



# @task

## Chapter 1

### API Overview

The following chapter introduces the user to the SDK, and @task's functionality. It is preparatory to understanding the use cases and JavaDocs described in Chapter 2. The sections in this chapter are:

- [Introduction to the SDK](#)
- [Structure of the SDK](#)
- [Understanding the Basics of @task](#)

## INTRODUCTION TO THE SDK

The synthesis of @task's collaborative platform with a powerful tracking and reporting system make it a project management tool that is second to none in its class. Even with all of the built-in functionality of @task, we recognize that our clients use various types of hardware and programs that do not directly interface and communicate with @task. In the hope of meeting the needs of the more enterprising customer we have developed a Software Developer's Kit (SDK). Using the tools and examples contained in the kit, any customer will be able to customize their instance of @task and get the maximum value out of their purchase.

Our APIs are accessible to a native Java client through EJB, a Macromedia Flash client through Flash Data Remoting, or a Web-Services client through SOAP. The advantage of this design is that it gives 3rd party developers complete access to all the functionality provided by the @task Project Management Engine. The following document will instruct the reader on the layout of the SDK, some basic functionality of @task, and some use cases that a user should be aware of. Information is included in the related documentation concerning how to set up your environment to access the APIs through the native Java client, SOAP, and Flash Data Remoting. We also include some documentation on how to extend the Java server through external login and the module development kit (MDK).

### A NOTE TO POTENTIAL USERS OF THE SDK

The type of application and customization that is possible with the SDK depends on the license type and service type that your company is using. The different license types are Enterprise and Professional, and the service types include On-Demand, and Onsite Installation. For example a user with an On-Demand service and an Enterprise Edition license can access the SOAP APIs, as seen in the upper left corner of the matrix. To avoid any unpleasant surprises, please ensure that you know your company's agreement with @task before you begin development.

TABLE 1.1: CUSTOMIZATION OPTIONS MATRIX

LICENSE TYPE	ON-DEMAND OPTIONS	ONSITE INSTALL OPTIONS
Enterprise	SOAP	SOAP EJB Flash
Professional	N/A	EJB Flash

## STRUCTURE OF THE SDK

The SDK is composed of multiple features and areas of emphasis. When the SDK has been unzipped, you should have noticed an README.txt file. This gives a very general description of what the SDK contains. There are three main directories: client, docs, and server.

The client directory includes all of the examples designed to communicate remotely with the @task server. These include examples in Java, .Net, PHP, and Flash. The Java examples are further broken down into EJB and SOAP. Each directory contains at least one code example and an accompanying document that explains how to build and run the sample code on your computer. All libraries are included in the desired directory. The client\docs directory contains all of the JavaDocs generated from our API.

The docs directory contains this document, API Overview, and the User's Guide to Translation. The translator's guide is designed for those individuals who are looking to customize the language of @task, and is a valuable resource to international customers.

The server directory deals with code examples that extend the @task server to support external authentication methods, such as LDAP, and the MDK. A src directory contains the code and both the external login and MDK contain documentation relating to the example code.

Future editions of the SDK will include different code examples such as Perl, and Ruby. The desire is that independently developed @task plug-ins will be submitted to us in various languages. Once they have been reviewed we will make them available to future users of the SDK.

Other resources of information concerning the SDK can be found in the developer's web page, that is accessible at <http://attask.com/developer>. A developer's forum is also available to registered users of @task that is accessible from the previous web page.

## UNDERSTANDING THE BASICS OF @TASK

Any user who wishes to successfully code a functioning plug-in via the SDK should have a basic understanding of how @task works. This will document will be very brief in its treatment of the subject, for a more detailed understanding please refer to the User's Guide included with your version of @task; but it is also available on the customer support page by accessing your instance of @task help via the "?" icon in the top right of the screen. Many of the data types in @task, like users and tasks, contain multiple fields of information that can be accessed and updated. The following attempts to address those components of a data type which are most important. We will discuss the data types of user, projects, tasks, issues, and templates.

### USERS

User objects represent the human element in @task. Users are assigned tasks and issues, and effect changes within the system. Each user has various dependent objects associated with them. @task displays these dependent objects as a set of information categorized in tabs: contact and related info, custom data and organizational charts. As with all users, each will have a set of contact information including a first and last name, a username, a password, an email address, and a home group. The related information tab has more influence on the ability of the user to effect changes in @task.

The access level associated with the account is one of the related information fields. Access levels determine which data objects a user has the power to view, edit, add and delete. For some applications, you will have to be sensitive to the access level of the user, and thus, it is an important consideration for your application design. For example, if you have created an application that allows a user to view a list of incomplete tasks outside of @task using their userID, then the list that is returned will depend on the user's access level.

The same consideration must taken with fields such as job roles and group, as they may limit what kind of information can be accessed by the user. Job roles are defined within the company and help guide project managers in selecting their project team. Groups are generally made to distinguish between different departments and geographical offices, like New York office and Los Angeles office, or accounting and sales.

### PROJECTS

Once a user has been created with at least the required fields, they can be assigned to a project. A project is an individual or collaborative process by which a stated objective can be accomplished. In @task a project can be created with as little as a name, a planned start date, and a group, because all other fields are set to the defaults. A planned start date and scheduling mode allow the user to determine the beginning or end of the project's timetable. Each project has an assigned group that influences which users are eligible to be added to the project. Other fields of information include what is the budget for the project, the update mode, and any attached templates. Projects also have a team of users that are assigned to them. Once the project has been created, a task or an issue can be attached to it. Thus, a project provides a framework for the addition of tasks and issues within @task.

### TASKS

Tasks are the steps or activities that must take place in order to accomplish the stated objective of the project. Each task has a handful of very important features, among them are: completion status and percentage, duration, predecessors, task constraint, assignments and approvals.

The completion status is an enumerated value which indicates when the task is New, In Progress, or Complete. Completion percentage is also used to update the completion status. For example, if the status is

currently In Progress and the completion percentage is changed to 100%, then the task status will be set to Complete, and vice versa.

The duration of a task affects the duration of a project. If a project has a single task with a duration of 10 days, then the project will have a planned duration of 10 days. Another influencing factor is the concept of a predecessor. A task with a predecessor implies that it is in some way dependent on the completion of another task. Predecessor relationships can come in many forms, and a more in depth understanding can be obtained from the User's Guide.

Task constraints determine when and how the task will be started or completed. There are multiple enumerated values that can be selected, the default being As Soon As Possible. This field enables proper timeline calculations based on the selected constraint.

Assignments are made by selecting a member or job role that is found on the project team. The assignee is responsible for the successful execution of a task. A user that has been associated with a task will have their userID included in the taskBean, and is therefore, a searchable field.

An approval is a method of ensuring that a responsible party will review a completed task before submission. The process of approving a task is completed when the selected approver reviews the information contained in the task and they pass it off. An approver does not necessarily have to be on the project team. Thus, a task may actually have to go through two steps to be fully completed.

## ISSUES

During the execution of a task there are occasional situations or questions that crop up which stall or impede the successful completion of the task or project. In @task these situation are known as Issues. A bug report is an excellent example of an issue. Bug reports are a type of issue that points out a flaw in a program that needs to be fixed before releasing the product. Issues have a unique place in @task as they are flexible data types. An issue can be converted into a project or a task. Issues can be converted to different data types because they are used to note a problem and not necessarily to fix the problem. An issue that is sufficiently complex to rectify could require the time and effort of a project with multiple tasks. On the other hand it might only need a quick fix and not require creating a task or a project.

@task implements four types of Issues: Issues, Bug Reports, Change Order, and Requests. Each has a different set of enumerated Issue Status. Please see the User's Guide for a further explanation. An issue also stores information concerning who was the originator, who it is assigned to, when it should start and the duration.

Issues can create problems for the unwary programmer, because of their versatility and complex functionality. It is strongly recommended that you review the information concerning Issues in the User's Guide.

## TEMPLATES

After a user has used @task for a period of time, they may notice that many of their projects or tasks have the same components, timelines, and team. To overcome a tedious process of creating and recreating projects with the same format, there is a data type known as a template. Templates save the framework of a project or a task and can be attached to future projects at any time. For example, if your company uses the same three steps for all billing transactions (e.g., invoice, receive payment and approve contract), instead of creating a new project and individually adding each task, you can create one project with three set generic tasks and save it as a template. This new template can be used to recreate the form of the billing transaction with a simple attachment.



## Chapter 2

# Developer's Tips and Use Cases

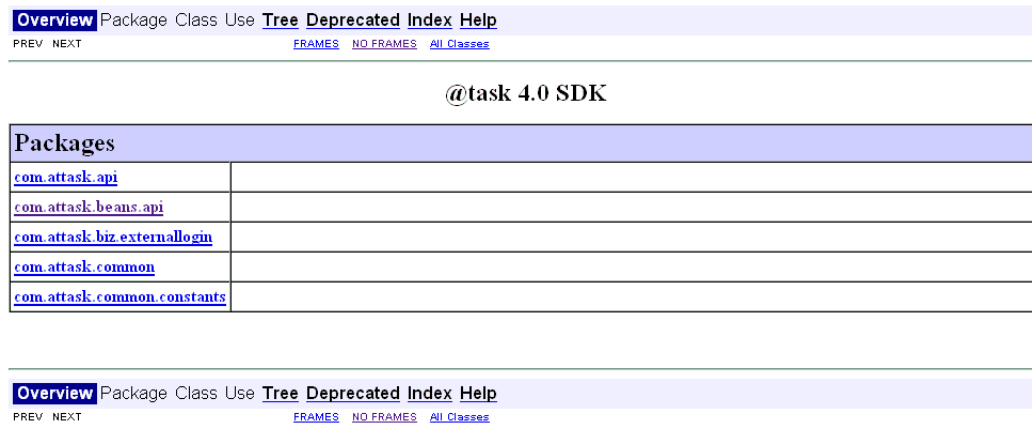
As most developers are new to the structure and design of our API, we offer the following helpful tips and use cases. This section should be used in conjunction with the @task JavaDocs that are bundled with the API.

- [How to Use the API JavaDocs](#)
- [API Tips](#)
- [Basic Operations](#)
- [Retrieving @task Objects](#)

## HOW TO USE THE API JAVADOCS

A complete description of the classes, interfaces, methods, and enumerated values in the API can be found in the following root path: ...sdk\_4\_0/client/docs/api/index.html. Opening this file will display the API JavaDocs in a web browser. It is similar in format to any generated Java documentation, as shown in Figure 2.1 there is a navigator bar displaying the following options: Overview, Package, Class, Use, Tree, Deprecated, Index, and Help.

FIGURE 2.1: OVERVIEW PAGE



From this page, any of the needed information concerning the API can be accessed. The packages contain different classes and interfaces. The names are indicative of what they contain, except for com.attask.common which includes all of the exceptions or fault that can occur in @task. Each package name is a link and opens up a view similar to that found in Figure 2.2.

FIGURE 2.2: INTERFACE SUMMARY

### Package com.attask.beans.api

Interface Summary	
<a href="#">AddAccessLevelMessage</a>	
<a href="#">AddAccessScopeMessage</a>	
<a href="#">AddAccountRepMessage</a>	
<a href="#">AddAppEventMessage</a>	
<a href="#">AddBillingRecordMessage</a>	
<a href="#">AddCategoryMessage</a>	
<a href="#">AddCompanyMessage</a>	
<a href="#">AddContactMessage</a>	

From here you can perform a search for the information that you desire. All of the class names are linked, and open to a class description that includes all of the methods, interfaces, searchable fields, etc., that are associate with the class.

## USING THE MESSAGES AND ENUM CLASSES

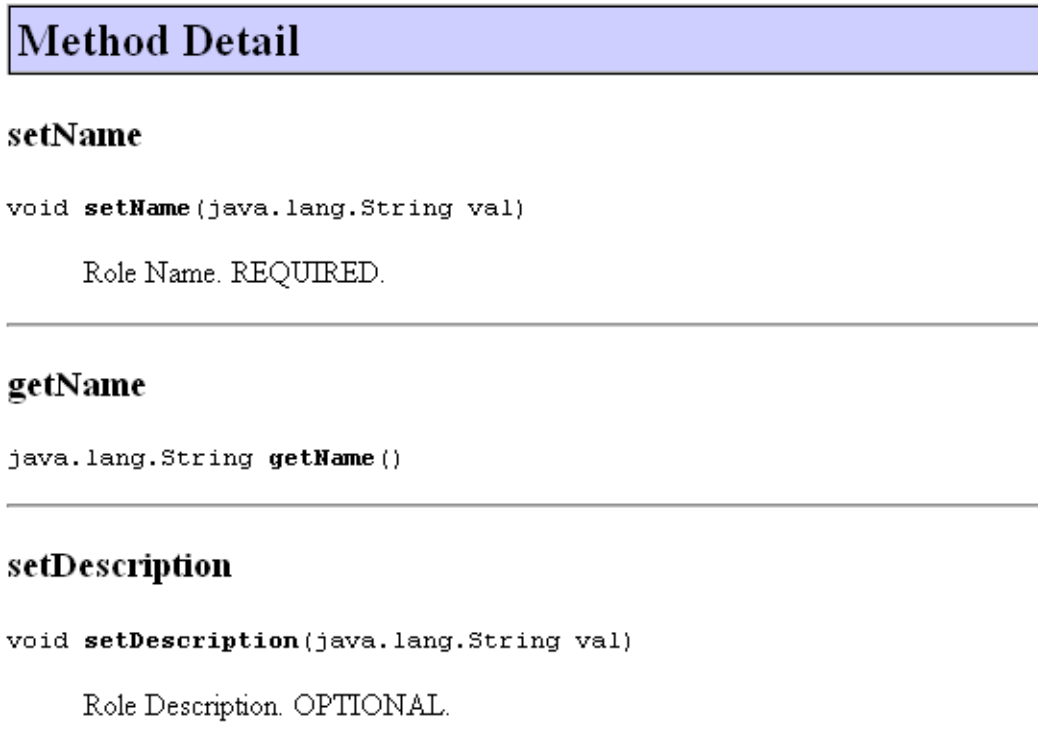
### MESSAGES

The AddXXXMessages or EditXXXMessages classes are only used internally, but are included for documentation purposes. For example, AddRoleMessage describes what fields are required for adding a Role, and EditRoleMessage describes what additional fields should be set when editing. The XXXMessages

interfaces **cannot** be used by EJB or SOAP users (XXXBean classes must be used instead), but are included to help with documentation.

Applying the same process as before to search for a class in the JavaDocs, we can locate the AddRoleMessage class. Once it has been found in the com.attask.beans.api package, the documentation indicates, under the Method Details section, that the following fields have to be set: setMaxUsers, and setName. A quick check is to look at the setXXX methods found under the Method Details as seen in Figure 2.3, and then read the requirements for those methods in the detail section. As you can see setName is a required field, while setDescription is not.

FIGURE 2.3: ADDROLEMESSAGES METHOD DETAILS



The screenshot shows a 'Method Detail' section with three methods: setName, getName, and setDescription. Each method is followed by its signature and a description of its parameters and requirements.

```
Method Detail
```

**setName**

```
void setName(java.lang.String val)
```

Role Name. REQUIRED.

---

**getName**

```
java.lang.String getName()
```

---

**setDescription**

```
void setDescription(java.lang.String val)
```

Role Description. OPTIONAL.

When accessing the EditRoleMessage interface there is only one required setXXX method, named setID, but this is in addition to the methods listed for AddRoleMessage.

## ENUMERATIONS

In some cases the fields that must be set are enumerated fields. These values are identified by the “Enum” suffix. For example, the TaskStatusEnum class includes the values New, In Progress, and Complete. To determine which values are enumerated, access the com.attask.common.constants package in the JavaDocs. To find all of the enumerated values found in @task please refer to the sdk\_4\_0/client/docs/api/com.attask.common.constants. Search for the enumerated class and search the Field Summary section for all of the static methods. An exemplary Field Details section is depicted in Figure 2.4.

FIGURE 2.4: FIELD DETAILS

### NEW

```
public static final OpTaskStatusEnum NEW
```

New. A newly entered Operational Task. This is the default status. Value is "NEW".

---

### IN\_PROGRESS

```
public static final OpTaskStatusEnum IN_PROGRESS
```

In Progress. A request is currently assigned and being worked on. Value is "INP".

---

### AWAITING\_FEEDBACK

```
public static final OpTaskStatusEnum AWAITING_FEEDBACK
```

Awaiting Feedback. Progress cannot continue until more feedback is gathered. Value is "AWF".

The Field Detail information can be accessed from the Field Summary section. Following the above example, the enumerated values for New, In Progress, and Complete, are respectively: "NEW", "INP", and "CPL". Whenever a TaskBean is retrieved, the Task Status will return one of these values. When editing the bean these values should be passed in.

Selecting one of the links such as New will display the associated description of that field and any constraints it has. There are enumerated values that have a raw value associated with them. For example, some issues have a field name Priority that ranges from Low to Urgent. The raw value is a range from 1-5, Low being 1 and Urgent being 5. There is no difference in choosing the enumerated value or the raw value.

**NOTE:** When Adding and Editing objects be careful to retrieve the object first, via a "get" method, and change the fields that need to be updated. If you do not retrieve the object first, all fields that are not populated will end up being set to null.

## API TIPS

### VOCABULARY DIFFERENCES BETWEEN THE API AND THE UI

During the development of @task a few data types have gradually changed from first conception. This includes the way they were named in the API and how they appear in the UI. To alleviate any future confusion while working with the API the following data types are known as one thing in the UI and as a another in the APIs, respectively:

1. A Library Task in the UI is a MasterTask in the API.
2. An Issue in the UI is an OpTask in the API.

### LOGIN, ACCESS LEVELS AND THE JNDI

The first call made in an application should be the login call. The reason is that a login will return the sessionID which will be crucial to making other calls in the application.

Be sensitive to the access level of the logged in user because it will be enforced through out the session. As described before, the access level of the user determines what data objects they can view, edit, add and delete. Thus, when designing the application consider how the access level could affect functionality.

Lastly, when you connect to the server using SOAP or EJB, there is no need to reconnect before each call. Instead, cache the API handle and use it until logout. For EJB, this is handled inside of APISupport (lines 48 through 54). For SOAP, an example of how to do this is done in the SDK SOAPExample (lines 55 and 56).

## BASIC OPERATIONS

### ADDING OBJECTS

As mentioned previously in this document, there are multiple fields that need to be set with a value in order to Add an object. These include all of the required fields listed in the AddXXXMessages.java file. You need to view the Message file to make sure that you are setting all the correct fields to the required value. The following is an example of how to add a Project:

```
// Create a project.
ProjectBean project = new ProjectBean();
project.setName("Project Name");
project.setGroupID(groupID);

// All of these fields must be set to some value, but NULL or empty
// values are OK.
project.setCategoryID(_intNull);
project.setScheduleID(_intNull);
project.setTemplateID(_intNull);
// The planned start date is required.
project.setPlannedStartDate(new GregorianCalendar());
```

Additionally, there are many fields in @task that require an Enumerated String value. These include dropdown fields. In this example, "CUR" is passed in as the value Status.

```
// This field is also required and must use one of the values listed
// in the provided Javadoc documentation.
project.setStatus("CUR");
project.setCompletionType("MAN");
project.setUpdateType("AUTO");

// Create the project and store its ID.
LOG.info("Creating a project");
projectID = _apiSEI.addProject(_sessionID, project);
```

### EDITING OBJECTS

Editing objects is a fairly straightforward process. As you can see from the example code below, it requires retrieving the desired task via the taskID and modifying the value(s) as desired. Please note that @task does not support single field editing. You must first retrieve the bean by calling a "get" method, which preserves all of the saved and unedited fields, editing the desired fields and then calling "edit". If you do not do this you run the risk of overwriting information with NULL or empty values. After the task has been edited we confirmed that the change had been modified, but you do not need to replicate this practice. An example of this process is found below.

```
// Edit the task.
TaskBean task2 = _apiSEI.getTaskByTaskID(_sessionID, taskIDs);
task2.setName("New Task Name");
_apiSEI.editTask(_sessionID, task2);

TaskBean task3 = _apiSEI.getTaskByTaskID(_sessionID, taskIDs[0]);
if (!"New Task Name".equals(task3.getName())) {
    LOG.error("Task not modified");
};
```

### DELETING OBJECTS

Due the final nature of a delete operation, @task takes care to protect the database from any errors that can be introduced by lost links and associations. The @task API supports three options for deleting an object. The first is Delete. This is always a safe option because it will only remove that object and nothing else. The following is an example of deleting a project from the database:

```
int deletedProject = _apiSEI.deleteProject(_sessionID, projectID);
```

The second option is to Replace the object. This will reset the values of the existing objects so it is an in-place swap of the current object. New values will be inserted, but all of the current associations and dependents will remain intact. This is an example of a replaceRole call:

```
int replaceRoleID = _apiSEI.replaceRole(_sessionID, roleID, replacementRoleID);
```

Force Delete is the most aggressive action you can take on an object, and should be reserved for objects that you have no use for. When an object is Force Deleted you remove all record of it and its dependents. This has the propensity to leave orphaned objects out there. For example, if you have a Task that has hours applied to it and you Force Delete the task, those hours have to be reapplied to the project. The rule of thumb would be to not use Force Delete unless you have to or the object is truly expendable. The following is example of how to use a Force Delete:

```
int deletedProjectID = _apiSEI.forceDeleteProject(_sessionID, projectID);
```

Please note that objects that do not have any dependents do not have a Force Delete method. An example of this would be the RoleBean.

## DEALING WITH DOCUMENTS

The only objects that do not follow the above conventions are documents. Documents are uploaded and downloaded. To upload a new document set the name, CategoryID and the version. The version label is stored as a String value. Once this is done, you read the document into a byte array. The following code example demonstrates how to accomplish the above steps:

```
// Add a document to the project.
DocumentBean document = new DocumentBean();
document.setName("hello.txt");

// All of these fields must be set to some value, but NULL is OK.
document.setCategoryID(_intNull);

// Set the document version.
DocumentVersionBean documentVersion = new DocumentVersionBean();
documentVersion.setVersion("Version 1");
document.setCurrentVersion(documentVersion);

// Create the document and store its ID.
LOG.info("Creating a document");
byte[] contents = new byte[] { 'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l',
'd' };
documentID = _api.uploadProjectDocument(_sessionID, projectID, document, con-
tents);
```

Documents can be updated with a new version by incrementing the version, and uploading the new document. Continuing the previous example, we upload a second version of the document:

```
// Update the contents of the document.
document.setID(documentID);
documentVersion.setVersion("Version 2");
contents[6] = 'H';
LOG.info("Uploading a new version of the document");
int documentID1 = _api.uploadProjectDocument(_sessionID, projectID, document,
contents);
```

Documents can be individually downloaded, and multiple document downloads will be supported in the near future (Version 4.0 R6). A multiple document download will output a .zip file. Additionally, it is possible for users to download if you know exactly what version you want to download. Downloading the contents of a document is accomplished like so:

```
// Download document contents.  
LOG.info("Downloading the document");  
byte[] contentsCheck = _api.downloadDocument(_sessionID, documentID);
```

## RETRIEVING @TASK OBJECTS

The @task search APIs use a name/value pair mechanism that allows clients to filter and “batch read” @task data similar to how SQL would be used to filter and load database results. This section explains the @task Query Framework and provides examples for performing various search operations using the @task APIs. Understanding how to retrieve objects in @task depends heavily on using the JavaDocs described above. Fields that can be queried are listed in the JavaDocs under the appropriate bean object. These fields can be filtered as described below, to take advantage of the robust search capability of @task.

### CREATING A FILTER

The name/value pair described above is called a NameValueBean. NameValueBeans are primarily used to represent search criteria for various "getXXXXFromSearch()" methods throughout the @task API. All of the @task data objects, called XXXBean, contain a list of "Searchable Fields" which can be used in search expressions. There are 2 basic kinds of Fields: Primitive Types (int, Calendar, String, etc.), and Bean Types (using other data objects). Each field has a name that is used in all expressions. Using these field names along with a standardized set of rules offers users an extremely powerful search capability. Creating a filter is done by setting up an array of NameValueBeans, and setting the name/value pair like so:

```
NameValueBean[] nv = new NameValueBean[1];
nv[0] = new NameValueBean();
nv[0].setName("name");
nv[0].setValue("value");
```

Alternatively this could be written as:

```
NameValueBean[] nv = new NameValueBean[] {
    new NameValueBean("name", "value")};
```

### TYPES OF SEARCHES

Now that we have a structure to create a filter, we can discuss the different types of searches you can perform.

#### EQUALS SEARCH

An Equals search is done with one filter as described above. Essentially the filter will find a field of name “x” with a value “y”. The following is an example of how to create a filter for an Equals search for a “username” of “admin”:

```
NameValueBean[] nv = new NameValueBean[] {
    new NameValueBean("username", "admin")};
```

#### SEARCHING WITH IN

The In search uses more than one NameValueBean, i.e., an array, on the same attribute. An example of an In search would be to search for all users in @task with a username of “admin”, “bob” or “mary”. This can be seen below:

```
NameValueBean[] nv = new NameValueBean[] {
    new NameValueBean("username", "admin"),
    new NameValueBean("username", "bob"),
    new NameValueBean("username", "mary")};
```

#### SEARCHING WITH AND

The And search is used to place multiple filters NameValueBeans on different attributes. For example, you could search for a customerID of “1” and a Role with the value of “Manager”. You can do this by setting up a

filter that searches for both of these attributes, and then calling the `getRolesFromSearch()` method with a `sessionID` and the filter you just created. This would look like the following code:

```
NameValueBean[] filter1 = new NameValueBean[] {
    new NameValueBean("customerID", "1"),
    new NameValueBean("name", "Manager")
};
RoleBean [] roles = APISupport.getRolesFromSearch(_sessionID, filter1);
```

## SEARCHING WITH OR

API Interfaces that accept a 2-dimensional array of `NameValueBean` (`NameValueBean[][]`) allow for multiple queries to be combined in a single search. In this way it is possible to perform an OR search. For example, if you had the following 2 separate Role queries defined, i.e.,

```
Query 1
role="Manager"

Query 2
role="Engineer"
```

and you wanted to combine them into a query that would return all `RoleBeans` where the role equals "Manager" OR the role equals "Engineer", you could send both of these `NameValueBean[]` queries as separate rows in the `NameValueBean[][]`, and their results would be combined to give you a single result set. Here is a code snippet:

```
NameValueBean[] filter1 = new NameValueBean[] {
    new NameValueBean("role", "Manager")};

NameValueBean[] filter2 = new NameValueBean[] {
    new NameValueBean("role", "Engineer")};
```

Now take both filters and get a result set of Documents that match filter1 OR filter2

```
DocumentBean documents = APISupport.getRolesFromSearchWithOr(ctxt, new NameValueBean[][] { filter1, filter2});
```

Another way to accomplish this is using the Query Framework without a 2-dimensional array. It is important to preface any search terms with "OR:". For example, "OR:a:plannedStartDate=2004-05-06T12:00:00:000" would put that `plannedStartDate` value in a separate OR branch.

## SINGLE VARIABLE MODIFIER EXPRESSIONS

You can also add the suffix "\_Mod" to a field to define an expression. Available expressions can be found at the end of this section. A few examples of these modifying expressions are used for examples below.

For example, to find `UserBeans` that have "matt" as part of their first name, you could use the "contains" modifier. This is similar to the SQL expression field of `LIKE"%matt%"`, and this is case-sensitive. It would match all Users with First Name of "matthew", "matt", "mattias", etc:

```
NameValueBean[] filter1 = new NameValueBean[] {
    new NameValueBean("firstName", "matt"),
    new NameValueBean("firstName_Mod", "contains")};
```

To do a case-insensitive search, you would do the following, which produces the SQL expression `UPPER(field) LIKE UPPER('%value%')`:

```
NameValueBean[] filter1 = new NameValueBean[] {
    new NameValueBean("firstName", "matt"),
    new NameValueBean("firstName_Mod", "cicontains")};
```

To find `UserBeans` where the ID is greater than 1000, and produce a SQL expression of `ID > 1000`:

```
NameValueBean[] filter1 = new NameValueBean[] {
```

```
new NameValueBean("ID", "1000"),
new NameValueBean("ID_Mod", "gt");
```

The key to using single variable modifier expressions is knowing what you want to accomplish and with what given query constant. Table 2.1 contains all of the query constants available in @task.

**TABLE 2.1: SINGLE VARIABLE MODIFIER EXPRESSIONS**

DESCRIPTION	STRING VALUE	EQUIVALENT SQL EXPRESSION
Case-Insensitive Contains	icontains	UPPER (field) LIKE UPPER('%value%')
Case-Insensitive Equal	cieq	UPPER(field) = UPPER(value)
Case-Insensitive Like	cilike	UPPER(field) LIKE UPPER('value')
Case-Insensitive Does Not Contain	cinotcontains	UPPER(field) NOT LIKE UPPER('%value%')
Case-Insensitive Does Not Equal	cine	UPPER(field) <> UPPER(value)
Contains	contains	field LIKE '%value%'
Equal	eq	field = value
False	false	false
Greater Than	gt	field > value
Greater Than or Equal	gte	field >= value
Is Blank	isblank	field IS NULL OR field = ''
Is Null	isnull	field IS NULL
Less Than	lt	field < value
Less Than Equal	lte	field <= value
Like	like	field LIKE 'value'
Not Blank	notblank	field IS NOT NULL AND field <> ''
Not Contains	notcontains	field NOT LIKE '%value%'
Not Equal	ne	field <> value
Not Null	notnull	field IS NOT NULL
SoundEX	soundex	SOUNDEX(field) = SOUNDEX(value)

### DUAL VARIABLE MODIFIER EXPRESSIONS

You can also add the suffix "\_Range" to a field to define a 2nd value to consider in comparisons. This is useful for BETWEEN and NOTBETWEEN-type searches. Available expressions can be found in QueryConstants.

An example of how to find UserBeans where the first name is between BETWEEN "a" AND "c":

```
NameValueBean[] filter1 = new NameValueBean[]{
    new NameValueBean("firstName", "a"),
    new NameValueBean("firstName_Range", "c"),
```

```
new NameValueBean("firstName_Mod", "between");
```

TABLE 2.2: DUAL VARIABLE MODIFIER EXPRESSIONS

DESCRIPTION	STRING VALUE	EQUIVALENT SQL EXPRESSION
Between	between	field BETWEEN value AND value_Range
Case-Insensitive Between	cibetween	UPPER(field) BETWEEN UPPER(value) AND UPPER(value_Range)
Not Between	notbetween	fieldNOT BETWEEN value AND value_Range
Case-Insensitive Not Between	cinobetween	UPPER(field)NOT BETWEEN UPPER(value) AND UPPER(value_Range)

### RELATED OBJECT MODIFIER EXPRESSIONS

It is also possible to search for an object based on its Related Object fields. The syntax for this is "Related Object Field:Primitive Type". Please note that it is only possible to span 1 Related Object field in a search. It is also required that the 2nd field after the 1st Related Object field is of Primitive Type. In cases where a Related Object field relates to more than one object, the search framework will return a match if any of the related objects match.

An exemplary Related Object modifier is written like this: `accessLevel:name=admin`. The following code implements that modifier, which finds UserBeans that have an Access Level with name = "admin":

```
NameValueBean[] filter1 = new NameValueBean[]{
    new NameValueBean("accessLevel:name", "admin"),
    new NameValueBean("accessLevel:name_Mod", "eq")};
```

In keeping with the restriction that a modifier can only span one Related Object field, we can search for UserBeans that have a Role with a name of either "Engineer" or "Salesperson". The syntax of the modifiers would be as such:

```
NameValueBean[] filter1 = new NameValueBean[]{
    new NameValueBean("roles:name", "Engineer"),
    new NameValueBean("roles:name", "Salesperson")};
```

### CUSTOM DATA MODIFIER EXPRESSIONS

Custom Data fields can be searched in the same way as Primitive Type fields. The same rules apply. If a UserBean is assigned to a Category that contains a Parameter named "Tenure", the following searches are valid. Additionally, for Custom Data parameters that allow for multiple values, the special filter key "allof" in the Query Constants will only match objects that contain all values listed.

The following finds UserBeans with a Custom Data field named Tenure having a value of "5".

```
NameValueBean[] filter1 = new NameValueBean[]{
    new NameValueBean("Tenure", "5")};
```

Like the example above this example finds UserBeans with a Custom Data field named Tenure and with the a modifier to find all those having a value  $\geq 5$ :

```
NameValueBean[] filter1 = new NameValueBean[]{
    new NameValueBean("Tenure", "5"),
    new NameValueBean("Tenure_Mod", "gte")};
```

## COMPARING FIELDS AGAINST OTHER FIELD VALUES

It is possible to run queries that filter data based on comparing fields against values contained in other fields. For example, it may be interesting to find all Projects that have a Projected Completion Date that is after its Planned Completion Date. Or, you may want to query for Tasks that have a total number of hours greater than the value in a numeric custom data field. This is noted by prefixing the value portion of the search term with "FIELD:" Query Constant followed by the field to compare with.

The following demonstrates how to find Tasks that with a Planned Start Date scheduled before its Projected Start Date (Tasks scheduling late):

```
NameValueBean[] filter1 = new NameValueBean[]{
    new NameValueBean("plannedStartDate", "FIELD:projectedStartDate"),
    new NameValueBean("plannedStartDate_Mod", "lt")};
```

To use Custom Data fields as comparisons, simply use the following syntax "FIELD:DE:value". This example applies this principle by finding Issues that were completed after the value in the "Ship Date" custom field attached to the Issue:

```
NameValueBean[] filter1 = new NameValueBean[]{
    new NameValueBean("actualCompletionDate", "FIELD:DE:Ship Date"),
    new NameValueBean("actualCompletionDate_Mod", "gt")};
```

## BATCH READING OBJECTS

In general, only Primitive Type fields are loaded when an object is retrieved. For example, Custom Data information is not populated for objects by default. This is for performance reasons. However, it is possible in a getXXXXFromSearch() method call to "Batch Read" these related fields when loading the object. This is done by appending the suffix "\_BatchRead" to the needed Related Object field. All Related Object fields listed in a Bean's available search fields are available for Batch Read.

A handful of examples would be instructive here. In this case we will retrieve a Project and all of the associated Tasks.

```
NameValueBean[] filter1 = new NameValueBean[]{
    new NameValueBean("projectID", "18"),
    new NameValueBean("tasks_BatchRead", "true")};

ProjectBean[] = APISupport.getProjectFromSearch(_sessionID, filter1);
```

Additionally, we can Batch Read the company name that is associated with a specific userID like so:

```
NameValueBean[] filter1 = new NameValueBean[]{
    new NameValueBean("userID", "281")
    new NameValueBean("userID_BatchRead", "true")};

ProjectBean[] = APISupport.getUserFromSearch(_sessionID, filter1);
```

The following code example tells the search engine to pre-populate Custom Data in the objects that are retrieved:

```
NameValueBean[] filter1 = new NameValueBean[]{
    new NameValueBean("projectID", "18"),
    new NameValueBean("parameterValues_BatchRead", "true")};

ProjectBean[] = APISupport.getProjectFromSearch(_sessionID, filter1);
```

If instead you would like to retrieve only a specific field you can do so. The syntax for this call is "<name>:<value>\_BatchRead". Following the example above we will retrieve a specific Custom Data field called "ParameterName".

```
NameValueBean[] filter1 = new NameValueBean[]{
```

```

    new NameValueBean("parameterValues_BatchRead:parameterName", "true");
ProjectBean[] = APISupport.getProjectFromSearch(_sessionID, filter1);

```

## DATE VALUES

In several cases it may be necessary to use a Date value in a NameValueBean. Since NameValueBean only accepts String values, a common datetime encoding is necessary for recognizing correct values. The date format recognized by the @task APIs is yyyy-MM-DD'T'HH:mm:ss:SSS. For example, 2005-01-01T20:30:00:000 would be interpreted as January 1, 2005, 8:30:00 PM 0 milliseconds. The TimeZone used is the default TimeZone defined for the server @task is running on.

For example, suppose you wanted to find all TaskBeans which have a plannedStartDate on March 18th, 2005. By default, @task interprets Calendar searches by matching on any objects that fall on the same day. That is, NameValueBean fields of name=plannedStartDate, value=2005-03-18T13:27:29:999 would return all TaskBean objects with a plannedStartDate "BETWEEN <2005-03-18T00:00:00:000> AND <2005-03-18T23:59:59:999>". This is because in the general case, searches are looking for items that occur at any time during the day being searched.

To disable this functionality to allow searching on an exact match, use QueryConstants.EXACT\_TIME as a field Modifier. The following code would generate a value of "<2004-05-24T09:00:00:000>".

```

NameValueBean[] filter1 = new NameValueBean[]{
    new NameValueBean("plannedStartDate", "2004-05-24T09:00:00:000")
    new NameValueBean("plannedStartDate_Mod", "exacttime")};

```

## USING OPERATORS TO RETRIEVE DATE VALUES

If you use the \_Range operator, then the earliest time of the day is used for the plannedStartDate and the latest time of the day is used for the \_Range operator. So, for plannedStartDate=2004-05-24T00:00:00:000, plannedStartDate\_Range=2004-05-25T00:00:00:000 the expression generated would be "BETWEEN <2004-05-24T00:00:00:000> AND <2004-05-25T23:59:59:999>". That is, it would include ALL tasks that land anywhere on 5/24 and 5/25

If you combine both \_Range and \_Mod=exacttime, then a BETWEEN clause is generated using EXACTLY the values provided. So, for plannedStartDate=2004-05-24T09:00:00:000, plannedStartDate\_Range=2004-05-24T12:00:00:000 the expression generated would be "BETWEEN <2004-05-24T09:00:00:000> AND <2004-05-24T12:00:00:000>"

For "IN" clause generation (when multiple values are provided), the search clause is ALWAYS exact. This is because, by nature, IN clause lists specific values.

## SORTING (ORDER BY)

Query-level sorting is done through the use of the Sort modifier, a sequence number and a directional modifier (asc, desc, ciasc or cidesc). For example, to sort by ID in ascending order, then by name in descending order, ignoring case-sensitivity:

```

NameValueBean[] filter1 = new NameValueBean[]{
    new NameValueBean("ID_1_Sort", "asc"),
    new NameValueBean("name_2_Sort", "cidesc")};

```

Sorting can also be done on a Custom Data field using the same convention as for normal fields. Simply use the Parameter Name as the field identifier. For example:

```
NameValueBean[] filter1 = new NameValueBean[] {
    new NameValueBean("My_Parameter_1_Sort", "ciasc")};
```

Please note that Custom Data fields can be defined to store more than 1 value. Attempts to sort on these fields that allow multiple values will produce an error.

**TABLE 2.3: SORTING MODIFYING EXPRESSIONS**

DESCRIPTION	STRING VALUE
Sorts Ascending	asc
Sorts Descending	desc
Case-insensitive Ascending Sort	ciasc
Case-Insensitive Descending Sort	cidesc

## AGGREGATE QUERIES

Aggregate Queries are defined using a combination of the AggFunc and GroupBy modifiers. Special rules that apply to Aggregate Queries include:

- Sort attributes are ignored. Sorting in Aggregate Queries is enforced using the GroupBy modifier.
- Batch Read attributes are ignored
- Results are returned as ResultRowBean[]. Field names used for column headers follow special rules. See ResultFieldBean for naming rules.

Aggregate Functions are defined using the "\_AggFunc" suffix QueryConstants.AGGFUNC. The following table lists of possible Aggregate Functions is:

**TABLE 2.4: AGGREGATE FUNCTIONS**

DESCRIPTION	STRING VALUE	EQUIVALENT SQL EXPRESSION
Summation	sum	SUM()
Count	count	COUNT()
Average	avg	AVG()
Minimum	min	MIN()
Maximum	max	MAX()
Standard Deviation	std	STD()
Variance	var	VAR()
Distinct Summation	dsum	SUM(DISTINCT)
Distinct Count	dcount	COUNT(DISTINCT)

**TABLE 2.4: AGGREGATE FUNCTIONS**

DESCRIPTION	STRING VALUE	EQUIVALENT SQL EXPRESSION
Distinct Average	davg	AVG(DISTINCT)
Distinct Minimum	dmin	MIN(DISTINCT)
Distinct Maximum	dmax	MAX(DISTINCT)
Distinct Standard Deviation	dstd	STD(DISTINCT)
Distinct Variance	dvar	VAR(DISTINCT)

For example, the following would create an Aggregate Function:

```
select COUNT(ID), AVG(PERCENTCOMPLETE), MAX(PLANNEDSTARTDATE) from...;

NameValueBean [] filter1 = new NameValueBean[] {(
    new NameValueBean("ID_AggFunc", "count"),
    new NameValueBean("percentComplete_AggFunc", "avg"),
    new NameValueBean("plannedStartDate_AggFunc", "max")};
```

Please note that data type of fields used in aggregate functions can effect whether a query will execute successfully. For example, it is possible that AVG(String Field) will cause an exception in some databases. Please refer the object documentation to verify field types.

Group By settings are defined in a similar manner as Sorting.

```
NameValueBean[] filter1 = new NameValueBean[]{
    new NameValueBean("groupID_1_GroupBy", "true"),
    new NameValueBean("categoryID_2_GroupBy", "true")};
```

This set of Group By attributes would generate a SQL clause as follows: "SELECT GROUPID, CATEGORYID.....GROUP BY GROUPID, CATEGORYID ORDER BY GROUPID, CATEGORYID". Please note that in addition to the "GROUP BY clause, this attribute will also generate an ORDER BY clause and will add these fields to the SELECT portion of the query.

**GROUP BY USING DATE RANGES**

It is common to group objects that contain dates, such as Project, Tasks, Hours, and Issues by ranges of dates. However, because dates are accurate to the millisecond, it is most useful to group by ranges of dates for anything useful. Because special functions are required to create SQL that will group objects by date ranges, special syntax is required.

```
NameValueBean[] filter1 = new NameValueBean[]{
    new NameValueBean("plannedStartDate_ByMonth_1_GroupBy", "true")};
```

This would generate a SQL clause on Oracle, for example:

```
"SELECT TO_NUMBER(TO_CHAR(TO_DATE('19700101','yyyymmdd')+NUMTODSINTER-
VAL(PLANNEDSTARTDATE/1000,'SECOND'),'WW')).....GROUP BY
TO_NUMBER(TO_CHAR(TO_DATE('19700101','yyyymmdd')+NUMTODSINTERVAL(PLANNEDSTART-
DATE/1000,'SECOND'),'WW')) ORDER BY TO_NUMBER(TO_CHAR(TO_DATE('19700101','yyyym-
mdd')+NUMTODSINTERVAL(PLANNEDSTARTDATE/1000,'SECOND'),'WW')).
```

Please note that in addition to the "GROUP BY clause, this attribute will also generate an ORDER BY clause and will add these fields to the SELECT portion of the query.

**VALID DATE RANGE GROUP BY ATTRIBUTES**

The following can be used by any date field: "\_ByWeekFromDate, \_ByWeek, \_ByMonth, \_ByQuarter, \_ByDay, \_ByYear".

## YEAR BOUNDARY AUTOMATICALLY ADDED

"\_ByYear" will be added automatically to "\_ByWeek, \_ByMonth, \_ByQuarter, and \_ByDay" group by queries so that queries that span multiple year boundaries can be identified as unique. If this is not desired behavior, add the "Only" suffix to these queries "\_ByWeekOnly, \_ByMonthOnly, \_ByQuarterOnly, \_ByDayOnly" Grouping By Week The "\_ByWeekFromDate" Group By is most useful for grouping dates by week. Because database vendors have not standardized on a common definition for date boundaries, and because different locales use different days to begin their weeks, the "\_ByWeekFromDate"

Group By attribute allows a developer to set the boundary date from which all weeks are calculated. Usually, this date is midnight of the Sunday, Monday, or Saturday that each week will be calculated from. For example:

```
plannedStartDate_ByWeekFromDate_1_GroupBy=2006-04-09T00:00:00:000
```

This would begin weeks from Sunday, April 9th, 2006. Dates falling from April 9th - April 16th would have a 1 and so on. Dates April 2nd, 2006-April 9th, 2006 would be week -1.

## WITH ROLLUP

The SQL ROLLUP clause can be added to an aggregate query by adding the \$\$ROLLUP parameter to any search. This will generate SQL with the following form:

```
"SELECT , . . . . GROUP BY WITH ROLLUP"
```

Please note that the ROLLUP function is only available if the database supports it.

TABLE 2.5: GROUPBY MODIFYING EXPRESSIONS

DESCRIPTION	STRING VALUE
Suffix for grouping date fields within the same day.	_ByDay
Suffix for grouping date fields within the same month.	_ByMonth
Suffix for grouping date fields within the same quarter.	_ByQuarter
Suffix for grouping date fields within the same week.	_ByWeek
Suffix for grouping date fields within the same week starting from an anchor date.	_ByWeekFromDate
Suffix for grouping date fields within the same year.	_ByYear
Key used to specify that a GROUP BY query should be done WITH ROLLUP.	\$\$ROLLUP

## CONCLUSION

Now that you have read through this entire document you should be well prepared to create your own applications and integrate almost any application you can use into @task. Remember you have resources such as the developer's home page and forum which are accessible at [www.attask.com/developer](http://www.attask.com/developer).