



easyprojects.net

API Documentation

Version 1.0

© 2003-2008 Logic Software Inc.
All rights reserved
Phone: +1 416 907 9944
Fax: +1 928 752 3905
E-mail: info@easyprojects.net
Web: www.easyprojects.net

TABLE OF CONTENTS

1. Overview of Easy Projects .NET API	3
1.1 Basic operation.....	3
1.2 Entities.....	3
1.3 Views	4
1.4 Descriptors	4
2. Setting up Easy Projects .NET API.....	5
2.1 Installing Easy Projects .NET API	5
2.1 Adding an API reference to your application	5
3. Using Easy Projects .NET API.....	6
3.1 Initial steps	6
3.2 Selecting entities	6
3.3 Selecting from views	6
3.4 Filters.....	6
3.5 Inserting entities.....	8
3.6 Updating entities.....	8
3.7 Deleting entities	9
4. Creating Sample Application with Easy Projects .NET API	10
4.1 General description	10
4.2 Creating the application and adding an API reference	10
4.3 Adding controls to the form.....	10
4.4 Creating a client stub instance member	10
4.5 Filling project list	11
4.6 Filling activity list	12
4.7 Adding an entry.....	14

1. OVERVIEW OF EASY PROJECTS .NET API

1.1 BASIC OPERATION

The Easy Projects .NET API provides web service-based access to the Easy Projects .NET service layer. There are four main operations that can be carried out:

1. Selecting data
2. Inserting data
3. Updating data
4. Deleting data

All Easy Projects .NET specific logical operations (such as object visibility handling according to permissions, or calculating activity dependencies) are performed behind the scene by Easy Projects .NET itself.

Easy Projects .NET API is based on Windows Communication Foundation. **You should have Microsoft .NET Framework 3.0 installed on the both client and server sides in order to use Easy Projects .NET API.**

For security purposes, Easy Projects .NET API uses secure channel (HTTPS) to exchange data between client and server.

Please note that Easy Projects .NET API functionality can be further extended in later Easy Projects .NET versions. However, API usage scenarios described in this documentation will remain unchanged.

1.2 ENTITIES

An entity is a basic data unit Easy Projects .NET operates with. Projects, activities, activity assignees, events, time entries and so on are all entities. On the other hand, an entity can be thought of as a record in a database table.

Entities are divided into classes. Entity class's identifier is its name. "Activity", "Project", "Event" are entity classes. Also, there is a database table with the same name for each entity class.

Easy Projects .NET assumes that the table representing an entity has an aggregated integer primary key. The name of a primary key field is a property an entity class is required to have.

The most important property of entity class is its fields. Each field corresponds to a database field, and is described by data type, access type, value limits and so on.

Some of entity class fields are reference fields. These fields describe one-to-many references of a given entity class to other entity classes. That is, the "Activity" entity class references the "Project" entity class and "Project" entity class references "Priority" entity class. Physically referenced fields represent foreign key database fields.

An entity can include calculated fields. In this case, an SQL expression for the field is provided.

The entities are described in App_Data\EntityClasses.config file.

1.3 VIEWS

A view is used to retrieve data from the database. A view can be treated as a collection of fields in different entities. When a query using particular view is performed, the framework generates an SQL query based on the view, and adds multiple JOIN statements to it according to relations between entities involved in the query. The view has the “root entity” parameter which helps Framework to generate the SQL query.

A view field is specified by the name of an entity field and that of the entity. Also the view field has additional parameters related to grid customization; these parameters do not apply directly when API is used.

A view can contain calculated fields the same way as an entity does.

Please note that some columns have complex aliases in form of <EntityName>_<ColumnName>, e.g. “Project_Name”, “Activity_ActivityID”. This also applies to parameters passed to the create, update, and delete operations. Expression column’s alias coincides with its name.

Some Easy Projects .NET predefined views have additional parameters that force the application core to insert extra columns into the resulting set. Usually such columns contain permission handling values that are not used in API applications most of the time.

The system contains a set of predefined views, but you are encouraged to create your own.

Views are described in App_Data\Views.config file.

1.4 DESCRIPTORS

When data is retrieved from the Easy Projects .NET service layer, it comes with an instruction on how this data should be treated. Thus each field in a data set retrieved in a typical case has an instance of FieldDescriptor class. The instance describes the field’s data type, and indicates whether it is required for user to enter it and so on. Descriptors can be specified for both entity fields and view fields. View field descriptors can lack some of the properties – the missing ones will be taken from the corresponding entity field descriptors.

2. SETTING UP EASY PROJECTS .NET API

2.1 INSTALLING EASY PROJECTS .NET API

First of all, you must uncomment the `<system.serviceModel>` section in *web.config* file in Easy Projects .NET installation folder, to enable Easy Projects .NET API.

Then you should install a certificate on your server to enable HTTPS communication. A temporary certificate can be used. You can refer <http://msdn2.microsoft.com/en-us/library/ms733813.aspx> to get information on how to create and install temporary certificates.

2.1 ADDING AN API REFERENCE TO YOUR APPLICATION

If you use Microsoft Visual Studio 2008, the API reference is no more than a Service reference in terms of Windows Communication Foundation. Consequently, you just need to add a Service reference to your project in order to reference Easy Projects .NET API. The service address you enter must consist of Easy Projects .NET base URL and a 'SharedServices.svc' service name, e.g. <http://localhost/EasyProjects/SharedServices.svc>. The system will generate all code stubs for you.

When using Microsoft Visual Studio 2005, you can use the SvcUtil command line utility from Windows SDK to generate the code using a command line like this:

```
"C:\Program Files\Microsoft SDKs\Windows\v6.0\Bin\SvcUtil.exe"  
/serializer:DataContractSerializer /config:app.config  
/reference:C:\WINDOWS\assembly\GAC_32\System.Data\2.0.0.0__b77a5c561934e089\System.Data.dll http://localhost/EasyProjects/SharedServices.svc?wsdl
```

You can download Windows SDK from <http://www.microsoft.com/downloads/details.aspx?familyid=C2B1E300-F358-4523-B479-F53D234CDDCF&displaylang=en>

Another way to reference Easy Projects .NET API is to add a stub code and make the configuration changes manually. To do this, you will have to add the `<system.serviceModel>` section to your application configuration file (app.config or web.config) as in the sample app.config file that comes with the sample application. The stub code is found in the CommonOperations.cs file supplied with the sample application.

You must add some assembly references to your project:

- System.ServiceModel
- System.Runtime.Serialization

3. USING EASY PROJECTS .NET API

3.1 INITIAL STEPS

Easy Projects .NET API is visible at the client side as `CommonOperationsServiceClient` class instance offering several methods to handle Easy Projects .NET data. Therefore, to start using Easy Projects .NET API, you must create an instance of that class.

Another important step is supplying credentials of the user who will be operating on the client side. These are supplied with `ClientCredentials.UserName` property of API proxy class instance. These credentials are automatically applied to all subsequent operations restricting object visibility and modification privileges.

3.2 SELECTING ENTITIES

Entity data is selected using `SelectEntities` method of the `CommonOperationsServiceClient` class. The method performs SELECT SQL query against the entity's table specified. By default, all data is returned (that is, all accessible items are selected). Data can be restricted using filters.

In `SelectEntities` method, one argument is used to specify which entity is selected, another one is a set of parameters used to set selection filters. For more details about filters, see chapter '3.4 Filters'.

The method returns a `Dataset` object with selected data.

All fields in the entity are included in the resulting dataset, including calculated ones. This can affect application performance. On the one hand, retrieving calculated fields can cause delays on SQL Server, and, on the other hand, this can result in having to transfer a vast amount of data. To this end, it is recommended that `UseOnlyRequiredFields` be used most of the time.

3.3 SELECTING FROM VIEWS

The `SelectFromView` method is used to retrieve data using views (see 1.3 Views). The method allows for generating a SELECT SQL query based on view data and parameters specified, and then executing it on the Easy Projects .NET database.

By default, all accessible data is retrieved, but restricting by means of filters is available.

The method returns a `Dataset` object with data selected.

In the `SelectFromView` method, one argument is used to specify which view is selected, while another one is a set of parameters used to set selection filters.

3.4 FILTERS

Filters are used to restrict data being selected. Effectively, what filters do is generate a WHERE clause in an SELECT SQL statement.

Each select operation can have several different filters. Restrictions of all filters overlap, and the result that meets the criteria of all filters involved is returned. In terms of SQL, each filter generates its own part of a WHERE clause, and then these parts are combined with AND.

Filters are generated based on parameters passed to the select operation. There are two types of filters – predefined filters and auto-generated filters.

Predefined filters are described in the *filters.config* file. Each predefined filter has the following properties:

- a set of parameter names that invoke the filter (<Parameters> section)
- a set of entities the filter operates on (<FilterEntities> section)
- the portion of a WHERE clause that the filter will generate (<SQL> section)

Predefined filter is invoked as soon as all of its parameters are included in the select operation parameters. Once the filter is engaged, it inserts its portion of a WHERE clause. The portion can refer filter parameter values with common SQL syntax, that is, if a parameter is named `Parameter1`, its value is referred to as `@Parameter1` in the <SQL> section text.

<FilterEntities> section lists all entities, whose fields are referred in the filter SQL text. These entities will be included into a FROM clause with an appropriate JOIN statement based on how the entities relate one to another.

An example of using a predefined filter can be found in chapter ‘4.6 Filling activity list’.

Auto-generated filters are generated automatically based on parameters with special names. These filters allow for injecting simple conditions to a WHERE clause, which compare a field with a certain value. The name of a parameter that invokes auto-generated filter has the following syntax:

`b_<EntityName>_<FieldName>_<Operator>`

Prefix `b_` indicates an auto-generated filter parameter, `<EntityName>_<FieldName>` specify the field, which will be compared, and `<Operator>` specify which type of comparison will be applied. The operator can be one of the following:

Operator	SQL comparison clause
Equal	<code><FieldName> = @Parameter OR (<FieldName> IS NULL AND @Parameter IS NULL)¹</code>
Greater	<code><FieldName> > @Parameter</code>
Less	<code><FieldName> < @Parameter</code>
LessOrEqual	<code><FieldName> <= @Parameter</code>

¹ This rather complicated syntax of an SQL clause is used to allow equality comparison with NULL values. Note that parameter value should be null (Nothing in Visual Basic) but not DBNull value to engage NULL value comparison. This also relates to field values when updating or inserting fields.

GreaterOrEqual	<FieldName> >= @Parameter
NotEqual	<FieldName> <> @Parameter
NotEqualOrNull	<FieldName> <> @Parameter OR <FieldName> IS NULL
EqualOrNull	<FieldName> = @Parameter OR <FieldName> IS NULL
IsNull	<FieldName> IS NULL
IsNotNull	<FieldName> IS NOT NULL
In	<FieldName> IN (<ValueList>) ²
NotIn	<FieldName> NOT IN (<ValueList>)
Like	<FieldName> LIKE @Parameter

Examples of auto-generated filters usage can be found in the sample application.

3.5 INSERTING ENTITIES

An entity is inserted with the `Insert` method. The method accepts an entity name to be inserted as the first argument, and a set of parameters which represent entity field values as the second one. Note that parameter names should coincide with full entity field names (that is, parameter must have names like “<EntityName>_<FieldName>”). The method will throw an exception if any of the entity’s required fields is missing.

The method returns ID of the entity inserted. If insertion has failed (e.g. due to lack of privilege), the method throws an exception.

3.6 UPDATING ENTITIES

Entity data is updated with the `Update` method. The method accepts entity name to be inserted as the first argument, and a set of parameters which represent entity field values as the second argument. As with the insert, parameter names should coincide with full entity field names. If you want to set some of the fields to NULL value, you must pass `null` (`Nothing` in Visual Basic) as a parameter value.

The entity primary key field must be present in the parameter set. The method will throw an exception if primary key is missing. Thus only one entity per operation can be updated.

² Parameter value should implement `ICollection` interface, which is used to generate value lists. Typically integer values are used. The same applies to `NotIn` parameter.

The method returns the number of records affected, namely, 1 on successful update and 0 if an entity is missing (ID is non-existent). If update has failed (e.g. due to lack of privilege), the method throws an exception.

3.7 DELETING ENTITIES

Entity is deleted with the `Update` method. The method accepts an entity name to be inserted as the first argument and a set of parameters for delete operation as the second one. It is only the entity primary key that must be present in the list of parameters; the method accepts sets of parameters to allow extensions. Only one entity per operation can be deleted.

The method returns number of records affected, that is, 1 on update success and 0 if entity is missing (ID is non-existent). If deletion has failed (e.g. due to lack of privilege), the method throws an exception.

4. CREATING SAMPLE APPLICATION WITH EASY PROJECTS .NET API

4.1 GENERAL DESCRIPTION

Below are the steps describing how to create a sample Easy Projects .NET application that allows adding time entries to the system. The application code is rather “raw” for simplicity’s sake (there are no error checks, etc.).

4.2 CREATING THE APPLICATION AND ADDING AN API REFERENCE

1. Open Microsoft Visual Studio and create a new Windows application.
2. Add a reference to Easy Projects .NET API, as described in chapter 0.

4.3 ADDING CONTROLS TO THE FORM

1. Add textBoxUserName and textBoxPassword text boxes to the application’s form.
2. Add comboBoxProjects and comboBoxTasks combo boxes.
3. Add textBoxHours and textBoxDescription text boxes.
4. Add dateTimePickerEntryDate date time picker. And,
5. Add buttonAddEntry button.

4.4 CREATING A CLIENT STUB INSTANCE MEMBER

We first create a variable that will hold the client stub instance. We then carry out all operations using this instance. The variable will be accessible via the property which initializes the client on the first use. To create a member, we add the following code to Form1.cs file:

```
private CommonOperationsServiceClient api;

private CommonOperationsServiceClient API
{
    get
    {
        if (api == null)
        {
            api = new CommonOperationsServiceClient();

            api.ClientCredentials.UserName.UserName = textBoxUserName.Text;

            api.ClientCredentials.UserName.Password = textBoxPassword.Text;

            ServicePointManager.ServerCertificateValidationCallback = ValidateServerCertificate;
        }
    }
}
```

```

    }
}
}

```

The client stub instance is now created and initialized when user name and password are entered on the first use. This is not handy in real applications but is OK as far as the example is concerned.

Also, there is a trick in the code above as it assumes server certificate to be always valid. Certificate validation may fail if a temporary certificate is used (e.g. with a name mismatch error). To override this, we replace the certificate validation callback function with one that always returns a valid certificate value:

```

public static bool ValidateServerCertificate(
    object sender,
    X509Certificate certificate,
    X509Chain chain,
    SslPolicyErrors sslPolicyErrors)
{
    return true;
}

```

4.5 FILLING PROJECT LIST

First we must create a view that will allow selecting a list of project. The view should contain two columns: ID and Project Name. Let's name the view "API_ProjectList". The XML code of the view will be as follows:

```

<View Name="API_ProjectList" RootEntityName="Project">
    <Fields>
        <Field Name="ProjectID" EntityName="Project" DefaultVisibility="true"/>
        <Field Name="Name" EntityName="Project" DefaultVisibility="true"/>
    </Fields>
</View>

```

We add the code to the </Views> element of App_Data\Views.config file.

There

Let our project list be filled as we focus it. To accomplish that, we write the following code in the OnEnter event handler of the project list combo box:

```

private void comboBoxProjects_Enter(object sender, EventArgs e)
{
    Dictionary<string, object> projectListParameters = new Dictionary<string, object>();

```

```

comboBoxProjects.DisplayMember = "Project_Name";

comboBoxProjects.ValueMember = "Project_ProjectID";

comboBoxProjects.DataSource = API.SelectFromView("API_ProjectList",
projectListParameters).Tables[0];
}

```

We use the `SelectFromView` method of the API client stub to retrieve data from the view. The selection does not require parameters, so we pass an empty parameter dictionary to the method. We use the `ProjectID` field as a value field, and the `Name` field as a display field.

4.6 FILLING ACTIVITY LIST

Next, we must fill the activity list. It should be filled with activities that correspond to the project selected on each change of the project. Hence the code to be placed in the `OnSelectedIndexChanged` event handler of the `comboBoxProjects` drop down list:

```

private void comboBoxProjects_SelectedIndexChanged(object sender, EventArgs e)
{
    Dictionary<string, object> taskListParameters = new Dictionary<string, object>();
    taskListParameters.Add("b_Task_ProjectID_Equal", comboBoxProjects.SelectedValue);
    comboBoxTasks.DisplayMember = "Task_Name";
    comboBoxTasks.ValueMember = "Task_TaskID";
    comboBoxTasks.DataSource = API.SelectFromView("API_TaskList", taskListParameters).Tables[0];
}

```

The `API_TaskList` view is used here, the code for this view is as follows:

```

<View Name="API_TaskList" RootEntityName="Task">
    <Fields>
        <Field Name="TaskID" EntityName="Task" DefaultVisibility="true"/>
        <Field Name="Name" EntityName="Task" DefaultVisibility="true"/>
    </Fields>
</View>

```

We must limit activities to those of a selected project, and we must show only those activities that are assigned to a current user. The former is achieved by `b_Task_ProjectID_Equal` parameter, which imposes a condition that only activities having specific `ProjectID` be selected. The other condition is more sophisticated, and it is implemented through a filter. One that performs the same functionality is already set up in the Easy Projects .NET configuration (no surprise, since the Time Logs page offers functionality similar to one we are developing here).

The filter has a definition as follows:

```

<Filter>

    <FilterEntities>

        <FilterEntity>Task</FilterEntity>

    </FilterEntities>

    <SQL>[Task].[TaskID] in (SELECT DISTINCT ta.TaskID FROM TT_TaskAssignees ta INNER JOIN
TT_ProjectMembers pm ON ta.ProjectMemberID = pm.ProjectMemberID WHERE pm.ProjectID =
@ForCreateTimeEntryTaskList_ProjectID AND pm.UserID = @ForCreateTimeEntryTaskList_UserID)</SQL>

    <Parameters>

        <Parameter>ForCreateTimeEntryTaskList</Parameter>

        <Parameter>ForCreateTimeEntryTaskList_ProjectID</Parameter>

        <Parameter>ForCreateTimeEntryTaskList_UserID</Parameter>

    </Parameters>

</Filter>

```

The filter selects only activities assigned to a specific user within a specific project as can be seen from the filter definition. User ID is specified with a @ForCreateTimeEntryTaskList_UserID parameter and project ID is specified with a @ForCreateTimeEntryTaskList_ProjectID parameter. The @ForCreateTimeEntryTaskList parameter is needed to engage the filter, the value of the parameter is not of significance.

To engage the filter, we must add all three parameters to the select request. The last thing we must find out is the current user's ID. We obtain the ID by querying Easy Projects for a user with the specified login name (login names in Easy Projects are unique). The property that returns user ID is implemented as follows:

```

private int userID;

private int UserID
{
    get
    {
        if (userID == 0)
        {
            Dictionary<string, object> userParameters = new Dictionary<string, object>();
            userParameters.Add("b_User_Login_Like", textBoxUserName.Text);
            userID = (int)API.SelectEntities("User", userParameters).Tables[0].Rows[0]["User_UserID"];
        }
        return userID;
    }
}

```

Note that we use a `Like` filter operator rather than `Equals` in order to do a case-insensitive lookup by login name. DB collation must be set to case-insensitive (this is the default).

4.7 ADDING AN ENTRY

To add an entry, we must specify the following data:

- ID of activity to add entry to
- ID of user who adds entry
- Entry duration
- Entry description
- Entry date
- Flag indicating if entry is billable (we shall assume it is)

Now we know all of them and can add an entry in the `OnClick` event handler of the `buttonAdd` button:

```
private void buttonAdd_Click(object sender, EventArgs e)
{
    Dictionary<string, object> timeEntryParameters = new Dictionary<string, object>();
    timeEntryParameters.Add("TimeEntry_TaskID", comboBoxTasks.SelectedValue);
    timeEntryParameters.Add("TimeEntry_UserID", UserID);
    timeEntryParameters.Add("TimeEntry_Duration", Decimal.Parse(textBoxHours.Text));
    timeEntryParameters.Add("TimeEntry_Description", textBoxDescription.Text);
    timeEntryParameters.Add("TimeEntry_EntryDate", dateTimePickerEntryDate.Value);
    timeEntryParameters.Add("TimeEntry_Billable", true);
    API.Insert("TimeEntry", timeEntryParameters);
}
```