



Journeyx jxAPI

Status: This document applies to Timesheet version 5.6 or higher.
Comments and suggestions are welcomed at: jxAPI@journyx.com

Please always check <http://www.journyx.com/jxapi.html> for the latest information about the jxAPI, including updates and source code examples.

Version: for Journeyx Timesheet 5.6 (jxAPI Version 5.6.0.0)
Specification Document version 1.6 (May 2004)

Authors: Journeyx Development Team

Contributors: Journeyx Professional Services Team

<u>LEGAL DISCLAIMER</u>	7
<u>COPYRIGHT</u>	7
<u>TRADEMARKS</u>	7
<u>1. JOURNYX TIMESHEET™</u>	8
<u>TABLES AND RECORDS</u>	8
<u>DOMAINS AND GROUPS</u>	9
<u>AUTHENTICATION</u>	9
<u>TIMESHEETS</u>	9
<u>2. CHANGES IN THE JXAPI FROM PREVIOUS VERSIONS</u>	10
<u>CHANGES FROM TIMESHEET 4.5 TO TIMESHEET 4.6</u>	10
<u>CHANGES FROM 4.6.0 TO 4.6M1P1 (AND 46M2)</u>	10
<u>CHANGES FROM 4.6M1P1 TO 5.0.0</u>	10
<u>CHANGES FROM 5.0.0 TO 5.0M2</u>	11
<u>CHANGES FROM 5.0M2 TO 5.0M3</u>	11
<u>CHANGES FROM 5.0M3 TO 5.5</u>	11
<u>CHANGES FROM 5.5 TO 5.5M1</u>	11
<u>CHANGES FROM 5.5M1 TO 5.5.2</u>	11
<u>CHANGES FROM 5.5.2 TO 5.5.2M1</u>	12
<u>CHANGES FROM 5.5.2 TO 5.6</u>	12
<u>3. THE INTERFACE</u>	12
<u>ACCESS</u>	12
<u>SOAP HEADERS</u>	12
<u>DATA TYPES</u>	12
<u>SOAP OBJECTS</u>	13
<u>EXCEPTION (ERROR) HANDLING</u>	14
<u>METHODS, PARAMETERS AND TYPES</u>	14
<u>return type methodName (parameter 1 name: data type, parameter 2 name: data type,...)</u>	14
<u>4. REFERENCE</u>	15
<u>SESSION MANAGEMENT</u>	15
<u>BASE OBJECT METHODS</u>	16
<u>list apiVersion ()</u>	16
<u>boolean versionCheck (major: integer, minor: integer, revision: integer, patchlevel: integer [optional])</u>	16
<u>string login (user: string, password: string)</u>	16
<u>void changeUserPassword (user: string, currentpassword: string, newpassword: string)</u>	16
<u>string version ()</u>	16
<u>string installDate ()</u>	16
<u>string expirationDate ()</u>	17
<u>string licensedUsers ()</u>	17
<u>string hostname ()</u>	17
<u>string licensedHost ()</u>	17
<u>string uname ()</u>	17
<u>string companyName()</u>	17
<u>string customerNumber()</u>	17
<u>string adminName()</u>	17
<u>string adminTelephone()</u>	17
<u>string adminEmail()</u>	17
<u>STRUCTS AND SESSION OBJECT METHODS</u>	17
<u>Privileges</u>	17
<u>Domains</u>	18
<u>Database Manipulation: General Considerations</u>	18

string addFullType (new record: struct)	18
string addFullRecord(table name: string, new record: struct)	18
string addType (name: string)	19
string modifyType (id: string, existing record: struct)	19
struct getType (a pattern: struct)	19
void removeType (id: string)	19
void removeType (pattern: struct)	19
void addTypeToGroup (id: string, group: string)	19
void removeTypeFromGroup (id: string, group: string)	19
struct getDefaultType ()	19
General Use Methods	19
list getSearchableTables ()	19
list getRecordStructure (table name: string)	19
list getRecordsList (table name: string, pattern: struct)	20
struct getDefaultRecord (table name: string, display name: string [optional])	20
Preference Records	20
list getPreferenceValue (string)	21
Users	21
User-Related Methods	21
string addFullUser (record: UserRecord)	21
string addUser (name: string)	21
string modifyUser (name: string, record: UserRecord)	21
UserRecord getUser (pattern: UserRecord)	21
void removeUser (name: string)	21
void removeUser (pattern: UserRecord)	21
void addUserToGroup (name: string, group: string)	21
void removeUserFromGroup (name: string, group: string)	21
UserRecord getDefaultUser ()	21
void assignRoleToUser (name: string, role: string)	21
void removeRoleFromUser (name: string, role: string)	22
list getUserRoles (name: string)	22
string getDefaultRole ()	22
void changePassword (newpassword: string)	22
void changePasswordForUser (name: string, newpassword: string)	22
list getUserPermissions (name: string)	22
Groups	22
Group-Related Methods	22
string addFullGroup (record: GroupRecord)	22
string addGroup (name: string)	22
string modifyGroup (id: string, record: GroupRecord)	22
GroupRecord getGroup (pattern: GroupRecord)	22
void removeGroup (id: string)	22
void removeGroup (pattern: GroupRecord)	22
GroupRecord getDefaultGroup ()	22
void addObjectToGroup (group id or name: string, object class: string, object id name: string)	22
void removeObjectFromGroup (group id or name: string, object class: string, object id name: string)	23
list listGroupObjects (group id or name: string, object class: string)	23
list getGroupObjectClasses ()	24
Projects	24
Project-Related Methods	25
string addFullProject (record: ProjectRecord)	25
string addProject (name: string)	25
string modifyProject (id: string, record: ProjectRecord)	25
ProjectRecord getProject (pattern: ProjectRecord)	25
void removeProject (id: string)	25
void removeProject (pattern: ProjectRecord)	25
void addProjectToGroup (id: string, group: string)	25
void removeProjectFromGroup (id: string, group: string)	25
ProjectRecord getDefaultProject ()	25
list getProjectList ()	25
list (or dictionary) getProjectDependencies (id: string, code type: string [optional])	25
void addProjectDependency (id: string, code type: string, code id: string)	26

void removeProjectDependency (<i>id: string, code type: string, code id: string</i>)	26
<i>Codes (Tasks), Subcodes (Pay Types), and Sub-Subcodes (Bill Types)</i>	26
Code-Related Methods	26
string addFullCode (<i>record: CodeRecord</i>)	26
string addCode (<i>name: string</i>)	26
string modifyCode (<i>id: string, record: CodeRecord</i>)	26
CodeRecord getCode (<i>pattern: CodeRecord</i>)	26
void removeCode (<i>id: string</i>)	26
void removeCode (<i>pattern: CodeRecord</i>)	26
void addCodeToGroup (<i>id: string, group: string</i>)	26
void removeCodeFromGroup (<i>id: string, group: string</i>)	26
CodeRecord getDefaultCode ()	26
list getCodeList ()	26
Subcode-Related Methods	27
string addFullSubcode (<i>record: SubcodeRecord</i>)	27
string addSubcode (<i>name: string</i>)	27
string modifySubcode (<i>id: string, record: SubcodeRecord</i>)	27
SubcodeRecord getSubcode (<i>pattern: SubcodeRecord</i>)	27
void removeSubcode (<i>id: string</i>)	27
void removeSubcode (<i>pattern: SubcodeRecord</i>)	27
void addSubcodeToGroup (<i>id: string, group: string</i>)	27
void removeSubcodeFromGroup (<i>id: string, group: string</i>)	27
SubcodeRecord getDefaultSubcode ()	27
list getSubcodeList ()	27
Subsubcode-Related Methods	27
string addFullSubsubcode (<i>record: SubsubcodeRecord</i>)	27
string addSubsubcode (<i>name: string</i>)	27
string modifySubsubcode (<i>id: string, record: SubsubcodeRecord</i>)	27
SubsubcodeRecord getSubsubcode (<i>pattern: SubsubcodeRecord</i>)	27
void removeSubsubcode (<i>id: string</i>)	27
void removeSubsubcode (<i>pattern: SubsubcodeRecord</i>)	27
void addSubsubcodeToGroup (<i>id: string, group: string</i>)	27
void removeSubsubcodeFromGroup (<i>id: string, group: string</i>)	27
SubsubcodeRecord getDefaultSubsubcode ()	27
list getSubsubcodeList ()	27
Expense Code-Related Methods	27
list getExpenseCodeList()	27
list getExpenseCurrencyList()	27
list getExpenseSourceList()	27
<i>Time Records</i>	27
Time Record-Related Methods	28
string addFullTimeRecord (<i>record: TimeRecord</i>)	28
string addTimeRecord (<i>comment: string</i>)	28
string modifyTimeRecord (<i>id: string, record: TimeRecord</i>)	28
TimeRecord getTimeRecord (<i>pattern: TimeRecord</i>)	28
void removeTimeRecord (<i>id: string</i>)	28
void removeTimeRecord (<i>pattern: TimeRecord</i>)	28
TimeRecord getDefaultTimeRecord ()	28
list getTimeList (<i>date: string</i>)	28
list getEnabledTimeFields ()	28
<i>Special Note about the 'Committed' field</i>	28
<i>Expense Records</i>	29
Expense Record-Related Methods	29
string addFullExpenseRecord (<i>record: ExpenseRecord</i>)	29
string addExpenseRecord (<i>comment: string</i>)	29
string modifyExpenseRecord (<i>id: string, record: ExpenseRecord</i>)	29
ExpenseRecord getExpenseRecord (<i>pattern: ExpenseRecord</i>)	29
void removeExpenseRecord (<i>id: string</i>)	29
void removeExpenseRecord (<i>pattern: ExpenseRecord</i>)	29
ExpenseRecord getDefaultExpenseRecord ()	29
list getExpenseList (<i>date: string</i>)	29
list getEnabledExpenseFields ()	29

<u>Time Sheets</u>	30
<u>Time Sheet-Related Methods</u>	30
<u>string getTimeSheetIDByDate (date: string)</u>	30
<u>string getNextTimeSheetID (id: string)</u>	30
<u>string getPreviousTimeSheetID (id: string)</u>	30
<u>list getDatesInTimeSheet (id: string)</u>	30
<u>float getTotalHoursInTimeSheet (id: string)</u>	30
<u>void addTimeRecordToSheet (id: string, trecid: string)</u>	30
<u>void removeTimeRecordFromSheet (id: string, trecid: string)</u>	30
<u>list getTimeRecordIDsInSheet (id: string)</u>	30
<u>void submitTimeSheet (id: string)</u>	30
<u>string getTimeSheetStatus (id: string)</u>	30
<u>string getTimeSheetReason (id: string)</u>	30
<u>string getLatestTimeSheetID ()</u>	30
<u>list getAllTimeSheetIDs ()</u>	30
<u>list getRecentTimeSheetStatus (number: integer)</u>	30
<u>list getTimeSheetRejectedStatuses ()</u>	31
<u>list getTimePeriodDates (date: string)</u>	31
<u>list getExpensePeriodDates (date: string)</u>	31
<u>list ChangeSheetStatus (sheet_recs list: list [of SheetRecords])</u>	31
<u>list getSheets (sheet_ids list: list [of Sheet IDs])</u>	31
<u>integer AssignApprovalTemplate (template_id_or_name: string, user_id list: list [of User IDs])</u>	31
<u>list getApprovalTemplates (template_ids list: list [of Template IDs – optional])</u>	32
<u>list getRecordsForSheet (sheet_type: string, sheet_id: string)</u>	32
<u>integer CreateApprovalTemplate (new_template: ApprovalTemplateRecord)</u>	32
<u>integer UnassignApprovalTemplate (template_id: string, user_id list: list [of User IDs])</u>	33
<u>Extra Field (Attribute) Methods</u>	34
<u>An Example</u>	34
<u>Historic Value Tracking</u>	35
<u>Extra Field Selection Lists</u>	35
<u>Extra Fields Data Types</u>	35
<u>Extra Field Value Methods</u>	36
<u>scalar getAttribute (object_type: string, object_id: string, id_attr_type: string)</u>	36
<u>void setAttribute (object_type: string, object_id: string, id_attr_type: string, value: scalar, overwrite_existing: bool [optional – defaults to true])</u>	37
<u>void deleteAttribute (object_type: string, object_id: string, id_attr_type: string)</u>	37
<u>list queryAttributes (object_type: string, object_id list: list [of Object IDs], id_attr_type list: list [optional – list of Attribute Type IDs])</u>	38
<u>Extra Field Management Methods</u>	39
<u>list queryAttributeTypes (search_pattern: AttributeTypeRecord)</u>	39
<u>string getAttributeTypeByName (object_type: string, type_name: string, full_record: boolean [optional – default is false])</u>	39
<u>list getAttributeObjectTypes ()</u>	40
<u>void deleteAttributeType (id_attr_type: string)</u>	40
<u>string addAttributeType (object_type: string, pname: string, data_type: string)</u>	40
<u>integer checkAttributeValue (data_type: string, value: scalar)</u>	41
<u>integer checkAttributeDataType (data_type: string)</u>	41
<u>void modifyAttributeName (id_attr_type: string, new_name: string)</u>	41
<u>void modifyAttributeTypeDescription (id_attr_type: string, new_description: string)</u>	42
<u>scalar getAttributeTypeDefaultValue (id_attr_type: string)</u>	42
<u>void setAttributeTypeDefaultValue (id_attr_type: string, default_value: scalar)</u>	42
<u>void removeAttributeTypeDefaultValue (id_attr_type: string)</u>	43
<u>list getAttributeTypeReportability (id_attr_type: string)</u>	43
<u>void modifyAttributeTypeReportability (id_attr_type: string, role list: list)</u>	43
<u>Historic Extra Fields Methods</u>	44
<u>list getHistoricalAttributeObjectTypes ()</u>	44
<u>list getHistoricalAttributeTypes (object_type: string)</u>	44
<u>void makeAttributeTypeAsHistorical (object_type: string, id_attr_type: string)</u>	44
<u>void dropAttributeTypeAsHistorical (object_type: string, id_attr_type: string)</u>	45
<u>Selection List Extra Field Methods</u>	45
<u>string addAttributeTypeSelectionValue (id_attr_type: string, the_value: scalar, is_default: boolean [optional – default is false])</u>	45

<u>void setAttributeTypeSelectionDefault (id_attr_value: string)</u>	46
<u>list getAttributeTypeSelectionValueRecords (id_attr_type: string)</u>	46
<u>list getAttributeTypeSelectionValues (id_attr_type: string)</u>	46
<u>void deleteAttributeTypeSelectionValue (id_attr_value: string)</u>	47
<u>Miscellaneous Session Object Methods</u>	47
<u>void logout ()</u>	47
<u>JXAPI BATCH COMMANDS</u>	47
<u>jxAPI Batch Record Details</u>	49
<u>jxAPI Batch Example</u>	51
<u>APPENDIX A. XML SCHEMA FOR RECORD TYPES</u>	54
<u>JOURNYX JXAPI V5.6.0.0 SCHEMA/DTD</u>	54

Legal Disclaimer

Neither Journyx, Inc. nor its members shall be responsible for any loss resulting from any use of this document or the specifications herein.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Copyright

©2004 Journyx, Inc. All rights reserved. Permission to use, copy, and redistribute this documentation for any purpose and without fee or royalty is hereby granted, provided that you include the following notice on ALL copies of the documentation or portions thereof:
"Copyright © Journyx, Inc. All Rights Reserved. <http://www.journyx.com>"

Trademarks

JxAPI, Timesheet, Timecard, Journyx and Journyx, Inc. are trademarks of the Journyx, Inc.

1. Journyx Timesheet™

Each Journyx Timesheet installation is conceptually divided into three parts: a relational database, which stores all time and expense information; a web server, which provides a browser-based user interface; and an application server, which performs the actual work of time, mileage and expense management. The interface described in this document allows client software to bypass the web-based user interface and communicate with the application server directly, to enable the creation of alternate user interfaces and applications that require programmed interaction with time and other business data.

In its current version, this interface is closely related to the internal structure of the Timesheet database. The following sections describe the kinds of data stored in the Timesheet database, and how they relate to each other; a detailed description of individual data items and interface methods is provided later, in the Reference chapter. (**Note:** This document describes only the parts of the Timesheet database and software that are supported by the current version of this interface.)

Tables and Records

Journyx Timesheet stores all data in a relational database. While it is certainly possible for an external program to interact directly with the database, one of the purposes of this interface is to make that unnecessary because of the risks involved. The danger is that the database might be changed in an inconsistent way, possibly rendering the Timesheet installation broken or corrupted. Normally, Timesheet 5.6 will be running with a basic set of Foreign Key Constraints to prevent most cases of referential integrity violation. However, these constraints can be easily disabled with a direct database connection, and furthermore there are certain relationships that are not protected by a constraint. These are some reasons why it is important to use the jxAPI instead of directly modifying the database.

Even though it is not necessary or desirable to interact directly with the database, it is still necessary to understand the relationship between the contents of the database and the information that passes through this interface. The object records that are used by the jxAPI closely resemble their corresponding database tables.

The Timesheet database contains many tables, each of which stores all records of a particular type. Related records are handled in typical relational database fashion, using unique keys to provide a linkage between separate records. This storage layout is reflected here by the use of an individual structure type for each kind of record, making it the responsibility of the client program to comprehend the relations between them. In order to facilitate relational links between records, every record has a column that contains a unique ID for that record.

The core record types used by this interface are:

Users: Each Timesheet user or administrator has a user record which records information such as their name, phone number, pay rate, and user interface preferences. The key field in the user table is the user's login name.

Projects, Codes (Tasks), Subcodes (Pay Types), and Subsubcodes (Bill Types): When a user reports time, mileage and expenses, they can be assigned to a particular project and to an activity (specified by up to three types of codes) within that project. The Timesheet administrator can define whatever projects and activities are relevant, so the list of possible projects, codes, subcodes, and subsubcodes are stored in the database. Note that since codes, subcodes, and subsubcodes are identical in structure and behavior, the rest of this document will refer to all of them simply as "codes" unless a distinction is important.

Time records: Each time record covers the hours reported for a single user, on a single day, for a single combination of project and codes.

Mileage records: Each mileage record covers a single mileage event, by a single user, on a single day, for a single combination of vehicle and measurement.

Expense records: Each expense record covers a single expenditure, by a single user, on a single day, for a single expense code.

This interface provides a consistent set of functions that allow basic database operations on the tables that hold each of these record types.

Domains and Groups

The Timesheet server allows users, projects, and codes to be divided into *groups*, so that each user can be presented with only the project and activity choices that are actually relevant. Each user, project, or code can be a member of more than one group at a time. This interface provides functions to perform basic operations on the group table, as well as functions to manipulate the group membership of users, projects, and codes.

In some installations, the Timesheet server may divide its contents into *domains*. Although they share a common database, each domain is logically separate from all of the other domains on the same server. (In other words, each domain has its own set of users, groups, projects, codes, etc.) Each user can only belong to one domain, so when a user is logged in he can only see things in his domain. The “top level” domain ID is always `install_root_dom` – all items belong to this domain unless they are specifically created in a different domain.

The creation and management of the actual Domains themselves is not currently supported in the jxAPI. In other words, you must create and modify Domains from the Timesheet web interface. However, most of the jxAPI record types include a “domain ID” field where appropriate, so it is possible to use the jxAPI with records that are in different domains.

Authentication

The Journyx Timesheet server uses a user- and session-based authentication scheme to restrict access to the database. Before a client application can do anything that would query or alter the database, it must establish a session using a valid username, password, and (in some cases) domain. The server will prevent the client application from doing anything that exceeds that user's privileges.

The privileges granted to a user are based on that user's *roles*, each of which grants the privileges necessary for the user to fulfill it. As with domains, creating and modifying the roles themselves must be done in the web interface, but there are jxAPI methods to grant and revoke roles from individual users.

Timesheets

A timesheet is a group of time records that may be submitted as a whole to some higher authority for approval. Each sheet covers a range of dates covering a single reporting period, and holds references to the time records (which are stored in the database in the normal manner) that are associated with it. This interface supports submission of time sheets, with limited support for the approval process and other sheet-management tasks.

2. Changes in the jxAPI from previous versions

Changes from Timesheet 4.5 to Timesheet 4.6

There was an error in earlier versions of this document in the description of the `addRecord` methods, where `Record` was one of `Project`, `User`, `Group`, `Code`, `Subcode`, `Subsubcode`, `TimeRecord`, or `ExpenseRecord`. It stated that these methods could be called with only a single string argument, and would create a record using that name based on default values. What was really happening was that the corresponding `addFullRecord` method was being called. These methods expected a full and complete record argument, not a single string. The specification document was later changed to reflect the underlying call to `addFullRecord`. However, in the jxAPI version 5.0 and higher, the `addRecord` method(s) were restored with the originally documented semantics. See the “Changes in 5.0” section below for more details.

There were no major changes to existing method calls, but several new methods were added in Timesheet 4.6. Here is the list of new methods in 4.6. See the reference material later in this document for a complete description of all public methods.

[ChangeSheetStatus](#)
[getSheets](#)
[AssignApprovalTemplate](#)
[getApprovalTemplates](#)
[getRecordsForSheet](#)
[CreateApprovalTemplate](#)
[UnassignApprovalTemplate](#)
[getProjectDependencies](#)
[addProjectDependency](#)
[removeProjectDependency](#)

Changes from 4.6.0 to 4.6m1p1 (and 46m2)

The jxAPI began reporting its own version as 4.6.1.1 (or 4.6.2.0) instead of 1.0.7.

The only functional change was the addition of several “Extra Fields” to the records returned by the `getProject` and `getUser` methods. These Extra Fields were limited to a few predefined fields, and they were read-only. (Values could not be inserted by adding the fields to records passed to `addFullProject` or `addFullUser`.) This “feature” has been removed in 5.0.0 in favor of the more comprehensive Attribute system (also called “Extra Fields”), whose features are fully supported by the jxAPI.

A bug was fixed where the “session URL” returned by the jxAPI `login` method was missing the “document root” for sites hosted under the ASP model. Now the correct, full URL is returned for all sites.

Changes from 4.6m1p1 to 5.0.0.

Both Timesheet and the jxAPI have a new version number. The jxAPI now reports its own version as 5.0.0.0 (or higher.) The version of this Specification document *and* the jxAPI XML Schema changed from 1.3.x to 1.4.0.

The Extra Fields included in the records that `getProject` and `getUser` returned are no longer present. The old Extra Field system in Journyx Timesheet has been replaced by a new set of database tables, Timesheet objects, and jxAPI methods, that provides a user extensible system of attaching “Attributes” to several different types of Timesheet objects, such as Users and Projects. See the [Extra Fields](#) section of the Reference chapter for more detailed information.

The originally documented behavior of the various `addType` methods was restored in version 5.0. These methods take a single string argument and create a record using that name and other appropriate

(automatically selected) default values. The ID of the new record is returned. See the appropriate Reference section below for a complete description of each method and other notes.

Changes from 5.0.0 to 5.0m2

These changes were made after the initial 5.0.0 release:

There is an *unsupported* `keep_old_sessions` parameter for the [login](#) method. Please read the notes in that method description before using this parameter.

The [apiVersion](#) method reports the version number as 5.0.2.0.

The two methods [assignRoleToUser](#) and [removeRoleFromUser](#) can now take the Role argument as either a database ID or a “pretty name.”

In some cases the [getProjectDependencies](#) method will return different names for the various code types than in previous versions. For instance, it will return a ‘tasks’ field instead of a ‘code’ field. The old style names will continue to work as parameters to this and the other project dependency methods (either kind of name is recognized.) This change was made so that the code names correspond to what they are called (by default) in the product GUI.

There are four new methods dealing with Group membership: [addObjectToGroup](#), [removeObjectFromGroup](#), [listGroupObjects](#), and [getGroupObjectClasses](#).

Two User related methods: [addFullUser](#) and [modifyUser](#), now allow you to specify the three GUI fields (`time_gui`, `expense_gui`, `mileage_gui`) by the GUI’s name (pretty name) as well as the GUI ID.

Changes from 5.0m2 to 5.0m3

A new method [jxAPIBatch](#) enables you to group separate jxAPI requests into a single SOAP message. This can give your client application a considerable boost in performance if used correctly. This capability was also added to Timesheet version 4.6m3.

Changes from 5.0m3 to 5.5

There were no major changes in the jxAPI in this release. Please see <http://www.journyx.com/jxapi.html> for up-to-date information about the jxAPI. We are planning to provide a full WSDL based interface in the very near future.

The [getAttributeTypeReportability](#) and [modifyAttributeTypeReportability](#) methods were added. These let you set who is allowed to run Timesheet reports that include the given extra field.

The [apiVersion](#) method reports the version number as 5.5.0.0.

Changes from 5.5 to 5.5m1

There were no major changes in the jxAPI in this release. [A note](#) was added in this document regarding the fact that the **Content-Type** and **Content-Length** HTTP request headers are required for Timesheet SOAP requests.

The [apiVersion](#) method reports the version number as 5.5.0.1.

Changes from 5.5m1 to 5.5.2

No changes to the jxAPI in this version other than that the [apiVersion](#) method reports the version number as 5.5.2.0

Changes from 5.5.2 to 5.5.2m1

No changes to the jxAPI in this version other than that the [apiVersion](#) method reports the version number as 5.5.2.1

Changes from 5.5.2 to 5.6

No changes to the jxAPI in this version other than that the [apiVersion](#) method reports the version number as 5.6

3. The Interface

This interface uses the Simple Object Access Protocol (SOAP), which is a data interchange protocol based on XML. One of the primary purposes of SOAP is to provide language-independent remote procedure calls between a client and a server with less overhead than traditional remote-object systems. To accomplish this, SOAP uses HTTP to transport calls and responses, meaning that it can be easily integrated into existing web infrastructure without adding a lot of complexity. The SOAP specification can be found at <http://www.w3.org/TR/SOAP/>.

Access

Since SOAP uses HTTP to transport calls and responses, the client will access this interface through the same server that provides the web-based user interface to Timesheet. If the client can access the web server portion of the Journyx installation, it can use the SOAP interface. This eliminates the need for special firewall / network configuration.

SOAP Client libraries are available for most popular programming languages, including Java, C++, Perl, and Visual Basic. Journyx does not supply any SOAP client tools; appropriate client access tools must be obtained from your language vendor or other third party.

SOAP Headers

All SOAP requests to Journyx Timesheet must be properly formatted according to the SOAP 1.0 standard. A SOAP request will not be recognized by Timesheet unless it has at least two HTTP request headers that are required by the SOAP standard.

Content-Type: `text/xml`

Content-Length: *[Length of request body in bytes]*

Normally most SOAP client software, such as Apache Axis or the Microsoft SOAP Toolkit, will automatically supply these headers. If you are using customized SOAP software, be sure to include these HTTP headers in your request.

Data Types

SOAP is a language-independent protocol, with its own object model and data types. Typically, a client library handles the mapping between SOAP and a specific language, so that in general, using the SOAP API is not much different than calling into “native” libraries or modules.

A complete description of how SOAP encodes data can be found in chapter 5 of the SOAP specification. In particular, it should be noted that this specification makes extensive use of SOAP "struct"s, which are compound values with named members (comparable to a structure in a traditional programming language).

The following table describes the relationship between the type names used in this interface specification, and their representation within a SOAP data packet. Application programmers must make sure that the client library maps these SOAP types to something appropriate in the target language.

<u>Generic Type</u>	<u>SOAP Type</u>
null object	<code>xsi:null='1'</code>
Integer	<code>xsi:type='xsd:int'</code>
Long	<code>xsi:type='xsd:integer'</code>
Float	<code>xsi:type='xsd:double'</code>
string ¹	<code>xsi:type='xsd:string'</code>
Struct	(intrinsic to SOAP)
List	<code>xsi:type='xsd:ur-type[]'</code>
Boolean	<code>xsi:type='xsd:boolean'</code>
date/time	<code>xsi:type='xsd:timePeriod'</code>
Binary	<code>xsi:type='enc:base64'</code>

The namespaces used in the table above are:

<code>xsi</code>	<code>http://www.w3.org/1999/XMLSchema/instance/</code>
<code>xsd</code>	<code>http://www.w3.org/1999/XMLSchema/</code>
<code>enc</code>	<code>http://schemas.xmlsoap.org/soap/encoding/</code>

In the remainder of this specification, only the generic data types will be mentioned, and it is assumed that an appropriate native representation will be provided by the client library.

SOAP Objects

SOAP represents available services by treating them as if they were methods on objects. In most languages, it is possible to represent a SOAP server with a proxy object, so that making calls to the server is no different than making calls to methods on a local object.

Objects in SOAP are referenced by URI. Since this interface uses HTTP to transport SOAP messages, this means that object references are simply URLs pointing to the Journyx Timesheet web server. The Timesheet server has a single publicly accessible *base object*, which acts as an entry point to the rest of the interface. Its URL is always:

`http://server/doc_root/jtcgi/soap_cgi.pyc`

Here, *server* is the name and port of the Journyx Timesheet web server. For example, if Timesheet is installed on the machine `test.journyx.com` at port 3333, the URL of the base object would be `http://test.journyx.com:3333/jtcgi/soap_cgi.pyc`. It is recommended to always include the port number even when Timesheet is running on the default web port (80.)

doc_root is optional, but is often used when more than one logical "site" runs on the same host and port, or when Journyx, Inc. is hosting Timesheet as an Application Service Provider (ASP.) In most cases, there will be no "document root" on locally hosted sites, and the *doc_root* part of the URL is not used when connecting to the SOAP service.

¹ Parameters of type **string** must contain only characters that are legal in XML text. If there is a need to pass arbitrary binary data, the **binary** type will be used.

Exception (Error) Handling

Unless otherwise specified, all errors or internal Python exceptions will be reported by the return of a SOAP Fault response, so that a successful return indicates that no error has occurred (unless the method specification states otherwise.²) In most languages, the client library will translate this into an exception, which will then be channeled through the native error handling mechanism. If the client language does not support exceptions, it is the application programmer's responsibility to check for a fault condition after each call to a SOAP object.

Methods, Parameters and Types

Every method in the jxAPI is specified in this document in this format:

return_type MethodName (*parameter_1_name*: **data_type**, *parameter_2_name*: **data_type**,...)

In addition to the usual types (strings, integers, floating point numbers) these types are used by the specification document:

`scalar` – can be either a string or a number.

`void` – no return value is expected. Normally the SOAP Null value (the Python `None`) is returned.

² For instance, the `ChangeSheetStatus` method can change the status of multiple sheets with a single method call. Some of the changes may succeed, and some may fail, but the failures will not cause a SOAP Fault. Instead, an indication of failure or success for each sheet will be returned as a structure.

4. Reference

Session Management

Most operations on time and expense data require authentication. The Journyx Timesheet server uses a session-oriented authentication system, based on named users with passwords. To use most of this interface, the client must identify itself to the server and establish access rights based on an existing user.

To establish an authenticated session, the client must call the base object's `login` method, passing a valid user name and password. If the login is successful, the return value will be a string URL of a *session object*. The client then establishes a *new* SOAP connection to the session URL. That connection is accorded the credentials of the logged-in user until the `logout()` method is called, or the session expires after too much idle time.³

In Python, the code to create and use a session object might look like this:

```
import SOAP

server = SOAP.SOAPProxy("http://test.journyx.com:3333/jtcgi/soap_cgi.pyc")
session_url = server.login("joe","joespass")

session = SOAP.SOAPProxy(session_url)

###
### now use the session object to call JX API methods
###

project_id = session.addProject("NewProject")
...
session.logout()
...

### you can now log in again using the server object
new_session = server.login("jane","janespass")
```

Although this example is in Python, the same concepts will apply in any SOAP client language such as Visual Basic. In this example, `server` is the Base object (the Base object methods listed below apply) and the `session` object is used to call all JX API methods except for those belonging to the Base object.

Another thing to keep in mind is that the `login()` method may throw an exception (generate a SOAP fault) if the username or password is incorrect, or in some other situations as documented below for `login()`. You should check for these faults and handle them appropriately.

In the web-based user interface, each session has a timeout after which its authentication cookie becomes invalid, forcing the user back to the login screen. Sessions established using SOAP also have a time limit; attempts to call any method on the session object after the session timeout has elapsed will result in a SOAP Fault. Once that occurs, the client will then be required to create a new session object in order to continue using the server.

In practice, the timeout will be long enough that it should not occur during a typical interactive session. The session timeout is configurable (through the web interface) in the Global Preferences section "Security and Access" under the preference named "Idle time before a user is forced to relogin (minutes)."

When you are finished with the session, it is recommended to call the `logout()` method of your session object (documented below.)

³ The length of idle session expiration is controlled by the Timesheet global preference "Idle time before a user is forced to relogin (minutes)" under the "Security and Access" section of the preferences page.

Base Object Methods

list apiVersion ()

Returns the version of the jxAPI specification that the server supports. The return value is a list of four integers, representing the major version, minor version, revision, and the patchlevel of the interface. The current version of the jxAPI as reported by this method is [5, 6, 0, 0]

boolean versionCheck (*major: integer, minor: integer, revision: integer, patchlevel: integer* [optional])

Checks whether the server is compatible with the specified version of the interface. If the return value is true, the server implements an interface that is compatible with the specified revision. See the [Changes from 4.6.0](#) section for a note about the API versioning.

string login (*user: string, password: string*)

Creates a new session, allowing access to the database with the rights of the specified user. If the login is successful, the return value is the URI of a session object. Otherwise, a SOAP Fault occurs.

login () may raise a SOAP Fault (exception) under these conditions:

- the user's password time limit has expired (if so, you can use `changeUserPassword ()` to fix this)
- the user's domain's license has expired and all users in that domain are locked out
- the username does not exist
- the password is incorrect

The Fault will provide a string that describes what the specific problem is.

Note: there is an *unsupported* additional option to the login method that will override the normal session behavior where all previous login sessions for the user are invalidated. The normal (default) behavior is that each new login under a particular user ID will effectively log out anyone else who was using the same user ID. However, if you set the third parameter of the jxAPI login method (called `keep_old_sessions`) to the integer value 1, then any existing login sessions are kept intact. (The default value of this parameter is 0, which will cause old sessions to be deleted.)

This parameter is unsupported because it is not normally desirable to have multiple separate clients using the same User ID at the same time. The interaction of multiple simultaneous sessions may produce unexpected results and must be carefully managed by the client. Please consult with Journyx before using this parameter in a production environment.

void changeUserPassword (*user: string, currentpassword: string, newpassword: string*)

Changes the password for a user without first establishing a session. This method is provided to allow users to change an expired password, since it is not possible to establish a session with an expired password. If the current password is incorrect, or the password cannot be changed for some other reason, a SOAP Fault will occur.

string version ()

Returns the version of the Journyx Timesheet server software. This is not necessarily the exact same as the version reported by the `apiVersion` method. Instead, it returns the version string listed in the site configuration file (the `config` file) which is currently "5.6".

string installDate ()

Returns the date on which the server software was installed, as a printable string.

string expirationDate ()

Returns the date on which the server license expires, as a printable string. This is the expiration date for the “master” site license, and does not reflect the expiration dates of any domains. If the license is set to never expire, the string “Never” is returned. Otherwise the expiration date is returned in YYYYMMDD format.

string licensedUsers ()

Returns the number of users allowed by the server license, as a printable string. This is the maximum number of licensed users for the whole site, and does not reflect any domain-specific restrictions. If an unlimited number of users are licensed, this method will return -1 (negative one.)

string hostname ()

Returns the hostname of the server.

string licensedHost ()

Returns the hostname specified in the server license.

string uname ()

Returns the name of the server operating system.

string companyName()

Returns the company name specified in the server license.

string customerNumber()

Returns the customer number specified in the server license.

string adminName()

string adminTelephone()

string adminEmail()

Return the name and contact information of the server administrator.

Structs and Session Object Methods

Privileges

Although every session object has the same methods, the server allows the client to perform only those operations allowed by the privileges of the user whose identity was used to create the session. If the client attempts to perform an operation that is not allowed, the result will be a SOAP Fault.

The specific operations available to a user depend on the *roles* that have been assigned to that user (or the user’s group) by the administrator. The *getUserRoles* session object method can be used to find out what roles have been assigned to a particular user. In the default server installation, the following roles are predefined; a particular server might also have customized roles defined by the administrator or provided as part of a specially licensed module.

Role	Description
install_admin	Administrator; full access to everything.
install_manager	Has authority over groups.
install_reporter	Can only run reports.
install_pmanager	Has authority over groups and projects.
install_cust1	Can enter time for other users.
install_user	An ordinary user.

Domains

Since users are *not* unique across Timesheet domains, it is not necessary to supply a domain ID when logging in. In addition, most methods that add records to Timesheet will automatically fill in the appropriate domain ID in the new record, except where specifically noted. Management of domains (creating and editing them, for example) is currently beyond the scope of the jxAPI interface.

Database Manipulation: General Considerations

The Journyx Timesheet database contains many different record types, but the basic operations on each are the same: addition, deletion, modification, and retrieval. To handle all of these record types, the session object has a number of methods that are almost identical in operation, differing only in what kind of data they handle.

Passing and returning records: Whenever a record is passed to, or returned from, one of these methods it is in the form of a struct (a record structure.) The contents of the struct representing each record type are described in more detail below. (A formal XML Schema for these structs is provided in [Appendix A](#) of this document.)

String sizes: In the struct definitions below, strings are referred to as "small", "medium", or "large". A "small" string is limited to 30 characters in length, a "medium" string is limited to 60 characters, and a "large" string is limited to 252 characters. If you try to pass in a string that exceeds the field's size limit, the server will throw an exception (which in turn generates a SOAP Fault message.) You must keep in mind that these are the "hard" limits that the database imposes on a given's field length. Extra characters may be used to quote or "escape" special characters, including single quotes. Issues like these may make a "30 character string" really become a 35 character string when it is time to insert into the database, so err on the side of caution.

ID Fields: Every record has an *id* field, which is a short alphanumeric string that uniquely identifies that record. The ID field must be unique among all records of the same type. In most record types, the ID has no real significance other than its usage as a way to refer to a particular record; if no ID is supplied when a record is created, the server will generate a new one. Please note that in the *user* record, the ID is used as the user's login name, so it must be specified when creating a new user.

Please note: it is important to realize that IDs are case-sensitive. That is, `xyzzy`, `xYzzY`, and `xYZZY` are three different and unique IDs. This is also true for User IDs.

The following methods are available for every record type. In each case, replace *Type* with the appropriate type of item: (User, Project, Code, Subcode, Subsubcode). Manipulation of the other Journyx Code types (mostly related to Expense and Mileage records) are not currently supported except where explicitly noted.

string addFullType (*new_record*: **struct**)

Adds a complete record to the database. All calls to these methods are now passed to the new method `addFullRecord(table_name, struct)` with the appropriate `table_name` parameter filled in. See that method for more information.

string addFullRecord(*table_name*: **string**, *new_record*: **struct**)

Adds a complete record belonging to the given `table_name` to the database. The record passed to this function must be complete; it will not be filled in with default values, but it will be checked for validity. The only exception to this is the *id* and *domain* fields: for most types, if it is empty the server will generate a value for it. The return value is the record's ID, whether or not it was generated by the server. (You are allowed to generate your own IDs, but they will be rejected if they conflict with an existing record in the same table.)

If the `table_name` parameter is not in the list of legal tables defined by the `getSearchableTables` method, a `ValueError` fault will be raised. You will also get `ValueError` if a column that is defined as

not-null has a null value in it. Foreign key constraint and other low-level database violations will be passed back as a SOAP Fault, usually named `JXDBQueryException`.

string `addType (name: string)`

For all different types of records, calling this method is like making two other method calls: first getting a default record with the given **name** by calling `getDefaultRecord()` and then adding the record with `addFullRecord()`. In fact, the `addType` methods are implemented by calling those methods for you using the correct table name and the **name** parameter.

Please note that the `name` parameter of this method may refer different fields depending on the record. For instance, for `code` records, the `name` parameter refers to the `pretty_name` field. (A “Pretty name” is the name that users see in the GUI, as opposed to the internal unique ID of the record.) However, for `User` records, the `name` parameter refers to the User ID. For `Groups`, it refers to the literal `name` field.

string `modifyType (id: string, existing_record: struct)`

Modifies the existing record with the given `id`, replacing its contents with the given `record`.

struct `getType (a_pattern: struct)`

Retrieves a record by matching it against a pattern. The pattern is simply a prototype record: the return from this function is the first record that exactly matches all of the non-zero (for numeric fields) and non-empty (for string fields) field values in the prototype. The prototype does not have to contain all possible fields; any field missing from the prototype will be treated as if it was empty or zero.

void `removeType (id: string)`

void `removeType (pattern: struct)`

Removes a record that is identified by its `id`, or by a pattern match similar to the one used for retrieving records. See the description of `getType`, above.

void `addTypeToGroup (id: string, group: string)`

Adds a record, identified by its key, to a group. Grouping applies only to `User`, `Project`, `Code`, `Subcode`, and `Subsubcode` records.

void `removeTypeFromGroup (id: string, group: string)`

Removes a record, identified by its key, from a group. Grouping applies only to `User`, `Project`, `Code`, `Subcode`, and `Subsubcode` records.

struct `getDefaultType ()`

Returns a prototype record filled in with default values. Note that some fields might not have a meaningful default, and will be filled with zeroes or empty strings. In particular, the `id` field of this default record will be empty, since the server will fill it in when a record is added to the database (except for `UserRecords` – the user ID must be supplied for those.)

General Use Methods

list `getSearchableTables ()`

Returns a list of table names that are valid as parameters to other methods below that require a table name as an argument. If any method described elsewhere lists `table_name` as a parameter, then the table *must* be in the list returned by this method (unless otherwise noted) or the method will raise an exception.

list `getRecordStructure (table_name: string)`

Given a searchable table (see above), this returns a special list describing the structure of the table’s records. This is not the same format as the actual records themselves (such as would be required by a call

to `addFullRecord`) but rather it is a format that is useful in converting raw record tuples (such as in a database record) to the dictionary-like structured records of the jxAPI.

Each element of the returned list is itself a list with 2 elements. The length of the overall list is the same as the length of the records in the given table.

Each 2-element sub-list consists of (“Field Name”, “Type”) indicators. Here, “Type” will be an empty string (‘’) for string fields, and the number 0 for numeric fields. This is *not* a default value; use the `getDefaultRecord` method to get default values. If the “Field Name” position is None, then that field is not used by the jxAPI.

list `getRecordsList (table_name: string, pattern: struct)`

This method takes two parameters, and is used to do generic searches on any of the supported tables. It returns lists of Records whose format depends on the table searched. For instance, if the `table_name` parameter is “users” then the return value will be a list of **UserRecords**.

`table_name` must be in the list of supported names returned by `getSearchableTables()`, or a `ValueError` exception is raised.

The `pattern` parameter must be a dictionary-format record structure which names the fields to be searched and the values to search on.

For instance, if you want to search for all projects that are children of Project ID 12345, your method call would look like:

```
search_pattern = { 'parent': '12345' }
results = getRecordsList('projects', search_pattern)
```

Each element in `results` would be a **ProjectRecord**. If no records matched your search query, an empty list with no elements is returned.

Currently LIKE style searches (partial string searches) or other advanced types of queries are not supported.

struct `getDefaultRecord (table_name: string, display_name: string [optional])`

This method returns a usable default record suitable for the given table name. The `display_name` parameter is optional, but if it is supplied, the record will use that name where appropriate, either in the [pretty / display] name or the description (or in the case of user records, as the user ID.) This method raises the `ValueError` fault if the `table_name` is not in the list of valid names returned by the `getSearchableTables` method.

Preference Records

Each preference record describes the preference and the current value associated with that. The **PreferenceRecord** struct contains these elements:

Element Name	Element Type	Description
<i>id</i>	String (small)	Key. A short alphanumeric string which uniquely identifies this preference record.
Description	String (large)	The description of the preference.
Value	String (large)	The value of the preference
RESERVED	Number	RESERVED
RESERVED	String (large)	RESERVED
RESERVED	String (large)	RESERVED
Domain	String (small)	The Timesheet domain to which this preference belongs.

list getPreferenceValue (string)

This method returns a list of preference records utilizing a sub-string match on both the id and Description fields. Searching for 'proj' will return all records with the word "Project" or "project" in either the id or description field.

Users

Each user of the Timesheet server is represented by a **UserRecord** struct, with the following elements:

Element Name	Type	Notes
<i>id</i>	String (small)	Key. The username; a short alphanumeric string which uniquely identifies the user and which is used as the user's login name.
num_input	integer	The number of blank input lines that are presented on this user's time entry screen.
num_dates	integer	The number of date columns that are presented on this user's time entry screen.
full_name	String (medium)	The full name of this user.
default_comment	String (medium)	A string which will be used as the default text for the <i>comment</i> field in time and expense records.
expense_gui	string (small)	The expense entry GUI id for this user.
time_gui	string (small)	The time entry GUI id for this user.
mileage_gui	string (small)	The Mileage / Travel entry GUI id for this user.
domain	string (small)	The id of the domain which this user is a member of.

Most operations on user records can only be performed by an administrator; ordinary users can change their own password and modify a few parts of their own records, but little else.

User-Related Methods

string addFullUser (*record*: **UserRecord**)

string addUser (*name*: **string**)

string modifyUser (*name*: **string**, *record*: **UserRecord**)

UserRecord getUser (*pattern*: **UserRecord**)

void removeUser (*name*: **string**)

void removeUser (*pattern*: **UserRecord**)

void addUserToGroup (*name*: **string**, *group*: **string**)

void removeUserFromGroup (*name*: **string**, *group*: **string**)

UserRecord getDefaultUser ()

Standard record access methods: see above for a complete description.

Special note (5.0m2 and higher): the two User related methods **addFullUser** and **modifyUser** have one change in this version in Timesheet. The three GUI fields (Time, Expense, and Mileage) may be specified by the GUI's pname (display name) as well as by the ID. A ValueError exception will be raised in the name cannot be converted to a unique ID. The **getUser** method does not understand this convention – searches for user records based on GUIs must use the GUI IDs.

void assignRoleToUser (*name*: **string**, *role*: **string**)

Assigns a role to a user. The *role* may specify either a Role ID, or the "Pretty Name" (display name) of the Role, provided that it can be unambiguously converted to a single Role ID. (If it cannot be converted to an ID, or the ID given is not found, a ValueError is raised.)

void removeRoleFromUser (*name*: **string**, *role*: **string**)

Removes a role from a user. The *role* may specify either a Role ID, or the “Pretty Name” (display name) of the Role, provided that it can be unambiguously converted to a single Role ID. (If it cannot be converted to an ID, or the ID given is not found, a ValueError is raised.)

list getUserRoles (*name*: **string**)

Returns a list of the roles of a user.

string getDefaultRole ()

Returns the default “basic user” role ID for the current domain (the domain of the jxAPI user.) Usually (but not always) this is `install_user`.

void changePassword (*newpassword*: **string**)

Changes the password of the current user.

void changePasswordForUser (*name*: **string**, *newpassword*: **string**)

Changes the password of any existing user. The normal password constraints will apply.

list getUserPermissions (*name*: **string**)

Returns a list of modules which the user is permitted to access. If the *name* parameter is empty, the current user's permissions will be checked; only an administrator may check permissions for a different user. The return value is a list of strings, each of which indicates one permission or feature that the user can access. Currently, the possible values are “time” and “expense” to indicate that the user is allowed report time and/or expenses.

Groups

A *group* is a collection of related projects, codes, and users. Each group is represented by a **GroupRecord** struct with the following elements:

Element Name	Element Type	Notes
<i>id</i>	string (small)	Key. A short alphanumeric string that uniquely identifies the group.
<i>name</i>	string (small)	The group's name, as it will appear in the GUI.
<i>description</i>	string (large)	A description of the group.
<i>parent</i>	string (small)	[Deprecated] Please leave this field blank (use an empty “” string) as it has no effect. (Groups <i>do not</i> have a parent-child relationship.)
<i>domain</i>	string (small)	The id of the domain this group is a member of.

Group-Related Methods

string addFullGroup (*record*: **GroupRecord**)

string addGroup (*name*: **string**)

string modifyGroup (*id*: **string**, *record*: **GroupRecord**)

GroupRecord getGroup (*pattern*: **GroupRecord**)

void removeGroup (*id*: **string**)

void removeGroup (*pattern*: **GroupRecord**)

GroupRecord getDefaultGroup ()

Standard record access methods: see above for a complete description.

void addObjectToGroup (*group_id_or_name*: **string**, *object_class*: **string**,
object_id_name: **string**)

This method can add any type of supported object to a group.

Param 1: *group_id_name* (string) the ID (or Pretty Name) of the group.

Param 2: *object_class* (string) the name of the 'class' of the object whose ID is the following parameter.

Param 3: *object_id_name* (string) the ID (or the Pretty Name) of the Object to be added.

This method returns None if the operation was successful.

A ValueError is raised if the Group OR Object ID/name cannot be mapped to a single unique ID. In other words, if you try to add an object to the group "Test Group", and there is more than one group named "Test Group", a ValueError will be raised. Likewise, if the Object name cannot be disambiguated, the ValueError will be raised.

In this case, you will need to provide the specific ID of the group and/or object, which can be found with the [getGroup\(search_rec\)](#) method, or the various other methods for searching for objects.

ValueError is also raised if the *object_class* is not one of the supported classes. See the [getGroupObjectClasses](#) method.

void `removeObjectFromGroup` (*group_id_or_name*: **string**, *object_class*: **string**, *object_id_name*: **string**)

This method will revoke the given Object's membership in a Group.

Param 1: *group_id_name* (string) the ID (or Pretty Name) of the group.

Param 2: *object_class* (string) the name of the 'class' of the object whose ID is the following parameter.

Param 3: *object_id_name* (string) the ID (or the Pretty Name) of the Object that we are removing from the Group.

This method returns None if the operation was successful.

A ValueError is raised if the Group OR Object ID/name cannot be mapped to a single unique ID. In other words, if you try to remove an object from the group "Test Group", and there is more than one group named "Test Group", a ValueError will be raised. Likewise, if the Object name cannot be disambiguated, the ValueError will be raised.

In this case, you will need to provide the specific ID of the group and/or object, which can be found with the [getGroup\(search_rec\)](#) method, or the various other methods for searching for objects.

ValueError is also raised if the *object_class* is not one of the supported classes. See the [getGroupObjectClasses](#) method.

list `listGroupObjects` (*group_id_or_name*: **string**, *object_class*: **string**)

This method returns a list of all the IDs of the members of a certain type of the given Group. The method can also return a dictionary listing all members of every type by using "all" for the *object_class* argument.

Param 1: *group_id_name* (string) the ID (or Pretty Name) of the group.

Param 2: *object_class* (string) the name of the 'class' of the object that you want to list. Use the special string "all" (which is not returned by `getGroupObjectClasses`) to return a dictionary mapping all the Group Object Classes to a list of members.

A list of strings consisting of the IDs of all the group members of the given type is returned. An empty list indicates that no objects of the given class are members of the group.

However, if the *object_class* parameter is "all", then a dictionary (hash) is returned. The keys are the different Group Object classes, and the values are a list of members in the given group. The lists may be empty if there are no members of that class.

A `ValueError` is raised if the Group ID/name cannot be mapped to a single unique Group ID. In other words, if you try to list objects from the group "Test Group", and there is more than one group named "Test Group", a `ValueError` will be raised.

`ValueError` is also raised if the *object_class* is not one of the supported classes (or "all".) See the [getGroupObjectClasses](#) method.

list `getGroupObjectClasses ()`

Returns a complete list of names that are valid to use for the *object_class* parameter of the above three methods. The list may include synonyms for the same class.

Here is the basic set of names of allowed Group members, not including any aliases:

```
users
roles
projects
time_guis
expense_guis
mileage_guis
punch_guis
tasks
paytypes
billtypes
expenses
expense_sources
expense_currencies
mileage_reasons
mileage_vehicles
mileage_measurements
```

Projects

A **ProjectRecord** structure has the following elements:

Element Name	Element Type	Notes
<i>id</i>	string (small)	Key. A short alphanumeric string which uniquely identifies the project.
name	string (large)	The project's name, as it will appear in the GUI.
description	string (large)	A description of the project.
parent	string (small)	The id of this project's parent project. If this project is at the root of the project tree, this field contains the string "root".
domain	string (small)	The id of the domain this project is a member of.
estimate	integer	The estimated hours necessary to complete this project.
percent_done	Integer	The percentage of this project which is complete.
creator	string (small)	The name of the user who created this project.

loggable	string (medium)	Use a “0” (string zero) to indicate that this Project is “reportable only” (users cannot create new time records with it) or any other value to indicate that it is both “loggable and reportable” (where users <i>can</i> create new time records with the Project.)
type	Integer	Use 1 for regular projects and 2 for subprojects (only if you have subproject feature in your Timesheet license.) If in doubt, always use 1 in this field, do not use 0.

Project-Related Methods

string addFullProject (*record: ProjectRecord*)

string addProject (*name: string*)

string modifyProject (*id: string, record: ProjectRecord*)

ProjectRecord getProject (*pattern: ProjectRecord*)

void removeProject (*id: string*)

void removeProject (*pattern: ProjectRecord*)

void addProjectToGroup (*id: string, group: string*)

void removeProjectFromGroup (*id: string, group: string*)

ProjectRecord getDefaultProject ()

Standard record access methods: see above for a complete description.

list getProjectList ()

Returns a list of project IDs and names, sorted in the same order as the web-based user interface. The return value is a list of strings, where each pair of elements is a project ID and a project name, for example ["p1", "Project One", "p2", "Project Two", "zzz", "Project zzz", ...] The list of projects returned is determined by which projects are “viewable” to the user logged into the jxAPI. This is regulated through group membership.

The following three methods are Project Dependency methods. Projects may have any number of code “dependencies.” If a particular Project has one or more dependencies of a particular code type (tasks, pay types, bill types, expense codes, etc) then when a user enters time against that project, that code field will be restricted to only those values listed as dependencies. So for example, you can specify that when a user enters time against project ID ‘1234’, only the Task codes “Sales” and “Management” will be available.

All of these methods take a *code_type* string parameter. This is a string which names which type of code you wish to use. This is the list of code type strings known to the Project Dependency methods: (those in parenthesis are equivalent synonyms.)

- code (task)
- subcode (paytype)
- subsubcode (billtype)
- expense
- expense_source
- expense_currency
- mileage_reason
- mileage_vehicle
- mileage_measurement

list (or dictionary) getProjectDependencies (*id: string, code_type: string [optional]*)

This method may be used to find out what dependencies a particular project (identified by its unique *id*) has, if any.

If you give it a code type string parameter (see above for a list) you will only get back dependencies of that code type, as a list of code ID’s (strings.) But if you leave off the *code_type* parameter, you will get back a dictionary that contains *all* the dependencies for the project. The dictionary will map the code type strings to a list of code ID’s.

In other words, call this *with* a `code_type` to get only the dependencies for that type, and call it *without* `code_type` to get *all* the project's code dependencies.

void addProjectDependency (*id*: **string**, *code_type*: **string**, *code_id*: **string**)

Adds the given `code_id` to Project `id`'s dependencies. This method will throw an exception if the `code_id` or project `id` does not already exist, or if the `code_type` is unknown. (See above for a list of valid `code_types`.)

void removeProjectDependency (*id*: **string**, *code_type*: **string**, *code_id*: **string**)

Similar to addProjectDependency except that it removes the dependency.

Codes (Tasks), Subcodes (Pay Types), and Sub-Subcodes (Bill Types)

Codes, subcodes, and sub-subcodes are used to subdivide reported time within a project. They are implemented identically to one another; the **CodeRecord**, **SubcodeRecord**, and **SubsubcodeRecord** structs all have the same elements:

Element Name	Element Type	Notes
<code>id</code>	string (small)	Key. A short alphanumeric string which uniquely identifies this code.
<code>description</code>	string (large)	A description of this code.
<code>autoadd</code>	Integer	set to 1 to have this code automatically added to new groups. Otherwise, set it to 0.
<code>domain</code>	string (medium)	The id of the domain this code is a member of.
<code>pretty_name</code>	string (small)	A printable, human-friendly name for this code for use in the GUI.

Code-Related Methods

string addFullCode (*record*: **CodeRecord**)

string addCode (*name*: **string**)

string modifyCode (*id*: **string**, *record*: **CodeRecord**)

CodeRecord getCode (*pattern*: **CodeRecord**)

void removeCode (*id*: **string**)

void removeCode (*pattern*: **CodeRecord**)

void addCodeToGroup (*id*: **string**, *group*: **string**)

void removeCodeFromGroup (*id*: **string**, *group*: **string**)

CodeRecord getDefaultCode ()

Standard record access methods: see above for a complete description.

list getCodeList ()

Returns a list of project IDs and names, sorted in the same order as the web-based user interface. The return value is a list of strings, where each pair of elements is a code ID and a code name, for example ["c1", "Code One", "c2", "Code Two", "zzz", "Code zzz", ...]. The viewability of codes is determined by the user's group membership.

Subcode-Related Methods

- string** addFullSubcode (*record: SubcodeRecord*)
- string** addSubcode (*name: string*)
- string** modifySubcode (*id: string, record: SubcodeRecord*)
- SubcodeRecord** getSubcode (*pattern: SubcodeRecord*)
- void** removeSubcode (*id: string*)
- void** removeSubcode (*pattern: SubcodeRecord*)
- void** addSubcodeToGroup (*id: string, group: string*)
- void** removeSubcodeFromGroup (*id: string, group: string*)
- SubcodeRecord** getDefaultSubcode ()

Standard record access methods: see above for a complete description.

list getSubcodeList ()

Returns a list of project IDs and names, sorted in the same order as the web-based user interface. The return value is a list of strings, where each pair of elements is a subcode ID and a subcode name, for example ["s1", "Subcode One", "s2", "Subcode Two", "zzz", "Subcode zzz", ...]

Subsubcode-Related Methods

- string** addFullSubsubcode (*record: SubsubcodeRecord*)
- string** addSubsubcode (*name: string*)
- string** modifySubsubcode (*id: string, record: SubsubcodeRecord*)
- SubsubcodeRecord** getSubsubcode (*pattern: SubsubcodeRecord*)
- void** removeSubsubcode (*id: string*)
- void** removeSubsubcode (*pattern: SubsubcodeRecord*)
- void** addSubsubcodeToGroup (*id: string, group: string*)
- void** removeSubsubcodeFromGroup (*id: string, group: string*)
- SubsubcodeRecord** getDefaultSubsubcode ()

Standard record access methods: see above for a complete description.

list getSubsubcodeList ()

Returns a list of subsubcode IDs and names, sorted in the same order as the web-based user interface. The return value is a list of strings, where each pair of elements is a subsubcode ID and a subsubcode name, for example ["ss1", "Subsubcode One", "ss2", "Subsubcode Two", "zzz", "Subsubcode zzz", ...]

Expense Code-Related Methods

The `expense_code`, `currency`, and `source` fields of an expense record are analogous to the code, subcode, and subsubcode of a time record. In a future version of this API functions will be added to manage them in the same way that codes, subcodes, and subsubcodes are. At this time, the following functions are available to work with them:

- list** getExpenseCodeList()
- list** getExpenseCurrencyList()
- list** getExpenseSourceList()

Returns a list of expense codes, currencies, or expense sources respectively, sorted in the same order as the web-based user interface. The return value is a list of strings, where each pair of elements is the ID and name of an item, for example ["US\$", "US Dollars", "C\$", "Canadian Dollars", "Y", "Yen", ...]

Time Records

Each time record describes the hours reported by a specific user, on a specific date, for a single combination of project, code, subcode, and subsubcode. The **TimeRecord** struct contains these elements:

Element Name	Element Type	Description
--------------	--------------	-------------

<i>id</i>	string (small)	Key. A short alphanumeric string which uniquely identifies this time record.
<i>user</i>	string (small)	The id of the user who reported this time.
<i>date</i>	string (small)	The date on which the reported activity occurred.
<i>code</i>	string (small)	The id of the code to which this time entry is assigned.
<i>subcode</i>	string (small)	The id of the subcode to which this time entry is assigned.
<i>subsubcode</i>	string (small)	The id of the subsubcode to which this time entry is assigned.
<i>project</i>	string (small)	The id of the project to which this time entry is assigned.
<i>group</i>	string (medium)	The id of the group to which this time entry is assigned.
<i>hours</i>	float	The number of hours in this time entry.
<i>comment</i>	string (large)	A comment about this time entry.
<i>committed</i>	Integer	Bit-flags indicating committed status of record. See note below.

Time Record-Related Methods

string addFullTimeRecord (*record*: TimeRecord)

string addTimeRecord (*comment*: string)

This method creates a default 1 hour time record for the current user with the comment field set to the given parameter. The ID of the new time record is returned. The codes used are the default codes for the user's current Entry Screen.

string modifyTimeRecord (*id*: string, *record*: TimeRecord)

TimeRecord getTimeRecord (*pattern*: TimeRecord)

void removeTimeRecord (*id*: string)

void removeTimeRecord (*pattern*: TimeRecord)

TimeRecord getDefaultTimeRecord ()

Standard record access methods: see above for a complete description.

list getTimeList (*date*: string)

This function returns a list of all of the time records for the current user on the specified date. The return value is not a list of structs, but a list of strings which contain the following comma-delimited fields: ID, committed flag (the string "true" or "false"), project name, hours (in hh:mm format), code, subcode, and subsubcode.

Please note: this method is deprecated and may not be present in future releases. Please use the `getRecordsList` method instead. For instance, here is an example (in Python) to retrieve all time records for the user "joe" on the date April 1st, 2002:

```
time_rec_search_params = { "date": "20020401", "user": "joe" }
time_records = Session.getRecordsList("time_recs", time_rec_search_params)
```

list getEnabledTimeFields ()

Returns a list of the supported time record fields for the current user. The return value is a list of strings where each pair of elements is the field name and printable name of a field that is enabled for the current user, in the order those fields appear on the user's time entry page. (Of the fields described above, the only configurable ones are `project`, `code`, `subcode`, `subsubcode`, and `comment`; the others are mandatory and will not be listed in the return from this function.)

Special Note about the 'Committed' field

The committed field is somewhat complex to understand and dangerous to modify. The field is stored in the database as an integer, and it is used by the application as a bit-flag field, but a few jxAPI functions represent it as a true or false value (Boolean.) In general, a 0 (zero) or false value means that the record has *not* be committed to the approvals process yet. Positive (non-zero) or true values generally mean that the

record is somewhere in the approvals process, or possibly that the record has been exported to an external system, depending on your site configuration.

Journyx does not recommend adjusting the committed field of any records through the jxAPI at this time. Easier to use methods will be provided in the next release.

Expense Records

Expense records are similar to time records, in that each one records a single expenditure by a single user, for a specific date, project, and code. The **ExpenseRecord** struct has the following members:

Element Name	Element Type	Notes
<i>id</i>	string (small)	Key. A short alphanumeric string which uniquely identifies this expense record.
<i>user</i>	string (small)	The id of the user who reported this expense.
<i>date</i>	string (small)	The date on which the reported expense occurred.
<i>project</i>	string (small)	The id of the project to which this expense is assigned.
<i>expense_code</i>	string (small)	The id of the expense code to which this expense is assigned.
<i>amount</i>	float	The amount of this expense, in the currency specified by the <i>currency</i> field.
<i>comment</i>	string (large)	A comment about this expense.
<i>committed</i>	Integer	See note above about Time Record committed field.
<i>currency</i>	string (small)	The currency type for this expense (the default of United States dollars is represented by the string "US\$".)
<i>source</i>	string (small)	The source of the expense, such as "Employee", "Company", or "Customer".

Expense Record-Related Methods

string addFullExpenseRecord (*record*: **ExpenseRecord**)

string addExpenseRecord (*comment*: **string**)

This method creates a default empty expense record (with an amount of 0.00) using the given comment. The ID of the new expense record is returned.

string modifyExpenseRecord (*id*: **string**, *record*: **ExpenseRecord**)

ExpenseRecord getExpenseRecord (*pattern*: **ExpenseRecord**)

void removeExpenseRecord (*id*: **string**)

void removeExpenseRecord (*pattern*: **ExpenseRecord**)

ExpenseRecord getDefaultExpenseRecord ()

Standard record access methods: see above for a complete description.

list getExpenseList (*date*: **string**)

This function returns a list of all of the expense records for the current user on the specified date. The return value is not a list of structs, but a list of strings which contain the following comma-delimited fields: ID, committed flag (the string "true" or "false"), project name, amount, code, subcode, and subsubcode.

Note that this function is only a temporary addition for efficiency purposes; it will be replaced by a more general query facility in a future version of this interface, once the Python SOAP implementation permits.

list getEnabledExpenseFields ()

Returns a list of the supported expense fields for the current user. The return value is a list of strings where each pair is the field name and printable name of a field that is enabled for the current user, in the order those fields appear on the user's expense entry page. (Of the fields described above, only the `project`, `expense_code`, `currency`, `source`, and `comment` fields can appear in this list; the rest are mandatory and will not appear in the return from this function.)

Time Sheets

Like a physical time sheet, a time sheet in the Journyx server represents a batch of time reports that are submitted to some higher authority for approval. A time sheet holds references to time records that already exist in the database, linked to the time sheet by their unique IDs.

Time Sheet-Related Methods

string getTimeSheetIDByDate (*date*: **string**)

Returns the ID of a timesheet covering the specified date (for the current user).

string getNextTimeSheetID (*id*: **string**)

Returns the ID of the timesheet after the one specified by the *id* parameter.

string getPreviousTimeSheetID (*id*: **string**)

Returns the ID of the timesheet before the one specified by the *id* parameter.

list getDatesInTimeSheet (*id*: **string**)

Returns a list of the dates covered by the specified timesheet.

float getTotalHoursInTimeSheet (*id*: **string**)

Returns the total hours reported in the specified timesheet.

void addTimeRecordToSheet (*id*: **string**, *trucid*: **string**)

Adds the time record with ID *trucid* to the specified timesheet.

void removeTimeRecordFromSheet (*id*: **string**, *trucid*: **string**)

Removes the time record with ID *trucid* from the timesheet specified by the first parameter. The time record itself is not deleted; only its association with that particular time sheet is removed. (The time record will still exist, but it will not be attached to that Time Sheet.)

list getTimeRecordIDsInSheet (*id*: **string**)

Returns a list of the IDs of all time records associated with the specified timesheet.

void submitTimeSheet (*id*: **string**)

Submits the specified timesheet for approval.

string getTimeSheetStatus (*id*: **string**)

Returns the status of the specified timesheet. The return value is a printable string such as "Approved" or "Rejected".

string getTimeSheetReason (*id*: **string**)

Returns a string describing why a timesheet was rejected. If the specified timesheet has not, in fact, been rejected the return value will be an empty string.

string getLatestTimeSheetID ()

Returns the ID of the most current timesheet for the current user.

list getAllTimeSheetIDs ()

Returns a list of all of the current user's timesheets.

list getRecentTimeSheetStatus (*number*: **integer**)

Returns a list of *number* recent timesheet IDs and their associated statuses, beginning with the most recent timesheet and going backwards in time. The return value is a list of strings, each of which contains five comma-delimited fields: the timesheet ID, the start and end dates of the timesheet, the status, and the reason (if any) associated with the timesheet.

list getTimeSheetRejectedStatuses ()

Returns a list of possible statuses for a rejected timesheet. The return value is a list of strings that can be compared against the timesheet status to determine if it has been rejected. (At the present time, the sole member of this list will be the string "Rejected", but this function is provided because that may change in the future, or in localized versions of the timesheet server.)

list getTimePeriodDates (*date*: **string**)

Returns a list of the dates in the time reporting period containing the specified date. *Note: This function may create a timesheet containing the specified date as a side effect.*

list getExpensePeriodDates (*date*: **string**)

Returns a list of the dates in the expense reporting period containing the specified date. *Note: This function may create a timesheet containing the specified date as a side effect.*

list ChangeSheetStatus (*sheet_recs_list*: **list** [of **SheetRecords**])

This method is used to change the approval status of any Time, Expense, or Mileage sheet.

The only parameter is a variable length List of **SheetRecords**.

Each **SheetRecord** shall be in the form of a key:value dictionary. (The format is described below.) The operation returns also returns a list of sheet records, where the status of each Sheet's change operation is noted as an item in the record dictionary. Changes are constrained by the user's authorization.

Each SheetRecord should have these keys and values:

Key: **id** Value: the database ID of the Sheet record to modify

Key: **change** Value: must be either "reject" or "approve"

Only if the `change` is `reject`, you must supply one more key/value pair:

Key: **reason** Value: (*string*) the reason for rejection that the end-user will see.

list getSheets (*sheet_ids_list*: **list** [of Sheet IDs])

This method returns a list of SheetRecords that match the given list of Sheet IDs. The IDs given in the parameter may be of any type (Time, Expense, or Mileage.) If the record associated with that ID is not found in the database, the jxAPI will silently skip it. Lists with one element are supported for the parameter. Record visibility is constrained by authorization level.

integer AssignApprovalTemplate (*template_id_or_name*: **string**, *user_id_list*: **list** [of User IDs])

Assigns an Approval Template to one more users. A `RuntimeError` exception will be raised if either the Template ID/Name cannot be found (you can supply either, see below), or if one of the users does not exist, or if the caller does not have the authority to do this. Returns a zero (0) if the assignment was successful.

Note that the first parameter can be either the unique ID of the Approval Template, or it can be the Name as it appears on the Template Modification screens. The Unique ID changes every time the administrator modifies the template, because the record is deleted and re-added, and so it is often convenient to assign templates based on their name.

Also note that Templates for Time, Expense, and Mileage can all be used with this function. But since each of these types of templates may have the same name, if you specify an ambiguous name they will be 'resolved' in the order of Time, then Expense, then Mileage. In other words, if you have both Time and Expense templates named "My Template", and call this function as `AssignApprovalTemplate("My Template", ["user1", "user2"])`, then only the Time template will get assigned to the users, not the Expense template. In that case, you must name your Expense and/or Mileage templates something unambiguous like "My Expense Template".

list getApprovalTemplates (*template_ids_list*: **list** [of Template IDs – *optional*])

This method returns a list of Approval Template records.

If called with no parameters, it returns a list of all visible Approval Template records. If one or more Template ID's are given as a list in the optional parameter, the returned records will be restricted to those named ID's. If no Approval Templates are visible, an empty list will be returned.

list getRecordsForSheet (*sheet_type*: **string**, *sheet_id*: **string**)

Given a string sheet type and ID, this returns a list of all records associated with that sheet. For example, if the *sheet_type* is 'time', then all the **TimeRecords** that belong to that sheet ID will be returned in an unordered list. The format of the Time, Expense, and Mileage Records is documented separately in this specification. The strings *time*, *expense*, and *mileage* are supported for the *sheet_type* parameter.

Note: in addition to the required parameters listed above, there is an optional parameter called *time_in_seconds*. If you leave this parameter out, or set it to 0, then the hours in the time records are returned in the normal database format (hours plus fractional hours.) However, if you set *time_in_seconds* to 1 (or another 'true' value) then each Hours column in the time records will be silently converted to pure integer seconds. Note however that the field itself will still be named *hours* even though the time is denoted in seconds. For instance, if a particular time amount is normally returned as "1.0166666667" (representing 1 hour and 1 minute, because 0.0166.. is 1/60th of an hour, or 1 minute) if *time_in_seconds* is set, then this time will be returned as 3660. Due to the nature of the time entry format, precision is limited to 1/100th of an hour. This parameter does not currently apply to any other method dealing with time records.

integer CreateApprovalTemplate (*new_template*: **ApprovalTemplateRecord**)

This method creates a new Approval Template for the approvals process and returns 0 (zero) to indicate success. This method raises an exception if the user does not have the authority to add an Approval Template, or if the record format is invalid.

ApprovalTemplateRecords must have these keys and values:

type – either *time*, *expense*, or *mileage*
bizid – the business process ID which defines the approvers structure (how many approvers there are.) Use one of these constants, which correspond to the number of approvers: *default1*, *default2*, *default3*, *default4*, *default5*.
name – the name of the Approval Template.
Userbefore – number of days before the end of a period to send a reminder email to the user.
Userafter – number of days after the end of a period before a reminder is sent to the user (if sheet has not been submitted)
Approverafter – number of days after the end of a period before a notice is sent to the approver (if they user has not submitted a sheet.)
Backupafter – number of days after the end of a period to notify the backup approver if a sheet has been submitted but not approved.
Userrejected – a string to send to the user when his sheet is rejected.
Userapproved – a string to send to the user when his sheet is approved.
Approversubmitted – a string to send to the approver when a sheet is in his approval queue.

In addition to the above keys, you must add all the approvers to the Template when you create it, as there is no other way to modify an Approval Template through the jxAPI once created. (However, you can modify the Approval Template through the normal web interface.) Backup approvers are optional but you must add a primary approver key (*approver_1* for instance) for each "level" of approval.

For instance, if you gave "default2" for the *bizid* field, this indicates a 2-level approval process. This means that you must give 2 primary approvers, and 0 to 2 backup approvers.

The valid keys for Primary Approvers and Backup approvers are:

```
approver_1
backup_approver_1
approver_2
backup_approver_2
approver_3
backup_approver_3
approver_4
backup_approver_4
approver_5
backup_approver_5
```

Again, the number of primary approvers must correspond exactly to the number of approval levels indicated by the `bizid` field. It is not possible to create an Approval Template through the jxAPI without specifying all the primary approvers.

You will get a `ValueError` exception if you are missing a primary approver.

integer `UnassignApprovalTemplate` (*template_id*: **string**, *user_id_list*: **list** [of User IDs])

Given a string Template ID, and a list of one or more user Ids, this removes the assignment of the template from the user(s).

An exception is raised if the user does not have the authority to do this, or if the Template ID supplied is not valid, or if any of the user ID's are not valid.

Returns a zero to indicate that the template was unassigned from all given users. An exception is *not* raised if a user was not previously assigned to the given Template ID.

Extra Field (Attribute) Methods

Journyx Timesheet 5.0 introduced a new API for managing user-configurable object attributes (extra data fields) that replaces the previous Extra Fields implementation. The main technical difference is that, in the old system, each object had only **one row** in the database for “extra fields,” and adding a new field meant adding a new column to the table. In the new system, each “extra field” (attribute) for every object is its own record for increased flexibility. Also, the new extra fields are fully manageable through the jxAPI. The old system was not supported at all in the jxAPI.

Each Attribute value record contains references to the object Class and ID, the ID of the Attribute Type, and the value itself. The Attribute Type describes the “extra field”, indicating its name, description, data type (numeric, string, date, etc) and possibly a default value.

In the rest of this text, the terms “Extra Field” and “Attributes” are used interchangeably.

So, to assign a value to an “extra field,” you must know four pieces of information. You must have:

1. The class (type) of Object you are working with, such as a user, a project, a group, and so on. Each of these classes must be referred to be a specific name; see the description of the [getAttributeObjectTypes](#) method below.
2. The ID of the Object itself. Every object has a unique ID, usually the first column in the associated database table. The ID is in most cases *not* the same as the name of the object as seen by the user.
3. The Name or ID of the Attribute Type, such as “Pay Type” or “Email Address.” If you only have the name of the Attribute Type, you must look up the correct ID using the `getAttributeTypeByName` method, as most of the Extra Field methods require the `id_attr_type` parameter, which must be the unique ID of an Attribute Type and *not* a name like “Pay Type.”
4. The actual value you wish to assign. The data type of the value must correspond to the data type of Attribute Type. For instance, values for the “Pay Type” Project extra field must be numbers.

An Example

Let’s say that we have a Project named “My Project”. We know that the class of Projects is simply `projects`. We happen to know that the ID of “My Project” is (say) 12345. If we only knew the name of the Project and not the ID, we would have to look that up as well. Let’s say that we want to assign a special “Tax Code” to each Project, so we have created a string Attribute Type named “Tax Code”, and that Type has an ID of (say) ABCDE.

So, to assign the “Tax Code” value of “Deductible” to “My Project”, we have the four key pieces of information we need to make the `setAttribute` method call:

```
Session.setAttribute('projects', '12345', 'ABCDE', 'Deductible')
```

To retrieve the “Tax Code” of “My Project” at a later time, we make this call:

```
my_project_tax_code = Session.getAttribute('project', '12345', 'ABCDE')
print my_project_tax_code # prints "Deductible"
```

Only these Object Classes are currently supported in either the jxAPI or the Timesheet product itself: `projects`, `users`, and `groups`. These are the actual names you use, but the authoritative list of names can always be seen by calling the `getAttributeObjectTypes()` method.

Historic Value Tracking

In addition to the object classes listed above, these classes can have “Historic” extra field tracking.

`time_recs` (regular Time Entry record)
`expense_recs` (Expense Entry record)
`travel_recs` (Mileage / Travel Entry records)

Every Attribute Type can be designated as “historic” for time, expense, or mileage, which means that when a new time/expense/mileage record is created, the value of the Attribute at the time when the record was created is saved for future reference. Historic attributes for Time, Expense, and Mileage only track the values of the “regular” Attribute Types at the time they were created, and are unaffected by changing the original values later. This is why they are called “Historic.” You cannot attach independent extra fields to Time, Expense, or Mileage; you can only track historical values of the other extra fields, such as for Projects and Users.

For example, let’s say that the user Jane is entering some Time Records, and Jane’s `userid` is `janesmith`. The user object `janesmith` has an `Pay Rate` extra field value of 25, because this is Jane’s hourly pay rate. If the Timesheet administrator designates the User `Pay Rate` extra field “historical” for Time entries, then every time Jane creates a time record, the `Pay Rate` of 25 is recorded with each time record, independently from the actual `Pay Rate` entry for `janesmith`.

Now, let’s suppose that Jane gets a promotion, and a subsequent pay rate of 30 per hour. (The `Pay Rate` extra field for the user object `janesmith` is changed from 25 to 30.) Jane goes and enters some new time records after her promotion, and these new records will have the new `Pay Rate` of 30. However, all the time records she entered *before* her promotion will still have the old `Pay Rate` of 25, even if those time records are modified in some way, such as if the comment was changed. This is useful for reporting and other analytical purposes.

Extra Field Selection Lists

Extra Fields (Attribute Types) can be designated as “Selection List” or enumerated types. This means that users will be presented with a pre-defined list of acceptable values for the Extra Field, instead of being able to enter any arbitrary values. Multiple selection lists are not supported at this time; only one “choice” per extra field is allowed.

Selection lists may have a default value that is always presented at the top of the list (as the default choice). The default value of selection lists is always assigned to new objects when they are created.

An Attribute Type is designated as a Selection List by creating the type with the `ENUM_` prefix for its data-type. (See the [Data Types](#) section below.) The actual values in the selection list are managed using the `addAttributeTypeSelectionValue` and `deleteAttributeTypeSelectionValue` methods.

Extra Fields Data Types

Each Attribute Type (or Extra Field definition) must have a designated “data type.” This determines what kinds of values are acceptable for the field. In the current implementation of this feature, you must supply a string that describes a valid data-type when you create the Attribute Type. You cannot modify the data-type of an Extra Field once it has been defined. (If you wanted to do that, you would have to define a new Extra Field, migrate the values over, and then delete the old Extra Field.)

The data-type is designated by the `attr_type` field of the `AttributeType` record, or the `data_type` parameter to the `addAttributeType` method. A valid `data_type` string is made of up the “Base Type” plus any modifying suffixes or prefixes.

There are five different “Base Types” supported in Journyx Timesheet 5.6:

- **NUMBER** – use for all ‘real’ (floating point) numbers. The range of numbers is the range supported by your database’s “double precision” type, which is usually the highest precision available.
- **INTEGER** – use this for purely integer (whole) numbers. The range of integers is also dependent on your database’s “integer” type, which usually allocates 4 bytes. This provides an effective range of -2147483648 to +2147483647
- **STRING** – use this for variable length text. If you do not specify a maximum length suffix, then the maximum length is always a “big string” (252 characters, not including [escapes](#).) **STRING** values are returned “stripped” with any leading or trailing spaces removed. The only character set that is supported by Journyx for **STRING** and **CHAR** types is standard 7-bit US ASCII text. Do not attempt to specify Unicode or other international characters out of this range, as they may not be stored in the database or returned properly.
- **CHAR** – use this for a *fixed-length* text string. **CHAR** (character) types *must* be specified with the maximum length suffix, and **CHAR** values will be returned “unstripped” with any trailing spaces out to the maximum length.
- **DATE** – this is a special type that *only* stores dates (not times) in a specific format: always YYYYMMDD. For example, April 17th, 2002 would be specified as 20020417. To store exact times, you will have to use either a **STRING** or **NUMBER** type and define your own time format.

In addition to the “base type”, you may append a length suffix to (only) the **STRING** and **CHAR** types. For instance, a string with a maximum length of 30 characters would be the **STRING_30** data type. With **CHAR** types, the length suffix *must* be included. Note the difference between **STRING** and **CHAR** types: if you have a **CHAR_30** type, the values returned are always exactly 30 characters, padded with extra spaces if need be. **STRING** types are returned with any leading or trailing spaces stripped off.

In indicate that the Extra Field should be a selection list, prepend the **ENUM_** prefix to the base type (along with any desired length suffix.) For instance, to make a selection list of strings, you can use the **ENUM_STRING** data-type. To make a selection list of strings with a maximum length of 25 characters, you could use **ENUM_STRING_25** for the data-type. Once you have created an **ENUM_** type, you can use the `addAttributeTypeSelectionValue` and `deleteAttributeTypeSelectionValue` methods to manage the values. Of course, `addAttributeTypeSelectionValue` will throw an exception if the value does not conform to the data-type of the Extra Field when you created it.

To avoid this problem, you can verify that your values match your data-types ahead of time with the handy `checkAttributeValue(data_type, value)` method. You can also verify that any particular string is a valid data-type field with the `checkAttributeDataType(data_type)` method.

Extra Fields may also be marked with a `visibility` attribute that determines whether the Extra Field is visible, hidden, or read-only. Currently these constants are used for the `visibility` field:

- 0 is Hidden. Hidden extra fields still exist in the database (and are usable through the jxAPI) but will not appear on any end-user GUI screens.
- 1 is Normal (Editable)
- 2 is Read-only. The type will appear on the GUI screens but the value will be read-only. You can still change the value through the jxAPI, so this is not rock-solid protection.

Extra Field Value Methods

`scalar getAttribute(object_type: string, object_id: string, id_attr_type: string)`

This method is used to fetch the value of an Extra Field (specified by the `id_attr_type`) for the given object type and ID.

Param 1: `object_type` – (**string**) must be in the list of valid Object Types as returned by `getAttributeObjectTypes`.

Param 2: `object_id` – (**string**) the Journyx database ID of the Object that we are interested in. It should belong to the object class (table) indicated by the first parameter.

Param 3: `id_attr_type` – (**string**) the ID of the Attribute Type that we're interested in.

RETURN VALUE

The value itself is returned, either as a string or as a number (depending on the data format of the Attribute Type.)

If the `object_id` and `id_attr_type` are valid, but that object does not happen to have an attribute set for that Attribute Type ID, then `None` (the null value) will be returned.

EXCEPTIONS

`RuntimeError` is raised if the Object Type is invalid, a `ValueError` is raised if the `object_id` or `id_attr_type` do not exist.

void `setAttribute` (*object_type*: **string**, *object_id*: **string**, *id_attr_type*: **string**, *value*: **scalar**, *overwrite_existing*: **bool** [optional – defaults to true])

This is the method you use to set an Extra Field value for some Object.

PARAMETERS

Param 1: `object_type` – (**string**) must be in the list of valid Object Types as returned by `getAttributeObjectTypes`.

Param 2: `object_id` – (**string**) the Journyx database ID of the Object that we are setting a value for. It should belong to the object class (table) indicated by the first parameter.

Param 3: `id_attr_type` – (**string**) the ID of the Attribute Type that we are going to set.

Param 4: `value` – (**string** or **number**) the actual value to set. The type should correspond to the data-type of the Attribute Type record.

Param 5: `overwrite_existing` – (**boolean**, defaults to 1/true) if True, any existing value for this ObjectID/AttributeTypeID combination will be silently overwritten with the new value. If False, and a value already exists, an exception will be raised to notify you (and no changes are made.)

RETURN VALUE / EXCEPTIONS

If setting the attribute was successful, then `None` is returned, otherwise an exception is raised in these circumstances:

- Object Type is invalid (`RuntimeError`)
- The Object ID does not exist (`ValueError`)
- The Attribute Type ID does not exist (`ValueError`)
- The value is inappropriate for the Attribute Type (`ValueError`.) For instance, a string was given when a number is required.
- If `overwrite_existing` is false, and a value already exists, a `ValueError` will be raised.

void `deleteAttribute` (*object_type*: **string**, *object_id*: **string**, *id_attr_type*: **string**)

This method is used to delete an Extra Field value (without specifying a new value.) This method causes the specified attribute value (the value at the 'address' of a specific Object Class and ID and an Attribute

Type ID) to be 'unset' or deleted. Subsequent calls to `getAttribute()` with the same parameters would then return `None`.

PARAMETERS

Param 1: `object_type` – (**string**) must be in the list of valid Object Types as returned by `getAttributeObjectTypes`.

Param 2: `object_id` – (**string**) the Journyx database ID of the Object that we are deleting an attribute value for. It should belong to the object class (table) indicated by the first parameter.

Param 3: `id_attr_type` – (**string**) the ID of the Attribute Type that we are to delete the values of.

RETURN VALUE / EXCEPTIONS

If deleting the attribute value was successful, then `None` is returned, otherwise an exception is raised in these circumstances:

- Object Type is invalid (`RuntimeError`)
- The Object ID does not exist (`ValueError`)
- The Attribute Type ID does not exist (`ValueError`)
- The Attribute Value cannot be deleted (due to a permissions error or other reasons)

If the Object ID and Attribute Type ID were valid, but no value was previously set, then `None` is returned (no exception is raised.)

list `queryAttributes` (*`object_type`: string, `object_id_list`: list [of Object IDs], `id_attr_type_list`: list [optional – list of Attribute Type IDs]*)

This method is generic way to query the Attribute table of this Object Type to get attribute values for a range of different Object IDs or Attribute Type IDs.

PARAMETERS

Param 1: `object_type` – (**string**) must be in the list of valid Object Types as returned by `getAttributeObjectTypes`.

Param 2: `object_id_list` – (**list of strings**) This array should contain the Journyx database IDs of the Objects whose attributes you wish to examine. Each Object ID should belong to same the object class (table) indicated by the first parameter. All Objects may be searched by providing a null or empty list.

Param 3: `id_attr_type` – (optional; **list of strings**) This array should contain the Attribute Type IDs to restrict the result set. If the list is empty or the parameter is Null or not provided, values for **all** attribute types (for this Object Type) will be returned.

RETURN VALUE

The return value of this method is a **list** of Attribute Arrays. Each Attribute Array in the return array has this format:

```
( Object ID, Attribute Type ID, Attribute Type PName, Value)
```

Note that the Value may be either String or Numeric depending on the data-type of the particular AttributeType record.

If no records are found that match, an empty list will be returned.

EXCEPTIONS

If the `object_type` is invalid, a `RuntimeError` will be raised.

`TypeError` is raised if the `object_id_list` and `id_attr_type_list` parameters are not in the proper array-of-strings format.

Extra Field Management Methods

list `queryAttributeTypes` (*search_pattern*: **AttributeTypeRecord**)

This method allows you to search the entire table of Attribute Types, searching on any of the fields.

PARAMETERS

Param 1: `search_pattern` – a SOAP Structure (or dictionary) of an Attribute Type record. The results that are returned by this method will be limited by the fields in this search record parameter. Passing an empty Structure will result in all Attribute Type records in the product (across all domains)! Likewise, not specifying the `id_domain` column in your search record will include all domains in the search.

RETURN VALUE

The return value of this method is an Array of Structures (**AttributeType** records) of attribute (Extra Field) types that match your search criteria. If no records match, an empty Array is returned.

EXCEPTIONS

This method raises `TypeError` when the search record parameter is not in the correct format (a Structure or dictionary / hash) or one of the keys is incorrect (not a field defined in the `AttributeTypeRecord` definition.)

EXAMPLE

This Python client example returns all 'editable' Project-related Attribute Types in the domain specified in the variable `my_domain_id`. (The 'visibility' codes are documented separately.)

```
search_rec = { 'table_name': 'projects',
               'id_domain': my_domain_id,
               'visibility': 1,}

results = Session.queryAttributeTypes(search_rec)

for result in results:
    print 'ID: ', result['id_attr_type']
```

string `getAttributeTypeByName` (*object_type*: **string**, *type_name*: **string**, *full_record*: **boolean** [optional – default is **false**])

Use this method to discover the appropriate Attribute Type ID for the class of objects you are interested in and for your domain.

'Hidden' names are set for only the built-in Attribute Types that ship with the Journyx Timesheet product. These names are also searched if no results can be found by searching the 'display names.' This feature allows you to change the name of the Journyx built-in Extra Fields, but this method will still find the correct Fields using the original (hidden) name.

PARAMETERS

Param 1: `object_type` – (**string**) must be in the list of valid Object Types as returned by `getAttributeObjectTypes`.

Param 2: `type_name` – (**string**) the display (or hidden) name of the Attribute Type to search for.

Param 3: `full_record` – (*optional*, **boolean**) set to true to return the full AttributeType record (as a SOAP Struct) instead of just the default string record ID.

RETURN VALUE

If no record is found, `None` (the null value) is returned.

If `full_record` is true, the entire AttributeType record is returned as a structure.

If `full_record` is false (the default) only the AttributeType ID is returned (as a string.)

EXCEPTIONS

If the `object_type` is invalid, a `RuntimeError` will be raised. Only Object Types returned by `getAttributeObjectTypes` are valid.

EXAMPLES

For instance, imagine that the Timesheet administrator has renamed the built-in Project attribute 'Budget' to 'Approved Budget'. The hidden name of this record is still 'Budget' so when you call this method like so:

```
budget_attr_id = Session.getAttributeTypeByName('projects', 'Budget')
```

... it will return the correct (built-in) Attribute Type ID for the Budget type, whatever that may be.

list `getAttributeObjectTypes ()`

Returns a list of strings indicating Object Types that are valid and legal to use with all Attribute (Extra Field) jxAPI methods (as the `table_name` parameter to most methods.)

`void deleteAttributeType (id_attr_type: string)`

This method deletes not only the Attribute Type record, but **all** attributes of that type, including any historical time/expense/travel attributes.

WARNING

Since this causes a lot of records to be deleted, it should be used with extreme caution. Only users with administrator privileges may run this command. Without a recent database backup, it is impossible to recover Types (and Values) once they are deleted.

PARAMETERS

Param 1: `id_attr_type` (**string**) – The ID of the Attribute Type that you wish to delete.

RETURN VALUE

Returns `None` if successful.

EXCEPTIONS

If you lack the requisite privileges (admin ability) `RuntimeError` will be raised.

If the Attribute Type ID you provide does not exist, `ValueError` will be raised.

string `addAttributeType (object_type: string, pname: string, data_type: string)`

This adds a new Attribute Type (Extra Field definition) with the given name to the current domain. Default values will be supplied (i.e., it will default to a regular String attribute.)

PARAMETERS

Param 1: `object_type` – (**string**) a valid table name as returned by `getAttributeObjectTypes`

Param 2: `pname` – (**string**) the display name of the new attribute type.

Param 3: `data_type` – (**string**) this describes what type of data this Attribute Type is allowed to hold, be it a string of characters, a date, or a number. See the [Extra Fields Data Types](#) section for more information about data-types.

RETURN VALUE

The ID of the newly created record is returned (as a **string**) if successful.

EXCEPTIONS

A `RuntimeError` is raised if the `object_type` is not valid.

A `TypeError` is raised if the `data_type` parameter is not a valid data-type description for Attribute Types. See the `checkAttributeDataType` method.

integer `checkAttributeValue` (*data_type: string, value: scalar*)

Use this method to determine if an attribute Value is valid for an Attribute Type's data-type, or to determine if the data-type itself is valid by checking for `TypeError`.

PARAMETERS

Param 1: `data_type` – (**string**) the data-type of the Attribute Type, for example `'ENUM_STRING_20'`. NOTE: you can supply an Attribute Type ID instead of just the data-type field, and the data-type will be looked up for you

Param 2: `value` – (**string or number**) the value that you wish to test for conformance.

RETURN VALUE

This method returns the integer 1 if the value is acceptable for the data-type.

EXCEPTIONS

A `ValueError` is raised if the value is not acceptable for the data-type.

If the data-type parameter is neither a proper data-type nor an existing Attribute Type ID, then a `TypeError` exception is raised.

integer `checkAttributeDataType` (*data_type: string*)

Use this method to determine if a particular string is a valid data-type for `AttributeType` records. (Here, data-type is a synonym for the `attr_type` field of `AttributeType` records.)

PARAMETERS

Param 1: `data_type` – (**string**) the string that you wish to check.

RETURN VALUE / EXCEPTIONS

This method returns the integer 1 if the given string is a valid data-type for `AttributeType` records.

If it is not a valid data-type, a `TypeError` exception will be raised.

void `modifyAttributeName` (*id_attr_type: string, new_name: string*)

This changes the display name (not the ID or hidden name) of the given Attribute Type.

PARAMETERS

Param 1: `id_attr_type` – (**string**) the ID of the Attribute Type we are changing the name of.

Param 2: `new_name` – (**string**) the new name of the attribute type. This is the name that end-users see, which is different than the ID or any original (hidden) name (for built-ins only)

RETURN VALUE

Returns `None` if the name change was successful.

EXCEPTIONS

A `ValueError` will be raised if the Attribute Type ID does not exist, or the new name is longer than 30 characters.

void `modifyAttributeTypeDescription` (*id_attr_type*: **string**, *new_description*: **string**)

Similar to `modifyAttributeTypeName`, this changes the text description associated with the Extra Field (Attribute Type) definition.

PARAMETERS

Param 1: `id_attr_type` – (**string**) the ID of the Attribute Type we are changing the name of.

Param 2: `new_description` – (**string**) the new description of the attribute type.

RETURN VALUE

Returns `None` if the description change was successful.

EXCEPTIONS

A `ValueError` will be raised if the Attribute Type ID does not exist, or if the new description is longer than 252 characters.

scalar `getAttributeTypeDefaultValue` (*id_attr_type*: **string**)

Returns the default value associated with the Extra Field, if any.

PARAMETERS

Param 1: `id_attr_type` – (**string**) the ID of the Attribute Type (Extra Field.)

RETURN VALUE

Returns the default value (which can be a string or a number) or `None` if there is no default value set.

EXCEPTIONS

A `ValueError` will be raised if the Attribute Type ID does not exist. If there is no default value, `None` is returned.

void `setAttributeTypeDefaultValue` (*id_attr_type*: **string**, *default_value*: **scalar**)

Makes the given value the default for that Attribute Type (Extra Field.) See the general Extra Fields [introduction](#) for more about default values. In brief, when any new Objects (users, projects, etc) are created, if any relevant Attributes Types have a default value, they will get that value as an attribute. Setting a default value does not apply the value to any existing attributes.

Note: This method is irrelevant for Selection List types (those with 'ENUM_' data types.) Use `addAttributeTypesSelectionValue` instead, with the `is_default` parameter set to 1.

PARAMETERS

Param 1: `id_attr_type` – (**string**) the ID of the Attribute Type

Param 2: `default_value` – (**string** or **number**) the new default value, which will overwrite any existing default.

RETURN VALUE

Returns `None` if successful.

EXCEPTIONS

A `ValueError` is raised if the Attribute Type ID does not exist, or if the `default_value` is not appropriate for the data-type, or if the Attribute Type is a selection list. (See the note above.)

void `removeAttributeTypeDefaultValue` (*id_attr_type*: **string**)

Removes the default value for an Attribute Type (sets it to null.) New Objects of that object type will no longer get any default value for this Attribute Type.

PARAMETERS

Param 1: `id_attr_type` – (**string**) the ID of the Attribute Type

RETURN VALUE

Returns `None` if successful.

EXCEPTIONS

A `ValueError` will be raised if the Attribute Type ID does not exist. If there is no default value, `None` is returned.

list `getAttributeTypeReportability`(*id_attr_type*: **string**)

Returns a list of role Ids that can report on the specified extra field (attribute type.)

PARAMETERS

Param 1: `id_attr_type` – (**string**) the ID of the Attribute Type

RETURN VALUE

Returns a list of Role Ids that can report on the specified extra field. May return an empty list if none can report on it.

EXCEPTIONS

A `ValueError` will be raised if the Attribute Type ID does not exist.

void `modifyAttributeTypeReportability`(*id_attr_type*: **string**, *role_list*: **list**)

Change the list of Roles that may report on the given Extra Field (attribute type.) The previous list is forgotten; only the members of the given list will be able to report on the extra field.

PARAMETERS

Param 1: `id_attr_type` – (**string**) the ID of the Attribute Type

Param 2: `role_list` (list) a list of Role Ids

RETURN VALUE

Returns `None` if successful.

EXCEPTIONS

A `ValueError` will be raised if the Attribute Type ID does not exist. A `TypeError` will be raised if `role_list` (the second parameter) is not a sequence of strings.

Historic Extra Fields Methods

list `getHistoricalAttributeObjectTypes ()`

Returns a list of strings indicating Object Types that are valid and legal to use as “Historical” attributes. See the Extra Fields [introduction](#) for more information about Historical attributes. Currently, the list returned by this method is [`'time_recs'`, `'expense_recs'`, `'travel_recs'`].

list `getHistoricalAttributeTypes(object_type: string)`

Returns a list of Attribute Type IDs that are currently active as 'historical record attributes' for the given object type across all domains.

If you need to filter the Attribute Type IDs by domain (or any other value) you will have to do it as post-processing to this method, by looking up each Attribute Type ID that is returned and deciding if it should be filtered (based on the domain or other criteria.)

See the jxAPI Attribute Types [documentation](#) for a general explanation of Historical Attributes.

PARAMETERS

Param 1: `object_type` – (**string**) a valid table name for Historical attribute types (as defined by the method `getHistoricalAttributeObjectTypes`.)

EXCEPTIONS

`RuntimeError` is raised if the given `object_type` is not a valid object-class for Historical Attributes.

void `makeAttributeTypeAsHistorical(object_type: string, id_attr_type: string)`

This method makes an Attribute Type (indicated by the second parameter) 'historical' for the `object_type` (either time, expense, or travel/mileage.) `None` is normally returned. See the general Attribute Type [documentation](#) for a more detailed explanation of what it means to be a Historical attribute type.

PARAMETERS

Param 1: `object_type` – (**string**) a valid table name for Historical attribute types (as defined by the method `getHistoricalAttributeObjectTypes`.)

Param 2: `id_attr_type` – (**string**) the ID of an Attribute Type

RETURN VALUE

Returns `None` if successful.

EXCEPTIONS

These exceptions are raised:

- `ValueError` if `object_type` is not in the allowed list.
- `ValueError` if `id_attr_type` is not a valid Attribute Type

- `RuntimeError` if `id_attr_type` is already historical for this `object_type`.

void `dropAttributeTypeAsHistorical(object_type: string, id_attr_type: string)`

This method drops a Historical Attribute Type. The Attribute Type record isn't deleted, only it is no longer tracked as a historical attribute. The actual values for historical attribute-types are not deleted either. This method only prevents future Time/Expense/Travel records from recording the historical value of the given Attribute Type. See the general Attribute Type [documentation](#) for a more detailed explanation of what it means to be a Historical attribute type.

PARAMETERS

Param 1: `object_type` – (**string**) a valid table name for Historical attribute types (as defined by the method `getHistoricalAttributeObjectTypes`.)

Param 2: `id_attr_type` – (**string**) the ID of an Attribute Type

RETURN VALUE

Returns `None` if successful.

EXCEPTIONS

These exceptions are raised:

- `ValueError` if `object_type` is not in the allowed list.
- `ValueError` if `id_attr_type` is not a valid Attribute Type
- `RuntimeError` if `id_attr_type` is not currently historical for this `object_type`.

Selection List Extra Field Methods

string `addAttributeTypeSelectionValue(id_attr_type: string, the_value: scalar, is_default: boolean [optional – default is false])`

This method creates a new Selection Value for an 'enumerated' attribute type (one with `ENUM_` at the beginning of its `attr_type` (data-type) column.) When a Selection Value is successfully created, it will show up in a drop-down selection list in any screen that allows users to input values for Extra Fields / Attribute Types. Each value has an ID (`id_attr_value`) that is used to refer to the value individually (for instance, to delete it with `deleteAttributeTypeSelectionValue`.)

PARAMETERS

Param 1: `id_attr_type` – (**string**) the ID of the Attribute Type that the selection value should belong to.

Param 2: `the_value` – (**string, number, or YYYYMMDD** date) the actual value that should show up in the selection list.

Param 3: `is_default` – (**boolean**, default is false) if true (non-zero), then the newly created value will be the default value for this attribute type, replacing any other as the default value. (The value that used to be the default is kept intact, but it will no longer be the default.)

RETURN VALUE

The Selection Value ID (`id_attr_value`) is returned if the value was successfully created.

EXCEPTIONS

`TypeError` is raised if `the_value` is not appropriate for the data-type of the Attribute Type.

`ValueError` is raised if the Attribute Type does not exist.

void setAttributeTypeSelectionDefault (*id_attr_value*: **string**)

This method makes a particular Selection Value the default for its Attribute Type (overriding any default that existed before.)

PARAMETERS

Param 1: *id_attr_value* – (**string**) the ID of the Selection Value (not the ID of the Attribute Type!) that is to become the new default.

RETURN VALUE

This method returns `None` if successful.

EXCEPTIONS

`ValueError` is raised if the Selection Value ID (*id_attr_value*) does not exist.

list getAttributeTypeSelectionValueRecords (*id_attr_type*: **string**)

This method returns a list (of lists) representing all the Selection Values that exist for the given Attribute Type ID. Both the Values themselves and their ID's are returned in pairs.

PARAMETERS

Param 1: *id_attr_type* – (**string**) the ID of the Attribute Type that we are interested in.

RETURN VALUE

An Array of 2-length string Arrays is returned, where each 2-length sub-array is the pair (`Value`, `SelectionValueID`)

The values are returned sorted (by value) in ascending order (A to Z), with the default value (if any) always first.

EXCEPTIONS

`ValueError` is raised if the Attribute Type does not exist.

list getAttributeTypeSelectionValues (*id_attr_type*: **string**)

This method returns a list representing all the Selection Values that exist for the given Attribute Type ID. Only the values themselves are returned, not the Selection Value ID's (for those, use the `getAttributeTypeSelectionValueRecords` method.)

PARAMETERS

Param 1: *id_attr_type* – (**string**) the ID of the Attribute Type that we are interested in.

RETURN VALUE

An Array / list of strings is returned, where each string is a Selection Value for the given Attribute Type. The values themselves are returned, not the Value IDs. The values are returned sorted in ascending order, with the default value (if any) always first.

EXCEPTIONS

`ValueError` is raised if the Attribute Type does not exist.

void deleteAttributeTypeSelectionValue (id_attr_value: string)

This method deletes the given Selection Value. It will no longer show up in Attribute Selection Lists. Any Objects which already have this Value will retain the value, but the value will not exist in future selection lists.

PARAMETERS

Param 1: id_attr_value - (**string**) the Selection Value ID to delete.

RETURN VALUE

This method returns `None` if successful.

EXCEPTIONS

`ValueError` is raised if the Selection Value ID (id_attr_value) does not exist.

Miscellaneous Session Object Methods

void logout ()

Terminate the session. A session will automatically be terminated after a certain amount of time, but it is recommended that you manually terminate the session when finished. The length of the session timeout is control by the Timesheet Global Preference “*Idle time before a user is forced to relogin (minutes)*” under the “Security and Access” section. (Global preferences are not adjustable through the jxAPI.)

jxAPI Batch Commands

Versions of Journyx Timesheet after 4.6m2 and 5.0m2 include a facility for grouping multiple jxAPI commands into a single SOAP request. This is called the “jxAPI Batch” feature, and is generally useful, particularly for enhancing performance. Grouping commands together in a batch avoids the overhead of handling each command as a separate CGI request.

There is no “hard” limit on the number of commands that may be grouped together in a single batch, nor on the type of command. Calls to any jxAPI “Session methods” may be mixed together in a single jxAPI Batch request. However, practical considerations such as HTTP timeouts may limit the number of commands that may be put into a single batch. In other words, your SOAP client software may give up before a result is ever returned if you specify a lot of time-consuming operations in your batch.

The performance and reliability of your jxAPI client application can be improved with careful use of the Batch architecture. It is neither necessary nor desirable to group *all* of your clients jxAPI requests into a single Batch request. For instance, depending on your applications design, the form of certain jxAPI requests may depend completely on the results of the previous requests. When using the jxAPI Batch, every command in the batch must be completely laid out with all parameters listed before the first command is executed. The only provision for controlling the “flow” of the batch is the `stop_on_error` parameter.

The jxAPI Batch is typically used to collect a group of similar jxAPI requests, such as setting Attribute (Extra Field) information for a long list of users. Setting the email address (for example) of 500 users would involve 500 separate jxAPI requests, which can be prohibitively slow, as each request is a complete HTTP transaction and (de)marshalling of the actual XML request.

Wrapping all 500 change email requests in a single jxAPI batch can bring the runtime for that operation down from several minutes to a few seconds.

At the heart of the jxAPI Batch system is a set of four new Structures (Records) which encapsulate all information about the batch request and the corresponding set of results.

`jxAPICommandSet` This represents a complete batch (list) of jxAPI commands, along with authentication and other meta-information.

`jxAPICommand` This represents a single jxAPI command and contains the method name and any arguments.

`jxAPIResultSet` This represents the corresponding set of results for a `jxAPICommandSet`.

`jxAPIResult` This represents an individual return value (or Fault) for the corresponding `jxAPICommand`.

A jxAPI Batch request is made by calling the `jxAPIBatch` method with a `jxAPICommandSet` as the only parameter. The server will step through each `jxAPICommand` in the `jxAPICommandSet` and execute the command, collecting the result of that command in a `jxAPIResult` record. The entire set of results is collected in a `jxAPIResultSet` object, and this is the return value of the `jxAPIBatch` method.

Conceptually this is rather simple, but as usual, the devil is in the details. A complete working Python example of the jxAPI Batch in action is available by contacting Journyx. Examples in other languages (particularly Visual Basic and Java) may be available by the time you read this.

Each `jxAPICommand` in the `jxAPICommandSet` is executed in the order that it is listed in the SOAP request. The commands are executed serially; that is, one after the other rather than concurrently. There is no provision to control the order of command execution other than by laying them out in the correct order in the SOAP request.

Note: The `jxAPIBatch` method is a “Base” method, meaning that it is accessed through the regular `soap.pyc` URL, and not using the URL that is the return value of the login method. A temporary login session is established using the username and password that are elements of the `jxAPICommandSet`. However, if you have already have a Session ID, you can use this instead. Session IDs are part of the URL that the jxAPI login method returns, and are also found in the web browser cookies.

The username and password (or a Session ID) are included as fields in the `jxAPICommandSet` record, and a temporary session is automatically established if need be. All `jxAPICommands` are always executed “on behalf of” the User given by the `jxAPICommandSet`. There is no provision for mixing operations on behalf of different users within a single Batch.

Each individual `jxAPIResult` record will be listed in the `jxAPIResultSet` in the same order as it was executed. However, for the convenience of the client, an “order” attribute is included with the `jxAPIResult` to indicate which order the command was executed in. The first result will have `order = 0`. The name of the corresponding command is also provided in the `jxAPIResult` as the “methodName” attribute.

An individual `jxAPICommand` may either return a single result (the “return value”, which may itself be a complex record) or there may be a Fault (an error or exception) during its execution. Whether or not a Fault occurred is indicated by the “fault” attribute, which will be either 1 or 0 in every `jxAPIResult` record. If `fault` is 0, then no error occurred, and the “result” attribute will hold the actual return value of the command. If instead `fault` is 1, then “result” holds the SOAP Fault object which indicates the problem with that specific command.

If there is for some reason a general fault running the batch itself, not related to a specific command in the batch, a regular SOAP Fault will be returned as with any other normal jxAPI call. This includes cases where the supplied username, password, or Session ID are invalid.

By default, an entire set of commands will be completely executed even if there was an error in the first few commands. Since later commands may depend on the results of previous commands, there is a way you can indicate that the first exception (error) should halt the entire batch. The `stop_on_error` attribute of the `jxAPICommandSet` controls whether processing of commands should be halted after the first error is encountered. The default is to *not* stop, but rather to continue executing all the commands in the batch. If `stop_on_error` is set, and an error occurs before the batch is finished, the `jxAPIResultSet` will be returned, with `jxAPIResult` objects inside for each command that was executed up to and including the one that caused the error. In that case, `jxAPIResult` objects will not be included for `jxAPICommands` that did not get to run because the earlier error stopped the processing.

jxAPI Batch Record Details

Record:
`jxAPICommandSet` (SOAP Struct)

Element Name	Element Type	Notes
<code>login</code>	String	The ID of the user who is running the commands in the batch.
<code>password</code>	String	
<code>session</code>	String	Optional – you may provide a Session ID in lieu of <code>login</code> and <code>password</code> attributes. The Session ID is everything after the <code>?</code> in the URL that is returned by the <code>jxAPI login</code> method. If using <code>login/password</code> , do not include <code>session</code> .
<code>stop_on_error</code>	Integer	Use 1 to force any errors to stop the rest of the batch from executing. 0 is the default.
<code>commands</code>	Array of <code>jxAPICommand</code> records	<p>The commands are executed in the order in which they appear in this array.</p> <p>The array itself should have attributes like this:</p> <pre>SOAP-ENC:arrayType="xsd:instance[X]" xsi:type="SOAP-ENC:Array"</pre> <p>Where X is the total number of commands in the array, starting from 1. The commands are executed in the order in which they appear here. An <code>arrayType</code> of “<code>xsd:ur-type[X]</code>” is also acceptable.</p>

Record:

`jxAPICommand` (SOAP Struct)

Element Name	Element Type	Notes
<code>methodName</code>	String	The name of the jxAPI method to execute.
<code>methodArgs</code>	Array	See below.
<code>methodKeywordArgs</code>	Struct	Optional – see below.

`methodArgs` will always be an Array. (`xsi:type="SOAP-ENC:Array"`). The type of the elements in the array may vary. If they are all the same type, that may be indicated. If they are heterogeneous types, then the `arrayType` should be `"xsd:ur-type[X]"` (again, replace the X with the actual number of elements.) If the method has no parameters at all, you can omit the `methodArgs` attribute. If the method has only one parameter, then `methodArgs` should be an array with one element.

`methodKeywordArgs` is a simple Struct (record) of name-value pairs representing keyword arguments to the method.⁴ This is completely optional; you do not need to include this attribute if there are no keyword arguments. All jxAPI method which have keyword arguments have default values for those, so they need not be specified unless you wish to override the default behavior. All keyword arguments may also be specified in the regular (positional) `methodArgs` instead, unless you are relying on default values for some keyword arguments and not others. If you are at all confused by this, you can just put all arguments in `methodArgs` and forget about `methodKeywordArgs`.

Record:

`jxAPIResultSet` (SOAP Struct)

Element Name	Element Type	Notes
<code>results</code>	Array of <code>jxAPIResult</code> records	This is the only element in the <code>jxAPIResultSet</code> record. It is an array of individual <code>jxAPIResult</code> records.

The `results` element of the `jxAPIResultSet` is an array, similar to the `commands` array element of the `jxAPICommandSet` record. The array type will be `SOAP-ENC:arrayType="xsd:instance[X]"` (where X represents the number of `jxAPIResult` records in the array.)

Record:

`jxAPIResult` (SOAP Struct)

Element Name	Element Type	Notes
<code>result</code>	(varies)	This is either the return value of the command, or the Fault object. (See below.)
<code>fault</code>	Integer	This will be 0 if there was no Fault (exception) running the command, otherwise it will be 1.
<code>methodName</code>	String	The name of the method that was invoked.
<code>order</code>	Integer	This indicates the order in which the command was executed, counting from 0.

As mentioned above, the type of the “`result`” attribute will vary depending on the method invoked. It may be a SOAP Fault structure if the fault flag is set to 1, otherwise it is the actual return value of the method. Obviously the type of an actual return value depends on the method itself. Some methods return

⁴ As opposed to positional arguments, which do not have a name.

simple results like a single string or number, but some methods may return complex results like an Array of Records.

jxAPI Batch Example

Below are two example jxAPI requests and responses. Both of the requests are functionally the same – to request a list of "Code" records which match the name "Clerical" and have a loggable value of 1. There should be only one Code with any given name, so only one record is returned (the array / list has one element.)

The difference is that in the second example the call to `getRecordsList` is encapsulated in a `jxAPIBatch` command. As you can see, the encoding of the actual parameters and return values remains exactly the same. The difference is that the `jxAPIBatch` format has additional layers of wrapping to enable you to specify multiple sets of commands.

In these examples, the XML indentation has been changed to make it more understandable.

Regular (non-batched) request:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>

    <getRecordsList SOAP-ENC:root="1">
      <v1 xsi:type="xsd:string">codes</v1>
      <v2>
        <pretty_name xsi:type="xsd:string">Clerical</pretty_name>
        <loggable xsi:type="xsd:int">1</loggable>
      </v2>
    </getRecordsList>

  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Response:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>

    <getRecordsListResponse SOAP-ENC:root="1">
      <Result
        SOAP-ENC:arrayType="ns1:SOAPStruct[1]"
        xsi:type="SOAP-ENC:Array" xmlns:ns1="http://soapinterop.org/xsd">
        <item>
          <reserved0 xsi:type="xsd:float">0.0</reserved0>
          <id xsi:type="xsd:string">Clerical</id>
```

```

        <reserved1 xsi:null="1"/>
        <autoadd xsi:type="xsd:float">0.0</autoadd>
        <pretty_name xsi:type="xsd:string">Clerical</pretty_name>
        <description xsi:type="xsd:string">
            Copying, Filing or Distributing Documents
        </description>
        <domain xsi:type="xsd:string">install_root_dom</domain>
        <loggable xsi:type="xsd:string">1</loggable>
    </item>
</Result>
</getRecordsListResponse>

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

Here is an example of the same `getRecordsList` request wrapped in a `jxAPIBatch` request:

```

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
  xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>

  <jxAPIBatch SOAP-ENC:root="1">
    <v1>
      <session xsi:type="xsd:string">
        192.168.100.140.1028758291.12.15186
      </session>
      <commands SOAP-ENC:arrayType="xsd:instance[1]" xsi:type="SOAP-ENC:Array">
        <item>
          <methodArgs
            SOAP-ENC:arrayType="xsd:ur-type[2]"
            xsi:type="SOAP-ENC:Array">
            <item xsi:type="xsd:string">codes</item>
            <item>
<pretty_name xsi:type="xsd:string">Clerical</pretty_name>
<loggable xsi:type="xsd:int">1</loggable>
          </item>
          </methodArgs>
          <methodName xsi:type="xsd:string">
            getRecordsList
          </methodName>
        </item>
      </commands>
    </v1>
  </jxAPIBatch>

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

And this is the response:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
  xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"

```

```
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>

<jxAPIBatchResponse SOAP-ENC:root="1">
<Result>
<results SOAP-ENC:arrayType="xsd:instance[1]" xsi:type="SOAP-ENC:Array">
  <item>
    <result
      SOAP-ENC:arrayType="ns1:SOAPStruct[1]"
      xsi:type="SOAP-ENC:Array"
      xmlns:ns1="http://soapinterop.org/xsd">
      <item>

<reserved0 xsi:type="xsd:float">0.0</reserved0>
<id xsi:type="xsd:string">Clerical</id>
<reserved1 xsi:null="1"/>
<autoadd xsi:type="xsd:float">0.0</autoadd>
<pretty_name xsi:type="xsd:string">Clerical</pretty_name>
<description xsi:type="xsd:string">Copying, Filing or Distributing
Documents</description>
<domain xsi:type="xsd:string">install_root_dom</domain>
<loggable xsi:type="xsd:string">1</loggable>

      </item>
    </result>
    <order xsi:type="xsd:int">1</order>
    <fault xsi:type="xsd:int">0</fault>
  </item>
</results>
</Result>
</jxAPIBatchResponse>

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Appendix A. XML Schema for Record Types

Journyx jxAPI v5.6.0.0 Schema/DTD

```

<?xml version="1.0"?>

<!DOCTYPE schema PUBLIC
    "-//W3C//DTD XMLSCHEMA 200010//EN"
    "/usr/local/lib/sgml/XMLSchema.dtd">

<!-- Journyx API XML Schema v1.6 -->

<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
    targetNamespace="http://www.journyx.com/jxapi"
    elementFormDefault="unqualified"
    attributeFormDefault="unqualified">

    <annotation>
        <documentation>Journyx API XML Schema v1.4</documentation>
    </annotation>

    <element name="UserRecord">
        <complexType abstract="false" block="" mixed="false">
            <all minOccurs="1" maxOccurs="1">
                <element name="id" type="string"></element>
                <element name="num_input" type="int"></element>
                <element name="num_dates" type="int"></element>
                <element name="full_name" type="string"></element>
                <element name="default_comment" type="string"></element>
                <element name="expense_gui" type="string"></element>
                <element name="time_gui" type="string"></element>
                <element name="mileage_gui" type="string"></element>
                <element name="domain" type="string"></element>
            </all>
        </complexType>
    </element>

    <element name="GroupRecord">
        <complexType abstract="false" block="" mixed="false">
            <all minOccurs="1" maxOccurs="1">
                <element name="id" type="string"></element>
                <element name="name" type="string"></element>
                <element name="description" type="string"></element>
                <element name="parent" type="string"></element>
                <element name="domain" type="string"></element>
            </all>
        </complexType>
    </element>

    <element name="ProjectRecord">
        <complexType abstract="false" block="" mixed="false">
            <all minOccurs="1" maxOccurs="1">
                <element name="id" type="string"></element>
                <element name="name" type="string"></element>
            </all>
        </complexType>
    </element>

```

```

    <element name="description" type="string"></element>
    <element name="parent" type="string"></element>
    <element name="domain" type="string"></element>
    <element name="estimate" type="int"></element>
    <element name="percent_done" type="int"></element>
    <element name="creator" type="string"></element>
    <element name="loggable" type="string"></element>
    <element name="type" type="int"></element>
  </all>
</complexType>
</element>

<element name="CodeRecord">
  <complexType abstract="false" block="" mixed="false">
    <all minOccurs="1" maxOccurs="1">
      <element name="id" type="string"></element>
      <element name="description" type="string"></element>
      <element name="autoadd" type="int"></element>
      <element name="domain" type="string"></element>
      <element name="pretty_name" type="string"></element>
    </all>
  </complexType>
</element>

<element name="SubcodeRecord">
  <complexType abstract="false" block="" mixed="false">
    <all minOccurs="1" maxOccurs="1">
      <element name="id" type="string"></element>
      <element name="description" type="string"></element>
      <element name="autoadd" type="int"></element>
      <element name="domain" type="string"></element>
      <element name="pretty_name" type="string"></element>
    </all>
  </complexType>
</element>

<element name="SubsubcodeRecord">
  <complexType abstract="false" block="" mixed="false">
    <all minOccurs="1" maxOccurs="1">
      <element name="id" type="string"></element>
      <element name="description" type="string"></element>
      <element name="autoadd" type="int"></element>
      <element name="domain" type="string"></element>
      <element name="pretty_name" type="string"></element>
    </all>
  </complexType>
</element>

<element name="TimeRecord">
  <complexType abstract="false" block="" mixed="false">
    <all minOccurs="1" maxOccurs="1">
      <element name="id" type="string"></element>
      <element name="user" type="string"></element>
      <element name="date" type="string"></element>
      <element name="code" type="string"></element>
      <element name="subcode" type="string"></element>
      <element name="subsubcode" type="string"></element>
    </all>
  </complexType>
</element>

```

```

    <element name="project" type="string"></element>
    <element name="group" type="string"></element>
    <element name="hours" type="double"></element>
    <element name="comment" type="string"></element>
    <element name="committed" type="int"></element>
    <element name="domain" type="string"></element>
  </all>
</complexType>
</element>

<element name="ExpenseRecord">
  <complexType abstract="false" block="" mixed="false">
    <all minOccurs="1" maxOccurs="1">
      <element name="id" type="string"></element>
      <element name="user" type="string"></element>
      <element name="date" type="string"></element>
      <element name="project" type="string"></element>
      <element name="expense_code" type="string"></element>
      <element name="amount" type="double"></element>
      <element name="comment" type="string"></element>
      <element name="committed" type="int"></element>
      <element name="currency" type="string"></element>
      <element name="source" type="string"></element>
      <element name="domain" type="string"></element>
      <element name="rocomment" type="string"></element>
      <element name="flags" type="string"></element>
      <element name="picture" type="string"></element>
    </all>
  </complexType>
</element>

<element name="MileageRecord">
  <complexType abstract="false" block="" mixed="false">
    <all minOccurs="1" maxOccurs="1">
      <element name="id" type="string"></element>
      <element name="user" type="string"></element>
      <element name="date" type="string"></element>
      <element name="project" type="string"></element>
      <element name="reason" type="string"></element>
      <element name="distance" type="double"></element>
      <element name="comment" type="string"></element>
      <element name="committed" type="int"></element>
      <element name="measurement" type="string"></element>
      <element name="vehicle" type="string"></element>
      <element name="domain" type="string"></element>
      <element name="extra" type="string"></element>
      <element name="flags" type="string"></element>
    </all>
  </complexType>
</element>

<element name="AttributeType">
  <complexType abstract="false" block="" mixed="false">
    <all minOccurs="1" maxOccurs="1">
      <element name="id_attr_type" type="string"></element>
      <element name="table_name" type="string"></element>
      <element name="pname" type="string"></element>
    </all>
  </complexType>
</element>

```

```
    <element name="description" type="string"></element>
    <element name="attr_type" type="string"></element>
    <element name="visibility" type="integer"></element>
    <element name="id_domain" type="string"></element>
    <element name="hidden_name" type="boolean"></element>
    <element name="string_default" type="string"></element>
    <element name="numeric_default" type="double"></element>
  </all>
</complexType>
</element>

</schema>
```