Profile Struts Actions 1.0 Component Specification

1. Design

The TopCoder registration process to become a member of TopCoder is going to be redesigned. The main goal is to allow for a minimal set of required info that the user needs to enter to sign up (handle, email, and password).

The existing TC web registration process will ask a new user to enter a lot of data when he/she tries to register to TC web-site. The registration process takes a lot of time, so many users can simply break the registration process and leave non-registered

Thus there is a need of an easy to use and user friendly web-application for supporting registration on the new users and management of registered user profiles at TC web-site.

The main goal of this application is to simplify registration of the new users on TC web-site by minimizing the count of mandatory data fields for new account registration, and to improve usability of user profile management for the registered users. Any additional profile information will be requested from the user by the system as it is needed. For example, if a user registers to compete in an assembly contest the site would prompt them for any required info that they have not yet entered.

This component provides the actions used for the profile process. There are two separate but related workflows here. Each workflow deals with the management of the user's name and contact info, the user's demographic info, and the user's account info. But the first workflow deals with the user completing the above-mentioned profile data in a wizard-like screen, with one master submission button. The second workflow, however, is launched from the user's profile page, where each of the above items is managed via a separate mini-workflow of view-edit-save steps.

The requirements call for 2 actions to service the profile completeness process, to load the profile data, and to save it. The requirements also call for 9 actions to manage the profile data, 3 per type of info. All actions are also required to load the completeness data.

This component provides all these actions, but provides a few changes in regards to the approach, consistent with such approved refactoring as per RS 1.2.7. View and Edit classes basically do the same work, except the Edit actions must also load lookup data, which is why in the inheritance tree, all of them inherit from BaseDisplayProfileAction, and Edit actions extend View actions.

All Save actions are modeled with their own class that extends BaseSaveProfileAction, since again, all of them deal with saving User data. Each concrete action will save the data it deals with, and also provide custom auditing and any additional programmatic validation. BaseSaveProfileAction itself extends EmailSendingProfileAction. We provide this separation for future extensibility in case other, non-saving actions want to send emails. One possibility would be some kind of timing action that would send a reminder to the user that he needs to complete registration, but without saving data.

The one exception to this approach is the CompleteProfileAction. All this action does is forward to the individual action that services the given tab, so it extends BaseProfileAction.

All actions inherit from the BaseProfileAction, which provides some of the base plumbing as well as some convenience methods and properties generally used by all actions.

Conventions:
When referring to bean properties in this component, the standard bean dot notation is used in lieu of quoting the actually get/set methods. So instead of having

```
initiationResponse.getProblem().getId()
```

We will write

```
initiationResponse.problem.id
```

Assumptions:
This component assumes that a resource bundle is being used for validation messages, and further assumes that the ones referenced here are the correct ones. If in later stages this is not the case, the references will need to be changed to reflect the correct paths.

1.1 Design Patterns

MVC – This is the foundation of this framework. STRUTS will do everything using this pattern and the actions, results, and interceptors will plug into this framework. Note that actions will be placed on the ValueStack with their data already properly set.

Strategy – The managers and DAOs are used as strategies by the actions

Template Method – The BaseDisplayProfileAction and BaseSaveProfileAction are base classes that do most of the work of displaying and saving, but leave the

concrete classes to fill the templates for certain steps, such as processing the specifics of the User entity, auditing, or other processing.

DAO – The DAOs used by this component are examples of the Data Access Object pattern.

DTO – The entities defined in this component and other entities used by it are data transfer objects.

1.2 Industry Standards

- Inversion of Control (IoC)
  - This requires that all properties of actions will provide at least setters to values can be injected. Complementary getters are also provided in case the classes are extended in the future.
- XML for validation
- OGNL used by the frontend to access action results

1.3 Required Algorithms

1.3.1 Logging

The application will log activity and exceptions using the Logging Wrapper in struts actions, and the Logging Wrapper should be configured to use Log4j.

It will log errors at Error level, and method entry/exit information at DEBUG level.

Specifically, logging will be performed as follows, if logging is turned on.

- Method entrance and exit will be logged with DEBUG level.
  - Entrance format: [Entering method {*className.methodName}*]
  - Exit format: [Exiting method {*className.methodName}]*. Only do this if there are no exceptions.
- Method request and response parameters will be logged with DEBUG level
  - Format for request parameters: [Input parameters[{request_parameter_name_1}:{ request_parameter_value_1}, {request_parameter_name_2}:{ request_parameter_value_2}, etc.)]]
  - Format for the response: [Output parameter {response_value}] . Only do this if there are no exceptions and the return value is not void.
  - If a request or response parameter is complex, use its toString() method. If that is not implemented, then print its value using the same kind of name:value notation as above. If a child parameter is also complex, repeat this process recursively.
- All exceptions will be logged at ERROR level, and automatically log inner exceptions as well.
  - Format: Simply log the text of exception: [Error in method {*className.methodName}*: Details {*error details*}]

In general, the order of the logging in a method should be as follows:

1. Method entry
2. Log method entry
3. Log method input parameters
4. If error occurs, log it and skip to step7
5. Log method exit
6. If not void, log method output value
7. Method exit

The toString() method of the entity can be used for the logging.

1.3.2 Validation

We will use declarative validation available with Struts. This will quickly state, what validation is to be done in each action. The developer will consult the Struts2 validation framework for the details on how to apply these rules to applicable validators. This framework provides such built-in validators to handle the above-mentioned rules. More can be read here: http://struts.apache.org/2.0.11.2/docs/validation.html.

Each validation would be created in a file per action name or alias depending on application integration requirements. Each of the following fields would be encompassed by a <validators>…</validators> element.

Only the individual save actions perform validation, and the rules are defined in the ARS sections 2.9.1.2, 2.10.1.2, and 2.11.1.2. An example of how a field is validated is shown below with a few selected fields. The others follow in the same manner, and are left to the developer.

SaveNameAndContactAction:

```
<field name=“savedUser.userProfile.firstName">
    <field-validator type="stringlength">
        <param name="maxLength>15</param>
        <message key="firstName.invalid.length" />
    </field-validator>
</field>

<field name=“savedUser.userProfile.lastName">
    <field-validator type="stringlength">
        <param name="maxLength>15</param>
        <message key="lastName.invalid.length" />
    </field-validator>
</field>

<field name=“savedUser.emailAddresses[0].address">
    <field-validator type="requiredstring">
        <message key="email.required" />
    </field-validator>
    <field-validator type="stringlength">
```

```
        <param name="maxLength>100</param>
        <message key="email.invalid.length" />
    </field-validator>
    <field-validator type="email">
        <message key="email.invalid.format" />
    </field-validator>
</field>
<field name=“savedUser.userProfile.contact.title">
    <field-validator type="stringlength">
        <param name="maxLength>15</param>
        <message key="jobtitle.invalid.length" />
    </field-validator>
</field>

<field name=“savedUser.userProfile.contact.company.name">
    <field-validator type="stringlength">
        <param name="maxLength>100</param>
        <message key="companyname.invalid.length" />
    </field-validator>
</field>

<field name=“savedUser.userProfile.addresses[0].address1">
    <field-validator type="stringlength">
        <param name="maxLength>100</param>
        <message key="address1.invalid.length" />
    </field-validator>
</field>

<field name=“savedUser.userProfile.addresses[0].address2">
    <field-validator type="stringlength">
        <param name="maxLength>100</param>
        <message key="address2.invalid.length" />
    </field-validator>
</field>

<field name=“savedUser.userProfile.addresses[0].address3">
    <field-validator type="stringlength">
        <param name="maxLength>100</param>
        <message key="address3.invalid.length" />
    </field-validator>
</field>

<field name=“savedUser.userProfile.addresses[0].city">
    <field-validator type="stringlength">
        <param name="maxLength>64</param>
        <message key="city.invalid.length" />
    </field-validator>
</field>

<field name=“savedUser.userProfile.addresses[0].state.code">
    <field-validator type="stringlength">
        <param name="maxLength>2</param>
        <message key="state.invalid.length" />
    </field-validator>
</field>

<field name=“savedUser.userProfile.addresses[0].postalCode">
```

```
    <field-validator type="stringlength">
        <param name="maxLength>15</param>
        <message key="lastName.invalid.length" />
    </field-validator>
</field>

<field name=“savedUser.userProfile.addresses[0].province">
    <field-validator type="stringlength">
        <param name="minLength>4</param>
        <param name="maxLength>15</param>
        <message key="lastName.invalid.length" />
    </field-validator>
</field>

<field name=“savedUser.userProfile.phoneNumbers[0].number">
    <field-validator type="stringlength">
        <param name="maxLength>64</param>
        <message key="lastName.invalid.length" />
    </field-validator>
</field>
```

SaveDemographicInfoAction:
As all the data is dynamic, the validation must be done programmatically.

SaveAccountInfoAction:

```
<field name=“savedUser.handle">
    <field-validator type="requiredstring">
        <message key="handle.required" />
    </field-validator>
    <field-validator type="stringlength">
        <param name="minLength">2</param>
        <param name="maxLength>15</param>
        <message key="handle.invalid.length" />
    </field-validator>
</field>

<field name=“savedUser.password">
    <field-validator type="requiredstring">
        <message key="password.required" />
    </field-validator>
    <field-validator type="stringlength">
        <param name="minLength">7</param>
        <param name="maxLength>30</param>
        <message key="password.invalid.length" />
    </field-validator>
</field>

<field name=“confirmPassword">
    <field-validator type="requiredstring">
        <message key="password.required" />
    </field-validator>
    <field-validator type="stringlength">
        <param name="minLength">7</param>
        <param name="maxLength>30</param>
        <message key="password.invalid.length" />
```

```
    </field-validator>
</field>

<field name=“savedUser.userProfile.secretQuestion.question">
    <field-validator type="stringlength">
        <param name="minLength">8</param>
        <param name="maxLength>254</param>
        <message key="secretquestion.invalid.length" />
    </field-validator>
</field>

<field name=“savedUser.userProfile.secretQuestion.answer">
    <field-validator type="stringlength">
        <param name="minLength">2</param>
        <param name="maxLength>254</param>
        <message key="secretquestionresponse.invalid.length" />
    </field-validator>
</field>

<field name=“savedUser.userSecurityKey.securityKey">
    <field-validator type="stringlength">
        <param name="maxLength>2000</param>
        <message key="securitykey.invalid.length" />
    </field-validator>
</field>

<field name="referrerHandle">
    <field-validator type="stringlength">
        <param name="minLength">2</param>
        <param name="maxLength>15</param>
        <message key="handle.invalid.length" />
    </field-validator>
</field>

<field name="password">
    <field-validator type="requiredstring">
        <message key="password.required" />
    </field-validator>
    <field-validator type="stringlength">
        <param name="minLength">7</param>
        <param name="maxLength">30</param>
        <message key="password.invalid.length" />
    </field-validator>
</field>
```

1.3.3 Action variable mapping

The Frontend will use the variable names defined in the Action classes for input and output parameters, in conjunction with OGNL. This also includes usage of complex entities where their fields are set. Below are the mappings.

Developer will consult OGNL more fully to see this at work.

1.3.3.1 Contact Info mapping

The field mapping will refer to the properties of the user field in the action, using OGNL.

Data Element	Field mapping
First Name	userProfile.firstName
Last Name	userProfile.lastName
E-mail	userProfile.emailAddresses[0].address
Job Title	userProfile.contact.title
Company Name	userProfile.contact.company.name
Current Address 1	userProfile.addresses[0].address1
Current Address 2	userProfile.addresses[0].address2
Current Address 3	userProfile.addresses[0].address3
City	userProfile.addresses[0].city
State	userProfile.addresses[0].state.code
Postal Code	userProfile.addresses[0].postalCode
Province	userProfile.addresses[0].province
Country	userProfile.addresses[0].country.id for editing and saving userProfile.addresses[0].country.name for viewing
Country to Represent	userProfile.coder.country.id for editing and saving userProfile.coder.country.name for viewing
Phone Number	userProfile.phoneNumbers[0].number
Timezone	userProfile.timezone.id for editing and saving userProfile.timezone.description for viewing

1.3.3.2 Demographic info mapping

The field mapping will refer to the properties of the user field in the action, using OGNL. The demographic data is dynamic, provided in a DemographicQuestion.

Each DemographicQuestion is effectively a line item, and its contents are used to display the question text, and based on the type, the selection of possible answers. When viewing and editing, the user.userProfile.demographicResponses provides the list of user demographic responses. Each response contains both the DemographicQuestion and DemographicAnswer or text. Each response of the user would be matched with the provided questions by ID to align the user's existing answers.

When saving, it is sufficient to map the user answers into the user.userProfile.demographicResponses, with the

- demographicResponse.question.id with the ID of the question
- demographicResponse.answer.id with the ID of the user-selected answer
- demographicResponse.response with the text of the response

1.3.3.3 Account Info mapping

The field mapping will refer to the properties of the user field in the action, using OGNL.

Data Element	Description
Username	handle
Password	password
Secret question	userProfile.secretQuestion.question
Secret question response	userProfile.secretQuestion.response
Student / Professional	userProfile.coder.coderType.id Student is ID 1 Professional is ID 2
Security Key	userSecurityKey.securityKey
Referrer handle	userProfile.coder.coderReferral.referral.id The ID is used to get the user.handle of the referral user

1.3.4 Audit

The audit will generally work as follows:

1. Create a new AuditRecord
2. Fill audit record (see below)
3. Use AuditDAO: audit(auditRecord)

The AuditRecord is filled as follows

AuditRecord field	value
operationType	Configurable. Possible values below
handle	authentication.user.userName
ipAddress	HttpServletRequest getRemoteAddr()
timestamp	new Date()
previousValue	Action-specific: See below
newValue	Action-specific: See below

Each action will require its own values for certain audit fields, although the operationType will be configurable and the value provided below is a possible value. The BaseProfileAction provides a convenience method getPreparedAuditRecord that sets the first four fields.

When auditing an updated entity, provide a name:value set, delimited by commas, as all updated fields must be audited using just one AuditRecord. For the name, we would use the Data Element name seen in CS 1.3. For example, suppose that in the contact info save operation, the first and last names have changed. Then the previous and new Values would look as below:

previousValue: “First Name:Jim, Last Name:Jones”
newValue: “First Name:John, Last Name:Smith”

ViewNameAndContactAction
One record will be created:
- operationType: “view profile contact info”
- previousValue: null
- newValue: null

EditNameAndContactAction
One record will be created:
- operationType: “edit profile contact info”
- previousValue: null
- newValue: null

SaveNameAndContactAction
One record will be created:
- operationType: “save profile contact info”
- previousValue: the old values of changed fields as per above
- newValue: the new values of changed fields as per above

If no fields were changed, then just one record will be created
- operationType: “save profile contact info”
- previousValue: null
- newValue: null

ViewDemographicInfoAction
One record will be created:
- operationType: “view profile demographic info”
- previousValue: null
- newValue: null

EditDemographicInfoAction
One record will be created:
- operationType: “edit profile demographic info”
- previousValue: null
- newValue: null

SaveDemographicInfoAction
One record will be created:
- operationType: “save profile demographic info”
- previousValue: the old values of changed fields as per above
- newValue: the new values of changed fields as per above

If no fields were changed, then just one record will be created
- operationType: “save profile demographic info”

- previousValue: null
- newValue: null

ViewAccountInfoAction
One record will be created:
- operationType: “view profile account info”
- previousValue: null
- newValue: null

EditAccountInfoAction
One record will be created:
- operationType: “edit profile account info”
- previousValue: null
- newValue: null

SaveAccountInfoAction
One record will be created:
- operationType: “save profile account info”
- previousValue: the old values of changed fields as per above
- newValue: the new values of changed fields as per above

If no fields were changed, then just one record will be created
- operationType: “save profile account info”
- previousValue: null
- newValue: null

ViewProfileCompletenessAction
One record will be created:
- operationType: “view profile complete info”
- previousValue: null
- newValue: null

CompleteProfileAction
One record will be created:
- operationType: “complete profile”
- previousValue: null
- newValue: null

1.3.5 Email Template Data

These are the Document Generator template data for the emails.

1.3.5.1 SaveNameAndContactAction

```
<DATA>
    <ID>user.id</ID>
    <HANDLE>user.handle</HANDLE>
    // the specific fields of the info type. See CS 1.3.3.1 for details
</DATA>
```

1.3.5.2 SaveDemographicInfoAction

```
<DATA>
    <ID>user.id</ID>
    <HANDLE>user.handle</HANDLE>
    // the specific fields of the info type. See CS 1.3.3.2 for details
</DATA>
```

1.3.5.3 SaveAccountInfoAction

```
<DATA>
    <ID>user.id</ID>
    <HANDLE>user.handle</HANDLE>
    // the specific fields of the info type. See CS 1.3.3.3 for details
</DATA>
```

1.4 Component Class Overview

BaseProfileAction
This is a base class for all profile actions. It provides a logger, audit DAO and user DAO for logging, auditing, and user management, respectively. It also provides the key for getting the authentication object from session. The session map is provided as part of this actions contract as being a SessionAware entity. The HttpServletRequest is provided as part of this actions contract as being a ServletRequestAware entity, although this is only for the sake of getting the caller's IP address for auditing. Similarly, the audit operation type is provided for auditing. A convenience method to get a pre-populated AuditRecord is also provided. Also provides the completeness report.

EmailSendingProfileAction
This is a base class for the actions that send an email, so it provides common fields and utilities for doing that, including a helper method to generate the body text, package the email message, and dispatch it. The generation of the body text is done with the Document Generator, and the email is sent with the EmailEngine. Also provides a flag whether the email should be sent.

BaseDisplayProfileAction
This action will perform some common tasks related to displaying profile information, regardless if it is in view or edit mode. Extensions will provide the specifics of how data is processed. It uses UserDAO to manage the user.

BaseSaveProfileAction
This action will perform some common tasks related to saving profile information. Extensions will provide the specifics of how data is processed. It uses UserDAO to manage the user. It will send an email if the configuration asks for it.

ProfileCompletenessRetriever
This interface defines the contract for getting information about the completeness of a profile registration process. It returns a report stating the percentage of

completion, as well as a list of tasks and whether each one of them is completed or not.

The percentage completion calculation algorithm will be determined by the implementations.

ProfileCompletenessReport
This is an entity class that contains information about the completion status of the registration for a user. It provides the info about the percentage of completion and the completion status of individual tasks associated with the completion.

TaskCompletionStatus
This is an entity class that contains information about the completion status of a task. The task name will be provided along with the flag whether that task has been completed.

ProfileTaskChecker
This interface defines the contract for getting information about how complete a given task is. It provides a report that states the task name, a count of total fields being checked and a count of how many fields are provided, and a flag to state if all fields that need to be provided are indeed provided.

The interface expects that the passed User instance is the complete instance that is to be checked, and that no additional steps are required to get more of the user information.

ProfileTaskReport
This is an entity class that contains information about the task. It provides the task name, a count of total fields being checked and a count of how many fields are provided, and a flag to state if all fields that need to be provided are indeed provided.

DefaultProfileCompletenessRetriever
This class implements ProfileCompletenessRetriever and provides an implementation that asks the ProfileTaskCheckers to state if they are completed, and to provide their totals of all applicable fields, and how many have been completed. It provides a report of the completion percentage by checking how many required fields have been set, and also states which stages (tasks) have not yet been completed.

BaseProfileTaskChecker
This is a base class for all ProfileTaskCheckers. It provides the task name field and initialization check that it is provided.

ContactInfoTaskChecker

This class implements ProfileTaskChecker to provide a check of the contact information task and determine how many of the required fields in this task has the passed user provided. The check on a field is simple and only verifies that the field is not null/empty.

DemographicInfoTaskChecker
This class implements ProfileTaskChecker to provide a check of the demographic information task and determine how many of the required fields in this task has the passed user provided. The check on a field is simple and only verifies that the field is not null/empty.

AccountInfoTaskChecker
This class implements ProfileTaskChecker to provide a check of the account information task and determine how many of the required fields in this task has the passed user provided. The check on a field is simple and only verifies that the field is not null/empty.

ViewNameAndContactAction
This action extends BaseDisplayProfileAction and processes the output by taking the passed user, which includes the name and contact info, and sets it to the output.

EditNameAndContactAction
This action extends ViewNameAndContactAction, but also gets the additional lookup lists required for the page.

SaveNameAndContactAction
This action extends BaseSaveProfileAction to save the name and contact info for the user, and generates the appropriate template data with the updated contact info data.

ViewDemographicInfoAction
This action extends BaseDisplayProfileAction and processes the output by taking the passed user and sets it to the output. It also provides the full set of questions.

EditDemographicInfoAction
This action extends ViewDemographicInfoAction, but does not provide any additional functionality, as the viewable action provides all required lookup data.

SaveDemographicInfoAction
This action extends BaseSaveProfileAction to save the name and contact info for the user, and generates the appropriate template data with the updated contact info data.

ViewAccountInfoAction

This action extends BaseDisplayProfileAction and processes the output by taking the passed user, which includes the account info, as well as the handle of the refarral user, and sets it to the output.

EditAccountInfoAction
This action extends ViewAccountInfoAction, but also gets the additional lookup lists required for the page.

SaveAccountInfoAction
This action extends BaseSaveProfileAction to save the account info for the user, and generates the appropriate template data with the updated account info data.

ViewProfileCompletenessAction
This action extends BaseDisplayProfileAction and processes the output by simply taking the passed user and sets it to the output. It also provides all the necessary lookups.

CompleteProfileAction
This action extends BaseProfileAction to simply forward to the appropriate save action based on the current tab.

CurrentTab
This enumeration provides the types of tabs that are available.

1.5 Component Exception Definitions

This component defines new exceptions

ProfileActionConfigurationException
This exception is thrown by various actions in their PostConstruct methods if these classes are not properly initialized (e.g. while required property is not specified or property value has invalid format).

ProfileCompletenessRetrievalException
This exception is thrown by the ProfileCompletenessRetriever if there is any error while executing the retrieval.

ProfileActionException
This exception is thrown by the execute method and any helper methods in all actions if there is any error while executing the operations.

1.6 Thread Safety

The actions and interceptors do not need to be thread-safe as they will be handled in a thread-safe manner by the web container. This holds for the used services as well, as they are assumed to be effectively thread-safe as well.

Effective thread-safety is achieved in general when a class does not introduce any state that changes during invocation or the process does not use mutable input/output in more than one thread. This is achieved in this component.

2. Environment Requirements

2.1 Environment

- Development language: Java 1.6, J2EE 1.5
- Compile target: Java 1.6, J2EE 1.5
- JBoss 4.0.4 GA
- Informix 11.5

2.2 TopCoder Software Components

- TopCoder Registration CodeBase
    - This is the existing codebase being managed.
    - Registration codebase: https://coder.topcoder.com/internal/web_module/trunk
    - Shared codebase: https://coder.topcoder.com/internal/shared/trunk
- User Profile and Audit Back End 1.0
    - Provides audit and user entities and DAOs.
- Document Generator 3.1.1
    - Provides the facility to generate messages.
- Base Exception 2.0
    - Provides the BaseRuntimeException, BaseCriticalException, and ExceptionData.
- Logging Wrapper 2.0
    - Used for logging activity and errors.

2.3 Third Party Components

- Struts 2.2.3: http://struts.apache.org/index.html
- Spring 2.5.6 (used to create objects and configure them by IoC injection): http://www.springsource.com/products/spring-community-download
- Log4j 1.2.x for logging, via the Logging Wrapper: http://logging.apache.org/log4j/1.2/download.html

3. Installation and Configuration

3.1 Package Names

com.topcoder.web.reg.actions.profile

3.2 Configuration Parameters

None

3.3 Dependencies Configuration

The developer should read the specifications for all components specified in section 2.2 to see how they are configured.

4. Usage Notes

4.1 Required steps to test the component

- Start email server applet at http://www.aboutmyip.com/AboutMyXApp/RunDevNullSmtp.jsp
- Configure paths in "build-dependencies.xml"

4.2 Required steps to use the component

*1. Configure libraries paths in **build-dependencies.xml**.*

*2. Execute **ant demo_war**.*

*3. Deploy **demo.war** into Tomcat server.*

*4. Go to **http://localhost:8080/demo/** and see what you can view and edit.*

NOTE: I'm using mock data for demo, so if you make some changes to data it will not saved to back-end, only exposed to front-end.

Deploy the application to J2EE container, for example:

```
demo.war
    -- WEB-INF
       -- applicationContext.xml
       -- web.xml
       -- classes
          -- struts.xml
          -- struts.properties
          -- ……
       -- lib
          -- ….. libraries
    -- example
    -- …… jsp pages
```

4.3 Demo

The application can be demonstrated using the following scenarios, from the perspective of JSP usage.

4.3.1 Setup

The struts.xml file would look approximately like this for the five actions:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
  <constant name="struts.devMode" value="true" />
```

```
  <package name="default" namespace="/" extends="struts-default">

            <action name="viewNameAndContactAction"
class="com.topcoder.web.reg.actions.profile.ViewNameAndContactAction" >
                        <result
name="success">myprofile_view_contact_info.jsp</result>
            </action>

            <action name="editNameAndContactAction"
class="com.topcoder.web.reg.actions.profile.EditNameAndContactAction" >
                        <result
name="success">myprofile_edit_contact_info.jsp</result>
            </action>

            <action name="saveNameAndContactAction"
class="com.topcoder.web.reg.actions.profile.SaveNameAndContactAction" >
                        <result
name="success">myprofile_confirm_contact_info.jsp</result>
            </action>

            <action name="viewDemographicAction"
class="com.topcoder.web.reg.actions.profile.ViewDemographicAction" >
                        <result
name="success">myprofile_view_demographic_info.jsp</result>
            </action>

            <action name="editDemographicAction"
class="com.topcoder.web.reg.actions.profile.EditDemographicAction" >
                        <result
name="success">myprofile_edit_demographic_info.jsp</result>
            </action>

            <action name="saveDemographicAction"
class="com.topcoder.web.reg.actions.profile.SaveDemographicAction" >
                        <result
name="success">myprofile_confirm_demographic_info.jsp</result>
            </action>

            <action name="viewAccountInfoAction"
class="com.topcoder.web.reg.actions.profile.ViewAccountInfoAction" >
                        <result
name="success">myprofile_view_account_info.jsp</result>
            </action>

            <action name="editAccountInfoAction"
class="com.topcoder.web.reg.actions.profile.EditAccountInfoAction" >
                        <result
name="success">myprofile_edit_account_info.jsp</result>
            </action>

            <action name="saveAccountInfoAction"
class="com.topcoder.web.reg.actions.profile.SaveAccountInfoAction" >
                        <result
name="success">myprofile_confirm_account_info.jsp</result>
            </action>

            <action name="viewProfileCompletenessAction"
class="com.topcoder.web.reg.actions.profile.ViewProfileCompletenessAction" >
                        <result
name="success">myprofile_view_registration_info.jsp</result>
            </action>
            <action name="completeProfileAction"
class="com.topcoder.web.reg.actions.profile.CompleteProfileAction" >
                        <result name="Save Contact">saveNameAndContactAction</result>
                        <result name="Save
Demographics">saveDemographicAction</result>
                        <result name="Save Account">saveAccountInfoAction</result>
            </action>
```

```
  </package>

</struts>
```

The configuration of the actions is straightforward with spring. Some of the base beans, such as the services and logger, would be configured in the following manner. A snippet of the configuration is shown, with assumed existing classes for the services and DAOs:

```
<bean id="logger" class="com.topcoder.util.log.LogFactory" factory-
method="getLog">
  <constructor-arg value="myLogger"/>
</bean>
```

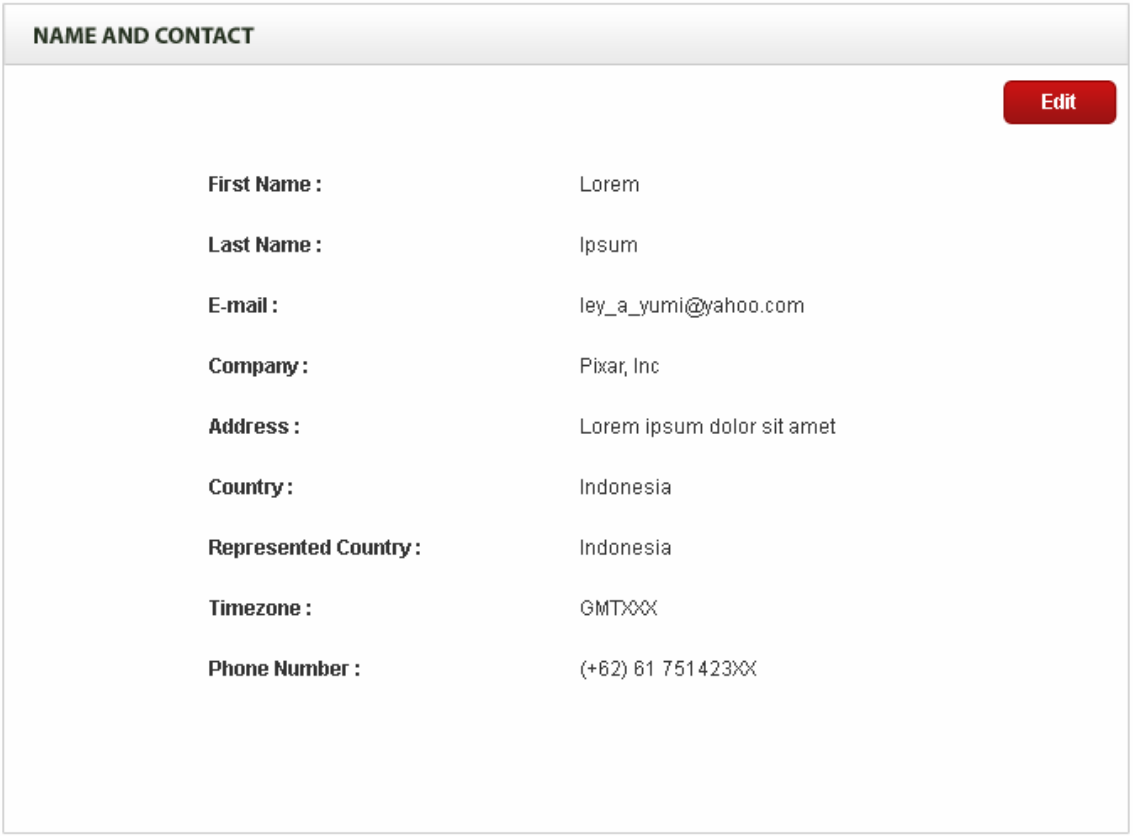
The actions would be configured as this example shows for the group for contact info. Other actions will be done in the same manner.

```
<bean id="viewNameAndContactAction"
class="com.topcoder.web.reg.actions.profile.ViewNameAndContactAction"
scope="prototype">
  <property name="logger" ref="logger"/>
  <property name="auditDAO" ref="auditDAO"/>
  <property name="authenticationSessionKey" value="authentication"/>
  <property name="userDAO" ref="userDAO"/>
  <property name="auditOperationType" ref="Save contact"/>
  <property name="profileCompletenessRetriever"
ref="profileCompletenessRetriever"/>
</bean>

<bean id="editNameAndContactAction"
class="com.topcoder.web.reg.actions.profile.SaveNameAndContactAction"
scope="prototype">
  <property name="logger" ref="logger"/>
  <property name="auditDAO" ref="auditDAO"/>
  <property name="authenticationSessionKey" value="authentication"/>
  <property name="userDAO" ref="userDAO"/>
  <property name="auditOperationType" ref="Save contact"/>
  <property name="profileCompletenessRetriever"
ref="profileCompletenessRetriever"/>
  <property name="countryDAO" ref="countryDAO"/>
  <property name="timeZoneDAO" ref="timeZoneDAO"/>
</bean>

<bean id="saveNameAndContactAction"
class="com.topcoder.web.reg.actions.profile.SaveNameAndContactAction"
scope="prototype">
  <property name="logger" ref="logger"/>
  <property name="auditDAO" ref="auditDAO"/>
  <property name="authenticationSessionKey" value="authentication"/>
  <property name="userDAO" ref="userDAO"/>
  <property name="auditOperationType" ref="Save contact"/>
  <property name="profileCompletenessRetriever"
ref="profileCompletenessRetriever"/>
  <property name="documentGenerator" ref="documentGenerator"/>
  <property name="emailSubject" value="Profile Contact Info Updated"/>
  <property name=" emailBodyTemplateSourceId" value="default"/>
  <property name="emailBodyTemplateName" value="peofileContactInfo"/>
  <property name="emailSender" value="ivern@topcoder.com"/>
  <property name="sendEmail" value="true"/>
</bean>
```

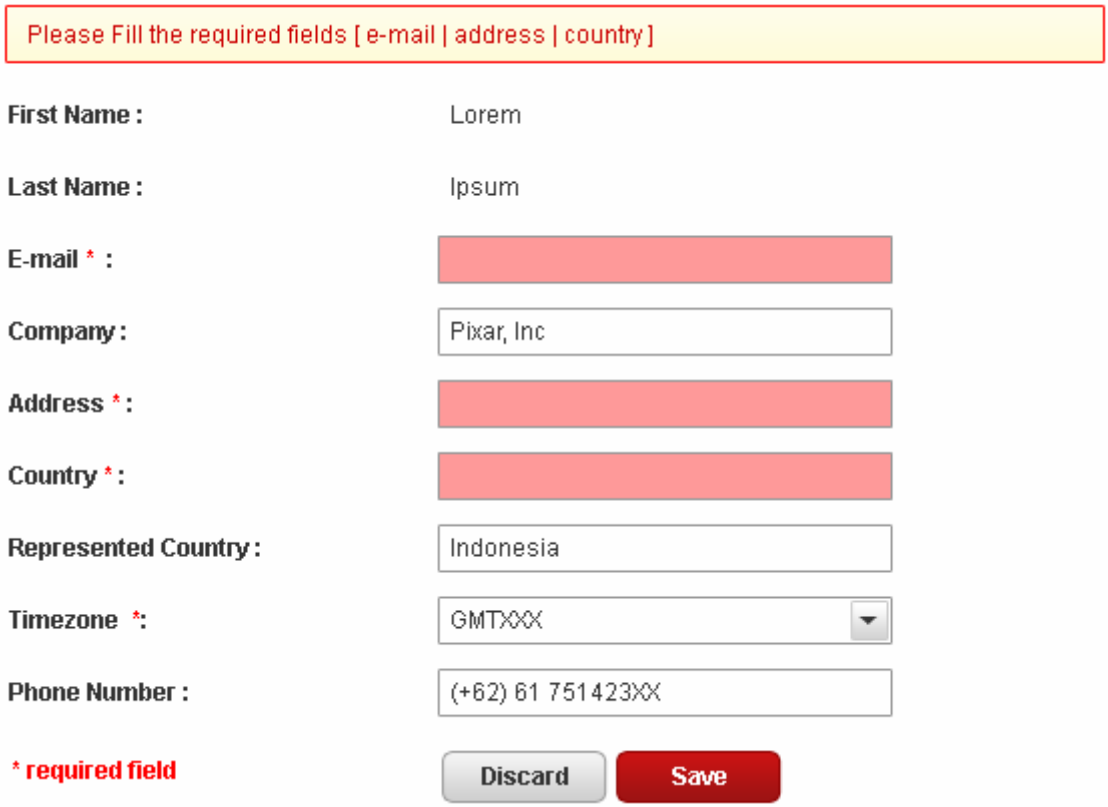
4.3.2 A step-through the management of contact info

The management of the contact info is a good representation of this component.

The actions beings with the invocation of the ViewNameAndContactAction, which loads the user contact data. It would appear as follows:



When the user presses the Edit button, the screen becomes editable, and the EditNameAndContactAction loads the lookup values.



When the user presses the Save button, the data is validated, and the SaveNameAndContactAction is activated. The result is that the data is saved, and an email is dispatched to the user stating the changes.

The other set of actions works in a similar way.

5. Future Enhancements

None.