3. A Standardised Test Data Format

3.1 Hierarchies and Decision Tables

Standardising the format of **test data** is a notoriously troublesome problem. But after much reflection while working through the recent SAM Pt I project, I'm inclined to think that these problems may be tractable after all.

Test Analysts have traditionally attached much weight to the idea of "Decision Tables", a concept which has been somewhat incorrectly borrowed from symbolic logic. In logic, a **decision table** has only Boolean values (True or False, Yes or No, √ or ×, 1 or 0). Along the top in a header row are arranged all the input variables for a Boolean expression, which we may regard as A, B, C, etc. with one final column for the Result of the expression, like this:

A	B	C	Result = (A ∨ B) ⟶ C
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(Don't take my word on the Boolean algebra here – I'm a bit rusty!)
The problem with these Truth Tables is that the number of combinations of input equals 2^n for *n* inputs (here $2^3 = 8$) and this rapidly becomes a very large number.

Test Analysts commonly use the term **Decision Table** more loosely to mean something more like a Table or View in a database. Again each column corresponds to a particular output variable, but its values may now be numerical or text values and not just Booleans. A relatively large number of rows is chosen that gives a reasonable *spread* over the sort of input combinations that might be expected rather than trying to be exhaustive, a practice known as a "limited entry" **Decision Table**. The test is run with the various input variables defined by a single *row* of the table *in turn*, a pass or fail being assigned to *the row*. So we might have:

Name	Basic Salary	Additions	Postcode	Age	Citizenship	Gender	Pass/Fail
Fred	22000	5000	CB3	45	USA	M	Pass
Alice	65000	0	NP25	33	GB	F	Pass
Jean			NR7		FR		Fail
Victor	44000	17000	N8	24	RUS	M	Pass
Beatrice	27000						Fail
Adolphe		5000	BM4		GDR	M	Pass

The option to have *blank* or *unknown* entries is important, and indeed some fields (columns) may actually be necessarily undefined for certain combinations of the values in other columns.
Generally, too, columns to the left tend to be in a certain sense *dominant* over columns to the right, carrying more deterministic information.
Sometimes one column (or more than one) may define an expected *output* and a Pass or Fail is determined by whether the given outputs arise from the given inputs.

In this looser concept of a **Decision Table**, the only necessarily Boolean column becomes the Pass/Fail one.

Such **Decision Tables** have been a common means of expressing differing data for use in testing in StudentSys, and most **scripts** make at least some allusions to them. For this reason I've expressly included in my Conversion System a set of tables for holding **Decision Tables** and for easily picking them out from existing **scripts** with, normally, a header row that defines the meaning of the various columns. This was a challenge for the Conversion System. Whereas in a *script*, the conversion algorithm need only look for certain well-defined columns like "Test Step", "Input Value" and "Expected Result" with a few variations in naming, there is no way of knowing in advance just how many columns a given **Decision Table** may include, and what sort of data types each column may use. Because of this subtlety, I also included as an auxiliary feature the automatic saving of **Decision Tables** as pure HTML tables, which being saved simply as text, are necessarily free-format. I will show presently that **Decision Tables** can be subsumed under the broader Data Definition Format I am about to propose.

But from what I have seen in my testing, **Decision Tables** are, like the databases which motivate them, too rigid a way of storing the variety of data that is used to drive a given test. A far more common pattern of data is a *nested hierarchy* or *tree structure*. In the SAM Pt I project, we start each test by logging in as one from a list of Admin Users or Administrators (this is almost always how StudentSys tests start), and each such Administrator has an Admin Role (College Admin, Department Admin, Clinical Schools Admin, etc.) which determines a list of "Child Jobs" that they can assign to the people they're administering, and a list of "Academic Organisations" to which they have access and

to which they can assign people, a list of what I'll here call Academic Categories of people they can administer (UGRD, PGRD, AUGD, etc., or in this project actual academic staff of the university) and a list of test "assignees" and so on.

Nested hierarchies are the backbone of object-oriented programming, and so excellent notations have been introduced to describe them. At the time of writing, two of these notations are way out in front in popularity: XML and JSON. The two are trivially interchangeable, so it would hardly matter which we chose, and in this document I'll use both. JSON is strictly a part of the programming language for the web 'Javascript'. The hierarchy sketched above would look like this in XML:

```
<AdminUserList>
    <AdminUser USN='123456789' ID='lucinda23' Role='College Administrator'>
        <ChildJobsList>
            <ChildJob Code='CST' Name='College Senior Tutor'/>
            <ChildJob Code='CGT' Name='College Graduate Tutor'/>
            <ChildJob Code='DOS' Name='Director of Studies'/>
            <ChildJob Code='TUT' Name='College Tutor'/>
            . . .
        </ChildJobsList>
        <AcadOrgList>
            <AcadOrg Code='SAC' Name='Santa Anna College'/>
            <AcadOrg Code='POOL' Name='Applicant Pool'/>
        </AcadOrgList>
        <AcadCareerList>
            <AcadCareer>
                <Code>UGRD</Code>
                <Name>Undergraduate</Name>
            </AcadCareer>
            <AcadCareer>
                <Code>PGRD</Code>
                <Name>Postgraduate</Name>
            </AcadCareer>
            . . .
        </AcadCareerList>
        <TestUserList>
            <TestUser USN='123456789' ID='annemarie12'>
            . . .
            </TestUser>
            <TestUser USN='123456789' ID='annemarie2000'>
            . . .
```

```
                    </TestUser>
                </TestUserList>
            </AdminUser>
            <AdminUser USN='123456789' ID='xyz31'>
            . . .
            </AdminUser>
            . . .
        </AdminUserList>
```

Here I've used ellipses[1] (". . .") to indicate that many other example may come in this category. The structure of XML is quite simple: it uses a system of "named brackets" called *tags* such as <TestUser> which is the *opening tag* for a Test User or "assignee" above, corresponding to an opening bracket "(", and </TestUser> which is the corresponding *closing tag* analogous to ")". Everything (all data) pertaining to that particular Test User has to go in between these two tags. The actual *tag names* within the tags, like "TestUser", cannot include spaces or any special symbols like "<" or ">".

For elements that don't have any further "parts" of their own like "Code" and "Name" above (known as "leaves", "atomic elements" or "terminals"), there are two ways of writing them: either as separate entries like <Code>PGRD</Code>, or *within the opening tag* of the entry that contains them like:

<AcadOrg Code='SAC' Name='Santa Anna College'></AcadOrg>.

It's because this introduces space into the opening tag that the tag name is not allowed spaces. Such "compact" entries are called *attributes* of their containing entry, and when the containing entry has nothing *other than attributes* within it, we can further condense the opening and closing tags into one using the notation with the closing "/" moved to the end of the would-be opening tag to indicate that the two are now combined:

<AcadOrg Code='SAC' Name='Santa Anna College'/>.

I would strongly recommend the convention of adding the suffix "List" to the tag for any list of similar objects as I've done above, as these will turn out to be central to our concept.

XML actually derives from HTML, the notation I have offered as one of two formats for holding **Decision Tables** in the conversion process. So converting from HTML to XML upon, or immediately after, conversion, is another possibility. Again the two formats are easily interchangeable.

[1] This is actually the plural of *ellipsis* not of ellipse, although both nouns are the same in the plural.

JSON ("Javascript Object Notation") is more compact and simpler, but arguably more liable to error. The same hierarchy in JSON would be written as:

```
var AdminUserList =
[
        {
                USN:'123456789',
                ID:'lucinda23',
                Role:'College Administrator',

                ChildJobsList:
                [
                        { Code='CST', Name='College Senior Tutor'},
                        { Code='CGT', Name='College Graduate Tutor'},
                        { Code='DOS', Name='Director of Studies'},
                        { Code='TUT', Name='College Tutor'},
                        . . .
                ],
                AcadOrgList:
                [
                        { Code='SAC', Name='Santa Anna College'},
                        { Code='POOL', Name='Applicant Pool'}
                ]
                AcadCareerList:
                [
                        { Code='UGRD',
                          Name='Undergraduate'},
                        { Code='PGRD',
                          Name='Postgraduate'},
                        . . .
                ],
                TestUserList:
                [
                        { USN='123456789', ID='annemarie12',  . . .},
                        { USN='123456789', ID='annemarie2000',  . . .},
                        . . .
                ]
        },
        {
                //Next Admin User
                USN:'123456789',
```

```
                ID:'xyz31',
                Role:'Clinical Schools Administrator',
                . . .
        },
        . . .
]
```

The "var" at the start merely indicates this is strictly a Javascript variable, and this can be ignored for now.
The main thing to note here is that *different* bracketing is used for *lists* "[", "]", from that used for the distinct *named properties* like Name, Code, USN, etc. which are separated by "{", "}". Individual list elements are not named here (we don't have "TestUser" or "ChildJob" labels), only the list as a whole being named, *because we are not allowed the same name for two subsidiary elements within another element*. Note that we may have list elements that have other elements within themselves, as in:

```
TestUserList:
[
        { USN='123456789', ID='annemarie12',  . . .},
        { USN='123456789', ID='annemarie2000',  . . .},
        . . .
]
```

where the elements of the list such as { USN='123456789', ID='annemarie12', . . .} are themselves complex objects (and may include other lists).

Note that I've also added a comment "//Next Admin User", which can be put in using the same standard syntax as in my main language proposal. Comments in XML are slightly more clumsy.

Because JSON actually derives from a programming language, its format is ideal for incorporation into my proposed test scripting notation; in newspeak, both are *programmatic* concepts.

Observe how either XML or JSON can subsume **Decision Tables** quite readily. If we take a sample **decision table** like that below (from my original example of a free-format **Decision Table**), then expand it first into XML:

Fred	22000	5000	CB3	45	USA	M
Alice	65000	0	NP25	33	GB	F
Jean			NR7		FR	

```
<EmployeeList>
        <Employee Name="Fred" Salary="22000" Additions="5000" Postcode="CB3">
                <Age>45</Age>
                <Citizenship>USA</Citizenship>
                <Gender>M</Gender>
        </Employee>
        <Employee Name="Alice" Salary="65000" Additions="0" Postcode="NP25">
                <Age>33</Age>
                <Citizenship>GB</Citizenship>
                <Gender>F</Gender>
        </Employee>
        <Employee Name="Jean" Postcode="NR7">
                <Citizenship>FR</Citizenship>
        </Employee>
</EmployeeList>
```

where I've freely and arbitrarily mixed attributes and full subsidiary tags.
Next, expand it into JSON:

```
        var EmployeeList =
        [ {Name:'Fred', Salary:'22000', Additions:'5000', Postcode:'CB3', Age:45,
                Citizenship:'USA', Gender:'M'},
          {Name:'Alice', Salary:65000, Additions:0, Postcode:'NP25', Age:33,
                Citizenship:'GB', Gender:'F'},
          {Name:'Jean', Postcode:'NR7', Citizenship:'FR'} ];
```

and here I've taken the opportunity to show that numbers may be written in Javascript in string form as '22000' or straight as 65000. Use of the string form is a practice which I think should be strongly deprecated! Just use the straight form, please.

3.2 Using the Data Format in the Scripts

JSON is the data definition format of the language Javascript, so I'll use the conventions of that language to show how the various elements of the **test data** can be accessed from the **script**.

The key idea is to replace the *explicit* items in the **test script** by *placeholders* that refer to items in the XML or JSON data, so that *the same* script can in fact refer to *different* items on different runs. The most basic requirement of this is to do the same **script** for each of several users in turn, like this:

Login(lucinda23)
    < *do script as lucinda23*>
Login(xyz31)
    < *do script as xyz31*>
Login(addin44)
    < *do script as addin44*>
etc.

An intuitive and easy construct for doing this would be something like this:

FOREACH(AdminUser IN AdminUserList)
{
    Login(AdminUser)
    <*do script as AdminUser*>
}

Here "AdminUser" is a placeholder for the individual users cited in AdminUserList, and all references in the **script** to the *actual* users such as lucinda23, addin44, etc. should now be replaced by this placeholder.
It's important to realize that "AdminUser" is now *defined* by this expression as referring to each member of the list in turn, and it retains that meaning *only between the brackets* "{" and "}".
The "FOREACH" expression here and the brackets "{" and "}" are technically known as *meta-language*, language that is strictly outside the **script** itself, but describing *how* the **script** should be run. Almost any **test script** uses meta-language, usually in the form of free English text at the end of a **script** saying something like "*Now repeat this procedure for various administrators in turn such as xyz31, addin44, etc.*". What I'm proposing is simply a formalisation of this idea that is in line with common programming practice.

Note that the brackets "{" and "}" must be strictly matched if we are to keep to the possibility of automating any of our **scripts**, so the machine running the automated **script** knows exactly where the bit to be repeated – the "loop" – begins and ends.

This rule must always be observed in any bracketing system, including the possible uses of brackets suggested in the Test Script Format Proposal of Chapter 2.

FOREACH statements could be nested, as in:

```
FOREACH(AdminUser IN AdminUserList)
{
        Login(AdminUser)
        <do preliminary part of script as AdminUser>
        FOREACH(TestUser in TestUserList)
        {
                <do part of script specific to each TestUser assigned to this AdminUser>
        }
        <do cleaning-up and finishing of part of script as AdminUser>
}
```

But this wouldn't be quite how we would approach it using the data format given in the last section. We want to have some way of saying that we want to use the *specific* TestUserList that *belongs* to *this particular AdminUser*. To do that we use the standard Javascript notation of a "." that attaches the TestUserList to this AdminUser. So the **script** above would instead be written as (the change is in the inner FOREACH):

```
FOREACH(AdminUser IN AdminUserList)
{
        Login(AdminUser)
        <do preliminary part of script as AdminUser>
        FOREACH(TestUser in AdminUser.TestUserList)
        {
                <do part of script specific to each TestUser assigned to this AdminUser>
        }
        <do cleaning-up and finishing of part of script as AdminUser>
}
```

This unambiguously defines TestUser as being an element in the particular TestUserList for this AdminUser.

We then replace any explicit reference to items that have themselves been replaced by placeholders by the placeholders *throughout the* **script** *contained in the brackets*"{}", just as I've done with the Login() statements above. So a **script** like:

```
Login( lucinda23 )  //login
Main menu > PeopleTools > Security > User Profiles > User Role Maintenance
ID: ↓begins with: ab
Search  //Take the Search button
```

```
Results(#1:USN)          //opens User Role Maintenance page for first user returned by search
{
        User Roles:#Last: +
        (
                //add a new entry after the last entry, this time using the standard + button
                User Role: ↓Director of Studies
                College/Dept: Lookup(SAC, Santa Anna College)
                Start Date: 22/10/2015
                End Date: 22/10/2024
                //Lookup() is a simple way of defining a routine operation in a compact way
        )
        Save    //return from this component
}
```

might become something like:

```
FOREACH(AdminUser in AdminUserList)
{
        Main menu > PeopleTools > Security > User Profiles > User Role Maintenance
        FOREACH(TestUser in AdminUser.TestUserList)
        {
                ID: ↓equals: TestUser.ID
                Search  //Take the Search button to open up the specific user's Role Maintenance
                        //page
                FOREACH(ChildJob in AdminUser.ChildJobsList)
                {
                        User Roles:#Last: +
                        (
                                //add a new entry after the last entry,
                                //this time using the standard + button
                                User Role: ↓ChildJob.Name
                                College/Dept: Lookup(SAC, Santa Anna College)
                                Start Date: 22/10/2015
                                End Date: 22/10/2024
                        )
                }
                Save
        }
}
```

Here I've put the references to placeholders in the **script** in red, which of course you won't see if you print this document in black and white!

This creates a **script** that is much more flexible, much more comprehensive in its coverage, and lending itself readily to automation. In the same amount of **script**, I've suggested the addition of a User Role for *every one* of the Child Jobs assignable by this user, and the **script** doesn't even need to know what they are.

The explicit reference to Santa Anna College would also most likely be replaced by something like:

College/Dept: *Lookup*(AdminUser.AcadOrg.Code, AdminUser.AcadOrg.Name)

where again I've marked the placeholder reference in red. In the sample XML/JSON above, I actually used a *list* for AcadOrg called AcadOrgList. (The use of a list here reflects the fact that most AdminUsers have access to *several* Academic Organisations.) How would we use this to select just *one* item from the list?
We could use another meta-language form:

```
College/Dept: RANDOM(AcadOrg in AdminUser.AcadOrgList {AcadOrg.Code,
                                                        AcadOrg.Name})
```

This proposed construct would indicate *any Acad Org chosen at random* from the list AdminUser.AcadOrgList. The syntax form is the same as in FOREACH, this code *defining* a new placeholder AcadOrg. So our "new entry" **script** would become:

```
FOREACH(ChildJob in AdminUser.ChildJobsList)
{
    User Roles:#Last: +
    (
        //add a new entry after the last entry,
        //this time using the standard + button
        User Role: ↓ChildJob.Name
        College/Dept: RANDOM(AcadOrg in AdminUser.AcadOrgList {AcadOrg.Code,
                                                               AcadOrg.Name})
        Start Date: 22/10/2015
        End Date: 22/10/2024
    )
}
```

Obviously we'd want to vary the dates too, and for that I'll suggest the special reserved word "Today" as a reference point, so we could use:

```
Start Date: Date(Today)
End Date: Date(Today + 9Y)
```

using a special function Date() to indicate that it's dates we're dealing with here, and a simple **script** syntax to indicate "Today" and nine years hence.

Decision Table logic can easily be handled in this same syntax. So, for example, taking the list of Employees defined from my original **Decision Table** example and re-formatted as at the end of the previous section, we might use this list as:

```
FOREACH(Employee in EmployeeList)
{
        Employees:#Last: +     //assume a panel "Employees" showing a table of employees
        (
                Name: Employee.Name
                Salary: Employee.Salary
                Additions: Employee.Additions
                Postcode: Employee.PostCode
                 Age: Employee.Age
                Citizenship: Employee.Citizenship
                 Gender: Employee.Gender
        )
}
```

Suitable conventions would need to be in place to handle undefined fields as we had in the actual EmployeeList table we used before.

3.3 Getting Data from the System

Commonly we want to select our lists of users or other data from the system *as it is now*, rather than be tied to a predefined set that was valid when the **script** was written.

There are two main ways in which this is done in StudentSys. Either we search for a relevant data set using the search options within the system, most notably StudentSys Search itself, or else we ask Zenobia (if we're stuck ourselves) to write us an SQL script that returns the data we want, save that data somehow, and use it to give us a selection of test items by referring to the saved list by inspection ("manually").

It's possible to save most lists returned by StudentSys Search as Excel spreadsheets using the Excel icon at the extreme right of the *Results* panel header. It's also possible to export the results of an SQL query from SQL developer into an Excel spreadsheet.

So whichever method we use, assuming for now we have no other way of getting these lists, we've got them into a standard format: Excel.

My conversion system has shown that any tabulated data in an Excel spreadsheet can be imported and re-formatted in whatever way we wish. So there would be no problem writing a VBA script that could take such externally defined data from an Excel file and convert it into either the XML or JSON formats that we could use for our **test data**.

But we could do much better than this: we could cut out the middle man – Excel. To do so we'd have to have **scripts** that could be run automatically that *either* ran a StudentSys Search or ran an SQL Query directly, and we could then bring in the data *direct* from the source into our test environment. To set up such automation from the scripting format I'm proposing we'd need to be able to run our **scripts** through Selenium, and I'll devote the whole of the next chapter to a discussion of how this could be done. Basically the **scripts** would be converted by a wholly automatic conversion procedure, the first part of which I would need to write, but the second part of which is already available in Selenium, into a Java or C# program which could both see what the browser sees and so run StudentSys through the browser automatically, and also connect to the database directly and so run SQL scrips directly on the server.

I would expect, and I would also suggest that this is an essential next step for the facilitation of testing in QRCIS *however else it's decided to go*, that a repository of such automatic **scripts** is gradually built up. They don't have to be based on this proposal: such a repository could be made available working directly in Selenium. This would be a vast saving in effort for the test team and would enormously improve their effectiveness and speed of delivery. The reason is that it is very similar lists of data, generally lists of people on StudentSys fitting certain criteria, that are needed *again and again* in testing. To have such a set of automated **scripts** available as a permanent and ever-expanding feature of StudentSys would be an enormous boon.

I'll just say a little more about this idea before resuming the discussion of how it could be used: an obvious way, but one that will probably be regarded as quite outside consideration, would be actually to have a special test navigation area built in to StudentSys. So whatever lists might be needed as a prerequisite to a particular **test script**, testers would know how to go in to the system, navigate to a special testing page, and request the data they need, which would be returned in a guaranteed correct and fully up-to-date form in any standard format that would conform to use in our test procedures. Built-in to the

request page would be a checkbox for any lists of StudentSys people requested "Set all to standard password □" and maybe a box to define that password for this test run. This would save a huge wasted effort in repeatedly going in to **User Profiles** to do this for each of maybe dozens of users in turn.
One level of indirection from this would be, as suggested above, to have an external Selenium script running under Java or C# (although it would appear to the tester simply to be another application tool on their desktop) that acted as a driver of StudentSys, logging in to the system as perhaps a special newly-defined user and using StudentSys's built-in functions to get the list required. This could still incorporate the automatic password-reset option as a Selenium can call up anything in StudentSys.
The third level of development would be to incorporate SQL searches into the Selenium application (it wouldn't necessarily need to use Selenium to do this as both Java and C# have full support anyway for database access) to abstract data that cannot readily be returned by the ordinary StudentSys interface.

Incorporating such an extension into my scripting data format couldn't be easier. I would propose two new terms to define out-of-system and within-system data access respectively. The syntax:

```
SQL.GetCollAdminUsers([SAC, DUC, JN], passwordTo:'zoggledutch27')
```

could imply going in to the SQL driver application to call the pre-defined SQL query GetCollAdminUsers() (which Zenobia would most likely be the one to define) for the given list of Colleges and resetting the returned College Administrator's passwords to 'zoggledutch27'.
To conform to my earlier suggestion for a standard data format, the result of this would need to be given a placeholder label, so that we'd have in full:

```
AdminUsersList = SQL.GetCollAdminUsers([SAC, DUC, JN], passwordTo:'zoggledutch27')
```

Such a call would only return a predefined set of properties for each Admin User, like ID, USN and Role. But we could use the FOREACH logic in an automated **script** to populate the extra fields like AcadOrgList, e.g.:

```
FOREACH(AdminUser in AdminUserList)
{
        AdminUser.AcadOrgList = SQL.GetAcadOrgList(AdminUser)
        AdminUser.ChildJobList = SQL.GetChildJobList(AdminUser.Role)
        AdminUser.OrgRolesList = SQL.GetOrgRolesList(AdminUser)
        AdminUser.TestUserList = SQL.GetTestUserList(. . . , . . . , . . .)
}
```

where "(. . . , . . . , . . .)" indicates various parameters that might be chosen peculiar to this test application.

Such calls could be placed in an initial block like this, or wherever in the code they're needed. In this I've adopted the Javascript convention that arbitrary subsidiary elements can be attached to any given element "on the fly", a convention that could be very useful for our proposed language.

I would suggest a distinct notation to indicate calls to automated Selenium *scripts* (as opposed to SQL queries) simply for clarity. So in the Selenium script version, our call above might become:

```
AdminUsersList = SELENIUM.GetCollAdminUsers([SAC, DUC, JN],
                                                    passwordTo:'zoggledutch27')
```

where the prefix "SELENIUM" is used to distinguish such calls from their SQL counterparts.

3.4 Using the Data in Testing

[Section largely omitted.]
Of course you can do this sort of thing after a fashion in Excel. But the big difference in using XML for **test data** and test progress recording is that it is *well-structured*. Everything is correctly assigned its place in the hierarchy.

If you did use the XML **test data** as a test notebook in this way, do make sure that at the start you save it under a *distinct* file name! You'll need to keep the original intact.

You could also add special nodes to indicate things like:

- Pass: YES
- Issue: #23882 - Failure to show dropdown.

Let's now suppose we're running through a **test script**, working in TestRail. TestRail, as do all our possible candidates for a test Management System, supports adding custom buttons to assist the way we work (see http://docs.gurock.com/testrail-custom/automation-trigger, or for analogous methods for Zephyr, http://developer.yourzephyr.com/zapi/index.php). I would propose adding a button labelled something like **Open and Expand Test Data**, looking a bit like one of the special buttons I used in the VBA conversion program in chapter 1.

The way we'd work would be like this. We browse to our chosen **test script** using whatever navigation options we've added to the system – **scenarios** or any other. The **script** opens with the first **test step** displayed. But before we run it, we need to open up the associated XML file (or JSON) that holds the **test data definition**. Because we will configure TestRail to keep this file associated with the **script**, we just click on our custom button **Open and Expand Test Data**, and the code triggered by this button will know where the XML (say) file is, and open it.
The custom code now zips through the file looking for any nodes in it involving calls like:

SQL.GetCollAdminUsers([SAC, DUC, JN], passwordTo:'zoggledutch27')

(Yes, we can have these in XML as well as JSON, as shown below, although this would of course be a somewhat unlikely place to put this particular **query**!)

If it finds any, it connects to CS_TEST or whatever the defined test environment should be (either defined explicitly in the **script** or set either when we open the **script** in TestRail or even on discovering a "SQL." node as we've just done – it would only need to do it for the first one it finds).
The custom code then calls the special **query** SQL.GetCollAdminUsers() with the given parameters, and replaces this position in the XML file *with the data returned from the query*, correctly formatted as XML.
When it's found and processed all such nodes, which may take a few seconds, StudentSys being the way it is, it's now converted the XML **test data definition** into an expanded XML file with the *fully up-to-date* data as required for the **test**.
It would now offer up a "File Save Dialog" to ask where you want to save this file *as being the defining state of the* **data** *when the* **test** *was actually run*.

Once the state of the **data** that will be used for this **test run** has been saved and this file's location recorded in TestRail, the custom code would open the new XML file in XML Notepad and testing can begin.

If you wanted to make notes in the XML file as you go, then you'd simply take the *File* → *Save As* option from XML Notepad's menus. You might need to make a note of this "notebook" copy independently in TestRail in the record for this **test run**, but that's going to be quite easy.

As the tester worked through the test, they would simply refer to the XML Notepad listing for the relevant *actual data* whenever they encountered a placeholder like:

- AdminUser
- TestUser.ID
- ChildJob.Name
- AdminUser.AcadOrg.Code

to take some from our previous listings, and replace the placeholder with the current actual value in performing that **test step**.