ceLTIc developers — Creating Environments for Learning Through Instigating a Community of Developers

A JISC-funded project

LTI saves lives

A case study of how e-Reflect was LTI-enabled

I've recently been involved with adding LTI support to an application as part of the JISC-funded e-Reflect benefits realisation project and thought that the process I went through might make useful reading, for both application providers and LTI developers.

One of my alternative titles for this case study was "*A day in the life of an LTI developer*". It is not far from the truth that the adaptation took about one day's effort, though the elapsed time was rather longer as you will see below. But the case study will use the life saving metaphor instead...

Contents

The symptoms

e-Reflect is a well-received tool but its adoption is hindered by the need to host the system and provision it with user accounts. This is a common scenario, especially with applications which may only have pockets of interest within institutions - it is similar to that addressed with WebPA as part of the JISC-funded ceLTIc project.

The diagnosis

The hosting issue can be overcome by enabling e-Reflect to support multiple institutions. The provisioning issue can be addressed by having user accounts created "on-the-fly" when they first connect from a trusted source. The IMS Learning Tools Interoperability (LTI) specification is an ideal remedy for both issues and so was prescribed for this case.

The treatment

Surgery is always risky, the last thing you want is for any treatment to cause new problems or lead to unwanted side-effects. So the main objective of this first treatment was to make the minimum of changes to the patient application in the hope that it also carried on working as before.

Getting to know the patient

Before embarking on the treatment it is essential to understand the patient. In the case of e-Reflect this involves a knowledge of

- the application and how it is used;
- the technology (C#, .NET, Visual Studio, SQL Server);
- how the source code has been wired together.

This was a bit of a challenge for me as I had never programmed in C# or .NET before, or used Visual Studio. However, I was familiar with SQL Server and had access to Google! I had experience of similar operations using different technology which was perhaps more important. Anyway, adapting an existing program is always easier than starting from scratch and any limitations can be blamed on the pre-existing conditions!

In fact I found e-Reflect to be well written and structured, so it was easy to follow and I almost enjoyed using C#, it has some really nice features.

Assembling the required equipment

Here's a list of the things I thought I would need:

- a .NET library for C# to verify OAuth signatures;
- any re-usable C# code for handling LTI launch requests;
- an LTI Tool Consumer emulator to allow test launch requests to be thrown at e-Reflect;
- a plan for how to perform the operation.

So, taking each of these in turn...

OAuth library

I have always used an OAuth library in my LTI projects with other programming languages and there are some available for .NET (check out the OAuth website). However, the ones which looked most promising were for .NET 4 and e-Reflect uses 3.5. Perhaps with more .NET experience this would not have been an issue, but instead I started looking for an alternative solution.

Re-usable code

This look elsewhere took me to see what other LTI code there was available for .NET. I found two: by Niall Barr and Andy Miller. Both the former, and an earlier release from the latter, coded the OAuth signature check directly without using a library. So for the purposes of this project, I saw no reason to look any further. As far as other aspects of the LTI integration are concerned, both sources were really just examples and not designed to be class libraries which could be plugged into other applications (unlike the PHP and Java class libraries created by the ceLTIc project).

Tool Consumer emulators

The only true way of testing an LTI Tool Provider integration is with a Tool Consumer emulator – a real Tool Consumer (such as a VLE) does not allow all the edge cases to be tested to ensure that it copes with all combinations of launch parameters. There is a dummy LMS application available on the IMS website, but a comprehensive version was written to accompany the ceLTIc:developers workshops. Clearly I am biased, but since it provides access to all the possible LTI 1.1.1 launch parameters, this was my preferred choice. However, in recent weeks an LTI Launcher for Windows has been released; whilst this does not allow control of all the launch parameters, it does provide a simple way of launching from different resource links and user accounts. So both were placed on my surgical tray.

Planning the operation

A plan is always good to have, especially if lives are at risk. But, as a developer, it is so tempting to just jump in and rummage around. Fortunately, I have previous experience with this type of surgery and so hopefully that helped to mitigate any risks. Having said that, I did have an outline plan; the process involved two main tasks:

1. the surgery: insert the LTI bypass;
2. the aftercare: managing the new bypass.

The operation

Inserting an LTI bypass can be a tricky operation, involving a number of important decisions:

- should the launch endpoint be a new URL or added to an existing one?
- how should contexts, resources, users and roles be mapped from the LTI launch into e-Reflect?
- what unique IDs should be used to cross-reference the different objects?
- which launch parameters should be required , optional and ignored?

Whilst the prime objective was to make as few changes as possible, using a new endpoint should have fewer side-effects. I started, therefore, with a new Launch.aspx page to which LTI launches were directed. This script was built up step-by-step; firstly, verify the authenticity of the launch request:

- check the HTTP POST parameters are being received;
- confirm that the request is an LTI launch (e.g. look for the `lti_message_type` parameter)
- check the OAuth timestamp is within a permitted margin from the system time;
- check the OAuth signature (using a hard-coded secret for now);

If the request is not valid, then the user should be redirected to a page with an error message. If a `launch_presentation_return_url` parameter is received, then the error message is returned via this URL (in the `lti_errormsg` query parameter), otherwise a new error page is used. Redirecting to the normal home page containing the login form is not very helpful here as LTI users will not have credentials to gain direct access. I also added a very general, apologetic message to the error page – the precise reason for the problem is unlikely to be of interest to a normal end-user. (In the PHP and Java class libraries, we used a custom parameter (`debug=true`) to indicate when more technical error messages should be displayed; but this has not been implemented here, yet.)

Once you have a valid request, the next question is what to do with it. This can involve mapping data in the launch into their equivalent objects in the application. In the case of e-Reflect, I took the following approach:

- *contexts and resource links* – e-Reflect is currently written such that users see the entire set of questionnaires when they log in; there is no association with courses. For now, therefore, these concepts are ignored and the current approach is continued, but this will be reviewed later.
- *users* – since there will not be any contexts or resource areas within e-Reflect, a single system-wide user account is all that is required for each user connecting.
- *roles* – e-Reflect has roles of Admin, Editor, Tutor and Student, and users may have one or more of these roles. The typical roles of users on the other end of an LTI launch request are Instructor, TeachingAssistant and Learner, but could also include ContentDeveloper. For now, Instructors, Teaching Assistants and Content Developers are mapped to being both an Editor and a Tutor, and Learners are mapped to Students. This may cause issues for a user who may have both Instructor and Learner roles in different courses in the VLE, but should be good enough for now. (The role of Admin will be left for direct logins only.)

The ID of a user can be uniquely defined as being a combination of the originating consumer key and the `user_id` parameter. This means that the `user_id` parameter will be a required parameter for this implementation, even though it is not required by the LTI specification. Since the e-Reflect database uses the username as the primary key in the `users` table, the simplest solution was to just concatenate the two elements to form a user ID, rather than add the consumer key as a separate column in the table. Either solution should work, but the former seemed like the least invasive to the existing code; the only change made was to increase the size of the `username` field as they are likely to be longer strings. The user ID value is also used for naming report files so the two elements were combined using a separator character of a backslash ("\"); this provides a neat way of separating the files for each consumer key into their own directory. Thus, a user having an ID of "`auser`" in the tool consumer assigned a key of "`testing.edu`" would be given a username in e-Reflect of "`testing.edu\auser`".

Processing a valid launch request is a case of assembling code which most likely already exists in the application. If you're lucky then this code is also encapsulated in useful class methods which can just be called, otherwise some copying of lines of code may be required. Fortunately for me, e-Reflect has been well written using classes. The code added performs the following tasks:

1. create an account for the user if one does not already exist;
2. update the user account with the name and email address passed on launch;
3. destroy any existing user session (in case they failed to logout from a previous launch);
4. establish a new user session, saving any return URL in the session for use on logout;
5. redirect the user to the appropriate entry page.

My normal practice is to use default values for a user's name and email address so as to avoid making these required launch parameters. If a VLE chooses not to pass the real name of its users, that is its choice; it should mean that we cannot offer a service. For example, a default name for a user with an ID of "`auser`" can be the imaginatively inventive "`User auser`". A default email address could be a standard "no-reply" address. However, e-Reflect makes this issue even easier; if a user's profile is not complete, they are forced to complete it when they log in. For some reason, the profile form did not include their email address, but this was simple to add. Since e-Reflect requires every user to have a title (which is not available via LTI), the profile page will be displayed anyway, so entering any other missing information can be done at the same time. Of course, at any time the data is received on a launch request it is used to override any existing values; the tool consumer is taken as the most authoritative source.

To complete the lifecycle of a user session, I like to handle logout events by returning the user to the system from which they launched, if a return URL is provided. Hence the reason for keeping a note of this URL in the user session. If no URL is supplied, then I will redirect the user to a new logout page with a simple thank you message so as to avoid displaying the login form (similar to the error page).

In e-Reflect, the logout request is handled by the login script (this may sound odd, but makes perfect sense when you look at the code). So to add the logout, I had to alter the login page. If I'm going to have to change this page anyway, and given the small number of lines required to handle a launch request, then I thought it would make good sense to move the code from the new launch page into the login page. So this is what I did. This means that the launch endpoint is essentially the same URL

as for the normal login page – one less URL to be confused about! Not a big deal, but keeps things simpler and tidier.

Just before resuscitating the patient, a comprehensive range of health tests was undertaken to ensure all is well. For example, launching:

- without all the required parameters;
- with an out-of-date timestamp;
- with an invalid consumer key;
- using the wrong secret;
- when the consumer is not enabled in e-Reflect;
- with and without the name and email parameters;
- with international characters in the user's name;
- with different combinations of roles, including different roles for the same user ;
- without logging out from the previous launch.

This is where the tool consumer emulator applications came in really handy.

When the application is brought round after this operation you should find that users can now connect into e-Reflect from any VLE without the need to have a pre-existing account in the system. Well, almost!

The aftercare

You may remember that my initial code for checking an OAuth signature was using a hard-coded signature. Clearly this is not acceptable; the patient needs a way in which they can manage the issuing of keys and secrets to interested parties. It would be a huge security hole if any key was accepted and the same secret was used with all keys. This is something which e-Reflect has not had to concern itself with in the past, so will require new code. However, it is not dissimilar to the management of user accounts, so this code can be copied and adapted to provide a simple, consistent solution.

For each tool consumer I chose to record the following data:

- a human readable name for the party;
- consumer key;
- secret;
- enabled status.

A new table named `lti_consumer` is added and all the fields are required. The naming of the table and the fields is arbitrary but I have used the same names as for the PHP and Java class libraries – one day we hope to have an equivalent library for C# so consistent naming will make it easier to adopt. (The class libraries include additional fields for setting from and until dates when a consumer key is available, recording details of the tool consumer, etc.; these have not been included in the current implementation for e-Reflect.) This table can now be used to look up the secret for a consumer key from an incoming launch request, and also check the enabled status (if the consumer is not enabled launch requests are declined, thereby providing a quick and easy to disable access to an existing consumer).

At the bottom of the admin page a table has been added to list all the currently defined consumer keys, with modify and remove options. The input form above it also allows new consumer keys to be created. This works in the same way as for managing users.

One beneficial side-effect of the operation is that the patient is suddenly transformed into a multi-institution system; users can connect from any registered tool consumer. However, this is not without some side-effects:

- the admin interface will display users from all tool consumers in a single list;
- all questionnaires will be visible to all students, irrespective of which tool consumer they launched from.

The antidote for these was to update the admin page to allow a tool consumer to be selected from a drop-down list (or a default option of "Direct logins") which is then used to determine which users are listed on the page. Since only users from a single tool consumer are displayed, the prefix of the consumer key is removed before displaying the username, to make for a cleaner interface. All other database select queries in e-Reflect were updated, where appropriate, to limit the records returned to those belonging to the same consumer key as the user.

Case review

It is imperative for any professional organisation to conduct a review of work undertaken to ensure that it was properly conducted, achieved its objectives and has not caused any unwanted side-effects. This is currently being undertaken independently of the developer by experienced users who can test the different use cases for which the system was designed. Any recommendations arising from this case review will be implemented to both repair any damage caused to this patient, and to improve the outcomes for any future patients requiring similar treatment.

In the meantime, a technical review identified the following issues for further consideration:

- keep a record of OAuth nonce values so these can also be checked as part of validating a launch request;
- adapt e-Reflect to associate questionnaires with courses and use the LTI context to limit the list displayed;
- consider having a one-to-one relationship between questionnaires and resource links; to create a new questionnaire an instructor would add a new launch link to their course, launches by all users would then go directly to the associated questionnaire only.

There are some who might say that e-Reflect was not dying, so LTI did not save its life. To them I would say that e-Reflect, like many other new and innovative applications, can easily be asphyxiated from a lack of wide adoption. In the long run, therefore, this LTI bypass may well have helped to save the life of e-Reflect. Its rejuvenation has made it easier for others to use from which a critical mass of users can be built to take good care of it into the future.

Stephen P Vickers
6 May 2013
www.celtic-project.org