# TraXPLORER = ExtJS 4 MVC + ASP.NET MVC 3 + CRUD + REST

In this post, I will try to build a RESTful ExtJS 4 MVC CRUD grid with ASP.NET MVC 3 as REST API provider. The application that I'm going to build is called "Traxplorer". It is ExtJS grid panel with rowEditing plugin containing list of music albums with columns for ID(Primary Key), Name, Artist and one actioncolumn for Delete functionality. Before we start , if you want to see it running or download code, use below links -

DEMO

CODE

Assumptions :

1. As soon as user adds new row to grid, it's actually added to the database as blank album with auto-generated ID value. User can add details afterwards.
2. It is not allowed to add more than one blank row. User has to update the incomplete blank row first.

BUILDING EXTJS 4 MVC CLIENT :

I started from where I left last post. The order manager application! I created similar grid and interface. The only thing that I clanged here is RESTPROXY configuration in Album model. If you have followed my last example, you know that it's very easy to pick up an Extjs MVC template and start from there. (Code Re-usability! ) So I replaced controller, model, store and view definitions according to design of Traxplorer.

So Basic structure of Traxplorer client is -

- Controller – AlbumsController.js
- Model – Album.js
- Store – Albums.js
- View – List.js

Special attention should be given to Model and Store classes as they will create REST API consuming client.

```
Ext.define('Traxplorer.model.Album', {

    extend: 'Ext.data.Model',

    fields: [ . . .
            ],
            ],
        ...
    proxy:
    {
        type: 'rest',
        url: '/Albums',
        timeout: 120000,
        noCache: false,

        reader:
        {
            type: 'json',
            root: 'data',
            successProperty: 'success'
        },

        writer:
        {
            type: 'json',
            writeAllFields: true
        },
              //clean up handlers
        afterRequest: function (request, success)
        {

            if (request.action == 'read') {
                this.readCallback(request);
            }

            else if (request.action == 'create') {
                this.createCallback(request);
            }

            else if (request.action == 'update') {
                this.updateCallback(request);
            }

            else if (request.action == 'destroy') {
                this.deleteCallback(request);
            }
        },

        //After Albums fetched
        readCallback: function (request)
        {
            if (!request.operation.success)
            {
            ...
            }
        },

        //After A record/Album created

        createCallback: function (request)
        {
            if (!request.operation.success)
            {
              ...
            }
        },

        //After Album updated

        updateCallback: function (request)
        {
            if (!request.operation.success)
            {
              ...
            }
        },

        //After a record deleted

        deleteCallback: function (request)
        {
            if (!request.operation.success)
            {
               ...
            }
        }
    }
});
```

I have set up a rest proxy and all the necessary CRUD operation callbacks. Most of the things are handled by ExtJS data package so you don't have to really try hard to make your proxy work. Just specify type as a by ExtJS data package so you don't have to really try hard to make your proxy work. Just specify type as a 'rest' and URL that will provide REST interface this client will talk to. Simple, isn't it?

Next is Store -

```
Ext.define('Traxplorer.store.Albums', {
    extend: 'Ext.data.Store',
    autoLoad: true,
    autoSync: true,
    model: 'Traxplorer.model.Album'
});
```

Keeping autoSync:true allows store to order it's proxy to carry out the CRUD data requests according to CRUD actions performed on it. e.g. If user removes record from a grid, proxy will initiate DESTROY operation and sync store with the database. Likewise record update will trigger UPDATE operation, new record will trigger CREATE and store load action will trigger READ operation.

Now let's move on to the Controller part -

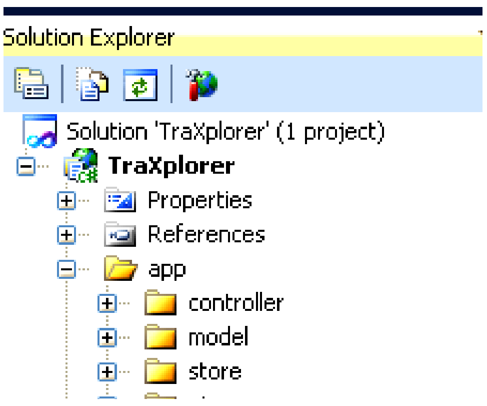
```
Ext.define('Traxplorer.controller.AlbumsController', {

    extend: 'Ext.app.Controller',
    stores: ['Albums'],
    models: ['Album'],
    views: ['album.List'],

    init: function () {
        this.control(
                {

                    'button[text=Add]':
                    {

                       //could have handled in # handler #
                       //of toolbar button but to keep view isolated
                       //from thick event handlers moved here.

                        click: this.addRow

                    }
```

```

                        });

    },

    addRow: function (e) {
                //[....could have restricted to create empty reco
rd in
                //database by toggling
                // this.getAlbumsStore().autoSync=false...]

              //...SKIPPED SOME VALIDATIONS REFER GITHUB CODE

        //Create empty album in database. Populate it's id when
 created.

        this.getAlbumsStore().insert(0, this.getAlbumModel().cr
eate());

          }
    }
});
```

Now view is simple Row Editor grid panel with toolbar containing add new row button and one action column for deleting row. You can have a look at the code here – List.js

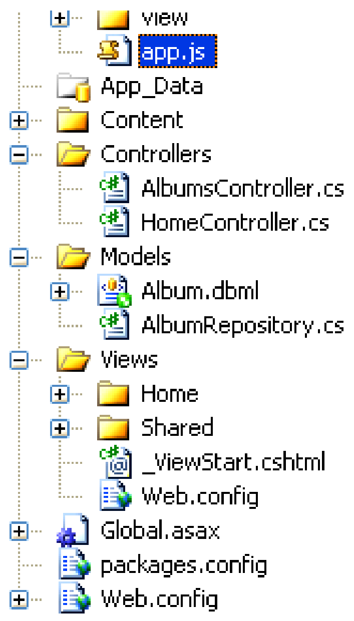
REST API PROVIDER USING ASP.NET MVC 3

We are almost done with our Traxplorer client. So ExtJS MVC part is over now and it's time to code REST API in ASP.NET MVC 3. Before that, I just want to show how Traxplorer file structure will look in Visual Studio.



There could be many approaches to integrate our client into ASP.NET MVC 3 project template. I just kept the default HomeController as is and blanked out its view content. I also modified the _Layout.cshtml in Views/Shared folder to embed Traxplorer client scripts. So HomeController simply returns Traxplorer ExtJS client.

Set Up Database -



Add a SQL database table Album with following configuration

1. id – int -not null -primary key – identity auto increment
2. name – varchar-nullable
3. artist – varchar- nullable

Setup Model, Repository Class -

Create LINQ to SQL class Album.dbml using above created Table. Create basic repository class AlbumRepository.cs containing four basic CRUD operations and persistence method. If you are unsure about the things, a good place to start ASP.NET MVC will be this.

Creating Controller -

Do you remember the URL our REST proxy was pointing to? Yeah it was '/ALBUMS'. When you implement rest provider, ideally all CRUD action methods get invoked by four specific type of HTTP requests.

1. HTTP GET (returns a resource identified by URI)
2. HTTP POST (Creates a resource & returns absolute URI for the created resource)
3. HTTP PUT (Updates/Replaces a resource identified by URI)
4. HTTP DELETE (Deletes a resource identified by URI)

Let's consider AlbumsController.cs now.

```
using ...;
namespace TraXplorer.Controllers
{

    //Provides REST API implementaion for Traxplorer ExtJS client.

    public class AlbumsController : Controller
    {
        AlbumRepository repoAlbum = new AlbumRepository();
        //
        // GET: /Albums/
        [HttpGet]
        [ActionName("Index")]
        public JsonResult Index(int? id)
        {

            try
            {
```

```
            {
                if (null != id)
                {
                    Album album = repoAlbum.GetAlbum((int)id);
                    return this.Json(new { success = true, data = album }, JsonRequestBehavior.AllowGet);
                }
                var list = repoAlbum.FindAllAlbums().ToList();
                return this.Json(new { data = list }, JsonRequestBehavior.AllowGet);
            }
            catch
            {
                return this.Json(new { success = false, data = "" }, JsonRequestBehavior.AllowGet);
            }

        }

        //PUT /Albums/Index/id
        [HttpPut]
        [ActionName("Index")]
        public JsonResult Update(int id, Album album)
        {

            try
            {
                Album dbAlbum = repoAlbum.GetAlbum(id);
                dbAlbum.name = album.name;
                dbAlbum.artist = album.artist;
                repoAlbum.Save();
                return this.Json(new { success = true, data = album }, JsonRequestBehavior.DenyGet);
            }
            catch
            {
                return this.Json(new { success = false, data = "" }, JsonRequestBehavior.DenyGet);
            }

        }

        //POST /Albums/Index/album
        [HttpPost]
        [ActionName("Index")]
        public JsonResult Create(Album album)
        {
            Response.BufferOutput = true;
            try
            {
                repoAlbum.Create(album);
                repoAlbum.Save();
                //Created
                Response.StatusCode = 201;
                //Set Location header to absolute path of entity.
                Response.AddHeader("LOCATION", Request.Url.AbsoluteUri + "/" + album.id);
                return this.Json(new { success = true, data = album }, JsonRequestBehavior.DenyGet);

            }
            catch
            {
                return this.Json(new { success = false }, JsonRequestBehavior.DenyGet);

            }

        }

        //DELETE /Albums/Index/id
        [HttpDelete]
        [ActionName("Index")]
        public JsonResult Delete(int id)
        {

            try
            {

                Album album = repoAlbum.GetAlbum(id);
                repoAlbum.Delete(album);
                repoAlbum.Save();
                //200 OK...could be 204 No Content if no stat
```

```
us describing entity in response.
                //Response.StatusCode = 204;
                return this.Json(new { success = true });
            }
            catch
            {
                return this.Json(new { success = false });
            }

        }
    }
}
```

All methods performing respective CRUD operations are decorated with ActionName attribute with value “Index”. Also make note of http[GET/PUT/POST/DELETE] attribute. This attribute makes them different from each other.

Also all methods return specific status codes according to action they perform. Ideally REST client should perform all response handling according to http status codes it receives from REST Service Provider very much like the provider identifies and deciphers the requests according to http VERBS.

Routing -

URL routing is configured in GLOBAL.asax.cs file

```
public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                "Default", // Route name
                "", // URL with parameters
                new { controller = "Home", action = "Index"} // Parameter defaults
            );
            routes.MapRoute(
               "Album", // Route name
               "{controller}/{id}", // URL with parameters
               new { controller = "Albums", action = "Index", id = UrlParameter.Optional } // Parameter defaults
           );
        }
```

First route maps to default controller that returns view comprising of Traxplorer extjs client.

The second route specifies  controller- Albums, action-Index as default values in the request URIs leaving ID parameter optional. That's enough! Rest of the things are self explanatory and can be found in code.

Now it's time to see Traxplorer running. It will be good to observe its behavior with FireBug with respect to Request types, Request and Response  Headers and URIs.

TRAXPLORER -EXTJS 4 MVC + ASP.NET MVC 3 + CRUD + REST- DEMO

CODE @GITHUB