Currying Logs

This article is a tangent on Currying Databases.

A little background: Our student portal was riddled with bugs. Since we use a monolithic system like ASP.Net Classic, any tiny issue could cause our entire page to crash. The codebase is riddled with sloppy code handed down from person to person through 5 generations of programmers who formerly held the position. These people had no interest in cleaning up the code or making it coherent. To make matters worse, the vendor themselves had bugs that they did not fix.

Faculties and students would encounter bug sometimes and technicians could not replicate the issue. At times there would be bugs that students knew, but no one reported it. These are some of the embarassing situations were common-place on the student portal.

To prevent an all-out-page-crash, a RESTful approach was used (when possible). Breaking **gigantic** operations into smaller units and wrapping each of those into a try-catch statements. Important operations that could fail should be inside a try-catch statements (e.g. database operations). When catching the error, supply as much information as possible such as variables and a simple and concise description of the error.

Currying is a great technique can be used to suite these purposes.

How?

Currying allows us to string together expressions and execute them.

This is particularly useful when we need to supply important information inputs into a function.

For instance, this a simple API route that searches for a user. This route is reserved only for a small group of administrators.

```
try
{
    var users = PortalUserQueries.SearchPortalUser(q);

    if (ReferenceEquals(users, null))
        return users.ToResult(2).Resp();

    return users.ToResult(1).Resp();
}
catch(Exception ex)
{
    ex.Error("[Get Portal User] Error when trying to search for users via
query.")
        .Add("q", q)
        .Add("qSearchPortalUser", PortalUserQueries.qSearchPortalUser)
        .Ok();
}
```

See the [bottom][SearchPortalUser] for a complete example.

Here's an example of a database call to get the Audit Records of a specific form entry.

```
try
{
    var auds = AuditQueries.GetAuditRows(_recid, _formid);

    if (ReferenceEquals(null, auds))
        "no data".ToResult(2).Resp();

    return auds.ToResult(1).Resp();
}
catch (Exception ex)
{
    ex.Error("[PZ Courses Audit] There was an error when trying to get audit
courses.")
        .Add("formid", formid)
        .Add("recid", recid)
        .Add("_formid (converted)", _formid.ToString())
        .Add("_recid (converted)", _recid.ToString())
        .Ok();

    return ex.Handle();
}
```

See the bottom for a complete example.

Here's another simple database call to get the Templates for a database form. If it fails, we get all the input variables that was used during the problem.

```
const string qGetTemplate = "select * from _form_templates where id = @id and
category = 'Course Forms (Pre-offer)' ";

try
{
    model = Db
        .Forms
        .Cmd(qGetTemplate)
        .Param("@id", info.formid)
        .Row();
}
catch (Exception ex)
```

```
{
    ex.Error("[PZ Courses Edit] Error when trying to select the form
templates.")
        .Add("cmdTemplate", qGetTemplate)
        .Add("model", model.Json())
        .Add("input", info.Json())
        .Ok();

    return ex.Handle();
}
```

See the bottom for a complete example.

What else?

Logging done correctly could eliminate hours or days of troubleshooting. A good logging system could allow the administrators to anticipate potential issues before a huge one occurrs. Another useful aspect of logging is it can help provide information and trace when necessary.

As an extra step, I created a service that would aggregate a list of errors within the past 24 hours at the end of the day. Looking at the daily report enough times, a pattern begins to emerge. When a problem occurs, I can respond to it before the users complain. Sometimes, the report can do checks or display patterns that are unusual and this can help anticipate possible issues.

Full Code

[SearchPortalUser]: Search Portal User Complete

```
public class PortalUserController : ApiController
{
    public HttpResponseMessage Post(string q)
    {
        if (PortalUser.Current.IsGuest)
            return new HttpResponseMessage(HttpStatusCode.Unauthorized);

        if (!PortalUser.Current.HasRole("Registrar")
            && !PortalUser.Current.IsSiteAdmin)
            return new HttpResponseMessage(HttpStatusCode.Forbidden);

        if(String.IsNullOrEmpty(q))
            return new HttpResponseMessage(HttpStatusCode.NotFound);

        try
        {
            var users = PortalUserQueries.SearchPortalUser(q);

            if (ReferenceEquals(users, null))
                return users.ToResult(2).Resp();
```

```
            return users.ToResult(1).Resp();
        }
        catch(Exception ex)
        {
            ex.Error("[Get Portal User] Error when trying to search for users
via query.")
                .Add("q", q)
                .Add("qSearchPortalUser", PortalUserQueries.qSearchPortalUser)
                .Ok();
        }

    }
}

public static class PortalUserQueries
{

    public const string qSearchPortalUser =
    @"select (FirstName + ' ' + LastName) fullname, substring(hostid, 4, 20)
cxid, email from fwk_user
      where FirstName like '%' + @name + '%' or LastName like '%' + @name +
'%' or _users.cxid like '%' + @cxid + '%'; ";

    public static Dictionary<string, object>[] SearchPortalUser(string q)
    {
        return Db.Jics
            .Cmd(qSearchPortalUser)
            .Param("@name", q)
            .Param("@cxid", q)
            .Rows();
    }

}
```