ModuleResourcesProposal

Problem Statement

(Apr 18 2004, not quite release 1.0-alpha-4)

HiveMind deals with two basic concepts; *services* and *configurations*. (There is no need to describe the HiveMind concept of *services* or_ configurations_ again as that has already been done on the HiveMind home page.).

Applications use **services** throughout and HiveMind provides a way to define, configure and manage these services using inversion of control meaning that the all the complex code is written just once in the framework (HiveMind). This is great and I love the way HiveMind does this but I believe that **services** are not the only thing applications need and that benefit from a micro-kernel/framework to define, configure and manage them. Applications also require **resources**.

A **resource** provides an application with the type of data that is not known or does not need to be known until runtime. This data could have many possible values, varies based on how and where the application is deployed and often needs to change without having to recompile and redeploy the application.

There are many examples of **resources**:

- Localized key/value properties used for error messages.
- Localized key/value properties used for enumerations in user interface (e.g countries/marital status etc.).
- Key/value properties used for smtp server address port or other similar configurations.
- Datasource to be used to connect to database.
- XML document used to configure a particular part of the application.
- Resource obtained from a lookup on a JNDI tree.
- and many more ...

The advantages of externalizing these resources via a framework or micro-kernel such as HiveMind are:

- Contract with framework defines only the name of the resource and the Java type of the resource.
- We can easily change implementation of ResourceFactory used to create the **resource** so as to use another source or backing store.
- Easier to change configuration and location of source/persistence of **resource**.
- We can monitor and manage all application **resources**.in one place.
- We can provide value added features such interceptors.

The idea of this proposal is to further explore the above with some concrete ideas and examples.

Definitions

Both services and resources are looked up via the registry using their unique id *moduleid.serviceid* or *moduleid.resourceid* respectivly.

Summary:

	Service	**Resource**	Comments
POJO	yes	yes	.
Funtion	Perfom unit of work	Provide runtime data	.
Implements Interface	yes	may do	.
Persisted	no	very often	.
Localized	no	yes	Services are localized via use of localized resources
Instaniation depends on	service model	resource type/store	.

Service: A POJO (Plain Old Java Object) used to perform a unit of work. A service implements an interface that defines its funtional interface. A service has no state apart from injected references to other services or resources that are needed by the service. A service is disposable and is not persisted. A service is not locale sensitive (although it can use a localized resource and therefore produce different execution results dependent on locale). Service instantiation is not dependent on the service implementation but only on the service model being used.

Services are already a HiveMind concept.

Resource: A POJO that does not perform work put rather provides services and framework users with some resource/data. A resource does not nessecarily have to implement a interface. A resource does have state and it is its state that is either the resource data itself (e.g Properties) or is used to gain access to the resource data (e.g. Datasource). Resource data is anything used in implementation, but external to it (Values from Properties /ResourceBundles, data from a DB, data or XML from a file/URL, link to a JNDI/LDAP tree etc.). A resource is locale sensitive. The instantiation of a resource changes depending on the type of resource and the resources backing store. A resource could in theory be writable. A resource has a state that tells us if it is in sync with its backing store if uses one as is the case with Properties/Messages etc.

Resources are not a HiveMind concept yet. I believe they could/should be and would be used widely.

My concept of a resource as i have attempted to describe above is similar in many ways to the way in which resources can be defined and used in Tomcat. Tomcat is a web container and provides resources defined/configured in the server.xml via JNDI. Hivemind is a micro-kernel and would provide resources defined/configured in hivemodule.xml's via the Registry. Take a look at http://jakarta.apache.org/tomcat/tomcat-5.0-doc/jndi-resources-howto.html. Don't let the mention of JNDI/J2EE or web applications make you think its something specific to that type of application. **Resources** are widely in nearly all applications to the same extent as **services**.

Suggested Solutions

Make 'resources' a HiveMind concept at the same level as, and complimentary to, services

This is the prefered solution to integrate the described functionality into HiveMind. It may seem slightly drastic but i believe it is definitly worth looking into.

An **application** has *modules*

A **module** has *services*, *configurations* and *resources*.

A **service** has dependent *services*, *configurations* and *resources*.

A **configuration** has *contributions*

A **resource** has attributes/contributions that are used to instantiate the *resource* using a *ResourceFactory*. A *resource* may also depend on other *resources* with optional resource references as a special type of attribute. (e.g. a resource that obtains properties from a database may depend on a DataSource resource)

Use:

```
// Mailer service obtained has been injected with its dependent services and resources by HiveMind.  In this
case the
// datasource from which the mailer service obtains the list of email addresses.
Mailer myMailer = (Mailer) registry.getService("com.myco.mypackage.Mailer", Mailer.class);

boolean mailerResult=myMailer.mail();

//Properties 'userMessages' obtained is created by HiveMind.  The factory used to create this resource along
with the
//source location of the data i.e file, database or web-services.  This resource can also be managed and
monitored by
//Hivemind in the same way services and have interceptors.
Properties userMessages=(Properties) registry.getResource("com.myco.mypackage.userMessages", Properties.class);

if (result){
   log.info(userMessages.get("mailerSuccess");
}
```

Module definition (hivemodule.xml):

```
<?xml version="1.0"?>
<module id="com.myco.mypackage" version="1.0.0">

  <service-point id="Mailer" interface="com.myco.mypackage.Mailer"/>

  <resource id="myDatasource" type="java.sql.DataSource" >
    <attribute name="username">sql</attribute>
    <attribute name="password">sql</attribute>
  </resource>

  <!-- This resource needs a factory because it is not JavaBean complient -->
  <resource id="userMessages" type="java.util.Properties" factory="com.myco.mypackage.DBPropetiesFactory" >
    <attribute name="table">USER_MESSAGES</attribute>
    <attribute name="pk">MESSAGE_KEY</attribute>
    <resource-ref id="myDatasource">
  </resource>

  <implementation service-id="com.myco.mypackage.Adder">
    <invoke-factory service-id="hivemind.BuilderFactory">
      <construct class="com.myco.mypackage.impl.DefaultMailerImpl">
      <!-- Here we inject the service's resource dependency.  A resource is not just a file but any pre-defined
           resource in the HiveMind registry from any module. -->
      <set-resource property="mailerDS" resource-id="myDatasource"/>
      </construct>
    </invoke-factory>
  </implementation>

</module>
```

Provide functionality via a 'service' included in HiveMind framework or library

This solution is an alternative which avoids modification of the core concepts of HiveMind. Initially it seems to have disadvantages but this needs to be looked into further.

Use:

```
//First we need to obtain the service that is used to obtain resources
ResourceService rs=(ResourceService) registry.getResource("org.apache.hivemind.ResourceService",
ResourceService.class);

//We then obtain the resource from the resource service.
Properties userMessages=(Properties) rs.getResource("com.myco.mypackage.userMessages", Properties.class);
```

Obtaining resources this way is not hard its just has an extra step, obtaining the ResourceService. The thing is how do define our resources in the hivemind.xml module definitions?

The only way that occurs to me is to along with the definition of the ResourceService define a configuration point whose contributions are in fact the resource definitions. This is workable and would provide the same result as the first proposal.

Module definition (hivemodule.xml):
Module definition (hivemodule.xml):

```
<?xml version="1.0"?>
<module id="com.myco.mypackage" version="1.0.0">

  <service-point id="Mailer" interface="com.myco.mypackage.Mailer"/>
  <service-point id="ResourceService" interface="org.apache.hivemind.ResourceService"/>

  <configuration-point id="resources">
    <schema>...</schema>
  </configuration-point>

  <contribution configuration-id="resources">
    <resource id="myDatasource" type="java.sql.DataSource" >
      <attribute name="username">sql</attribute>
      <attribute name="password">sql</attribute>
    </resource>
    <resource id="userMessages" type="java.util.Properties" factory="com.myco.mypackage.DBPropetiesFactory" >
      <attribute name="table">USER_MESSAGES</attribute>
      <attribute name="pk">MESSAGE_KEY</attribute>
      <resource-ref id="myDatasource">
    </resource>
  </contribution configuration-id="resources">

  <implementation service-id="com.myco.mypackage.Adder">
    <invoke-factory service-id="hivemind.BuilderFactory">
      <construct class="com.myco.mypackage.impl.DefaultMailerImpl">
      <!-- To be able to inject resources here the BuilderFactory would have be dependent on the
ResourceService and
           would obtain the resource via the ResourceService -->
      <set-resource property="mailerDS" resource-id="myDatasource"/>
      </construct>
    </invoke-factory>
  </implementation>

</module>
```

Provide resources via HiveMind services

It has been pointed out to me that in fact HiveMind *services* can actually be used to provide resources as well as services (following my definitions). This is a quick and simple and clean solution but which has one major hurdle. The hurdle is the fact that HiveMind *services* have to implement an interface wheras in the real world *resources* do not always implements an interface. So although this is a satisfactory solution for some types of resources it does not satisfy the proposal.

Feel free to add suggested solutions here..

Discussion

The disadvantage of the second proposal is that with this this proposal *resources* are not considered as core module artifacts along with *services* meaning that the concept of injection of dependent services and/or resources is not so simple and direct and would mean implementation that attempted to use it (such as the BuilderFactory for resource injection) would be dependent on the resource service. Also if *resources* are not considered at the same level as *services* within HiveMind it will be far harder to provide a integrated and uniform management interface (via JMX or not) for both **services** and **resources**; the two key and complimentary module artifacts needed by applications that should be provided by HiveMind.

Commentary

Please add your comments, ideas and suggestions here ...

HowardLewisShip: Not quite seeing what a resource can do that a service can not. For example, a data source: For this, I would picture a special factory, specific for creating a service that implements the DataSource interface. The parameters would be the same as what you are contributing to a <resource> element. For another service to obtain JDBC connections, it would simply inject the correct data source service. What your syntax (in the second implementation suggestion) seems to allow is a bit more succinct approach; annoymous <resource> elements, in place, vs. well known services. Lets noodle on this more and consider wider use cases.

DanielFeist: I agree there is a fine line between the concept of a service and of a resource. The main difference is that resources have different concerns as i tired to describe in the definitions above. What you seem to be suggesting is that the <service-points> are not in fact limited to services as I have defined but should be considered as a something which, through the use of factories, are able to provide services, resources or any number of other things. I see your point. That would mean that a service could be anything including a DataSource or Properties etc. and that the factory it uses in <invoke-factory> would create these resource object in a similar way to in my examples. Is this correct? There are concerns specific to resources that would need to be thought though particularly thinking of resources that have a backing store such as change notification etc. I will look into this and add these specific concerns above.

HowardLewisShip: If it can be represented as a Java interface, it can be a service. So certainly for things like DataSource, this is practical. Strictly speaking, then, we don't need this concept of resources. However, strict is not always *correct*. HiveMind should be easy to use, and if people are going to get constantly bogged down creating resource-like services and get annoyed or confused doing so, then we should adapt the framework to address this. For example, the <conversion> element was a recent addition: it doesn't do anything you can do with <rules>, but it makes it *much* easier. Something simiilar, to streamline your use cases may, or may not, be reasonable.

DanielFeist: Ah, that's something i had in mind but forgot to mention. A service is always an implementation of an interface but it is unreasonable to try to fit resources into the same box. Some resources implement an interface (DataSource, Context etc.) but others don't (e.g. ResourceBundle, Properties, File, URL).

ChristianEssl: While this is off-topic IMO it is a general problem that any injected Object has to implement an interface and has to be defined as a service. Further some service-implementations may implement different interfaces and so represent more than one service. Maybe implementations could be defined without necesseraly implenting a certain service and have in such a case their own id. I somehow imagine services as the interface of a module while implentations are the private - implementation - part.

DanielFeist: Not particulary off topic as what you mention is the hurdle which seems to prevent resources being defined are services. My idea was that its fair enough for a service to implement an interface (maybe its good to ensure this) whereas with resources this is not the case, resources often dont implement an interface. This led me to the idea of defining resources as something complimentary to services that are biased to the needs of resources such as not having to implement an interface and managing the issue of backing store syncronization etc.. What you suggest would go towards a solution to my proposal in that it would allow a resource to be defined as a service as Howard suggests. It would be good to see how this would work out in practice, although i personally am not sure if it's a good idea to have services that done implement an interface. What do people think? Make services more flexible and don't force then to implement an interface or try to solve my proposal without going down this route?