Teams in Business Process Manager V8.5, Part 2: Developing services for dynamic team resolution

Gabriel Dermler (gabriel.dermler@de.ibm.com)
Senior Software Engineer
IBM

Markus Reichart (markus.reichart@de.ibm.com)
Software Architect
IBM

Thomas Bernhardt (thomas.bernhardt@de.ibm.com)
Development Team Lead
IBM

Thomas Duerr (thomas.duerr@de.ibm.com)
Software Engineer
IBM

19 February 2014

Part 2 of this series provides insight into developing dynamic teams to get the most out of the new team functionality. The Team concept has been explained in the first article in this series. In this article, we get practical with dynamic teams. We introduce team retrieval and team filter services and we provide simple and advanced examples, including a sample implementation that shows how to use services to apply substitution and ensure separation of duties.

View more content in this series

Introduction

The team functionality introduced in IBM® Business Process Manager V8.5 (IBM BPM) enables you to develop services to customize the logic that is used to determine users and groups for user assignment. Team retrieval and team filter services allow you to programmatically derive users and groups to use at runtime. This article gives an overview of how to model these services in IBM Process Designer, and it provides examples for exploiting the IBM BPM JavaScript API (JS API) to develop team retrieval and team filter services.

Developing team retrieval services

A team retrieval service is an implementation of an integration service based on the predefined Team Retrieval Service Template that is included in IBM BPM V8.5. The template mandates that
the service returns a team parameter of the predefined BO (business object) Team type and takes a name (String) parameter as input. You cannot change the output definition for a team retrieval service. However, you can change the input definition by adding custom parameters to it. The example in Figure 1 shows the predefined parameters with an additional region custom input parameter. The output BO Team type contains a list of (user or group) member names and the name of the associated manager team. The team name parameter is currently not exploited.

**Figure 1. Defining the interface of a team retrieval service**

![image of variables](image)

At runtime, when the team details are required for an assignment, for example, for assigning a team to a task, the service is invoked with the required input values and it produces the resulting team object as shown in Figure 2.

**Figure 2. Modeling a team retrieval service**

![image of process flow](image)

A team retrieval service can be implemented in the same way as other types of integration services. It can include any of the following components:

- Server scripts that define JavaScript code, which uses the JS API methods.
• Java integration components that use predefined integration utility classes, for example, to define SQL queries or HTTP client calls.
• Web service integration components to define access to web services.
• Nested services to define other (system or integration) services as part of the team retrieval service.
• Nested services to enable scenarios, such as business rules for determining the users and groups of a team.

Using the JS API to develop a team retrieval service

The IBM BPM JavaScript API (JS API) provides a set of team, user, and group methods that make it easy to design and implement a team retrieval service. These methods can access the native BPM user repository to retrieve and store the same user and group information that is also exposed in the Process Admin Console. Methods are also provided to access team definitions that are in a process application or toolkit.

These methods can be grouped into the following categories. Team retrieval services use mainly methods in the first two categories.

• Team, user, and group finder methods. The methods allow you to create objects that represent team, user, or group descriptions: TWTeam, TWUser, TWGroup. For a list of these methods, see Appendix A.
• TWTeam, TWUser, TWRole class methods to read and manipulate team, user, or group data. You can also use these class methods in a team retrieval service implementation. For a list of these methods, see Appendixes B through D.
• TWTask methods to read and manipulate team, user, or group data for a task

Example 1: Basic Team Retrieval Service

The Basic Team Retrieval Service example in Figure 3 shows how to use a service to construct a team from individual users and group members. It uses the interface defined by the Team Retrieval Service template. Its implementation includes a single script element as shown in Listing 1.

Figure 3. Modeling a simple team retrieval service
Listing 1. Team computation implementation

```javascript
// **************************************************
// approach 1: specify user and group names
var resultTeam = new tw.object.Team();
resultTeam.members = new tw.object.listOf.String();

resultTeam.members[0] = "David";
resultTeam.members[1] = "Linda";
resultTeam.members[2] = "Region North";

// provide the result
tw.local.team = resultTeam;
```

Both user and group members are collected in the same list. If a user and a group have the same name, it is assumed that the team member is an individual user. This is a limitation of the current JS API support.

User and group names can be specified, regardless of whether they exist in the BPM user repository. If the names are not in the user repository, they are ignored at runtime. If users and groups are added later, they are included in the team result when the team is refreshed. For more information, see the article in this series on managing teams.

To ensure that users and groups are available for the team retrieval service, you can use the corresponding JS API finder methods in the service implementation. The Team Computation implementation shown in Listing 2 includes a BPM user repository look-up for users and groups.

Listing 2. Team Computation implementation with lookup

```javascript
// **************************************************
// approach 2: lookup the users and groups by their names
var resultTeam = new tw.object.Team();
resultTeam.members = new tw.object.listOf.String();
var count = 0;
var user = tw.system.org.findUserByName("David");
if (user!=null) {
  resultTeam.members[count++] = user.name;
}
var user = tw.system.org.findUserByName("Linda");
if (user!=null) {
  resultTeam.members[count++] = user.name;
}
var group = tw.system.org.findRoleByName("Region North");
if (group!=null) {
  resultTeam.members[count++] = group.name;
}
// provide the result
tw.local.team = resultTeam;
```

Alternatively, you can look up another team and clone its definition for your new team definition as shown in Listing 3.
Listing 3. Team computation implementation with cloning a team

```javascript
// approach 3: lookup a team and copy its definition
var resultTeam = new tw.object.Team();
var team = tw.system.org.findTeamByName("Audit Team");
if (team != null) {
    // optionally, refresh team definition to be up-to-date
    team.refresh();
    // clone and convert into Team BO
    resultTeam = team.asTeam();
} else {
    // some fallback
    resultTeam.members = new tw.object.listOf.String();
    resultTeam.members[0] = "David";
}
```

Example 2: Using parameters in team retrieval services

The Sales Team Service example in Figure 4 shows how to use parameters in team retrieval services. It uses a simple extension of the interface defined by the Team Retrieval Service template. The additional `region` parameter is used to describe the scope of the team definition. It is exploited in the service implementation to use specified groups as the basis for team members (see Listing 4). The `managerTeam` parameter will be explained in a later example.

Figure 4. Using parameters in a team retrieval service
Listing 4. Team computation implementation

```javascript
// initialize team BO
var resultTeam = new tw.object.Team();
resultTeam.members = new tw.object.listOf.String();

// populate team members depending on received param value
if (tw.local.region == "North") {
    resultTeam.members[0] = "Bonnie";
    resultTeam.members[1] = "David";
} else if (tw.local.region == "South") {
    resultTeam.members[0] = "Region South";
} else {
    var team = tw.system.org.findTeamByName("Account Team");
    resultTeam = team.asTeam();
}

// set the manager team
resultTeam.managerTeam = tw.local.managerTeam;

// set team as output
tw.local.team = resultTeam;
```

Example 3: Combining groups and teams

The Team Intersection Service example in Figure 5 shows how to create a team by combining groups and teams. It allows you to specify a list of group names and a list of team names as the input. The resulting team contains the users who belong to all the specified groups and teams. The service interface contains two corresponding input parameters `groups` and `teams`, each represented by comma-separated value lists as shown in Listing 5.

Figure 5. Using groups in a team retrieval service

Listing 5. Team computation implementation

```javascript
// list of all TwRole objects for input groups AND teams
var allGroups = new Array();
var groupCount = 0;

// get the TwRole objects for input groups
var groupsParam = tw.local.groups;
if (groupsParam != null) {
    var splitGroups = groupsParam.split(",");
    for (var i = 0; i < splitGroups.length; i++) {
        var groupName = splitGroups[i].trim();
        var group = tw.system.org.findRoleByName(groupName);
    }
```
if (group!=null) {
    allGroups[groupCount++] = group;
}

// get the Team objects from input param teams
var teamsParam = tw.local.teams;
if (teamsParam!=null) {
    var splitTeams = teamsParam.split(",");
    for(var i=0; i<splitTeams.length; i++) {
        var teamName = splitTeams[i].trim();
        var team = tw.system.org.findTeamByName(teamName);
        if (team!=null) {
            // use the team's run-time definition
            allGroups[groupCount++] = team.associatedRole;
        }
    }
}

// get the users belonging to the intersection of all groups
var commonUsers = tw.system.org.findCommonUsersForRoles(allGroups);
var resultTeam = new tw.object.Team();
resultTeam.members = new tw.object.listOf.String();
for(var i=0; i<commonUsers.length; i++) {
    resultTeam.members[i] = commonUsers[i].name;
}

// set the result
tw.local.team = resultTeam;

The implementation uses the associatedRole property of a team to determine the supported users. This property corresponds to the team’s runtime definition and will, therefore, reflect updates that are done in the Process Admin Console.

The internal implementation of tw.system.org.findCommonUsersForRoles(...) implies that the complete set of users is retrieved for each specified group. Performance issues can therefore result if you implement this example with very large groups, such as "All Users".

A similar solution can be designed to retrieve the union of group and team members by using the tw.system.findAllUsersForRoles(...) JS API method.

**Example 4: Filtering by user attribute**

The User Attribute Driven Service example in Figure 6 shows how to select team members based on user attribute values. The service interface in Listing 6 includes parameters for specifying the team name and the condition to be used for filtering team members, such as “Locale=en”.
Figure 6. Filtering in a team retrieval service

Listing 6. Team computation implementation

```javascript
// get the attribute condition
var attrFilter = tw.local.attributeFilter;
var filterParts = attrFilter.split("=");
var reqAttrName = filterParts[0].trim();
var reqAttrValue = filterParts[1].trim();

// get the team members
var reqTeam = tw.system.org.findTeamByName(tw.local.teamName);
var allUsers = reqTeam.allUsers;

// get the filtered users
var filteredUsers = new tw.object.listOf.String();
var userCount = 0;
for (var i=0; i<allUsers.length; i++) {
    var user = allUsers[i];
    if (reqAttrValue == user.attributes[reqAttrName]) {
        filteredUsers[userCount++] = user.name;
    }
}

// set the result
var resultTeam = new tw.object.Team();
resultTeam.members = filteredUsers;
tw.local.team = resultTeam;
```

Example 5: Users and their backups

The User Substitution Service example in Figure 7 exploits a Substitute user attribute that is defined as an additional user attribute in either a process application or toolkit. This attribute contains the name of the backup person for when the original user is on leave. The team retrieval service returns the user and the backup as team members. If a backup is not defined, only the original user is included in the resulting team.

In Listing 7, the service interface contains an input parameter for the name of the original team that includes the users before the user substitution is applied. A second parameter denotes the name of the substitution attribute.
**Figure 7. Filtering in a team retrieval service**

![Image of filtering in a team retrieval service](image-url)

**Listing 7. Team computation implementation**

```javascript
// get the substitution attribute name
var substAttrName = tw.local.substitutionAttribute;

// get the team members
var reqTeam = tw.system.org.findTeamBy_Name(tw.local.baseTeam);
var teamUsers = reqTeam.allUsers;

// get the users including substitutes
var collectedUsers = new tw.object.listOf.String();
var userCount = 0;
for(var i=0; i<teamUsers.length; i++) {
    // add user
    var user = teamUsers[i];
    collectedUsers[userCount++] = user.name;
    // add substitute, if available
    var substitute = user.attributes[substAttrName];
    if (substitute!=null) {
        collectedUsers[userCount++] = substitute;
    }
}

// set the result
var resultTeam = new tw.object.Team();
resultTeam.members = collectedUsers;
tw.local.team = resultTeam;
```

**Example 6: Member-independent and member-specific managers**

You can easily extend the previous examples to include the name of a manager team in the resulting Team object as shown in Listing 8.

**Listing 8. Extending Listing 7**

```javascript
// set the result
var resultTeam = new tw.object.Team();
resultTeam.managerTeam = "Region Managers North";
tw.local.team = resultTeam;
```

This is a good approach to take if a team and its associated manager team can be defined independently of one another, that is, if the manager team does not change when the members
of the managed team change. In most scenarios, in particular when leveraging the Process Portal Team Performance dashboard in IBM BPM V8.5, this assignment pattern should suffice. When this assumption does not hold, a different approach is possible as shown in Figure 8.

Each user is associated with a specific manager. This means that the manager team comprises the managers of all users. When a user joins or leaves a team, the corresponding manager has to be added to or removed from the associated manager team.

The **Managers Retrieval Service** returns a manager team that takes a `managedTeam` input parameter, which indicates the name of the managed team. It assumes that for every user a Manager user attribute can be set, for example, by using user attribute administration in the Process Admin Console.

**Figure 8. Defining a team’s manager team in a team retrieval service**

The implementation looks up the users who form the members of the `managedTeam` team. If the `Manager` attribute is set, it adds the name to the member list of the manager team. If the attribute is not set, the `defaultManager` name is added to the member list of the manager team (see Listing 9).

**Listing 9. Team computation implementation**

```javascript
// get the managed team
var managedTeamName = tw.local.managedTeam;

// get the team members
var managedTeam = tw.system.org.findTeamByName(managedTeamName);
var allUsers = managedTeam.allUsers;
log.info("allUsers:" + allUsers);

// get the managing users
var managers = new tw.object.listOf.String();
var userCount = 0;
var defaultManager = tw.local.defaultManager;
for (var i=0; i<allUsers.length; i++) {
    var user = allUsers[i];
    log.info("user" + i + ":" + user);
    var manager = user.attributes["Manager"];
    if (manager!=null) {
        managers[userCount++] = manager;
    }
}```
else {
    // use default, if manager is not set
    managers[userCount++] = defaultManager;
}
log.info("managers:" + managers);

// set the result
var resultTeam = new tw.object.Team();
resultTeam.members = managers;
tw.local.team = resultTeam;

The implementation is generic so you can use it to compute the manager team for any managed team.

Assume that the managed team is computed by a service as well. Then the two services (generating the managed team and generating the manager team, respectively) can be linked in the following way:

- The service computing the manager team must get the name of the managed team as an input parameter because it needs the name to resolve the contained users.
- The service computing the managed team must get the name of the manager team as an input parameter so that it can set the name in the Team result object, as shown in Listing 8.

In other words, the computed managed and manager teams have to be aware of each other, as shown in Figure 9 for Sales Team North and Managers Sales Team North.

**Figure 9. Defining a team and its manager team**

When defining or refreshing a team, first the team members and then the associated manager team are determined. For this reason, the preceding example determines first the members of the Sales Team North, and then the managers of the team, Managers Sales Team North. This
coupling ensures that a team and its manager team stay synchronized in the face of dynamic membership changes.

Developing team filter services

Team filter services are used when modeling human activities of a business process to customize the team that is used for people assignment at runtime (see Figure 10). You can use team filter services, for example, to filter out users that do not have the required skill level to work on a particular task, or to enable scenarios, such as the separation of duties.

A team filter service is an implementation of an integration service that is based on the Team Filter Service Template. The template mandates that the service returns a `filteredTeam` parameter of predefined BO `Team` type and takes as input an `originalTeam` parameter of the BO `Team` type. You cannot change the output definition of a team filter service, but you can change the input definition by adding additional parameters.

Figure 10. Defining the interface of a team filter service

Team filter services are used at runtime to determine which team members and which managers' teams are assigned to an activity. When a task with a team filter service is created at runtime, the service is invoked with required input values and receives the filtered team as the result.

A team filter service can be implemented in the same way as other services. They can include server scripts, Java integration, web service integration, or nested services components.

Team filter services can be defined in the context of a User Task activity in a process only. As a consequence, you can use the process instance variables as input parameters to the team filter service.

While the life-cycle of a team is bound to the life-cycle of the enclosing process application, the life-cycle of a filtered team is bound to the life-cycle of the enclosing process instance.

Moreover, a team with a given name is unique and the team identified by this name is shared across the process application. In contrast, a filtered team is associated only with its corresponding task; therefore, it cannot be shared in any way.
Example: Filter Service ensuring separation of duties

This example demonstrates how a team filter service can access context data of the enclosing instance context. Its interface includes a `userFilter` parameter for the user that is to be excluded from the team associated with the task.

In Figure 12, the example process definition contains two activities, `Offer Submission` and `Offer Evaluation`. A variable `userToExclude` is defined for the process as shown in Figure 11.

**Figure 11. Defining the variables of a process**

![Variables diagram](image)

The variable value is defined via a `post mapping` in the definition of the `Offer Submission` activity. The variable then contains the user who completed the activity `Offer Submission` as shown in Figure 12.

**Figure 12. Modeling a process**

![Process diagram](image)

Both the `Offer Submission` and the `Offer Evaluation` activities are associated with the same team `Sales Team North`. The latter activity is associated in addition with the `Separation of Duties Service`. For this service, an input mapping is defined from the `tw.local.userToExclude` instance variable to the `userFilter` service input parameter as shown in Figure 13.
The team filter service interface includes the additional `userFilter` parameter. Its implementation is based on a server script element as shown in Figure 14.

**Figure 13. Modeling a task with a team filter service**

![Diagram showing a task with a team filter service](image)

The script implementation retrieves all the users belonging to the team that is associated with the task and then filters out the specified user to exclude as shown in Listing 10.

**Figure 14. Modeling this team filter service**

![Diagram showing the implementation of a team filter service](image)
Listing 10. Team computation implementation

```javascript
// get all users of the original team
var origTeam = tw.system.org.findTeamByName(tw.local.originalTeam.name);
var allUsers = origTeam.allUsers;

// compute result team members
var resultTeam = new tw.object.Team();
resultTeam.members = new tw.object.listOf.String();
var userCount = 0;
for(var i=0; i<allUsers.length; i++) {
    var user = allUsers[i].name;
    if (tw.local.userFilter != user) {
        resultTeam.members[userCount++] = user;
    }
}

// set the result
tw.local.filteredTeam = resultTeam;
```

Note that the filtered team name for a created task is not related to the original team name and it cannot be used for explicit processing as shown in Figure 15.

**Figure 15. Naming of a team defined by a team filter service**

![Figure 15](image-url)

Also note that in the Team Performance dashboard, tasks that are associated with filtered teams are aggregated under the original team to which filtering was applied. In the example, this is the *Sales Team North* as shown in Figure 16.

**Figure 16. Aggregation of teams defined by a team filter service**

![Figure 16](image-url)
Caching team retrieval or team filter service results

Any integration service can be defined to cache its results for a specified duration (see Figure 17). For every combination of (service name, (input par1 name, input par1 value), ..., (input parN name, input parN value)), a distinct service result is automatically added to the cache when caching is enabled. When the service is invoked, the cache is checked. If a matching result is found, the cached result is used directly without calling the service. If a matching result is not found in the cache, or the cached result has expired, then the service is invoked and the cache is refreshed.

Figure 17. Defining the caching of service results

![Diagram showing cache settings for a team service](image)

For team retrieval or team filter services, this means that when a team definition is determined or refreshed, the associated service uses the cached result, as long as it has not expired.

Conclusion

In Part 2 of the series, we have discussed team retrieval services and team filter services and we looked at the built-in capabilities for caching service results. We have demonstrated by example how to leverage the new service-based capabilities to create simple and sophisticated services for dynamic teams, including services to support participant substitution and separation of duties. In Part 3, we are going to discuss the administrative aspects of teams.

Appendices

This section displays an extract of the BPM JavaScript API that is related to teams.

Appendix A: JS API team, user, and group finder methods

Table 1 provides an overview of the finder methods for teams, users, and groups.

Table 1. Finder methods for teams, users, and groups

<table>
<thead>
<tr>
<th>Package / Result type</th>
<th>Call name</th>
<th>Search scope</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>tw.system.org TWTeam</td>
<td>findTeam(String teamId, String processAppId)</td>
<td>Specified process app</td>
<td>Find a team given its ID, in the specified process application.</td>
</tr>
<tr>
<td>tw.system.org TWTeam</td>
<td>findTeamByName(String teamName)</td>
<td>Current process app</td>
<td>Find a team given its name, in the current process application.</td>
</tr>
</tbody>
</table>
Teams in Business Process Manager V8.5, Part 2: Developing services for dynamic team resolution

The search scope of the team finder methods is restricted to the process application in which a specific integration service is specified. As a result, only the team that is defined in this process application (and toolkits included in the process application) will be found. In contrast, users are defined and treated globally. All users or groups in the BPM user repository are covered by the finder methods.

### Appendix B: TWTeam class methods

This class is used to represent a team definition, whether a team with standard members, a team is defined by a team retrieval service, or any of the deprecated definition types.

The following methods shown in Table 2 support the retrieval of team definition information.

#### Table 2. TWTeam class methods

<table>
<thead>
<tr>
<th>Return type</th>
<th>Property / method name</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>id</td>
<td>The team ID.</td>
</tr>
<tr>
<td>String</td>
<td>name</td>
<td>The team name.</td>
</tr>
<tr>
<td>String</td>
<td>getTypeName()</td>
<td>The team definition type (&quot;TWTeam&quot;).</td>
</tr>
<tr>
<td>TWRole[]</td>
<td>roles</td>
<td>The roles used in the team definition.</td>
</tr>
<tr>
<td>TWUser[]</td>
<td>users</td>
<td>The users used in the team definition.</td>
</tr>
<tr>
<td>TWTeam</td>
<td>getManagerTeam()</td>
<td>The defined manager team.</td>
</tr>
</tbody>
</table>

#### Table 3. Team definition modifications

<table>
<thead>
<tr>
<th>Return type</th>
<th>Property / method name</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The following methods in Table 4 perform explicit user membership operations.

### Table 4. User membership operations

<table>
<thead>
<tr>
<th>Return type</th>
<th>Property / method name</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>TWUser[]</td>
<td>allUsers</td>
<td>All users who are directly defined or indirect (for example, group) team members.</td>
</tr>
<tr>
<td>boolean</td>
<td>hasUser(TWUser user)</td>
<td>True, if the user is a direct or indirect member of the team.</td>
</tr>
</tbody>
</table>

For any given team, two definitions are tracked:

1. The *modeled team definition* as defined in Process Designer
2. The *runtime definition* corresponding to what was modeled, but with runtime changes done in the Process Admin Console or using JS API commands

A team's runtime definition is refreshed before being exploited for a task assignment or due to an administrative action. By default, the result of a refresh action leads to re-evaluating the modeled definition and using the result as runtime definition.

For example, for a team that is defined using a team retrieval service, the service is invoked again to determine the team definition users and groups. The obtained users and groups are then used as runtime definition.

The runtime definition is additionally affected by changes applied using the Process Admin Console to add or remove users or groups. When such a change is applied, the runtime definition gets decoupled from the model definition. In consequence, no refresh can delete the applied administrative changes. In order to re-enable refreshes to the runtime definition, the team's Automatic Refresh flag has to be explicitly enabled (see Table 5).
Table 5. Methods to refresh the runtime definition of a team

<table>
<thead>
<tr>
<th>Return type</th>
<th>Property / method name</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>TWRole</td>
<td>associatedId</td>
<td>The associated group representing the team's runtime definition.</td>
</tr>
<tr>
<td>Void</td>
<td>setAutomaticRefresh(boolean enable)</td>
<td>Enable or disable refresh enforcement between the modeled definition and the runtime definition.</td>
</tr>
<tr>
<td>Void</td>
<td>refresh()</td>
<td>Explicitly refresh a modeled team definition. For teams defined using a team retrieval service: Re-invoke the service. Refresh the runtime definition, if the AutomaticRefresh flag is set to true. For any other team type: Refresh the runtime definition if the AutomaticRefresh flag is set to true.</td>
</tr>
</tbody>
</table>

Note that the TWTeam class is different from the BO Team type considered earlier (see Using the JS API to develop a team retrieval service). The BO Team type was introduced as part of the Team Retrieval Service Template because a service implementation has to provide a BO Team object as the output. The implementation can exploit JS API methods to establish internal TWTeam objects (which correspond to existing database entries). As Table 6 shows, a method is provided to convert TWTeam objects to BO Team objects.

Table 6. Convert a JS TWTeam object into a Team data type

<table>
<thead>
<tr>
<th>Return type</th>
<th>Property / method name</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Team</td>
<td>asTeam()</td>
<td>Converts a TWTeam into a BO Team type.</td>
</tr>
</tbody>
</table>

Listing 3 contains an example on how to use it.

Appendix C: TWUser class methods

This class represents the description of a user, its attributes, and group and team memberships, as given in the BPM user repository.

The following methods in Table 7 support the retrieval of user information (skipped methods without obvious use for teams).

Table 7. TWUser class methods

<table>
<thead>
<tr>
<th>Return type</th>
<th>Property / method name</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>Name</td>
<td>The user name (must be unique).</td>
</tr>
<tr>
<td>String</td>
<td>fullName</td>
<td>The full name of the user.</td>
</tr>
<tr>
<td>int</td>
<td>Id</td>
<td>The user ID in the BPM user repository.</td>
</tr>
<tr>
<td>String</td>
<td>getTypeName()</td>
<td>Returns the name of the type (&quot;TWUser&quot;).</td>
</tr>
<tr>
<td>Record</td>
<td>Attributes</td>
<td>The defined attributes for the user.</td>
</tr>
</tbody>
</table>
Predefined user attribute types are provided in the System Data toolkit. You can define additional user attribute types in Process Designer as part of a process application or toolkit. The JS API provides the following methods shown in Table 8 for manipulating the user attributes.

### Table 8. Attribute-related methods in TWUser class

<table>
<thead>
<tr>
<th>Return type</th>
<th>Property / method name</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>removeAttribute(String name)</td>
<td>Removes the given attribute and its values for the given user.</td>
</tr>
<tr>
<td>void</td>
<td>setAttributeValue(String name, String value)</td>
<td>Set the specified attribute to the indicated value.</td>
</tr>
</tbody>
</table>

### Appendix D: TWRole class methods

This class represents the description of a group, along with its direct or indirect members (users or groups).

The following methods in Table 9 support retrieving and changing group information (skipped some methods).

### Table 9. TWRole class methods

<table>
<thead>
<tr>
<th>Return type</th>
<th>Property / method name</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>Name</td>
<td>The group name (has to be unique).</td>
</tr>
<tr>
<td>int</td>
<td>Id</td>
<td>The group ID in the BPM user repository.</td>
</tr>
<tr>
<td>String</td>
<td>getTypeName()</td>
<td>Returns the name of the type (&quot;TWRole&quot;)</td>
</tr>
<tr>
<td>TWUser[]</td>
<td>allUsers</td>
<td>All direct or indirect user members.</td>
</tr>
<tr>
<td>TWUser[]</td>
<td>Users</td>
<td>All direct user members.</td>
</tr>
<tr>
<td>void</td>
<td>addUser(String[] users)</td>
<td>Add the specified users as direct members to the given group.</td>
</tr>
<tr>
<td>void</td>
<td>addUser(TWRole[] users)</td>
<td>Add the specified users as direct members to the given group.</td>
</tr>
<tr>
<td>TWRole[]</td>
<td>containerRoles</td>
<td>All groups which contain the actual group as a direct member.</td>
</tr>
</tbody>
</table>
### Teams in Business Process Manager V8.5, Part 2: Developing services for dynamic team resolution

<table>
<thead>
<tr>
<th>TWRole[]</th>
<th>Roles</th>
<th>All direct group members.</th>
</tr>
</thead>
</table>
| **void** | addRoles(TWRole role)  
addRoles(TWRole[] roles)  
addRoles(String roleCsvList)  
addRoles(String[] roles) | Add the specified groups as direct members to the given group. |
| **void** | removeRoles(TWRole role)  
removeRoles(TWRole[] roles)  
removeRoles(String roleList)  
removeRoles(String[] roles) | Remove the specified groups from the direct members of the given group. |
Resources

- IBM Business Process Manager Information Center: Creating a team
- IBM Business Process Manager Information Center: The Team Performance dashboard
- IBM Business Process Manager Information Center: JavaScript API for IBM Process Designer
- developerWorks Business Process Management zone
About the authors

Gabriel Dermler

Gabriel Dermler is a Senior Software Engineer on the Business Process Manager team in Böblingen, Germany. He has been involved with the design and implementation of Process Management and Portal solutions for many years, with one focus area in user and security management. Prior to that, Gabriel spent three years with IBM Research working on Quality of Service issues for IP telephony and web offerings. He holds a PhD degree in Computer Science from the University of Stuttgart, Germany.

Markus Reichart

Markus Reichart is a member of the IBM WebSphere Business Process Management architecture team at the IBM Research and Development Lab in Böblingen, Germany. He has twelve years of experience as a developer and architect in the areas of Business Process Management for WebSphere MQ Workflow, WebSphere Business Integration Server Foundation, WebSphere Process Server, and IBM Business Process Manager. His architectural responsibilities include administration and operations and for IBM Business Process Manager 8.5, Work Management. Markus holds a Master’s degree in Computer Science from the University of Stuttgart and joined IBM Germany in 2001.

Thomas Bernhardt

Thomas Bernhardt is Team Lead of the development team for IBM Business Process Manager, focusing on the implementation of team concepts. He has worked for 17 years on IBM BPM and its predecessor products, both for human-centric and workflow-centric features.

Thomas Duerr

Thomas Duerr is a Software Engineer at the IBM Research and Development Lab in Böblingen, Germany. He has seven years of experience as a developer in the areas of WebSphere Process Server and IBM Business Process Manager. Currently, he is part of the development team for the IBM BPM offering and focuses on integration.
with Enterprise Content Management Systems. Thomas holds an Engineering degree in Information Technology and joined IBM in 2006.

© Copyright IBM Corporation 2014
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)