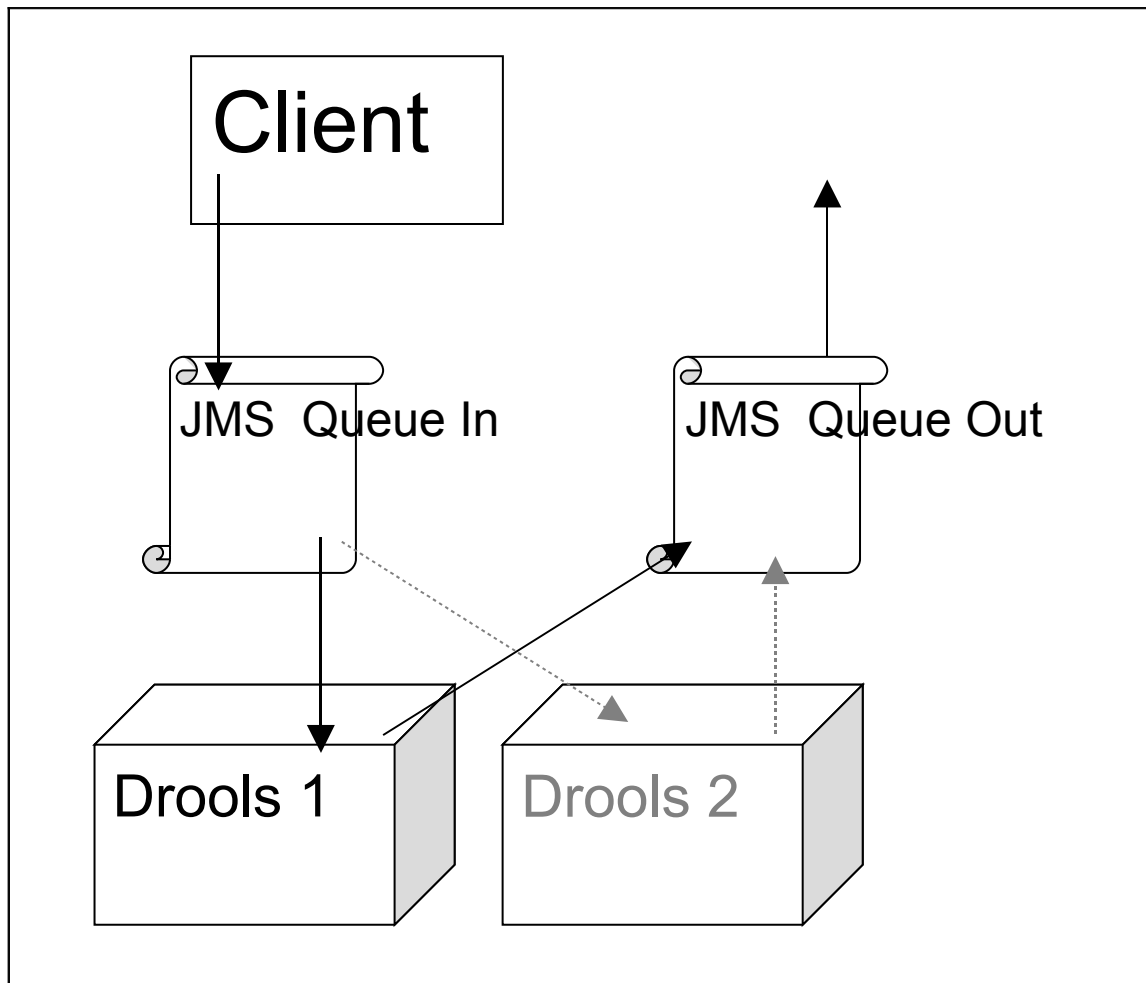


However ***you must have Stateless sessions in Drools*** to make this option work. If stateful Sessions are required – (e.g. the effort of transmitting / inserting the facts, or outweighs the cost of holding a session on a server), then consider Options 2 and 3

Option 2 - Stateful Drools Servers using JMS to synchronise (External)

JMS is the Java Messaging Service – a standard part of any Java Enterprise Edition (JEE) server including Jboss Application Server. Think of Email between computers but with guaranteed delivery. JMS can be backed up by a database (or other store) so even if **all** your servers fail the messages will be held and delivered when service is restored.

JMS makes applications a lot more reliable because it is asynchronous. It says 'I guarantee to respond to your request, probably within milliseconds, but maybe (at times of peak load) longer than that. But I **will** respond to your request eventually'. For many applications (especially those involving workflow) this is good enough and gives the best possible robustness and scalability.



In the above example, the client sends a message containing the information to be added / updated in the Rules Session. Drools Server 1 picks up the message from the queue, applies it to the StatefulSession and fires the rules. It puts the results onto the outbound queue where the Client can retrieve them when it is ready.

In the background, the backup Drools 2 server is also monitoring the queue and applying the same information to the working memory. In effect, it is shadowing the actions of Drools Server 1. However, unless the first Drools server fails, its output is not added to the output queue (or the output is filtered from the queue).

Apache ActiveMQ / Apache Camel allows this option to be quite sophisticated. It can filter duplicate messages from the outbound queue (so that both Drools Servers can be active at the same time, but with the duplicate output messages being removed). This means that if one server fails, we do not have to actively switch to the other.

Another option with ActiveMQ is to load balance between the servers. Each server might take half the messages (according to pre-defined criteria, so that the same client will always go to the same server). The servers would shadow

the remaining messages so that they could pick up the remainder of the load if the other fails.

An important note is that the messages passed to the rule engine server should contain **all** the information that the rules need. Consider an example where the rules calculate 'days since trouble ticket raised'.

If we use the date on the server, what happens if one server processes the message before midnight, while the other server processes it 2 minutes later (but technically the next day)?

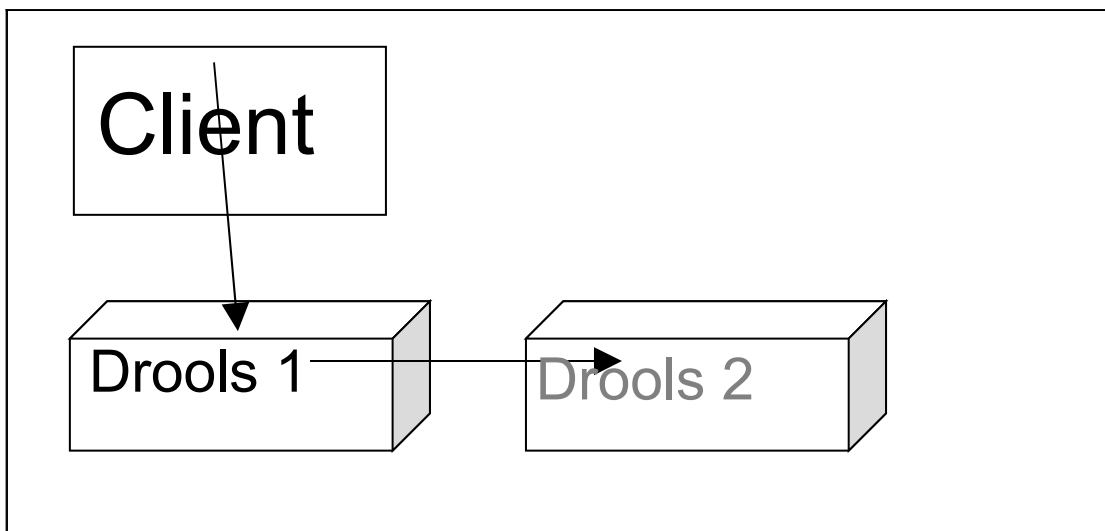
The solution is to pass in the 'current date' as part of the message. That way, both servers will use the same dates for their calculations and the problem is avoided. It also helps for auditing.

The use of JMS also can give us an audit trail, and the opportunity to 'replay' the messages should we need to recover our state.

Option 3: Stateful Drools Servers using JBoss Cache to synchronise (Internal)

The previous option 2 synchronised the Drools **outside** of the Drools code. Each instance of Drools worked as normally; Our application code ensured that data given to Drools 1 and Drools 2 was the same. This can be a lot of work and add unneeded complexity to our application.

Option 3 makes our application code simpler by having synchronisation between multiple live instances happen 'behind the scenes' in the Drools code. The actual synchronisation mechanism can be the same (e.g. Messaging) but is hidden from us.



In this option, the Client updates a single instance of Drools (StatefulSession) with the relevant facts. Drools is configured, using Jboss Cache, to replicate changes to this StatefulSession (and other relevant objects) to other Drools servers. This replication happens 'behind the scenes' and other clients need not be aware of this. From the replicated server's (Drools 2) point of view, it thinks the client has also updated it directly.

Once the facts are in each instance of Drools, we can fire the rules and use the results as normal. If we lose an instance of Drools, we can switch to an alternative server which will always be in the same state.

This option has the advantage that (a) changes required to Drools would likely be folded back into and be supported by JBoss team and (b) require other similar projects only need to 'switch on' and not have to build synchronisation from scratch.

Summary of options

- *By default, choose option 1: stateless sessions.* Only choose the other options if you must have Stateful Sessions (e.g. The cost of transmitting data, or the cost of evaluating it repeatedly in Stateless sessions is too high).
- Option 2 is the best option if you must have Stateful sessions and you want full control of your synchronisation. This is a good option if you are familiar with both Enterprise Java and JMS and are willing to manage the additional complexity in your application.
- Option 3 is the best for the community, and possibly best for your application. Once it is built, it becomes transparent to the application (i.e. It uses configuration, not code) and means that synchronisation is carried out by mainstream Drools code.