Project Service 1.0 Component Specification

1. Design

This component provides a web service interface to allow users to perform CRUD operations on project data. A Project is used to group different multiple competitions. Competitions can be one of the standard TopCoder competitions or a custom competition. The service only allows access to a limited set of project properties. It supports role-based access to these properties - with users only allowed to access their own projects and administrators allowed complete access.

The design is very simple. The web interface ProjectService defines the methods that are exposed by the web interface. Two marker extensions of this interface provide local and remote interfaces for access through EJB containers. A ProjectPersistence interface lays the contract for persisting projects. The ProjectData, Project and Competition classes serve as data transfer objects. A host of faults as well as three exceptions allow for clean reporting of errors.

We provide two contract implementations. ProjectServiceBean acts as the bean implementation for both the local and remote beans. It is also an implementation for the web service. JPAProjectPersistence persists the Project entity using a configurable JPA persistence provider. This component supports using Hibernate as the JPA persistence provider.

1.1 Design Patterns

Strategy Pattern - The strategy pattern is used to abstract the persistence of project entities into the ProjectPersistence contract. Instances of the contract may be plugged into the ProjectServiceBean.

Data Transfer Object Pattern - The ProjectData, Project and Competition classes are all serializable business entities whose purpose is to act as data transfer objects.

Data Access Object Pattern - The ProjectPersistence contract acts as the data access object for the Project entity.

1.2 Industry Standards

SOAP - This component provides a SOAP-based web service.

1.3 Required Algorithms

1.3.1 Authentication

The service defined in this component must authenticate users before performing actions. EJB authentication is used with all service methods, to allow the EJB container to automatically verify the roles of the calling principal and ensure necessary authorization. Two security roles will be used - "Administrator" and "User". Security will be ensured by using annotations as shown below.

```
/* To allow only users */
@RolesAllowed("User")
ProjectData getProject(...) { ... }

/* To allow only administrators */
@RolesAllowed("Administrator")
List<ProjectData> getProjectsForUser(...) { ... }

The fact that an administrator is usually also a user should be setup by the security
policy. This component will only allow "User" access to methods which do not require
"Administrator" access.
```

Another authentication related task is to get the ID of the current user and the roles

satisfied by the current user. The first step is to acquire a UserProfilePrincipal object. This is done through the session context, which is a resource injected automatically by the EJB container. Once the principal has been retrieved, we simply need to make simple method calls to extract the roles and user ID.

```
/* Obtain the user profile principal */
UserProfilePrincipal principal = (UserProfilePrincipal)
   sessionContext.getCallerPrincipal();

/* Obtain the user ID */
long userId = principal.getUserId();

/* Obtain the roles from the profile */
Map<Object,String> roles =
(Map<Object,String>)principal.getProfile().getAttribute(rolesKey).getValue();

/* Check if the administrator role is satisfied */
if(roles.containsValue(administratorRole))
  //Do stuff
  ...
```

Note that the UserProfilePrincipal class (which was part of the Authentication component) is now being redesigned as part of the JBoss Login Module 2.0 component. Therefore, the actual method calls may vary.

1.3.2 DDL and Hibernate mapping

In this component, we provide database persistence support for the Project and Competition entities. The DDL for the tables used is shown below.

```
CREATE TABLE project (
  project_id  INTEGER NOT NULL,
  name       CHAR(50) NOT NULL,
  description VARCHAR(10000),
  user_id    INTEGER NOT NULL, -- id of user who creates project
  create_date DATETIME YEAR TO FRACTION(3) NOT NULL,
  modify_date DATETIME YEAR TO FRACTION(3) NULL,
  PRIMARY KEY(project_id) --primary key description
);

CREATE TABLE competition (
  competition_id INTEGER NOT NULL,
  project_id    INTEGER NOT NULL,
  PRIMARY KEY(competition_id),
  FOREIGN KEY(project_id) REFERENCES project(project_id) -- foreign key to PROJECT table
);

CREATE SEQUENCE project_sequence MINVALUE 1 START WITH 1 INCREMENT BY 1;
CREATE SEQUENCE competition_sequence MINVALUE 1 START WITH 1 INCREMENT BY 1;
```

The database persistence is handled by using Hibernate as a JPA provider. The XML files provided below allow Hibernate to map the Project and Competition classes to the their tables in the database. Note that column names are omitted where they are the same as the property names.

```
<hibernate-mapping>
  <class name="com.topcoder.service.project.Project" table="project">
    <id name="projectId" column="project_id">
         <generator class="sequence">
           <param name="sequence">project_sequence</param>
         </generator>
    </id>
    <property name="name"/>
    <property name="description"/>
    <property name="userId" column="user_id"/>
    <property name="createDate" column="create_date" type="date"/>
    <property name="modifyDate" column="modify_date" type="date"/>
    <set name="competitions">
      <key column="project_id"/>
```

```
      <one-to-many class="com.topcoder.service.project.Competition"/>
    </set>
  </class>
  <class name="com.topcoder.service.project.Competition"
        table="competition">
    <id name="competitionId" column="competition_id">
      <generator class="sequence">
            <param name="sequence">competition_sequence</param>
      </generator>
    </id>
    <many-to-one name="project"
              class="com.topcoder.service.project.Project"
              column="project_id"/>
  </class>
</hibernate-mapping>
```

1.3.3 *Using Hibernate as a JPA persistence provider*

To use Hibernate as a JPA persistence provider, we need to setup the property file for JPA. There are many equivalent ways of specifying the properties for JPA. Here we simply reference the XML file containing the Hibernate configuration shown in the previous section.

```
<persistence version = "1.0">
  <persistence-unit name="HibernateProjectPersistence">
    <properties>
      <property name="hibernate.ejb.cfgfile" value = "filename"/>
    </properties>
  </persistence-unit>
</persistence>
```

With the configuration in place, we can use the Hibernate entity manager as just another JPA entity manager.

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory(
   "HibernateProjectPersistence");
EntityManager manager = factory.createEntityManager();
...
```

For any JPA operation, we will always follow the following structure.

```
EntityManager manager = factory.createEntityManager();
EntityTransaction transaction = manager.getTransaction();
transaction.begin();

//Do stuff
...

transaction.commit();
manager.close();
```

For each of the persistence 6 operations of this component, we show sample code below. This code must be inserted in place of the "Do stuff" comment in the above code.

```
Project entity;
List<Project> entities;

//Create
manager.persist(entity);

//Get by ID
Class projectClass = // the class for Project entity
entity = manager.find(projectClass, id);

//Get by user ID
Query query = manager.createQuery(
   "SELECT p FROM Project p WHERE p.userId=:userId";
entities = query.getResultList();

//Get all
```

```
Query query = manager.createQuery(
   "SELECT p FROM Project p";
entities = query.getResultList();

//Update
manager.merge(entity);

//Delete
Query query = manager.createQuery("DELETE FROM Project p WHERE
p.projectId=:projectId");
query.setParameter("projectId", projectId);
query.executeUpdate();
```

1.3.4 Logging

Only the service/bean methods of this component will log useful information. Logging should be performed only if a non-null Log object has been plugged into the bean.

- Error Level - Any exceptions that are raised or caught will be logged at this level.
- Info Level - All method entries and exits will be logged at this level. For method entry, the log must contain the method arguments and for method exit, the return value if any.

1.3.5 Service WSDL description

For JBoss 4.2.x, it is not need to mannually generate the wsdl, this file will be published dynamically, please check http://localhost:8080/jbossws/services after deploy the ear.

1.4 Component Class Overview

ProjectService - This interface is annotated as a web service and defines the contract for the service methods which must be implemented. The service methods provide CRUD functionality for the ProjectData entity.

ProjectServiceLocal - This interface is annotated as a stateless, EJB 3.0 local bean. It is a marker interface which simply extends the ProjectService contract. It allows the ProjectService to also be accessed as a local bean.

ProjectServiceRemote - This interface is annotated as a stateless, EJB 3.0 remote bean. It is a marker interface which simply extends the ProjectService contract. It allows the ProjectService to also be accessed as a remote bean.

ProjectServiceBean - This class implements the ProjectServiceLocal as well as the ProjectServiceRemote interfaces. It performs authorization tasks and delegates persistence to a pluggable instance of ProjectPersistence.

ProjectPersistence - This interface lays the contract for persisting Project entities. It consists of CRUD operation methods for the Project entity.

JPAProjectPersistence - This class implements the ProjectPersistence contract and uses JPA to persist entities.

ProjectData - This data object serves to communicate between project data between the bean and its client. It also serves as a base class for the Project class.

Project - This data object extends the ProjectData class and adds additional properties, including a set of associated competitions.

Competition - This data object represents a competition and is a dummy class whose only properties are an ID and a parent project. Further components may design and use sub-classes of this class.

1.5 Component Exception Definitions

ProjectServiceFault - This class serves as the base fault for all faults thrown by the various service methods. It also serves as a convenience for service clients.

UserNotFoundFault - This fault indicates that a required user was not found.

ProjectNotFoundFault - This fault indicates that a required project was not found.

PersistenceFault - This fault indicates a generic persistence error.

AuthorizationFailedFault - This fault indicates that authorization failed.

ProjectHasCompetitionsFault - This fault indicates that a project cannot be deleted since it is still associated with competitions.

IllegalArgumentFault - This fault indicates that an argument passed to a service method was illegal.

ProjectServiceException - This class serves as the base exception for all exceptions thrown in this component. It also serves as a convenience for applications using this component.

ConfigurationException - This exception indicates an error during configuration. It is thrown by constructors of contract implementations.

PersistenceException - This exception indicates a generic persistence error. It is thrown by methods of the persistence contract.

java.lang.IllegalArgumentException - This exception is thrown if the arguments passed to a method are not legal.

1.6 Thread Safety

This component is, strictly speaking, not thread safe since the data objects provided are mutable and thus not thread safe. The bean class as well as the persistence implementation are thread safe. This allows for two ways in which the component may be used in a thread safe manner.

- If the bean is accessed as a remote interface or as a web service, thread safety is automatically achieved. This is because all data objects used during communication will be serialized/deserialized at either end, thus ensuring that there are no shared data object references.
- If the bean is accessed as a local interface, then all data objects used to communicate must be used from a single thread. This can be easily done by only creating and using data objects within method scopes and not having data objects as shared instance members. If shared data objects are a must, some locking mechanism must be used so that the shared instances are not modified concurrently.

2. Environment Requirements

2.1 Environment

- Java 1.5, EJB 3.0 and JPA 1.0 are required for compilation and execution.

2.2 TopCoder Software Components

- **Base Exception 2.0** - This component provides the base exception for the exceptions of our component.
- **JBoss Login Module 2.0 -** This component provides us with the UserProfilePrinicipal class which is used for authentication.
- **Logging Wrapper 2.0 -** This component allows our component to use logging functionality.

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the

component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

2.3 Third Party Components

- Hibernate Core - The core allows us to map Java classes to database tables using Hibernate and also allows us to use Hibernate persistence.
- Hibernate Entity Manager - The entity manager allows us to use Hibernate as if it were just another JPA persistence provider.

Both of these components may be downloaded from http://www.hibernate.org/ The site also contains useful information on how to setup and use the components. The Hibernate version used must be 3.2 or higher.

NOTE: The default location for 3rd party packages is ../lib relative to this component installation. Setting the ext_libdir property in topcoder_global.properties will overwrite this default location.

3. Installation and Configuration

3.1 Package Name

com.topcoder.service.project

com.topcoder.service.project.impl

com.topcoder.service.project.persistence

3.2 Configuration Parameters

Note the all configuration properties should be set as environment entries in the EJB container.

3.2.1 Configuration for ProjectServiceBean

Name	Description
project_persistence_class	This property gives the fully qualified class name of the instance of ProjectPersistence to create. **Required.**
log_name	This property gives the name of the log to create using the LogManager class. **Optional.**
roles_key	This property gives the name of the key used to fetch the roles from the attributes of the user profile. **Required.**
administrator_role	This property gives the name of the administrator role. **Required.**
user_role	This property gives the name of the user role. **Required.**

3.2.2 Configuration for JPAProjectPersistence

Name	Description
persistence_unit_name	This property gives the name of the persistence unit to use to create the EntityManagerFactory. **Required.**

3.3 Dependencies Configuration

To use named logs, the LoggingWrapper component must be configured with a Log under the appropriate name.

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute 'ant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

In order to use Hibernate as the JPA persistence provider, an appropriate XML configuration file must be setup for Hibernate to map the entities of this component to database tables. A sample of this configuration is present in section 1.3.1 and 1.3.2.

Configuration is also required for the bean as well as the JPA persistence implementation. A sample set of configuration entries is shown below.

```
<!-- For the bean -->

<!-- As JPAProjectPersistence manually control the transaction, the Stateless session
bean should be configured as Bean-Managed Transaction -->

<env-entry>
  <description>Class that implements ProjectPersistence</description>
  <env-name>project_persistence_class</env-name>
  <env-type>java.lang.String</env-type>
  <env-value>
     com.topcoder.service.project.persistence.JPAProjectPersistence
  </env-value>
</env-entry>

<env-entry>
  <description>The name of the log to use</description>
  <env-name>log_name</env-name>
  <env-type>java.lang.String</env-type>
  <env-value>some_log_name</env-value>
</env-entry>

<env-entry>
  <description>The key under which the roles will be found</description>
  <env-name>roles_key</env-name>
  <env-type>java.lang.String</env-type>
  <env-value>roles</env-value>
</env-entry>

<env-entry>
  <description>The name of the administrator role</description>
  <env-name>administrator_role</env-name>
  <env-type>java.lang.String</env-type>
  <env-value>Admin</env-value>
</env-entry>

<env-entry>
  <description>The name of the user role</description>
  <env-name>user_role</env-name>
  <env-type>java.lang.String</env-type>
  <env-value>User</env-value>
</env-entry>

<!-- For the JPA persistence -->

<env-entry>
  <description>
     The persistence unit name, when setup as shown in section 1.3
  </description>
  <env-name>persistence_unit_name</env-name>
  <env-type>java.lang.String</env-type>
  <env-value>HibernateProjectPersistence</env-value>
```

```
</env-entry>
```

Since we use EJB 3.0, deployment of the service is simple. Depending on the application server used, all necessary artifacts (such as the deployment descriptor) can be automatically generated. The class files and any generated artifacts need to be packaged in an EJB JAR file, which should then be deployed in an EJB container.

Any client which wishes to use the bean/web service of this component will first need to obtain a stub to reference the bean. This may be done either through annotations or through code. Once the stub is obtained, it may be used as if it were the bean itself. We show sample code below.

```
/* Obtain the stub programmatically */
Context context = new InitialContext();

ProjectServiceRemote remote = (ProjectServiceRemote)
   context.lookup(<jndiNameRemote>);
ProjectServiceLocal local = (ProjectServiceLocal)
   context.lookup(<jndiNameLocal>);
/* Note that the web service has the same interface as either bean, the difference lies
in the name used to lookup the web service as opposed to looking up the bean */
ProjectServiceRemote remoteService = (ProjectServiceRemote)
   context.lookup(<jndiNameRemoteWebService>);

/*Obtain the stub through annotations */
@EJB
ProjectServiceRemote remote;

@EJB
ProjectServiceLocal local;

@WebServiceRef(wsdlLocation=<WSDL URL>)
ProjectService service;
```

There are many more ways of obtaining a stub - for example the web service can be obtained as a XXXService auto-generated stub statically, and getXXXPort may be used every time an interaction is required. Also a deployment descriptor may be used to inject bean references.

4.3 Demo

The service of this component is likely to be used by users and administrators to CRUD project data. The front-end is likely to be a standalone GUI or a JSP web page.

```
public class ProjectDataGUI implements ActionListener
{
  /* We assume that the bean is accessed remotely */
  @EJB
  private ProjectServiceRemote remote;

  /* Widgets used to enter data */
  private JTextField id,name,description,userId;
  /* A list to show project data */
  private JList list;
  /* Buttons to execute commands */
  private JButton create,retrieve,retrieveUser,retrieveAll,update,delete;

  public static void main(String[] args)
  {
    ProjectDataGUI gui = new ProjectDataGUI();
  }

  public ProjectDataGUI()
  {
    /* Setup a frame, add the widgets, setup button listeners etc. */
    ...
  }
```

```
public void actionPerformed(ActionEvent e)
{
  try
  {
    String command = e.getActionCommand();
    if(command.equals("create"))
    {
      ProjectData projectData = new ProjectData();
      projectData.setName(name.getText());
      projectData.setDescription(description.getText());
      projectData = remote.createProject(projectData);
      show(projectData);
    }
    else if(command.equals("retrieve"))
    {
      long id = Long.parseLong(this.id.getText());
      ProjectData projectData = remote.getProject(id);
      show(projectData);
    }
    else if(command.equals("retrieveUser"))
    {
      long userId = Long.parseLong(this.userId.getText());
      List<ProjectData> projectData = remote.getProjectsForUser(userId);
      show(projectData);
    }
    else if(command.equals("retrieveAll"))
    {
      List<ProjectData> projectData = remote.getAllProjects();
      show(projectData);
    }
    else if(command.equals("update"))
    {
      ProjectData projectData = new ProjectData();
      projectData.setProjectId(Long.parseLong(id.getText()));
      projectData.setName(name.getText());
      projectData.setDescription(description.getText());
      remote.updateProject(projectData);
      show(projectData);
    }
    else if(command.equals("D"))
    {
      long id = Long.parseLong(this.id.getText());
      boolean result = remote.deleteProject(id);
      if(result)
        alert("Project found and deleted.");
      else
        alert("Project does not exist.");
    }
  } catch (ProjectServiceFault fault)
  {
    alert(fault.getMessage());
  }
}
public static void show(ProjectData projectData)
{
  /* Clear the JList and show the given item */
  ...
}
public static void show(List<ProjectData> projectData)
{
  /* Clear the JList and show the given items */
  ...
}
public void alert(String message)
{
  /* Show an alert box, with the given message */
  JOptionPane.showMessageDialog(null, message, "Message",
     JOptionPane.PLAIN_MESSAGE);
}
```

```
}
```

However, the service only provides limited capabilities. Many administrative duties will be done directly through the JPA persistence, with appropriate setup of the context for configuration properties. We show below, how to create and populate an arena project consisting of 3 competitions. Note that the Competition class is likely to be extended by further components.

```
Project project = new Project();
project.setName("Arena");
project.setDescription("A new competition arena");
/* Similarly set the other properties */
...

/* We assume DesignCompetition and StudioCompetition are subclasses of Competition */
Set<Competition> competitions = new HashSet<Competition>();
competitions.add(new DesignCompetition("GUI design"));
competitions.add(new DesignCompetition("Backend design"));
competitions.add(new StudioCompetition("Arena skins"));
for(Competition c:competitions) {
  c.setProject(project);
}

/* Create everything. Note that to fully persist the Competition subclasses, custom
Hibernate configuration will be required. */
JPAProjectPersistence persis = new JPAProjectPersistence();
persis.createProject(project);

/* This will create 4 records in the database - 1 in the project table and 3 in the
competition table. */

/* Similarly we may update, delete, and retrieve by different criteria as shown in the
demo above. */
```

5. Future Enhancements

None presently.