NewAndNoteworthyFeaturesInAnnotationBranch

New an noteworthy features

This page describes changes and new features that have been made in this branch: http://svn.apache.org/repos/asf/hivemind/branches/branch-2-0-annot

Main goal of the branch was AnnotationSupport for HiveMind which required the introduction of a pure java API for registry definition. XML support is completely encapsulated in a separate module now.

Pure Java Registry Definition API

The definition of service points, configuration points, contributions etc. is possible now without xml.

More info (outdated): RegistryDefinitionAPI

Annotated Modules

The hivemodule.xml can be replaced by a annotation based module definition.

More info: AnnotationSupport

Registry Building

To reflect the various possible sources for module definitions the process of building a registry has changed. Two things are needed: First of all a registry definition, which contains a description of the registry with all its modules, extension points and extensions. The definition is to a registry instance what a class is to an object. The definition is handed over to a registry builder which is responsible for consistency checks and the construction.

```
RegistryDefinition registryDefinition = new RegistryDefinition();

...  add modules, services, configurations etc.

RegistryBuilder builder = new RegistryBuilder(registryDefinition );
Registry registry = builder.constructRegistry(Locale.getDefault());
```

A xml module that conforms to the 1.1 HiveMind deployment descriptor schema can be processed using a XmlModuleReader:

```
RegistryDefinition registryDefinition = new RegistryDefinition();

XmlModuleReader reader = new XmlModuleReader(registryDefinition);
reader.readClassPathModule("META-INF/hivemodule.xml");
reader.readClassPathModule("foo/hivemodule2.xml");

RegistryBuilder builder = new RegistryBuilder(registryDefinition );
Registry registry = builder.constructRegistry();
```

That is working similar for the new annotated modules:

```
AnnotatedModuleReader reader = new AnnotatedModuleReader(registryDefinition);
reader.readModule(org.apache.hivemind.SimpleModule.class);
```

And finally it is possible to create the module definition by hand:

```
ModuleDefinition module = new ModuleDefinitionImpl("test");

ServicePointDefinition service1 = new ServicePointDefinitionImpl(module, "Service1",
  null, Visibility.PUBLIC, Runnable.class.getName());

module.addServicePoint(service1);
registryDefinition.addModule(module);
```

Registry Autoloading

HiveMind 1.x was able to load all "META-INF/hivemodule.xml" files automatically. The concept of autolading was broadened and made more extensible. The RegistryBuilder now has a method autoDetectModules. It tries to detect so called RegistryProviders by searching the classpath for a global manifest entry "hivemind-providers".

Example of MANIFEST.MF:

```
Manifest-Version: 1.0
hivemind-providers: org.apache.hivemind.impl.HivemoduleProvider
```

An instance of the specified class is created and the registry definition is passed via the RegistryProvider interface:

```
public interface RegistryProvider
{
    public void process(RegistryDefinition registryDefinition, ErrorHandler errorHandler);
}
```

For example the HivemoduleProvider loads all hivemodule.xml files. It just delegates the work to a XmlModuleReader:

```
public void process(RegistryDefinition registryDefinition, ErrorHandler errorHandler)
{
   XmlModuleReader xmlModuleReader = new XmlModuleReader(registryDefinition);
   xmlModuleReader.readClassPathModules("META-INF/hivemodules.xml");
}
```

Extensible Autowiring Service

Autowiring was separated from the xml processing and is now a regular service in the framework:

```
public interface Autowiring
{
    public Object autowireProperties(Object target, String[] propertyNames);
    public Object autowireProperties(Object target);
    public Object autowireProperties(String strategy, Object target, String[] propertyNames);
    public Object autowireProperties(String strategy, Object target);
}
```

The autowiring service delegates the wiring to implementations of AutowiringStrategies. If no strategy is specified the all available strategies are tried until one strategy succeeds. The list of strategies and its order is configurable. The AutowiringStrategy interface:

```
public interface AutowiringStrategy
{
    public boolean autowireProperty(RegistryInfrastructure registry, Object target, String propertyName);
}
```

Currently AutowiringByTypeStrategy is available only.

Arbitrary Configuration Types

Configurations are no longer restricted to lists. A configuration can be of any type whereas collections will be the most common case. Like service points that separate between the service interface and the service implementation a configuration has a type and an implementation (called container) which is responsible for the aggregation of the configuration data. To be backward compatible any configuration defined in a xml module is by default of type java.util.List and uses an ArrayList as container (there are some exceptions to this rule when dealing with unique attributes and mapped configurations).

The container object is passed around to the different contributions which can add new data to it.

Example of a map based configuration:

```
@Configuration(id = "FactoryConfig")
public Map getFactoryConfig()
{
  return new HashMap();
}

@Contribution(configurationId = "FactoryConfig")
public void contributeToFactory(Map container)
{
  container.put("map", "java.util.HashMap");
  container.put("collection", "java.util.ArrayList");
  container.put("inputStream", "java.io.ByteArrayInputStream");
}
```

Example of a complex configuration object:

```
@Configuration(id = "strutsModule")
public ModuleConfig getStrutsModule()
{
  return new ModuleConfigImpl();
}

@Contribution(configuration-id = "strutsModule")
public void contributeToStrutsModule(ModuleConfig config)
{
  config.addActionConfig(new ActionConfig());
  config.addFormBeanConfig(new MyFormBeanConfig());
}
```

The type of an xml configuration point is optionally defined by the type attribute (defaults to java.util.List). The schema needs to know a concrete class to use as top level element while parsing. This class is defined by the attribute root-element-class and defaults to java.util.ArrayList.

```
<configuration-point id="ObjectProviders" type="java.util.Map">

  <schema root-element-class="java.util.HashMap">
    <element name="provider" >
      ...
      <rules>
       <push-attribute attribute="prefix"/>
       <push-attribute attribute="service-id"/>
       <invoke-parent method="put" parameter-count="2"/>
      </rules>
    </element>
  </schema>
</configuration-point>
```

Subsequent Schema Assignment

The new element <schema-assignment> in module descriptors allows to assign schemas to configuration points which have been defined in a non xml module.

```
<schema-assignment configuration-id="hivemind.SymbolSources" schema-id="SymbolSources" />
<schema id="SymbolSources">
   ...
</schem>
```

It's mainly used to retain backward compatibility. Certain configurations are defined in the core framework now which knows nothing about schemas.