Article
Dmitrii Kuznetsov · Jan 21, 2019 10m read

Should you trust Codd or your objects?

Headache-free stored objects: a simple example of working with InterSystems Caché objects in ObjectScript and Python

Neuschwanstein Castle

Tabular data storages based on what is formally known as the relational data model will be celebrating their 50th anniversary in June 2020. Here is an official document – that very famous article. Many thanks for it to Doctor Edgar Frank Codd. By the way, the relational data model is on the list of the most important global innovations of the past 100 years published by Forbes.

On the other hand, oddly enough, Codd viewed relational databases and SQL as a distorted implementation of his theory. For general guidance, he created 12 rules that any relational database management system must comply with (there are actually 13 rules). Honestly speaking, there is zero DBMS's on the market that observes at least Rule 0. Therefore, no one can call their DBMS 100% relational :) If you know any exceptions, please let me know.

The relational data model is not very complex and is very well studied. Maybe even overstudied. In the meantime, we'll be celebrating another anniversary in 2019 - exactly 10 years ago, when very first #NoSQL hashtag that later became known as "not only SQL" appeared on Twitter and started its rapid expansion into database development models.

Why such a lengthy introduction? Because the software development universe consists of data (and algorithms, of course), while the monopoly of the relational model leads, to put it mildly, to the divergence of the developer's consciousness. Because all the objects in the head of a developer (OOP is also global, right?) – all these lists, queues, trees, heaps, dictionaries, threads, and so on and so forth – are not tables.

What if we go on and recall the storage architecture in modern DBMS's? Let's be honest, no one stores data in tables. DBMS developers typically use variations of the B-tree (in PostgreSQL, for example) or, far less often, dictionary-based storage. On the other side of the barricade, developers using DBMS's for storing data do not work with tables either. And it makes developers constantly bridge the semantic gap using a cumbersome intermediate data layer. And by doing so, cause an internal dichotomy, system-wide discomfort and debugging sleeplessness.

That is, briefly speaking, we have a contradiction here – we pack data into objects suitable for our purpose, and at the same time have to care about some tables when saving data.

Dead end? Nope :) And what about object-relational representation, more commonly known as ORM? Let's leave this holy war to Egor Bugaenko & Co. And this whole last-century story, according to Uncle Bob, shouldn't be too much of a worry.

We should definitely mention that the "bag of bytes" (Robert Martin, Clean Architecture) can be serialized or dumped to a file or pushed to a suitable thread. First, it will immediately restrict us to a particular language. Second, we are going to focus on saving to a DBMS only for now.

All of these "bag of bytes" twists and turns have a pleasant exception - InterSystems Caché DBMS (now the InterSystems IRIS data platform as well). It may be the only DBMS in the world that doesn't hide the obvious from developers and takes it even further by saving them the trouble of thinking about "storing all this stuff the right way". Suffice it to say that the class extends the Persistent metaclass and it's in the bag - or, rather, in globals (don't confuse them with global variables!)

You can store all types of data, including character and binary streams. Here is the simplest example:

```
// a class for the object being saved with a single string parameter

// surprise: you can name objects with any string. We used quote marks in our example
, but they are obviously not needed for identifiers without spaces

Class FW.Events Extends %Persistent {

    Property "My name" As %String;

}



// trying to run it through the terminal

// creating a new "clean" object

set haJS = ##class(FW.Events).%New()



// save it

write haJS.%Id()
```

What's truly remarkable is that you can access stored objects not just via ObjectScript that is native for Caché but can also retrieve and save them directly in your programs written in Python, Java, JavaScript, C++, C#, Perl. And even - oh my! :) - retrieve information from these objects directly using SQL queries, as well as call your own methods. To be precise, methods in this case automatically (with a little help of SqlProc) turn into stored procedures. The Caché DBMS has all this magic under the hood.

> How to get free test access to the InterSystems Caché DBMS?
>
> It's 100% possible, no matter what some people say! :) Download and install the single-user, fully functional version of Caché by clicking on this link (requires free registration). MacOS, Windows, and Linux builds are available.
>
> The most convenient way of working with ObjectScript code with direct access to the Caché DBMS server (and the InterSystems IRIS platform as well) is to use the Atelier IDE based on Eclipse. All download and

> installation instructions can be found here.

If it's more convenient for you, you can use the simple and comfortable Visual Studio Code in combination with the ObjectScript plugin developed and supported by the community.

A few practical examples now. Let's try to create a couple of associated objects and work with them in ObjectScript and Python. Integration with other languages is implemented in a very similar manner. Python was chosen on the grounds of its "maximum kinship" to ObjectScript – both languages are script languages supporting OOP and lacking strong typing :)

To get some ideas, let's turn our sights to the popular "Framework Weekend" projects in Khabarovsk. The sample source code is available here: github.com/Hajsru/framework-weekend. Ours is provided in the text below.

> An important detail of macOS users. When launching support modules for Python, don't forget that you need to specify the DYLDLIBRARYPATH path to the Caché installation folder. For instance, like that:
>
> export DYLDLIBRARYPATH=/application/Cache/bin:$DYLDLIBRARYPATH
>
> This is highlighted in the documentation.

Creating stored ObjectScript classes

Let's go then. Our Caché classes will be very simple. We can do without an IDE - just copy the code of the classes directly through the portal of your instance of the Caché platform (yes, the Caché DBMS is far from being just a DBMS): System browser > Classes > Import (Namespace USER).

After saving, objects will appear in globals with names matching those of corresponding classes. Look for them also in the Caché management portal: System browser > Globals (Namespace USER).

```
// an event object includes a name, a description, a date, and a list of participants

Class FW.Event Extends %Persistent {

    Property title as %String;

    Property description as %String;

    Property date as %Date;

    Property visitors as list of FW.Attendee;

}



// the participant object has a name/nickname

Class FW.Attendee Extends %Persistent {

    Property name As %String;

}
```

Getting access to Caché objects from Python

First, let's connect to the Caché DBMS database. Let's do everything exactly as specified in the documentation.

> Just a useful fact for work. The free educational version of the Caché DBMS allows you to do everything its paid counterpart can do but supports two active connections only. Therefore, you won't be able to keep one connection from the IDE and concurrently try to run some code for interacting with the server. The simplest solution is to close the IDE while Python code is being executed.

```
# import of the Caché module for integrating it with Python3

import intersys.pythonbind3

# connecting to the server

conn = intersys.pythonbind3.connection()

conn.connect_now("localhost[1972]:USER","_SYSTEM","SYS", None)

# checking the connection descriptor

print ("conn = %d " % conn.handle)

# connecting to the database

database = intersys.pythonbind3.database(conn)
```

Now, let's create an object database with data on IT events and their participants. A very, very simple one. First, let's create a class for registering users and storing participants' details. For simplicity, the class only contains the participant's name.

```
// a class for objects containing information about registered participants

class Attendee:


# initialization of a new empty object in RAM

    def __init__ (self):

        self.att = database.create_new("FW.Attendee", None)



# writing the participant's name and saving the object to the database with a unique
id

    def new (self, name):

        self.att.set("name", name)

        self.att.run_obj_method("%Save",[])



# loading an object by id from the participant database
```

```
    def use (self, id):

        self.att = database.openid("FW.Attendee",str(id),-1,-1)


# removing an object from the participant database

    def clean (self):

        id = self.att.run_obj_method("%Id",[])

        self.att.run_obj_method("%DeleteId", [id])
```

As you can see, we are using ready wrapper functions for methods intended for accessing object fields in Caché: "set", combined with passing a property name in quotes in the parameters, and "openid" with the name of the package and class. There are examples for the identical "get" function below. To access any other methods, including those inherited by the class from its ancestors, use the runobjmethod() function with the name of the method and call parameters, if necessary.

The magic is in this line: self.att.runobjmethod("%Save",[])

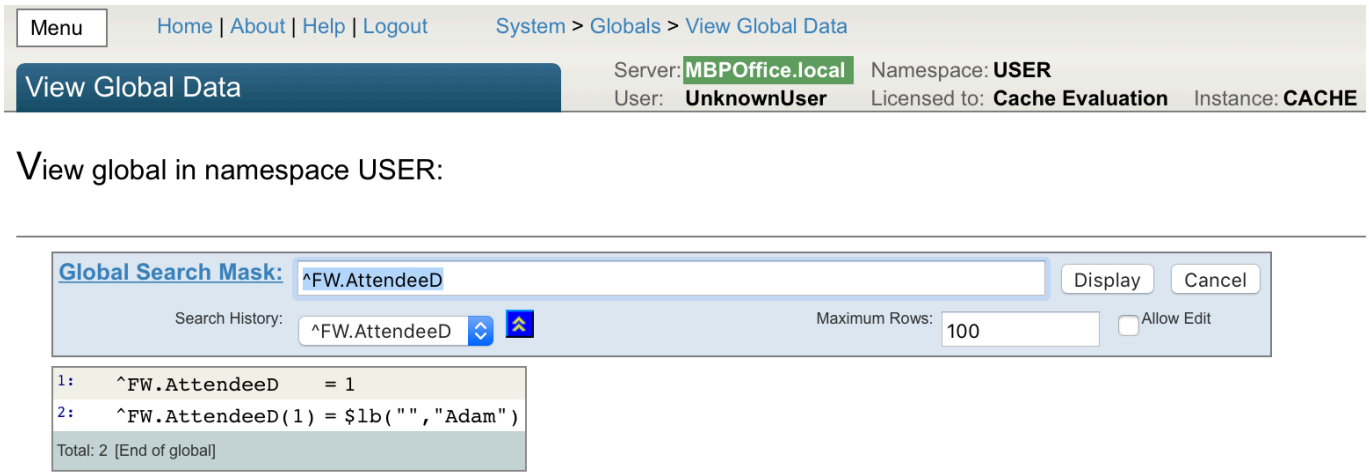
That's how we can save objects directly and without having to involve additional libraries and frameworks, such as the ubiquitous and inconvenient ORM's.

Besides, thanks to the OOP nature of ObjectScript along with the methods of your class (in our example, we didn't create them), we get access from Python to the entire set of methods inherited from the Persistent class and its children as a free bonus. Here's the complete list, just in case.

Let's create the first participant:

```
att = Attendee()

att.new("Adam")
```

Running this code will add a new global called FW.AttendeeD containing the newly saved object to the database (as seen in the screenshot):



After saving, this object gets a unique id (number 1). Therefore, you can load it to our program using this id:

```
att = Attendee()
```

```
att.use(1)

print (att.att.get("name"))
```

Knowing the id, we can now remove an object from the participant database:

```
att = Attendee()

att.use(1)

att.clean()
```

Make sure to check that after this sample code is run, the object record should disappear from the global. Although the loaded data still remain in the "memory" of your object until the program is closed.

Let's make the next step. Let's create actual event records.

```
# a class for objects containing  event details

class Event:

# initializing a new empty object in RAM

    def __init__ (self):

        self.event = database.create_new("FW.Event", None)


# populating an object with data and saving it in the database with a unique id

    def new (self, title, desc, date):

        self.event.set("title", title)

        self.event.set("description", desc)

        self.event.set("date", date)

        self.event.run_obj_method("%Save",[])


# loading an object by id from the database

    def use (self, id):

        self.event = database.openid("FW.Event",str(id),-1,-1)


# adding a participant to the event participants list

    def addAttendee (self, att):

        eventAtt = self.event.get("visitors")

        eventAtt.run_obj_method("Insert", [att])
```

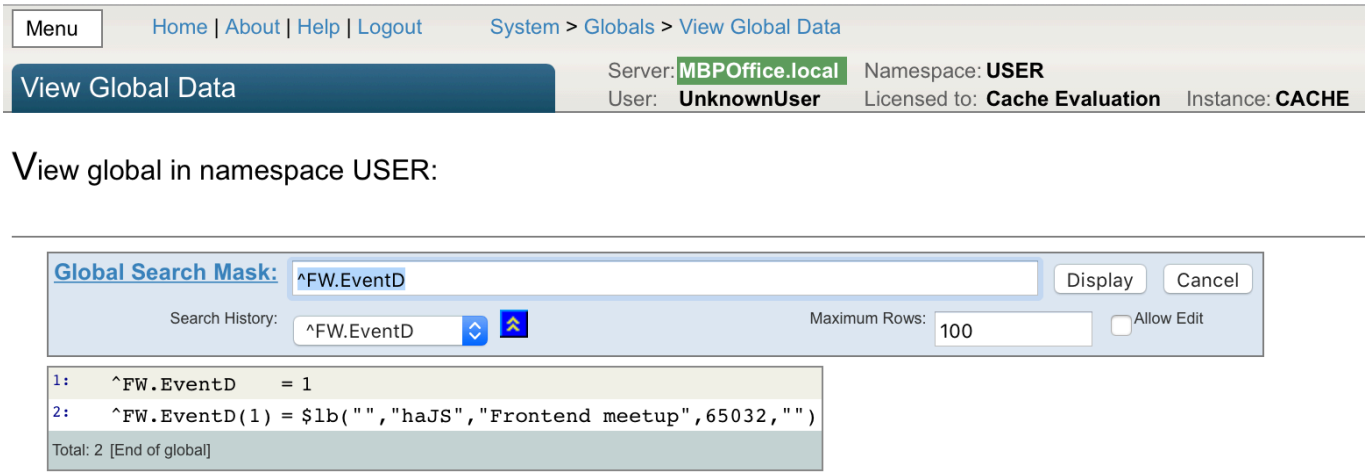
```
        self.event.set("visitors", eventAtt)

        self.event.run_obj_method("%Save",[])


# removing an object from the database

    def clean (self):

        id = self.event.run_obj_method("%Id",[])

        self.event.run_obj_method("%DeleteId", [id])
```

The structure of the class is almost identical to that of the participant class above. Most importantly, we now have a method for adding participants to the list of event participants: addAttendee(att).

Let's try to create an object record about the new event and save it in the database.

```
haJS = Event()

haJS.new("haJS", "Frontend meetup", "2019-01-19")
```

The result should look like this (please note that the date was automatically converted to the ObjectScript format and will be returned in the default format when loaded into a Python object):



All we need to do is to add the participant to the event:

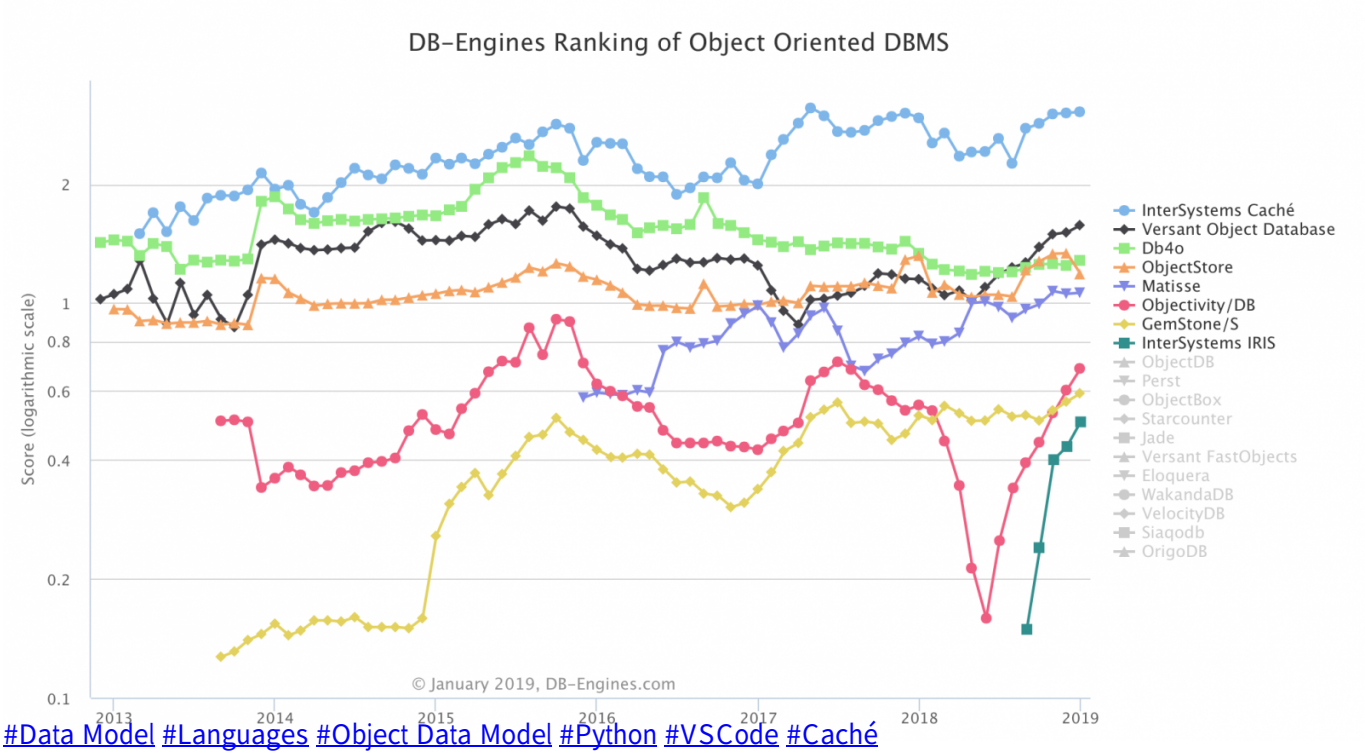
```
# loading the previously saved event

haJS = Event()

haJS.use(1)

# creating a new participant

att = Attendee()

att.new("Mark")

# adding the participant to our event
```

```
haJS.addAttendee(att.att)
```

As you can see from these examples, you don't necessarily need to simultaneously think about your data model and its tabular storage schema. You can use more obvious tools for your tasks.

Detailed instructions on connecting and using Caché with Python and other languages are always available in the official documentation and on the InterSystems developer community portal – and it's as many as 5000 members talking to each other here at community.intersystems.com

Fact: the multi-model InterSystems Caché DBMS remains a global leader on the object database market



#Data Model #Languages #Object Data Model #Python #VSCode #Caché

Source URL:https://community.intersystems.com/post/should-you-trust-codd-or-your-objects