Core Framework And Ontology

Overview

This document looks at some possibilities of extending version 1.0 of the KNSG Architecture to include semantic technologies that will improve the framework and add value to the user experience. KNSG is a non-domain specific application framework that is built upon Bard, PTPFlow, and MyProxy for setting up, launching and managing HPC application workflows through an easy to use set of user interfaces. The framework has simple facilities for working with data including import/export, annotation and tagging. The user can create scenarios, add data to them and then launch workflows to HPC machines using PTPFlow. There is also a facility for retrieving result data from completed jobs so the user can continue to work with it and if possible, visualize it. The intent of this document is to lay the foundation of how the core components and views will be enhanced in version 2.0 by adding in the semantic capabilities provided by Tupelo and replacing the current frameworks repository system with a Tupelo context and Tupelo beans. It will also give users information on how to extend the framework for their domain specific application.

Core Application Management

The central management piece for each KNSG application is KNSGFrame, an extension of BardFrame that registers tupelo utility methods for CETBeans used by the KNSG framework. For the remainder of this document we will use the term BardFrame since our extension (KNSGFrame) only overrides the method for registering beans, the rest is the same. BardFrame provides an interface for working with the Tupelo semantic content repository and is responsible for managing contexts, bean sessions, data, firing events, etc. The use of Tupelo beans will be a core concept for persisting information in the KNSG framework so all beans will need to descend from CETBean to remain compatible with the framework and other CET projects. Because every application will have its own bean requirements, each KNSG application should have its own instance of BardFrame to handle this along with an ontology to define domain specific concepts. All application bean types should register with BardFrame and the IBardFrameService should provide the correct instance of BardFrame at runtime.

Tupelo Beans

This section outlines the bean classes that will be required for the KNSG framework. Where possible, the core CETBeans will be used to minimize the work required and maximize compatibility across projects. Some beans will be marked optional if they are part of PTPFlow and if it is uncertain that they will be managed by Tupelo or continue to be managed by PTPFlow's current repository. If a bean is from the set of beans provided by *edu.uiuc.ncsa.cet.bean* it will be noted so we differentiate between new beans for KNSG and the use of existing beans (e.g. the ScenarioBean below is new, but has the same name as the scenario bean in *edu.uiuc.ncsa.cet.bean*).

Scenario Bean

A scenario bean will be used to organize things such as user data and workflows specific to a scenario (or project). This will include datasets (input and output), workflows, and possibly the RMI service for launching jobs. The RMI service was previously part of the scenario, but since this is a system wide object, it will probably not be tracked as part of the scenario in version 2.0. A snippet of what the scenario bean might look like is below:

ScenarioBean extends CETBean implements Serializable, CETBean.TitledBean

```
private String title;  // scenario title
private String description;  // scenario description
private Date date = new Date();  // date scenario created
private PersonBean creator;  // scenario creator
private Set<DatasetBean> dataSets;  // datasets associated with scenario
private List<WorkflowBean> workflows;  // workflows associated with this scenario, if possible, this needs to
be able to wrap ptpflow workflow xml files, or we need our own bean type
private List<VisualizationBean> visualizations; // it is possible in the future that visualizations might be
part of a scenario so we know which datasets/tools are required to generate visualizations
```

While this bean looks very similar to the ScenarioBean in the cet bean plug-in, it is unclear if some of the internal bean types will match what is required for eAIRS/KNSG (e.g. the workflow bean, the visualization bean). As the scenario bean evolves, it will become more clear whether we can replace our bean with the one in the cet bean plug-in. More final documentation will be put here as this design matures.

The main parts of this bean are: DatasetBean's will be used to manage all of the input/output datasets and the WorkflowBean List will contain the workflows associated with this scenario. A user might extend the ScenarioBean if their application has other things that logically belong to their scenarios, but it is envisioned that most changes will happen at the metadata level (e.g. this dataset is a mesh, a result, etc) since the scenario bean should be a generic container that satisfies most users needs.

Dataset Bean

This section is intended to talk about the types of concepts that the Ontology needs to capture. We will break this into two parts: general framework concepts (e.g. a result) and eAIRS specific (e.g. a mesh). We don't anticipate any changes to the DatasetBean class that is provided as part *edu.uiuc.ncsa.cet.bean* plug-in.

WorkflowBean & WorkflowStepBean

Below you will find an example of a PTPFlow workflow.xml file. Right now, this file cannot be altered since it is understood by PTPFlow and outlines the steps in the workflow including which resource to run on, executables that will be launched, input files to use, etc. Ideally, this file would be wrapped into the current WorkflowBean and/or WorkflowStepBean in the *edu.uiuc.ncsa.cet.bean* plug-in. If this is not possible, the KNSG framework will need its own workflow bean.

```
<workflow-builder name="eAIRS-Single" experimentId="singleCFDWorkflow" eventLevel="DEBUG">
  <!-- <global-resource>grid-abe.ncsa.teragrid.org</global-resource> -->
  <global-resource></global-resource>
  <scheduling>
    <profile name="batch">
      <property name="submissionType">
        <value>batch</value>
      </property>
    </profile>
  </scheduling>
  <execution>
     <profile name="mesh0">
              <property name="RESULT_LOC">
                      <value>some-file-uri</value>
              </property>
              <property name="executable">
                      <value>some-file-uri</value>
              </property>
         <property name="meshType">
           <value>some-file-uri</value>
         </property>
         <property name="inputParam">
           <value>some-file-uri</value>
         </property>
     </profile>
  </execution>
  <graph>
    <execute name="compute0">
      <scheduler-constraints>batch</scheduler-constraints>
      <execute-profiles>mesh0</execute-profiles>
      <payload>2DComp</payload>
    </execute>
  </graph>
  <scripts>
    <payload name="2DComp" type="elf">
      <elf>
        <serial-scripts>
          <ogrescript>
            <echo message="Result location = file:${RESULT_LOC}/${service.job.name} result directory is
file:${runtime.dir}/result, copy target is file:${RESULT_LOC}/${service.job.name}"/>
            <simple-process execution-dir="${runtime.dir}" out-file="cfd.out" >
              <command-line>${executable} -mesh ${meshType} -param ${inputParam}</command-line>
             <!-- <command-line>${runtime.dir}/2D_Comp-2.0 -mesh ${meshType} -param ${inputParam}</command-
line> -->
            </simple-process>
            <mkdir>
                    <uri>file:${RESULT_LOC}/${service.job.name}</uri>
            </mkdir>
            <copy sourceDir="file:${runtime.dir}/result" target="file:${RESULT_LOC}/${service.job.name}"/>
          </ogrescript>
        </serial-scripts>
      </elf>
    </payload>
  </scripts>
</workflow-builder>
```

RMIService Bean (optional)

PTPFlow's RMI Service is the service that manages the execution of PTPFlow workflows and records event information. It is the service through which clients communicate to find the status of their workflows. This information could be managed by Tupelo (at some future date) using an RMIServiceBean. This information is currently stored in xml files and managed by PTPFlow's repository system.

RMIServiceBean extends CETBean implements Serializable

```
// Service Info
private String name;
private String platform;
private String deployUsingURI;  // e.g. file:/
private String launchUsingURI;
private String installLocation;  // e.g. /home/user_home/ptpflow
private String rmiContactURI;
private int rmiPortLowerBound;
private int rmiPortUpperBound;
private int gridftpPortLowerBound;
private int gridftpPortUpperBound;
private Date installedDate;
private boolean running;
private Set<HostResourceBean> knownHosts;  // all of the known hosts associated with this service
```

Host Resource Bean (optional)

Below is the bean structure that is anticipated:

A HostResourceBean defines the hpc host and its properties.

HostResourceBean extends CETBean implements Serializable

```
private String osName;  // host os name
private String osVersion; // host os version
private String architecture; // host architecture
private String id; // host id
private Set<PropertyBean> envProperties;  // environment properties on host
private Set<NodeBean> nodes;  // properties of each node
private Set<UserPropertyBean> users;  // user properties on the host - userHome, userNameOnHost, userName
```

A NodeBean defines an HPC nodes properties such as the protocols used and nodeId.

NodeBean extends CETBean implements Serializable

```
private String nodeId;  // id of the node, e.g. grid-abe.ncsa.teragrid.org
private List<FileProtocolBean> fileProtocols;
private List<BatchProtocolBean> batchProtocols;
private List<InteractiveProtocolBean> interactiveProtocols;
```

A UserPropertyBean defines the users properties on the host

UserPropertyBean extends CETBean implements Serializable

```
private String userHome;
private String userName;
private String userNameOnHost;
```

X509CertificateBean

X509CertificateBean extends CETBean implements Serializable

```
private String credential = null;

public String getCredential() {
  return credential;
}

public void setCredential(String credential) {
  this.credential = credential;
}
```

X509CertificateBeanUtil extends AssociatableTupeloBeanUtil<X509CertificateBean>

```
  public X509CertificateBeanUtil(BeanSession beansession) {
    super(beansession)
  }

  public Resource getAssociationPredicate() {
    return KNSG.HAS_CREDENTIAL;  // "http://cet.ncsa.illinois.edu/2011/security/hasCredential"
  }

  public Resource getType() {
    return KNSG.X509_CERT;  //"http://cet.ncsa.illinois.edu/2011/X509Certificate"
  }

  public BeanMapping getMapping() {
    BeanMapping map = super.getMapping();

    // Java class representing the bean
    map.setJavaClassName(X509CertificateBean.class.getName());

    // Properties for the bean
    map.addProperty(KNSG.X509_CREDENTIAL, "credential", String.class);

    return map;
  }

  // Associate a credential with a given item (e.g. WorkflowStepBean, PersonBean)
  public void addCredential(CETBean item, String credential, Date expiration) throws OperatorException {
    addCredential(Resource.uriRef(item.getUri()), credential, expiration);
  }

  public void addCredential(Resource item, String credential, Date expiration) throws OperatorException {
    TripleWriter tw = new TripleWriter();

    // Create credential bean to store credential
    Resource credentialBean = Resource.uriRef(new X509CertificateBean().getUri());

    // Create the credential triple
    tw.add(Triple.create(credentialBean, KNSG.X509_CREDENTIAL, credential));

    // associate the credential bean with the user
    tw.add(Triple.create(item, KNSG.HAS_CREDENTIAL, credentialBean ));

    getBeanSession().getContext().perform(tw);
  }

  public String getCredential(Resource item) throws OperatorException {
    String credential = null;

    Unifier uf = new Unifier();
    uf.addPattern(item, KNSG.HAS_CREDENTIAL, "thecred");
    uf.addPattern("thecred", KNSG.X509_CREDENTIAL, "credential");
    uf.addColumnName("credential");
```

```
    getBeanSession().getContext().perform(uf);

    for(Tuple<Resource> row : uf.getResult()) {
      if(row.get(0) != null) {
        credential = row.get(0).getString();
      }
    }
    return credential;
  }

  public void removeCredential(Resource item, X509CertificateBean cred) throws OperatorException {

    Context context = getBeanSession().getContext();

    TripleWriter tw = new TripleWriter();
    tw.remove(Triple.create(item, KNSG.HAS_CREDENTIAL, Resource.uriRef(cred.getUri())));
    context.perform(tw);
  }
```

Metadata Requirements

This is a list of information we would like to capture with Tupelo using an ontology that is built by NCSA and KISTI (note that KISTI has not provided input on the domain concepts they would like captured).

General Framework Metadata

What we need to capture:

- Is this dataset a result or output dataset?
- Which workflow created this dataset?
- Is this dataset an input dataset?
- Who imported the dataset and when was it imported
- What tags and annotations are associated with the dataset

KNSG

```
public class KNSG {

//Namespace for KNSG
public static final String NS = "http://cet.ncsa.illinois.edu/2011/";

// KNSG Scenario
public static final Resource SCENARIO = newResourceSuffix("/knsg/scenario/KNSGScenario");
public static final Resource HAS_DATA = newResourceSuffix("/knsg/scenario/hasData");
}
```

eAIRS Metadata

What we need to capture:

- Is this dataset an eAirs mesh (*.msh)?
- Is this dataset an eAirs input file (*.inp)
- Result files: coefhist.rlt, error.rlt, result.rlt, time.rlt, cp.rlt, force_com.rlt, result.vtk. We should capture enough information to know what each of these files represent as far as outputs (need input from KISTI).

Mime types of the files associated with the eAIRS workflow

- Input
  - .inp
  - .msh
- Output
  - .rlt
  - .vtk