Permission Analysis Layer

A dynamic Permission analysis layer

2nd version

By: Capt. Gal Kahana

Table of Contents:

1. Introduction

The DAIL/PAL methodology introduces an XML based approach to building the lower levels of an n-tier IT application. First there's a Request/Response XML standard. Then there's DAIL component that address the problem of accessing data and manipulating it. This essay deals with a component that deals with permission handling.

What do I mean by permission handling? I'll start by saying what I don't mean - authentication. There are numerous methods of authenticating a user identity for applications and they are not discussed here. It is taken for granted that you have a decent enough method to get the user identity.

Permission handling is about when you know who the user is and you wish to limit the user ability to perform actions on certain items. We would like the limitation to be able to utilize all data available both on the user, and the item/items for its purposes.

An example for permission handling may be: we have an application for buying cucumbers via the internet. We have different types of cucumbers: green, extra green and green with bits of mud on them. We sell the extra green only to our gold members. John is not a gold member but he orders 3 kilos of green cucumbers and 2 kilos of those delicious extra green. The application should be able to either: allow the buy of the 3 kilos, but deny the extra green buy or simply tell john that the whole action is canceled altogether. To implement The first response (allow some but not all) we have to see that the order has some permitted items and some that are not, and then send modify the request so that only the permitted items are allowed. To implement the second response we should be able to tell from the request that there exists in the order a special type of items.

In DAIL/PAL methodology this is easily done by adding to the profile section of a request another profile parameter that allows only cucumbers that are green to be bought. This is one of the functionalities of PAL - Permission Analysis layer.
PAL is a layer that business objects use before sending a request to DAIL to limit the request so that it fits the user permissions or to deny it altogether. It is also used to censor data that has been brought from DAIL before sending it back to the user. A third ability of PAL is to retrieve data regarding the user permissions in regard to a collection of actions on an item type.
The following sections will discuss what are the requirement from PAL and how are they achieved.

A note for the 2nd version:
This version uses a new concept - a transformer. The transformer is used to transform the requests to raw data that data storages can use. The 1st version forced the use of XSL files for transformation, which denied the ability to cache a transformation and required IO. It also enables he user to create a less sophisticated method of transformation, that is cheaper then XSL, for certain usages. For a discussion of

Transformers and the ITransformer interface please refer to the "DAIL Component for effective data access" document.

2. Issues when handling permissions

A user must not be allowed to perform an action she's not allowed to perform. That part is obvious. Most applications solve it by using a single method that checks the action and the identity of the object on which it is done and by using a role – based method determine whether the user has a role that's allowing that action on the type of object. This works well on a screen-based application – meaning that a user is allowed to enter a screen if she has enough permissions – and not allowed if not.

Applications may require a bit more then that. The list of cases here describes more functionality that is required:

- A user can perform an action on some of the items of a certain type, but not on others, and with a certain set of parameters or another
- A user can see some properties on some items, but may not see them on others – this is different from the above, because at the time of performing a retrieve action for a certain profile of multiple properties, simply not retrieving those fields for the not allowed items, may not be possible – because its too complicated and may involve multiple retrieve sets, which is undesirable in a performance oriented application (most of them are, aren't they?)
- An application may want to perform a query on the permissions by handing a given profile of items and asking whether some actions (with some parameters) are allowed or not, or if the action is allowed on all items, regardless of the profile. (the letter is normally what is required by apps, the rest of the cases should offer new functionality that will enable expending this case)

As an example to the required complexity the application I've been working on recently has offered apart from a role-based security check, also an organization hierarchy permission test – meaning that a user was able to perform some actions only on the items that were below her in the organization hierarchy. And much more – I could even use the same method for doing private data permission testing – just by issuing a new kind of test, checking whether the item belonged to that user.

Definition: a "test" or a "permission test" refers to a single check of the user properties or any other state property in regards to the query. A combination of tests yields a final answer to the ability of the user to perform the action.

The next paragraph will expand on each of those requirements with reference to how requests/responses are handled with DAIL.

3. Expanding each requirement

"A user can perform an action on some of the items of a certain type…"

An action according to the Request/Response methodology is determined by the id of an item type, the name of the action and the parameters: what (Data) and who (Profile).
A user should be either:

- Disallowed from the action altogether because she just doesn't have enough permissions to perform this kind of an action on the certain kind of item
- Allowed to perform the action only on some of the items but not on others.
- Allowed to perform the action on all items.

This may be expended further to another capability:
If the user is disallowed from the action altogether perform more tests to see if the user is allowed to perform the action with a certain limitation and allow her to do that with that limitation.

"A user can see some properties on some items, but may not see them on others…"

A data retrieval action retrieves a certain data set. This data set may have to be censored, so that the user will not be able to see properties on the items she's not supposed to see, but should be able to see other properties, or should be to see other properties on the above mentioned items.
"Why not avoid getting them in the first place altogether?" you ask.
Well, the justification for that is all around us: why do we need censorship? Can't we just tell the people not to write what's not to be seen by whom they send the things too???
Well, no. Think of what is simpler:
Method A: out of the query dig out the many possibilities of different queries (as determined by the tests done to determine the user ability to perform the action) – some queries will use a certain profile and will retrieve some set of properties; some will use another profile and retrieve another set of properties. Later those sets should be merged to a single response. Assume that you have n permission tests, this will probably yield 2^n sets. The merge is also painful.
Method B: lets it retrieve whatever it wants. Later for each set of fields that should be seen only for a certain profile of items, retrieve that profile and for each of the items that are not in that profile censor the fields. This will result in n iterations for n tests.

So I like method B better. You may say that I have a certain cheat in my calculations – because method B requires an actual pass on the items. Well, so does merge which is used in method A. so all in all I'll always get a better performance when working in this method.

"An application may want to perform a query on the permissions…"

Let's say you're an application. You want to know if the user is allowed to press on one of you buttons. Better yet, you want the user to be allowed to press on that button only if she points to an item which belongs to a subset of items out of a table that you display. How do you get that?

That's Simple. There should be a method for querying for a certain list of actions, on an item type what are the allowed actions, or even easier – what are the actions allowed on that item type, without a given list of actions. Even more – I may want to add a simple profile indicating a certain list of items, and I'd like to know for each item what actions out of the list I gave are allowed.

4. PAL approach

PAL assumes that each application has its own tests that it applies on a user request in order to limit it or on a response in order to censor it.
A test provider varies in the parameters it receives but it may return one of 3 results:

1. An "ABORT" signal. This means that the user fail this test. Usually this happens when a user does not have a certain role – in a role test.

2. A limiting profile. This means that the user is allowed to do actions on items with a specific profile parameter. A limiting profile describes via the profile parameters it returns the range of items the user is allowed for. For example – a hierarchy test that allows an action to be performed only on items "under" the user in the hierarchy tree generates a string similar to
<Organization>
<Item>*userOrganiztion*</Item>
</Organization>
as its result. The string tells that the profile of the request should be limited to items that which their Organization profile parameter is equal to "*userOrganization*" which is replaced by the user organization. Notice that this kind of a test does not really have to know what the type of the item is to generate the string, only the details of the user.

3. An empty signal. This means that the user passed the test with no limitation.

When combined together, the results of a series of tests tells us how to limit the request in order to express the permission behavior for a certain type of an action on an object.

A business logic object passes a request to PAL to limit it. Apart from the request the object also passes a transformer class that creates a list of tests that should be made based upon the request parameters (much like the transformation class that transforms XML to SQL in DAIL). A test is recognized by a label (again, much like data sources in DAIL) and refers to a test class. The test class has been created by the application programmer who also built a method through which PAL may receive a class by handing a label. The tests are now being performed. If one of them aborts, then the response from PAL will be that the action is disallowed. If some of them returned limiting profiles they are collected and added to the request and the request is returned to the business object to be performed instead of the original request. It may also happen that nothing is modified in the request, if all tests passed with empty signals. The business object may now choose to either respond that the action is not allowed, or to continue with the new (modified or unchanged) request.

Censorship is very similar. A response and a transformer are passed to PAL. This time the transformer creates out of the response a list of entries of fields and tests. Each entry of fields and tests stands for conditions that the listed fields should appear in an item. The tests are performed, and for each item in the data set that does not fit the resulting limiting profile the fields are censored. When the tests fail completely the listed fields are removed from all items. When the tests result in an empty profile, nothing is done.

The last action - retrieving permission data - is very similar to doing a request limitation many times. It will be discussed later on.

5. AbstractPAL class

AbstractPAL is the main class in the PAL library. The class has a method for each of the above described requirements. Clients should subclass this class, and implement a method that returns a test class for a label (DAIL rings a bell?). The methods of AbstractPAL are hereby described:

5.1 getRequestLimitation method

The getRequestLimitation method signature is:

```
public string getRequestLimitation(
      string strRequest,
      ITransformer transformer,
      string strID,
      string strAction)
```

This method is used before a request is performed. It accepts the request, a transformer that describes a list of tests, an item type id and an action.
The id and action does not have to be the same as the id and action of the request itself. For example, assume that the request action is "changeAddress" or "removeDriverLicense". Both actions relates to a modification action. If the tests are the same in modification actions, one may pass "modify" in strAction to signal that.
The method returns a modified request, if a limiting profile was collected, or the same request, if no test generated a limiting profile. It may also return an abort string, namely "ABORT" if one of the tests failed completely.

The transformer is an object that can create out of a combination of the request, strID and strAction a series of tests. The XML that the object should transform looks like this one:

<PermissionRequest id=*strID* action=*strAction*>
    strRequest
</PermissionRequest>

(The words in italics should be replaced by the matching input variables)
The transformer transforms this XML to a list of tests, in a certain XML format. A template to that XML looks like this:

```
<Tests>
        <Test id="testID">
                test specific content
        </Test>

        <Test id="anotherTestID">
                test specific content
        </Test>
        ….
</Tests>
```

Each "Test" tag in the XML signals a single test. The "id" attribute of the "Test" tag is the label passed to the method implemented in the sub-class and returns a test class. The test class is given the content of the "Test" tag as parameters.

The algorithm of the getRequestLimitation is:

- Create an XML string which is a combination of strRequest, strID and strAction
- Activate the transformation on the XML
- Loop the created tests:
    - Get the test provider labeled by the id attribute
    - Call the getLimitation method of the test provider with the "Test" tag content
    - If the result of the method is abort - abort the loop and return "ABORT"
    - Else append the result (may be empty!) to a collected limiting profile
- If a limiting profile has been collected (and non-"ABORT") append the string collected to the profile of the request and return the new request.
- Otherwise return the original request (if the limiting profile is empty) or "ABORT" (if "ABORT" was returned from one of the tests)

As you may have noticed, the whole concept of "limiting profile" relies on the fact that concatenating a profile parameter (such as "Organization") to a profile string creates an "AND" relation with the rest of the profile - this is supported by the Request/Response standard.
One may receive the collected limiting profile after running the getRequestLimitation method by getting the recentLimitingProfile property of AbstractPAL.
Last thing, which actually is true for all methods and not just this one, is that a test may fail (an exception). In this case XBLIP.PAL.TestProviderActionFailedException exception is thrown with the exception that had failed the test.

5.2 recentLimitingProfile property

After executing getRequestLimitation it may be required to be able to tell the accumulated limiting profile that was appended to the request (by itself). recentLimitingProfile property will have that value.

5.3 censorResponse method

The censorResponse method signature is:

```
public string censorResponse(
      string strResponse,
      ITransformer transformer,
      string strID,
      string strAction,
      CensorPolicy censorPolicy,
      IItemsListRetriever itemsRetriever)
```

censorResponse may be used after a retrieve request has been performed to censor the retrieved data set fields that the user is not allowed to view. The method removes or marks fields that should not appear for certain items in the data set.
strResponse is the retrieved response string. Transformer is an object that transforms the response (combined with strID and strAction, in the same form of getRequestLimitation). censorPolicy is an enumeration value which may be one of:

CPRemove - when censored, the censored field tag is removed from the response

CPClearMark - when censored, the censored field content (text or sub nodes inside the field tag) is emptied and an attribute named "censored" with the value of "1" is added to the field tag

CPMark - when censored, an attribute named "censored" with the value of "1" is added to the censored field tag

The last parameter - "itemsRetriever" - will be discussed later.

When censoring the transformer should transform the response to an XML that describes groups of fields and tests. Each group stands for a censorship behavior. The tests result for each group tells for which item in the retrieved data set, the fields are allowed. For those that are not allowed according to the tests - either because the tests returned "ABORT" or they were not in the limiting profile created be the tests - the listed fields are censored, according to the policy.
The form of the XML that the transformer should create is in the following template:

```
<Groups>
      <Group>
            <Tests>
                  <Test id="testID">
                        test specific content
                  </Test>
                  <Test id="anotherTestID">
                        test specific content
                  </Test>
                  ….
            </Tests>
```

```
            <Fields>
                <Field id="fieldTagName"/>
                <Field id="anotherFieldTagName"/>
                ….
            </Fields>
        </Group>
        <Group>
            <Tests>
                ….
            </Tests>
            <Fields>
                ….
            </Fields>
        </Group>
        ….
</Groups>
```

The XML may include any number of groups to describe censorship behaviors.
It is recommended that the transformer will create censorship groups only for fields that do exist in the response, to save time.

The algorithm of the censorResponse is:

- Create an XML string which is a combination of strRequest, strID and strAction
- Activate the transformer on the XML
- Loop the created groups:
    - Get the limitation generated by the tests under the "Tests" tag
    - If the limitation that is returned is ABORT censor the fields indicated under the "Fields" tag from all items in the response
    - If the limitation is empty, leave the response as is
    - **If a limiting profile is returned, create a "Retrieve" request and use itemsRetriever to retrieve a list of the items which the limiting profile represents**. For each item that is in the response and not in the returned list, censor its fields according to the fields indicated
- Return the censored response

The bold section of the algorithm refers to "ItemsRetriever". ItemsRetriever is a class that implements the IItemsListRetriever interface. This interface implements a single method:

```
string retrieveItemsList(string strRequest)
```

This single method should know to accept a request string of the following form:

```
<Request id="strID" action="Retrieve">
        <Data/>
        <Profile>
            Items profile
        </Profile>
```

</Request>

Where strID is the item id passed to censorResponse method and under the profile tag appears a profile valid for that item (according the profiles generated by the tests).

The response from that method is a Response string which lists the items (empty "Item" tag with only the id attribute) that are in that given profile. Using this response AbstractPAL can determine what items require censoring, when limiting profiles are returned from tests.

While this seems to be non-trivial, it is very possible that the same business object that handles the entity may be already (in one of its private methods) implementing the above method. It is strongly suggested that if one entity object does have a retrieve method and wish to use censorship via PAL it will have a private object that will be creatable only by the entity object and will implement this interface by performing the retrieve. This way both the censorship method, and the entity object will be able to use it to perform the actual retrieve of the data.
DON'T USE THE ENTITY OBJECT OWN RETRIEVE METHOD (IF YOU HAVE ONE), so that the security check will not be made - again.

5.4 retrievePermissions method

The retrievePermissions method signature is:

```
public string retrievePermissions(
      string strRequest,
      ITransformer tranaformer,
      IItemsListRetriever itemsRetriever)
```

The getRequestLimitation method mentioned earlier is helpful when a request is being made by limiting (or denying altogether) the execution of the request. Sometimes we would like to know whether an action (or actions) is possible before executing it. This is mainly important to GUI display. One may use the retrievePermissions method for that. By calling this method, the application can get a list of items (in a Response standard), and for each item whether an action is allowed. The general algorithm of the method is that for each queried action a dummy request is built and using getRequestLimitation algorithm it is determined what items are allowed for this action by the calling user.
The first parameter to retrievePermissions is a Request string of the following form:

<Request id="*itemTypeID*" action="PermissionRetrieve">
<Data>
<ActionPermitted>
<Params>
<Param id="Action">*actionID*</Param>
<Param id="Data">
specific data section
</Param>

```
                    </Params>
                </ ActionPermitted >
                < ActionPermitted >
                    ….
                </ ActionPermitted >
                ….
        </Data>
        <Profile type="IncludeAll">
            item type specific profile
        </Profile>
</Request>
```

The request asks PAL about the permitted actions of a user on a certain list of actions (with certain parameters).

The "id" attribute of the Request tag identifies the item type id (for example: "employee"). It should match the entity id of the entity on which the request is being made. The "action" attribute is always "PermissionRetrieve". This is done so one may aggregate this query as one of the queries allowed on the entity.

Under the "Data" tag there are "ActionPermitted" tags. One exists for each action which we would like to check. The "Action" param that appears under this tag holds the action identifier. This should match the "action" attribute of a matching request for this action. The second param - "Data" is an optional parameter (this tag may not appear) and in it one can describe a typical "Data" section that is included with a request for that action. Using this param enable a test with specific parameters for this action. This is good when certain parameters for a request of that action trigger one set of tests normally, and another set of parameters trigger another set. The content of this tag are the internals of the tested "Data" tag - meaning without the enclosing "Data" tag.

The "Profile" tag in this request is optional. When it is not used, the response string will hold a single item with the id value of "All". In this tag there will be a list of actions that do not trigger "ABORT" - which means that on some items they are possible for the user. When a "Profile" tag does exist, its profile parameters should match the tags that are possible for the actions listed in the "Data" tag, and for a "Retrieve" action. This time, the response string returned from the method will hold a list of all items described by that profile, and for each the list of permitted actions will be returned. The Profile is used for building the dummy query for each action, and also for retrieving items according to each action permission profile. If one wish to also include an "All" item in the list, it is an option to add an attribute named "type" with the value of "IncludeAll" in the "Profile" tag.

The second parameter to the method is transformer. This transformer object is very similar to the one used for getRequestLimitation, only that it should be able to handle all the action ids given in this request. Choosing XSL as the transformation method and Having xsl:choose…xsl:when in the XSL should do the trick, for example.

The third parameter is an IItemRetriever class, same as the one used for censoring. This class should know to accept a Retrieve request with the profile parameters introduced in the "Profile" tag of the request, and also the ones that may be produced by the permission tests.

The XML string returned from this method is of the following template:

```
<Response id="itemTypeID" action="PermissionRetrieve">
        <Item id="All">
                <ActionPermitted id="actionID1"/>
                <ActionPermitted id="actionID2"/>
                ….
        </Item>
        <Item id="anItemID">
                <ActionPermitted id="actionID1"/>
                ….
        </Item>
        ….
</Response>
```

For each queried item + the "All" item, the list returns an "Item" tag. In each tag there are "ActionPermitted" tags with an id attribute which text matches an id of a permitted action (which is one of the queried actions).

The algorithm of retrievePermissions is:

- If a profile exists, retrieve the list of items and save it for building a response
- If no profile exists or the type="IncludeAll" attribute exists, save an "All" item
- For each "ActionPermitted" tag under data:
    - Build a dummy request having as the id the item id in the xml request, action as the action signaled by the action permitted. If there exists a "Data" param, use it as "Data" of the request, otherwise have an empty one. The Profile should be either empty (if no profile is defined) or the Profile defined in the xml request
    - Get the request limitation using getRequestLimitation
    - If the limitation is empty, signal that this action is allowed for all items
    - If the limitation is non empty, retrieve the resulting limiting profile combined with the xml request profile and for each item that exists in the result signal that this action is allowed
- Build the Response string out of the results of the loop

5.5 getTestProvider method

The getTestProvider method is an abstract method used by AbstractPAL to get a test provider by handling a label. Its signature is:

```
protected abstract IPALTestProvider getTestProvider(
   string strTPLabel)
```

The method should throw an XBLIP.PAL.UnknownTestProviderException exception.

6. IPALTestProvider - interface of a test provider

Writing a test class is done by implementing the IPALTestProvider interface. The interface includes a single method that AbstractPal uses for executing a test.
The method signature is as follows:

```
string getLimitiation(string strParams)
```

The method accepts a single parameter which is equal to the content of a "Test" tag in a tests descriptor XML. The expected return value of the test is one of three : an empty string, an "ABORT" string or a limiting profile - depends on the test result.

7. Groups of tests

The tests list created by the transformer holds tests that should be ran for a user in order to test the possibility or range of execution of an action.
It may happen that we would like to define optional tests when there is more then a single possibility for the permissions on an action. For example, as the library supervisor I'm allowed to by books for both the team library and for myself. Another guy in my team can only buy books for himself. We have a web application for placing orders on books, and I'd like it to limit the profile of entities to which it is possible to buy books in a way that I (having a role of a library manager) will be limited to my team, and myself, and he will be limited to himself only. This may be achieved by using a TestGroup. A
It is used instead of a "Test" tag in the list of tests created by transformers to define a group of tests and test groups and the relations between them.
A TestGroup may either be an "OR" group or an "AND" group.
An "OR" group runs the tests and test groups in it until one of them returns a non-"ABORT" result (limiting profile or empty) and this is assigned as the TestGroup result. If no non-"ABORT" results of tests are encountered the result of the TestGroup is "ABORT".
An "AND" group is very similar to the whole "Tests" tag. It defines tests and test groups. Its result is the result of looping the tests, executing them and concatenating the limiting profiles. If one of them results in "ABORT" the result is "ABORT".
A template for a test group looks like this:

```
<TestGroup type="OR/AND">
        <Test…./>
        ….
        <TestGroup …>
                ….
        </TestGroup>
        ….
</TestGroup>
```

The type attribute defines what the type of the group is. If it is dropped, OR group type is assumed. The Test tags and TestGroup tag define its sub tests.
For example, the following is a list of tests created for getRequestLimitation:

```
<Tests>
        <Test id="ApplicationUser"/>
        <TestGroup type="OR">
                <Test id="LibraryManager"/>
                <Test id="Private"/>
        </TestGroup>
</Tests>
```

Assume that I have the following type of tests:
ApplicationUser - A simple test that checks if I'm an application user. If I am, it returns an empty response. Otherwise it returns "ABORT".
LibraryManager - A test that checks if I'm a library manager. If I am, it returns an empty response. Otherwise it returns "ABORT".
Private - a test that adds a limiting profile that limits the request to items that have my id as the "Owner" property. Meaning that it always returns "<Owner>*myID*</Owner>" based upon the user id.

The above series of tests will always fail for someone who's not an "ApplicationUser". For a library manager, the OR test group will result in an empty string, because the first test will not file. For someone who's not a library manager, the first test will fail, and then the second test will succeed, resulting in a limiting profile of "<Owner>*myID*</Owner>"
If we wanted to allow only "Buyer"s to buy books, then we could modify the above test series to something like this:

```
<Tests>
        <Test id="ApplicationUser"/>
        <TestGroup type="OR">
                <Test id="LibraryManager"/>
                <TestGroup type="AND">
                        <Test id="Buyer"/>
                        <Test id="Private"/>
                </TestGroup>
        </TestGroup>
```

</Tests>
Notice the new TestGroup. The new test means that apart from being an ApplicationUser, you must also be either a LibraryManager or a Buyer to buy books, but if you're a Buyer, you can only buy books for yourself.

Finally, TestGroups may also be used when censoring, or for a retrievePermissions query.

8. Design Considerations

This last section of this essay includes a list of considerations that should effect an implementation when using PAL.

8.1 designing a test class

A test class should be a simple class defining very specifically a single character that the user should have in order to achieve the actual action execution. A Role test encapsulating all tests that belong to role-based permission is an example of a test class and is easily implemented by keeping in the database all possible actions on different items and attaching them to roles. Each user can then have a list of roles, and the test should just make sure that the user has a role that can execute the method on an item.

Limiting profile tests should match in the field they issue to actual fields of entities. Otherwise a failure will occur. This should not be a problem, because a test for a certain feature of an item is possible only when this feature is a characteristic of the item. Otherwise it's meaningless.

To create a safe mechanism, the test should not rely on the XML request parameters to give the user name to identify the user. Especially in web applications this may be a breach through which intruders could disguise themselves as usernames with higher permissions. For example, in DNA/.Net the usage of COM+ allows the test class to get the username of the calling user directly, and not through the request string.

8.2 The IItemsListRetriever interface

In order to force PAL to know about actual data access methods, it requires a client to hand an IItemsListRetriever implementer when censoring or querying for permissions through retrievePermissions. This is quite a pain, unless the entity object has already a "Retrieve" action in which the profile parameters contains all of the profiles parameters used by other actions. With careful planning of the entity object this is an easily achievable task. And if one does not have a "Retrieve" method, build one, but make it simple enough to be only valid for the actions you'd like to user censorship and the query. Remember that getLimitedRequest (which is mostly used by an application) does not require this.

8.3 Censoring and Limiting

It is advised that if a field may not be retrieve on an item for a certain user, that she may not also query items by it. This way no binary or other search method may be used to discover the value for that field.

8.4 Knowing what was limited

If the application allows for multiple items to be acted on in the same request we may sometimes like to know, whether a limitation occurred. First, if it's a modification action use the response to know how many entities were modified (The return result of an action is determined by the entity object, but it has already been advised to return the number of modified entities in the Request/Response standard definition. So if you wanted to know why - this is a good reason). Then you can use the limiting profile saved in the recentLimitingProfile property of abstract PAL to know on which exact items it was allowed to perform the action. Using the original Profile or the request, you can get the exact items on which the request was allowed on, and the items it were not.

9. Summery

The essay introduced a layer for handling permissions on actions in a very flexible method. The method allows full usage of all data that exists to get the correct data for permitting a user an action on items. This is true because of the introduction of test objects, which encapsulate all actions for a certain test for the user control over items. PAL is heavily based on the Request/Response standard, but is not necessarily connected to DAIL, although using both should simplify greatly the creation of a dynamic and yet complex IT application.