

Article

[Olga Phomina](#) · Nov 1, 2016 9m read

Reflection in Caché



The topic of reflection hasn't been raised in Caché forums or blogs too often. Perhaps, it happened because the notion of reflection is not explicitly defined in Caché. However, it exists in Caché and can be a very useful development tool.

What is reflection?

The notion of [reflection](#) refers to a process where a program can monitor and modify its own structure and behavior at runtime.

Some programs can process their own constructs as data by performing reflexive modifications.

Reflection-oriented programming encompasses self-checks, self-modification and self-cloning. However, the greatest advantage of the reflection-oriented paradigm is in the dynamic modification of the program, which can be defined and performed at runtime.

This notion was first introduced by Brian Cantwell Smith in his [master's thesis work](#).

If you go to oracle.com, you will find out that reflection is mostly used by programs that require that their behavior be tested or modified during code execution. This is a relatively advanced technology that should be used only by developers with a very good command and understanding of the basics of the corresponding programming language. With this in mind, you can use reflection as a powerful tool and will be able to do things that seemed impossible to do in your application.

If you are not familiar with reflection yet, then this notion is not quite self-explanatory.

As a rule, reflection is limited to changing the behavior of the program in some way by preparing instructions for its operation without knowing the exact moment it will be working with a particular object.

Reflecion in Caché

I have not come across a separate section or special definition.

However, some functions do relate to this remarkably interesting topic.

This said, let me introduce them:

[\\$CLASSMETHOD](#) - execute the defined class method in the desired class (from any point of the program);

[\\$CLASSNAME](#) - returns the name of the class;

[\\$ISOBJECT](#) - checks whether the specified expression is an object or not;

[\\$METHOD](#) - lets you call a method of a particular class instance;

[\\$NAME](#) - returns the name of a variable;

[\\$PROPERTY](#) - refers to a particular object property and returns its value;

[\\$PARAMETER](#) - returns the value of a specified class parameter.

And this one is quite special

[\\$EXECUTE](#) - executes the code sent as a string with specified parameters.

You can read more about these functions and find usage examples in the [documentation](#).

Usage example:

```
$XECUTE("set name = ##class(Data.SampleDict1).%OpenId("_param_").Name")
```

That is, we can run any line as code. That'd be fine, but we should not forget about security. If we are developing a web application and someone manages to feed something like "kill the system" to this command... (there could be a smiley here, but it's not allowed).

Also, don't forget that this is runtime compilation, so use it very carefully.

Other reflexive functions work more or less like their counterparts in other languages.

Let's practice now

For example, we need to create a web application that consists of a single work area with two tabs. Tabs should contain different sets of fields and be associated with real database objects.

For instance, the first tab will be responsible for one type of information, the second one - for a different type.

The user should be able to create a dictionary entry and view the results of work in the work area in the form of a log. It should be possible to search and sort by a specified parameter. Our application is a demo of applying several reflexive functions while working with a database.

You can download the source code using this link: [Application sample](#) - simply import to Studio.

Let's take a look how the dict1.CSP page will look:

```
<script language="Cache" runat="Server">
    do ##class(Front.Blocks).PrintHeader("sampleDict",%session)
</script>
<script language="Cache" method="logout">
    do %session.Logout()
</script>
<script language="Cache" runat="Server">
    // for search:
    set searchFields = ##class(%ListOfDataTypes).%New()
    do searchFields.Insert(
        ##class(Front.Helpers.SearchColumn).%New("Name","String")
    )

    // - for list form
    set listFields = ##class(%ListOfDataTypes).%New()
    do listFields.Insert(
        ##class(Front.Helpers.ListColumn).%New("ID","#",100)
    )
```

```

do listFields.Insert(
    ##class(Front.Helpers.ListColumn).%New("Name","TestName",200)
)
do listFields.Insert(
    ##class(Front.Helpers.ListColumn).%New("Code","TestCode",300)
)

// - for edit form
set detailFields = ##class(%ListOfDataTypes).%New()
do detailFields.Insert(
    ##class(Front.Helpers.DetailColumn).%New("Name","TestName",300, 1)
)
do detailFields.Insert(
    ##class(Front.Helpers.DetailColumn).%New("Code","TestCode",300, 1)
)

// - for tabs header
set titleFields = ##class(%ListOfDataTypes).%New()
do titleFields.Insert(
    "ID"
)
do titleFields.Insert(
    "Name"
)
do ##class(Front.Blocks).PrintAngularJs()
do ##class(Front.Blocks).PrintGridJs()
do ##class(Front.Blocks).PrintNgGridFlexibleHeightPluginJs()
do ##class(Front.Blocks).PrintBootstrapJs()
do ##class(Front.Blocks).PrintDatePickerJs()
do ##class(Front.Blocks).PrintDftabmenuJs()
do ##class(Front.Blocks).PrintModalJs()
do ##class(Front.LDController).Initialize(searchFields, listFields,
detailFields,"Data.SampleDict1","Test area",titleFields, 1)
do ##class(Front.Blocks).PrintFooter()
</script>

```

Special methods in Front.Blocks are used to render particular page elements (Header, Footer).

To set the field to be searched, we need to form a searchFields list. For the editing area, the list changes to detailFields, and for the viewing area, the name of the list will be listFields.

Then initialize our content in LDController.

```

do ##class(Front.LDController).Initialize(searchFields, listFields,
detailFields,"Data.SampleDict1","Test area 1",titleFields, 1)

```

In dict2.CSP, it will look like this

```

do ##class(Front.LDController).Initialize(searchFields, listFields,
detailFields,"Data.SampleDict2","Test area 2",titleFields, 1)

```

LDController is a universal class. It contains the key features used by typical CSP pages, such as our test dictionaries.

In this case, it allows us to get a list of database records, filter it by a certain field (we have only set a name filter), get the total and shown number of records, create and edit records.

Methods of the LDController class

initialize - the HTML code of the editing form and viewing area. It's a mad mix of scripts and Caché code, I will not go into details here.

saveItemData - saves data from the editing area to the database.

Generally speaking, we managed to implement the necessary functionality and achieve scalability everywhere, except, perhaps, the initialize method. Even a novice won't have any problems creating views for other database objects by creating a new CSP page. Functional parts are broken into logical components, making it easy to understand what will need to be changed and where, if necessary.

The app looks the following way.

An area for viewing and searching for existing records

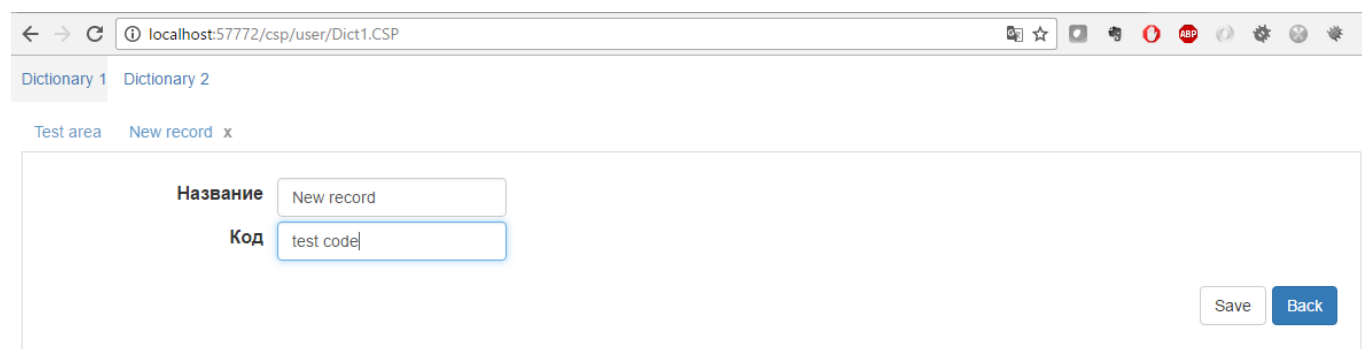
The screenshot shows a web browser window with the address bar displaying `localhost:57772/csp/user/Dict1.CSP`. The page has two tabs: "Dictionary 1" (active) and "Dictionary 2". Below the tabs is a "Test area" section. It contains a search form with a text input labeled "Name", a "Clear" button, and a "Search" button. Below the search form is a table with the following structure:

#	Название	Код
1	New record	test code

At the bottom of the table area, there is a pagination control showing "Items on page : 10" and a page indicator "1 / 1". At the very bottom right of the page, there is an "Add" button.

Reflections sample ©

An area for editing the element of the first directory



Dictionary 1 Dictionary 2

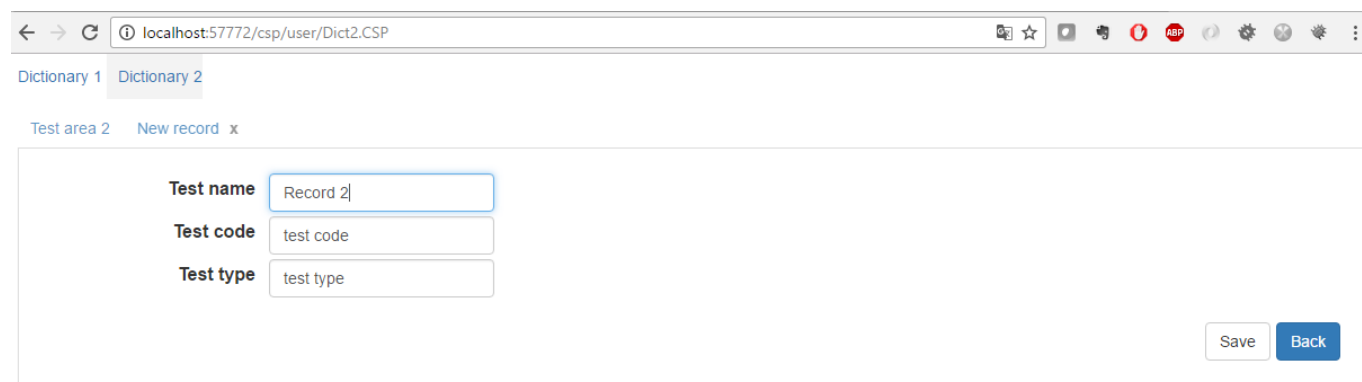
Test area New record x

Название New record

Код test code

Save Back

An area for editing the element of the second directory



Dictionary 1 Dictionary 2

Test area 2 New record x

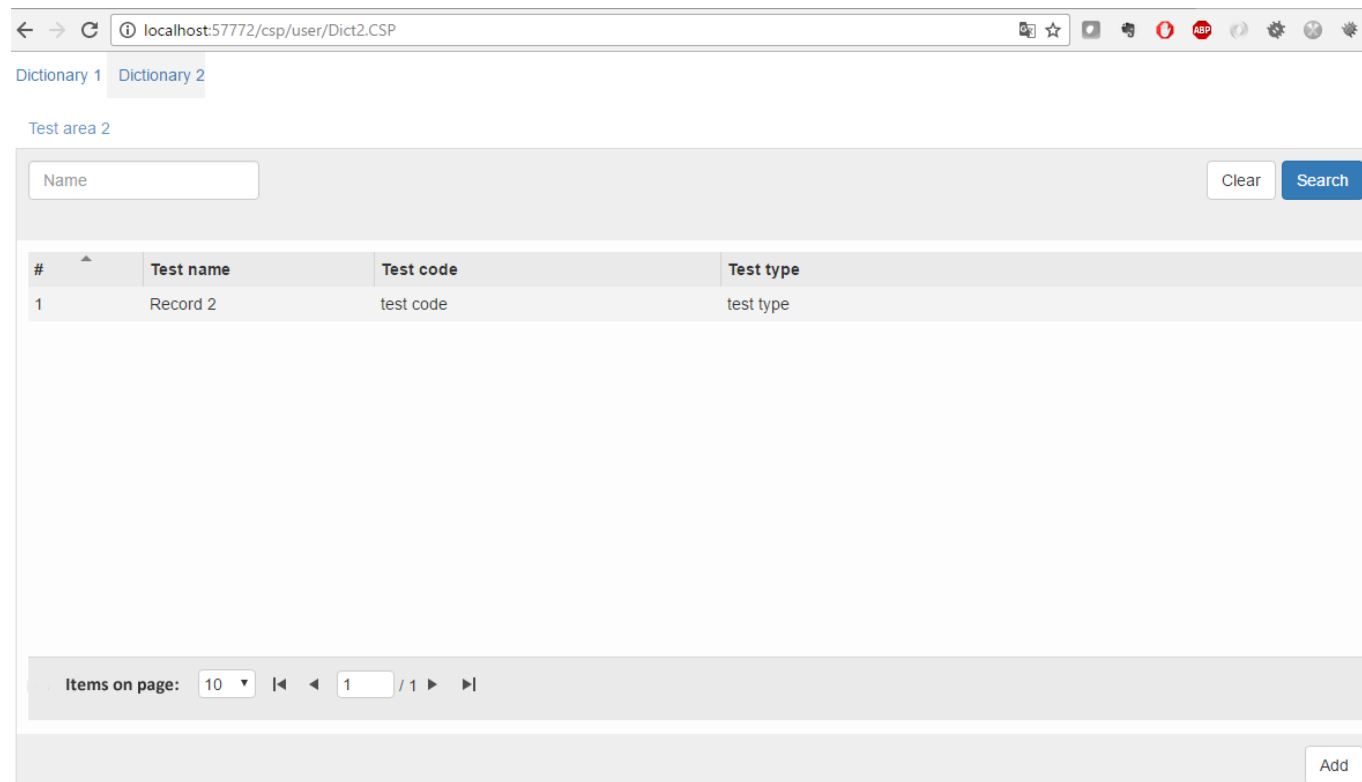
Test name Record 2

Test code test code

Test type test type

Save Back

An area for viewing the saved record



Dictionary 1 Dictionary 2

Test area 2

Name Clear Search

#	Test name	Test code	Test type
1	Record 2	test code	test type

Items on page: 10 1 / 1 Add

Reflections sample ©

Let's consider an example of creating/editing an object based on the `saveltemData` method. Parameters include a

list of object fields, form data as a JSON structure and a class name. Using the [\\$CLASSMETHOD](#) function, we can create/open an object of the required class (by name) Fill class properties with values from the data structure on the tab using the [\\$PROPERTY](#) function and save the object to the database.

ode of the saveItemData method

```
ClassMethod saveItemData(ListColumnsJSON As %String, ItemData As %String, DatasourceC
lassName As %String)
{
    set listColumns = ##class(Utils.JSON).Decode(ListColumnsJSON)
    $$$THROWONERROR(st,##class(%ZEN.Auxiliary.jsonProvider).
%ConvertJSONToObject(ItemData,,.itemData,1))

    if (itemData.ID =0) {
        set obj = $CLASSMETHOD(DatasourceClassName,"%New")
    }
    else {
        set obj = $CLASSMETHOD(DatasourceClassName,"%OpenId",itemData.ID)
    }

    for {
        set field=listColumns.GetNext(.idx)
        quit:idx=""
        set fieldName = field.GetAt("Field")
        if (fieldName != "ID") {
            set $PROPERTY(obj,fieldName) = $PROPERTY(itemData,fieldName)
        }
    }
    do obj.%Save()
}
```

The same can be done when obtaining database records and deleting them.

As you can see, using reflection in Caché is really easy.

We may not have covered a lot of opportunities, but solved our task, which allowed us to avoid objects and working with them altogether, as well as simplify the code structure and make it easier to understand. For those who are just starting to learn the language, this tutorial may become a good starting point for own discoveries.

Sources:

1. [Reflection \(computer programming\), Wikipedia](#)
2. [Procedural reflection in programming languages, Brian Cantwell Smith](#)
3. [The Reflection API, Oracle](#)
4. [Caché ObjectScript Functions, Intersystems](#)
5. [CacheJSON is a JSON encoder/decoder for Intersystems Cache](#)

[#Caché #Tutorial](#)

Source URL: <https://community.intersystems.com/post/reflection-cach%C3%A9>