Registration Services 1.0 Component Specification

1. Design

This component implements the business logic for managing registrations to contests. The main interface of this component is fine grained enough to provide a useful API to client applications, but coarse grained enough to offer transactional atomic services and allow the presentation layer to minimize the calls to this layer.

This Service captures the registration process for any type of competition, such as assembly, testing, and component. The focus is on the type of registration as opposed to the type of competition registering for.

In this version, registration as Free Agents and Team captains will be supported.

A user can already be registered for the selected project; this allows a user to alter their desired role for this project. For example, a user registered as a free agent may re-register for the project as a Team Captain. Doing so removes their free agent role and grants them the Team Captain role.

A user may only have one role at any time for a particular project; but the user may, subject to eligibility and rules, alter their role in a project.

This service returns whether the validation was successful, the previous registration if there was one and the error messages in case of a validation failure.

Overall, the design is very straightforward. All interfaces except RegistrationValidator have implementations provided. The role of this service is to coordinate the registration process using a number of other services and managers.

This design makes use of the Logging Wrapper to facilitate informational and debugging logging in the services implementation. Logging is not performed inside the bean implementations as there is no need to know such granular and trivial information.

1.1 Design considerations

1.1.1 EJBs, ConfigManager, and Logging Wrapper

The EJB specification stipulates that File I/O is not allowed during the execution of the bean. At first glance this might mean that the use of the Config Manager, and by extension Object Factory, could not be used because it performs property retrieval and storing using files. However, the restriction is only placed on the bean's lifetime, not the ConfigManager's. Therefore, as long as the Config Manager does not perform I/O itself during the execution of the bean, or to be more accurately, that the thread performing a bean operation does not cause the ConfigManager to perform I/O, the use of Config Manager to hold an in-memory library of properties is acceptable. Therefore, this design makes extensive use of Config Manager and Object Factory, again, with the stipulation that the Config Manager implementation does not perform I/O when this component is retrieving properties.

The issue of using Logging Wrapper is similar. The user of this component must ensure that logging is performed in a manner that does not violate EJB standards, such as not writing to a file or console.

1.2 Design Patterns

1.2.1 Strategy

RegistrationServicesImpl uses RegistrationValidator as a strategy. Also, the various external managers and services are used in a transparent manner.

The various bean implementations, such as OperationResult and RegistrationInfo are also used transparently.

1.2.2 Façade

The **Façade** pattern can be said to be used in the RegistrationServices, as it provides streamlined access to managing registrations that uses numerous other components.

1.3 Industry Standards

EJB (specifically, to ensure this component is compatible with EJB restrictions, as defined in http://java.sun.com/blueprints/qanda/ejb_tier/restrictions.html).

1.4 Required Algorithms

There are no complex algorithms here. Implementation hints are provided in Poseidon documentation as “Implementation Notes” sections. Information about logging is provided here.

1.4.1 Logging

This section is the central place to describe how logging is performed in lieu of stating this in each method in the zuml.

All methods in RegistrationServicesImpl have access to a Log and should log in the following manner:

- Method entrance and exit at INFO level.
- All exceptions and errors at ERROR level. This includes illegal arguments.
- All calls to external TopCoder classes (just to managers and services classes, not beans) at DEBUG level. This includes before the call and after the call.
- Any additional logging is at the developer’s discretion.

Note that if the RegistrationServicesImpl's log is null, no logging is to be performed.

1.4.2 *Template XML Data Generation*

The registration removal operation requires the generation of messages using the Document Generator component. Part of the task will be to create the required layered XML that will pass the data to the generator. This section will define the structure that the service method involved in messaging will adhere to.

The following shows the XML format that the developer will assemble. Simple string concatenation will be sufficient. Note that all the information will be available for this in the service method.

```
<DATA>
    <TEAM_NAME>name</TEAM_NAME>
    <TEAM_DESCRIPTION>description</TEAM_DESCRIPTION>
    <PROJECT_NAME>project name</PROJECT_NAME>
    <HANDLE>handle</HANDLE>
    <ROLE_NAME>role name</ROLE_NAME>
<DATA>
```

1.5 Component Class Overview

1.5.1 *com.topcoder.registration.service*

This package holds all the interfaces and constants in this component.

RegistrationServices
This represents the interface that defines all business methods for registration services. The services include being able to register a member in a project under a specific role, validating, retrieving and deleting the registration. Furthermore, one can find all available positions in any given project that is in a registration phase, one can get all current members in a project, and all projects a member is registered in.

It has one implementation: RegistrationServicesImpl.

Thread Safety: There are no restrictions on thread safety in implementations.

CustomResourceProperties
A simple listing of the names of properties to be used as custom properties in a Resource. These include an external reference ID, i.e. user ID, a user handle, user email, and the date of the registration. These are added when the user registers in the RegistrationServices.registerForProject method.

Thread Safety: This class is thread-safe.

RegistrationInfo
The interface that defines the properties of a registration, which include a project, user, and role ID. This information basically represents the role a user has in a project.

This interface follows java bean conventions for defining setters and getters for these properties.

It has one implementation: RegistrationInfoImpl.

Thread Safety: There are no restrictions on thread safety in implementations.

RegistrationResult
The interface that defines the result of the registration attempt, generally referring to the validation of the registration information. It defines a flag for success, potential error messages if the registration was not successful, and the previous registration information, if available, if the registration was successful.

This interface follows java bean conventions for defining getters for these properties.

It has one implementation: RegistrationResultImpl.

Thread Safety: There are no restrictions on thread safety in implementations.

RegistrationPosition
The interface that defines the information about the available positions/roles in a project during registration. It also provides detailed information about the registration phase of this project, such as its start and end date.

This interface follows java bean conventions for defining setters and getters for these properties.

It has one implementation: RegistrationPositionImpl.

Thread Safety: There are no restrictions on thread safety in implementations.

RemovalResult
The interface that defines the result of the registration removal attempt. It defines a flag for success, potential error messages if the registration removal was not successful.

This interface follows java bean conventions for defining getters for these properties.

It has one implementation: RemovalResultImpl.

Thread Safety: There are no restrictions on thread safety in implementations.

RegistrationValidator

The interface that defines the contract for validating registration information. The results of this validation are returned as an OperationResult object from the Team Services component.

There are no current implementations available in this component version.

Thread Safety: There are no restrictions on thread safety in implementations.

1.5.2 *com.topcoder.registration.service.impl*

This package holds the implementations to all interfaces but the RegistrationValidator in the com.topcoder.registration.service package.

RegistrationServicesImpl
Full implementation of the RegistrationServices interface. This implementation makes use of a large array of components to accomplish its task of managing registrations. First, it relies on the RegistrationValidator to perform validations in both the validateRegistration and registerForProject methods. It makes use of the User Data Store Persistence component to retrieve full user information. It uses the Phase and Resource Management, Project Phases, Project Services, and Team Services components to create, get, find, and delete registration information.

To provide a good view as the steps are progressing in each method, as well as full debug information for developers, the Logging Wrapper component is used in each method. To configure this component, the ConfigManager and ObjectFactory components are used.

Thread Safety: This class is immutable but operates on non thread safe objects, thus making it potentially non thread safe.

RegistrationInfoImpl
Simple implementation of the RegistrationInfo interface. Implements all methods in that interface, and provides a default constructor if the user wants to set all values via the setters, and one full constructor to set these values in one go.

Thread Safety: This class is mutable and not thread safe.

RemovalResultImpl
Simple implementation of the RemovalResult interface. Implements all methods in that interface, and provides a default constructor if the user wants to set all values via the setters, and one full constructor to set these values in one go

Thread Safety: This class is mutable and not thread safe.
.
RegistrationPositionImpl

Simple implementation of the RegistrationPosition interface. Implements all methods in that interface, and provides a default constructor if the user wants to set all values via the setters, and one full constructor to set these values in one go.

Thread Safety: This class is mutable and not thread safe.

RegistrationResultImpl
Simple implementation of the RegistrationResult interface. Implements all methods in that interface, and provides a default constructor if the user wants to set all values via the setters, and one full constructor to set these values in one go.

Thread Safety: This class is mutable and not thread safe.

1.6 Component Exception Definitions

This component defines four custom exceptions. Note that all are runtime exceptions.

RegistrationServiceException
Extends BaseRuntimeException. Called by all RegistrationServices methods if an unforeseen error occurs. All exceptions that occur from other TopCoder components are deemed to be exceptional, and would be wrapped in this exception.

RegistrationServiceConfigurationException
Extends RegistrationServiceException, Called by the RegistrationServicesImpl constructors if a configuration-related error occurs, such as a namespace not being recognized by Config Manager, or missing required values, or errors while constructing the managers and services.

InvalidRoleException
Extends RegistrationServiceException. Called by RegistrationServices.removeRegistration method if the passed roleId does not correspond to the roleId that the existing registration (and only if there is one) has.

RegistrationValidationException
Extends RegistrationServiceException. Called by the RegistrationValidator.validate method if there is an unexpected error during the call.

1.7 Thread Safety

Thread safety is not mentioned as a required part of this component. The component is virtually thread-safe, and under expected conditions, it is effectively thread-safe.

Objects such as RegistrationInfoImpl and RegistrationPositionImpl are not thread-safe, and if it had to be made thread-safe, its read/write operations would have to lock on the fields they access.

Another aspect is the use of mutable, non-thread-safe objects, such as RegistrationInfoImpl, in the RegistrationServicesImpl methods. This effectively renders RegistrationServicesImpl non-thread-safe and the only way to make it thread-safe is to ensure all objects it uses are thread-safe. However, it is not anticipated that these bean instances, such as RegistrationInfoImpl, will be used by multiple threads.

2. Environment Requirements

2.1 Environment

- Development language: Java 1.4
- Compile target: Java 1.4

2.2 TopCoder Software Components

- Configuration Manager 2.1.5
  - Used for configuration of RegistrationServicesImpl
- Object Factory 2.0.1
  - Used to obtain instances of required objects in RegistrationServicesImpl. Some of the required objects are instances of PhaseManager, ResourceManager, etc…
- Base Exception 2.0
  - Topcoder standard for all custom exceptions.
- Logging Wrapper 2.0
  - Used for logging actions in RegistrationServicesImpl. A Log instance is obtained from LogManager.
- Project Management 1.1
  - Provides the classes and interfaces used for phase management: ProjectManager, Project, and ProjectCategory.
- Resource Management 1.1
  - Provides the classes and interfaces used for resource management: ResourceManager, Resource, and ResourceRole.
- Project Phases 2.0
  - Provides the phase model: Project and Phase.
- User Project Data Store 2.0
  - Provides the UserRetrieval manager used to retrieve ExternalUsers.
- Phase Management 1.1
  - Provides the interface used for phase management: PhaseManager.

- Project Services 1.0
    - Provides the ProjectServices and FullprojectData interfaces. This component is used for getting active project data.
- Team Services 1.0
    - Provides the TeamServices and OperationResult interfaces. The first is used to remove members during registration removal, and the later is used in the RegistrationValidator.
- Ban Manager 1.0
    - Provides the BanManager interface. Used to ban resources for day periods.
- Contact Member 1.0
    - Provides the ContactMemberService, OperationResult, and Message interfaces and class used to send messages to resources.
- Search Builder 1.3.1
    - Provides the Filter interface and classes that enable querying of data with filters.
- Document Generator 2.0
    - Provides the facility to generate messages.

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

There are no third party components that need to be used directly by this component.

3. Installation and Configuration

3.1 Package Names

com.topcoder.registration.service
com.topcoder.registration.service.impl

3.2 Configuration Parameters

3.2.1 *RegistrationServicesImpl*

Parameter	Description	Details
specNamespace	Namespace to use with the ConfigManagerSpecificationFactory. Required	Example: “com.topcoder.specify”
validatorKey	Key for the RegistrationValidator to pass to ObjectFactory. Required	Valid key
userRetrievalKey	Key for the UserRetrieval to pass to ObjectFactory. Required	Valid key
resourceManagerKey	Key for the ResourceManager to pass to ObjectFactory. Required	Valid key
projectServicesKey	Key for the ProjectServices to pass to ObjectFactory. Required	Valid key
phaseManagerKey	Key for the PhaseManager to pass to ObjectFactory. Required	Valid key
banManagerKey	Key for the BanManager to pass to ObjectFactory. Required	Valid key
teamServicesKey	Key for the TeamServices to pass to ObjectFactory. Required	Valid key
contactMemberServiceKey	Key for the ContactMemberService to pass to ObjectFactory. Required	Valid key
loggerName	The name of the log to get from the LogManager. Optional	A valid name of the log. If not given, logging is not performed.
registrationPhaseId	The ID of the registration phase. Required	A non-negative long number.
availableRoleIds	The IDs of the available roles. Optional	Non-negative long numbers. Note that roles are not strictly required.
externalRoleIds	The IDs of the roles external to registration. Optional	Non-negative long numbers. Note that roles are not strictly required.
teamCaptainRoleId	The ID of a team captain resource role. Required	A non-negative long number.
operator	The ID of a operator of this component. Required	A non-negative long number.
activeProjectStatusId	The ID of the active project status type. Required	A non-negative long number.
removalMessage-TemplateName	The name of the template to use for generating messages for member removal. Required.	A valid template name in DocumentGenerator

3.3 Dependencies Configuration

3.3.1 TopCoder dependencies

The developer should refer to the component specification of the TopCoder components used in this component to configure them. Please see section 2.2 for a list of these components.

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

None

4.3 Demo

4.3.1 Setup

Much of the setup for any demo involves properly configuring all the components. This demo will not go into that depth, but it will use a sample configuration XML file.

Here we define the configuration parameters in ConfigManager:

```
<Config name="TestConfig">
   <Property name="specNamespace">
         <Value>SpecificationNamespace</Value>
   </Property>
   <Property name="validatorKey">
         <Value>validatorKey</Value>
   </Property>
   <Property name="userRetrievalKey">
         <Value>userRetrievalKey</Value>
   </Property>
   <Property name="resourceManagerKey">
         <Value>resourceManagerKey</Value>
   </Property>
   <Property name="projectServicesKey">
         <Value>projectServicesKey</Value>
   </Property>
   <Property name="phaseManagerKey">
         <Value>phaseManagerKey</Value>
   </Property>
   <Property name="banManagerKey">
         <Value>banManagerKey</Value>
   </Property>
   <Property name="teamServicesKey">
         <Value>teamServicesKey</Value>
   </Property>
   <Property name="contactMemberServiceKey">
         <Value>contactMemberServiceKey</Value>
   </Property>
   <Property name="loggerName">
         <Value>defaultLogger</Value>
   </Property>
   <Property name="registrationPhaseId">
         <Value>1</Value>
   </Property>
   <Property name="availableRoleIds">
         <Value>1</Value>
         <Value>2</Value>
   </Property>
   <Property name="externalRoleIds">
         <Value>5</Value>
         <Value>6</Value>
   </Property>
   <Property name="teamCaptainRoleId">
         <Value>2</Value>
   </Property>
   <Property name="operator">
         <Value>1</Value>
   </Property>
   <Property name="activeProjectStatusId">
         <Value>2</Value>
   </Property>
   <Property name="removalMessageTemplateName">
         <Value>removalMessageTemplate</Value>
   </Property>
</Config>
```

In the above configuration, we assume that roleID = 1 is a Team Member role, and roleId=2 is a Team Captain role.

The actual construction of the services class would be done in the following manner.

```
// Create RegistrationServicesImpl from configuration.
RegistrationServices service = new
RegistrationServicesImpl(“TestConfig”);
```

For the benefit of the scenario, we will provide a simple state of the system in terms of projects that are in registration phase. These projects are arbitrary:

- Design: Survey Data Model (Id=10): 2 registrants (1 member, 1 captain)
- Design: Survey Persistence (Id=11): 0 registrants
- Development: Survey Servlet (Id=12): 1 registrant (1 member)

For the demo, assume that user 1 is registered as a member in Survey Data Model and Survey Servlet. User 2 is the captain in Survey Data Model.

4.3.2 *Usage*

A typical usage scenario involves the full use of the methods to manage registrations. We use the service instance and scenario setup from above.

```
// Register a user (id=3) for a captain role for Survey Persistence
project
RegistrationInfo registrationInfo = new RegistrationInfoImpl(11,3,2);
```

```
RegistrationResult result = service.registerForProject(registrationInfo);
// The result would indicate that the validation was successful, and that
there was no previous registration for this user in this project. Survey
Persistence would now have this user registered as a Team Captain.

// Note that since the registration provides automatic validation,
explicit validation is not necessary but could be explicitly performed
with the following call:
RegistrationResult result =
service.validateRegistration(registrationInfo);
// The result would hold the same information as the previous shown call

// If we now attempt to retrieve information about this registration, we
will confirm the registration.
RegistrationInfo retrievedRegistrationInfo =
service.getRegistration(3,11);
// This would confirm role as Team Captain (roleId=2)

// If this person decided to change their role to Team Member, the can
repeat the call to registration:
RegistrationResult result2 = service.registerForProject(new
RegistrationInfoImpl(11,3,1));
// The result would contain the previous registration info, where roleID
was 2.

// Should this user decide to remove himself/herself from the project,
they could use the removeRegistration method
RemovalResult result3 = service.removeRegistration(new
RegistrationInfoImpl(11,3,1));
// The result would confirm deletion of the resource, and Survey
Persistence project would once again have no registrants

// We can query which design projects are available
ProjectCategory category = a design category
RegistrationPosition[] positions =
service.findAvailableRegistrationPositions(category);
// The result would state that Survey Data Model and Survey Persistence
are in design stage, with the same configured roles available: Team
member, and Team captain.

// Suppose user 1 (introduced at the end of section 4.3.1) wants to know
in which projects he/she is:
Project[] projects = service.getRegisteredProjects(1);
// The result would be two projects: Survey Data Model and Survey Servlet
projects

// If the aim was to find out who is currently registered in Survey Data
Model, the following method call can be made:
Resource[] resources = service.getRegisteredResources(10);
// The result is
// - a resource for user 1 as Team member
// - a resource for user 2 as Team captain

// This demo has provided a typical scenario for the usage of the
service.
```

5. Future Enhancements

Registration to non-team competitions will be added, along with new services like Unregister to Project.