



# HENRY-Archive Performance History Database

Installation and User Guide

Developed at Boston Symphony Orchestra

2014 (rev. 2016)

## Table of Contents

About the BSO Performance History Search "HENRY" .....	3
Contact .....	3
Architectural Overview.....	4
Project Solution (.NET Framework) .....	4
Database .....	4
Sample Data .....	5
Source Code .....	5
Adding Work Document .....	6
System Requirements.....	7
Installation .....	8
Pre-requisites .....	8
Step 1 .....	8
Step 2 .....	8
Step 3: Publishing Project to web server .....	8
Step 4: Data Import .....	8
Data Update .....	10
Updating with New Fields.....	12
Licensing and Disclaimer.....	14

## About the BSO Performance History Search "HENRY"

"HENRY" is the Boston Symphony Orchestra's Performance History Search module, which contains all documented concerts of the orchestra beginning with October 21, 1881 through the current season. In addition, documented concerts by the Boston Pops Orchestra (from 1885 to the current season) and the Tanglewood Music Center (from 1940 to the current season) are added to the database on an ongoing basis, with the anticipated completion of this addition to occur in late 2017. The module is named after the BSO's founder, Henry Lee Higginson. The search function provides access to the performance history of every work, and of all artists – conductors, ensembles, and soloists – who have performed with the orchestra.

BSO Performance History Search module "HENRY" was underwritten by:



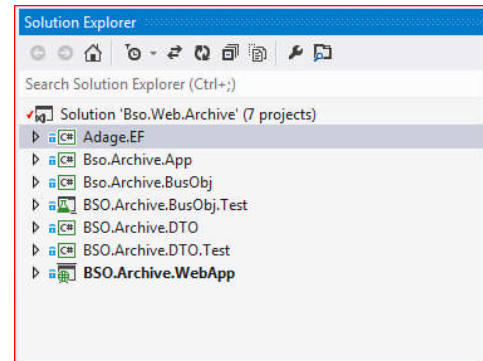
## Contact

Please tell us about your experience or how we can make this application better by emailing [archives@bso.org](mailto:archives@bso.org)

## Architectural Overview

### Project Solution (.NET Framework)

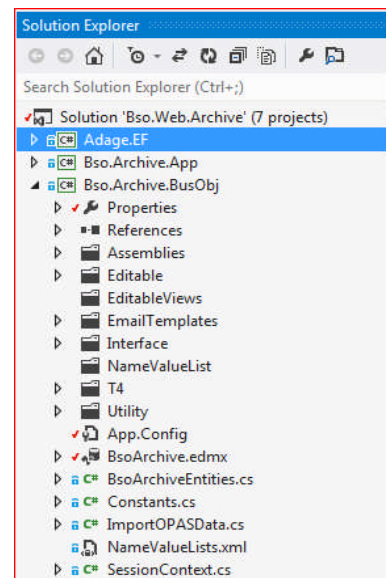
The solution is comprised of seven projects developed in Visual Studio. The projects ending in .Test are the unit tests for the corresponding projects. Adage.EF contains custom classes and interfaces. Bso.Archive.BusObj and BSO.Archive.DTO are class projects, while BSO.Archive.WebApp is an ASP.NET Web Forms Application. The discussion in this document assumes use of Visual Studio for code management.



### Database

The database that holds the performance history records is modeled after the XML file that holds the information. Each parent tag has its own table with a foreign key relationship to its child. As an example, every Event has works, every work has a composer.

The database is represented by an Object Relational Mapping using Microsoft.NET Entity Framework 4.0 model in the Bso.Archive.BusObj project in the solution. This project also contains the code required for running the import of data from the xml file to the database. Within the Editable folder, there are class files which represent every table in the database in it contains customized code to read that tag within the xml for that particular node. The ImportOpasData class file within that same project contains the initialization for the import process. From there the nodes are pulled from the XML file and sent to their individual class files to be read and added to the database.

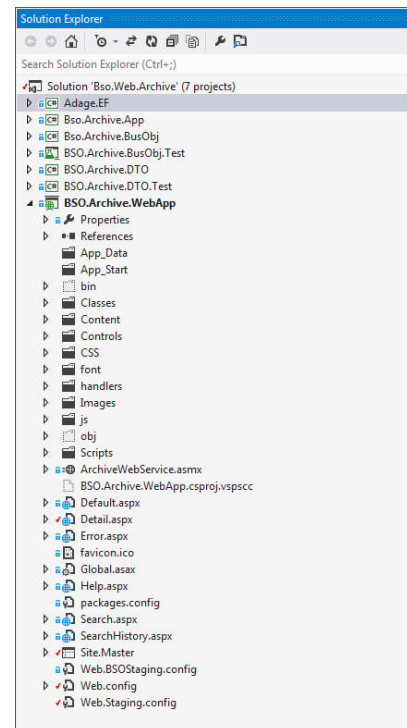


The database also contains a table named OPASUpdate, which is used in the update database process. This table contains columns for the table name, column name and tag name of the value to be updated. It also contains a Boolean value to indicate whether or not that entry has

been update already. In the Bso.Archive.BusObj Editable folder, the OPASUpdate class file contains the code for the update process. The update process reads from an update table, and an XML file that is in the same format as the import file. The update process updates the database based on the new values in the XML.

## Sample Data

Within the Bso.Archive.BusObj project, there is a folder labeled “ProjectFiles”. Within here you will find two sample files. SampleXML.XML is an XML file that can be used to test the import process. The file contains sample data that is correctly formatted to work with the importer. There is also an ArchiveSampleDatabase.bak file. This is a backup of the database that has been populated by running the import process on SampleXML.XML. Creating a new database by restoring from this file will allow you test all the functionality without having to manually build the database. Information on restoring a database from a backup can be found at <http://msdn.microsoft.com/en-us/library/ms177429.aspx>.



## Source Code

The BSO.Archive.WebApp project within the solution contains the code for the archive site itself. Within the Controls folder, the Performance, Artist, and Repertoire controls each represent one of the tabs on the Performance History Search page. When a search is run, the values in the text fields are used as search expressions to query specific columns in the database. The Performance Search includes within its results links to a detailed description page. This page provides more detailed information of the specific event that is not included in the results table. The results from Performance search also provide a link to the PDF image of the original program book for that event.

There are two tables within the database that are used for searching; the EventDetail table and the ArtistDetail table. These tables were created using the CreateArchiveSearchTables stored procedure and were designed to be single table representations of all the imported data tables that the different search types could potentially need to query from. The query returns a list of either

EventDetailIds or ArtistDetailIds, depending on the search type. On the client side, a series of asynchronous calls are made from JavaScript using these Ids to create data transfer objects (DTOs) which are representations of the data that the client can more easily display.

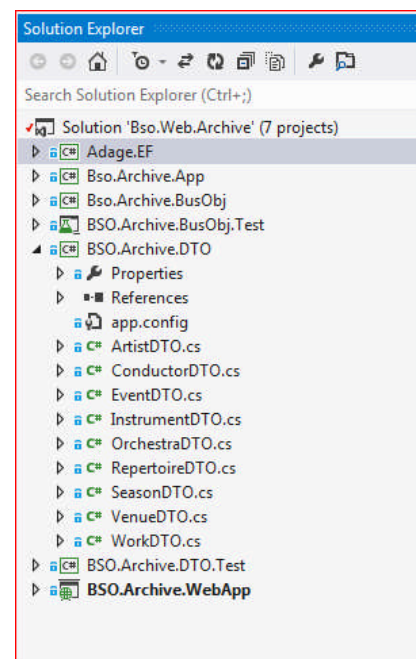
The class files for DTOs are within the BSO.Archive.DTO project. They use the Ids to query the database for specific data and build the objects to return back to the client. The client side technology utilizes Backbone.js and uses models to represent both the search results and the search filters. JavaScript asynchronously loads event DTOs using event detail ids generated from the server. They are displayed in the table and ordered using filters and sorted generated from the file in filterConfig in the /js/app/config folder. FilterConfigs are objects returning true or false for whether or not events passed into them apply, and SortConfig returns -1/0/1 (should be sorted before/same/after) based on the comparison of two key sorting values. The grid results are reloaded when filters or sorting change.

The performance archive search also provides functionality for saving search history. The search history is stored in Cookies and can be displayed using the SearchHistory.aspx page. The search history page also includes links allowing the user to navigate back to the search page and regenerate the results. This functionality is also present in the search results themselves, allowing users to click on linked text within the search results which will re generate searches based on that keyword. The search page also provides functionality for exporting the search result tables to Excel files.

Within the BSO.Archive.WebApp project there is also a Web.Config file. This file contains projects' applications settings. These include the file paths for the import and update processes, URLs for both the site itself and the link to PDF of the original program book and other general settings.

### Adding Work Document

The OPAS export includes fields for “Work Document” which represent the recording of a concert. These fields are found in the <WorkItem/> section of a record for a concert. A full concert record is enclosed with <eventItem/> tags, so the <WorkItem/> tags are nested within the <eventItem/> tags. The two most relevant fields (into which we are currently



putting data) are <workDocument\_Name/> and <workDocument\_FileLocation/>.

The batch import now reads these fields and imports them in to the HENRY database. A table was added named WorkDocument to store the corresponding information.

The data stored was:

- Work Id
- Work document name
- Work document notes
- Work document summary
- Work document file location

If the data in <workDocument\_Name/> contains the text “Audio” AND there is data in <workDocument\_FileLocation/> field, HENRY will display an audio icon on the search results and work details pages. Clicking on the icon will open a new tab to access the audio where it is hosted at collections.bso.org website.

## System Requirements

The Archive application runs using .NET Framework 4. It is hosted using Internet Information Services (IIS) 7.5. The data is stored using Microsoft SQL Server 2008 R2.

These are the suggested minimal hardware specifications for a consolidated server:

“Consolidated” meaning the server is responsible for both web and back-end SQL duties in a simple deployment.

- 1 x CPU Core
- 4GB Memory
- 40GB Disk Capacity
- 1Gbp Network
- Windows Server 2008R2
- .NET Framework 4.5.1
- SQL Server 2008R2 Web Edition or higher

## Installation

### Pre-requisites

1. Installation of .NET 4.0, Visual Studio, and SQL Server should be completed.
2. Data must be structured in the format depicted in the sample XML.

### Step 1

1. Download HENRY-Archive Project from GitHub's web site.
2. Unzip the project locally in a working folder.

### Step 2

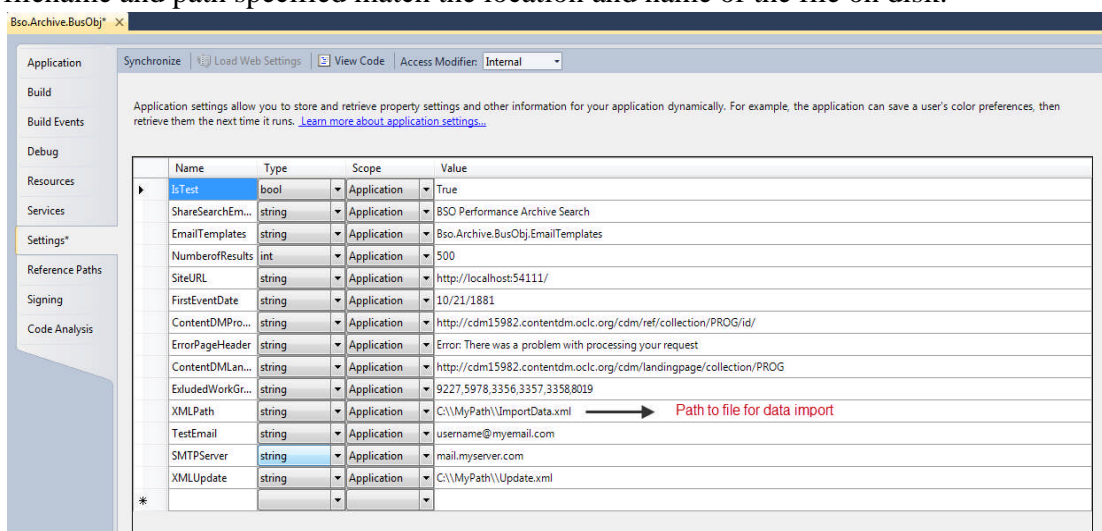
1. Open the Project Solution (Bso.Web.Archive.sln) in Visual Studio
2. Configure web.config to reflect your settings.
3. Build Solution

### Step 3: Publishing Project to web server

Publishing your code to your servers can be done through the one-click publish functionality in Visual Studio. Microsoft provides a tutorial (<http://msdn.microsoft.com/en-us/library/dd465337%28v=vs.110%29.aspx>) on how to create a publishing profile. Once the profile is created, future publishing can be done by right-clicking on the project in the solution folder and selecting publish, or by going to **Build** then **Publish BSO.Archive.WebApp** from the Visual Studio menu bar. Once you are presented with the following dialogue, you can select your previously created Publish Profile and publish.

### Step 4: Data Import

1. Add file location settings to point the importer to the full xml file. Make sure the filename and path specified match the location and name of the file on disk.





2. Go to the web server where the site is being hosted and in the browser, enter the address: [http:// <localhost or IP address:port number>/ArchiveWebService.asmx](http://<localhost or IP address:port number>/ArchiveWebService.asmx). From the list of links at the top, select RunImport. In the textbox to the right of update, **type in false** to indicate that this is an import not an update. Then press invoke. **It takes about 1.5 to 2 hours minutes to run the import for a 400 Mb file.**

The import process runs the following stored procedures in the database.

- a. DropForeignKeys – Removes the foreign key relationships from the tables in the database to allow all the data to be removed.
- b. CleanDatabase – Removes data from tables in database.
- c. AddZeroRows – Sets the default values for the tables.

These are followed by the actual import process. After the import process has finished, if there were no errors you will receive a success message.

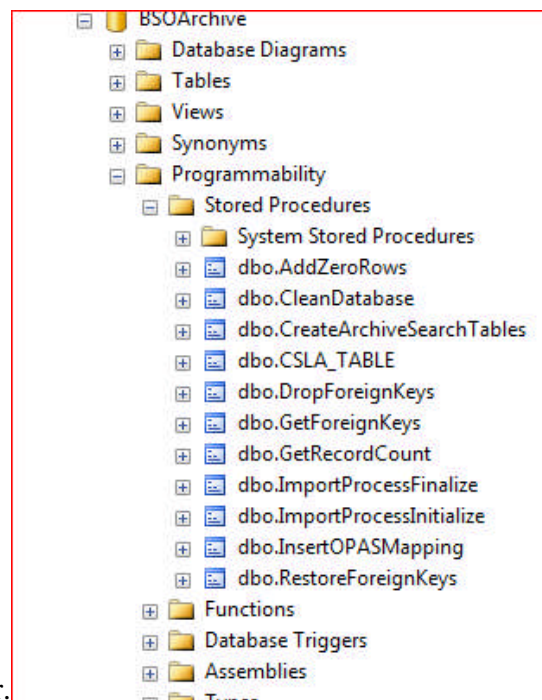
This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<string xmlns="http://tempuri.org/">Success</string>
```

Otherwise, you will receive an error message.

After you receive the Success message, go to the database in SQL Management studio.

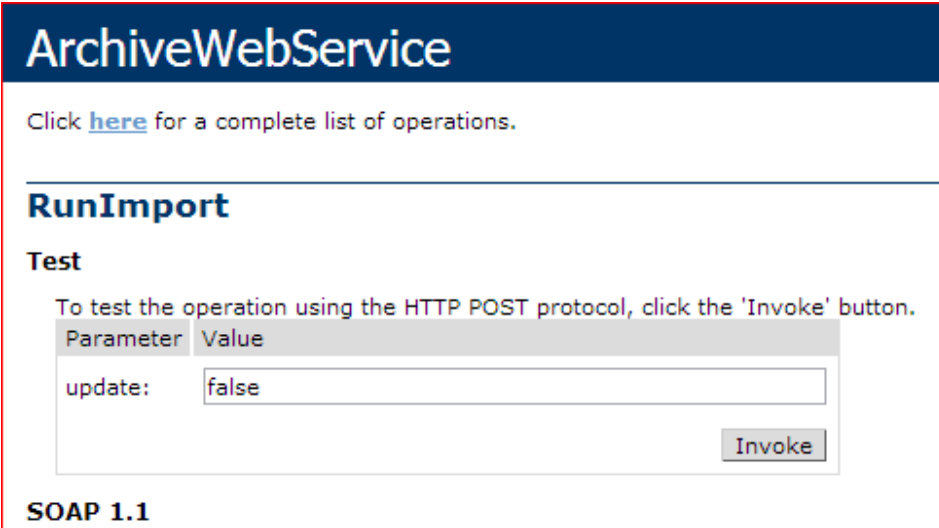
Navigate to the stored procedures folder.



- Right click on the dbo.**ImportProcessFinalize** procedure and select “Execute Stored Procedure...” This procedure will rebuild the foreign key relationships in the table. After this execute stored procedure **CreateArchiveSearchTables** which will create the search tables. **It takes about 10 minutes to run the SP.**

## Data Update

- Go to the database and modify the table OPASUpdate. Add a new entry based on the



**ArchiveWebService**

Click [here](#) for a complete list of operations.

---

**RunImport**

**Test**

To test the operation using the HTTP POST protocol, click the 'Invoke' button.

Parameter	Value
update:	false

**SOAP 1.1**

Invoke

field you wish to update. TableName and ColumnName refer to the table and its column that you want to update. The TagName refers to the tag in the

xml that you want to update the table with.

	OPASUpdateId	TableName	ColumnName	TagName	HasBeenUpdated	Created
	1	Work	WorkGroupID	workId	True	0
	2	Work	WorkGroupID	workId	True	0
	3	Venue	VenueName	eventVenueName	False	0
	NULL	NULL	NULL	NULL	NULL	NULL

The OPASMapping table provides a mapping between table, column, and tag names for reference.

TableName – BSOArchiveTable

ColumnName – BSOArchiveColumn

TagName - OPASColumn

- Add the file that will be used to update the database. Make sure the filename and path specified match the location and name of the file on disk.

3. The structure of the file should be in the same format as the import data file. Meaning if an element has a parent element it must be included in the update file even if it is not being updated itself. So if you were to update the name of an event venue the file would look like this.

```
1  <?xml version="1.0" encoding="utf-8" ?>
2  <!-- EXPORT 06/04/2013 -->
3  <DataRecordCollection>
4
5  <eventItem>
6    <eventVenue>
7      <eventVenueID>3</eventVenueID>
8      <eventVenueName>New Venue Name</eventVenueName>
9    </eventVenue>
10 </eventItem>
11
12 </DataRecordCollection>
```

If you were to update work artist or work composer, you would need to nest them inside the work parent element, which itself is nested in the event element.

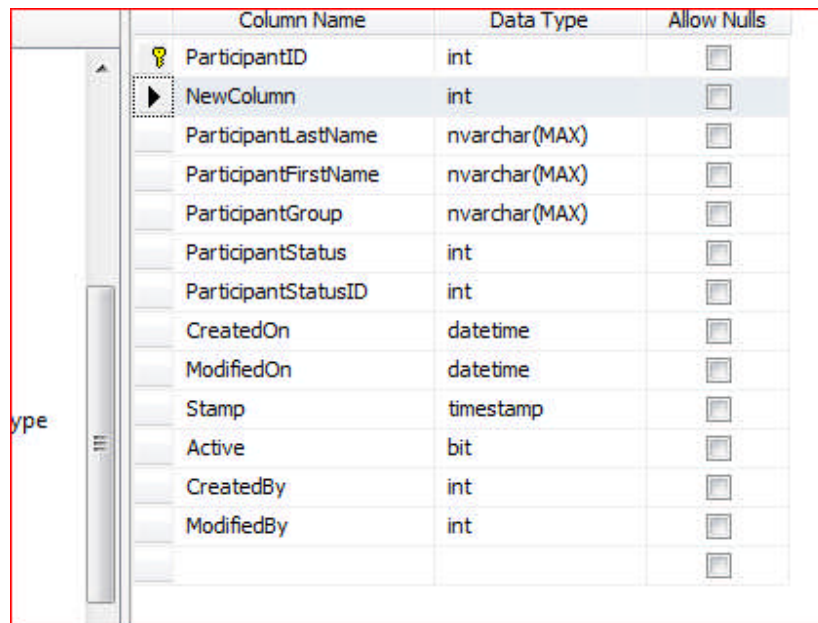
```
1  <?xml version="1.0" encoding="utf-8" ?>
2  <!-- EXPORT 06/04/2013 -->
3  <DataRecordCollection>
4
5  <eventItem>
6    <workItem>
7      <DateWorkId>5556</DateWorkId>
8      <workId>1177</workId>
9      <workComposer>
10        <workComposerId>150</workComposerId>
11        <workComposerLastname>New Composer Last Name</workComposerLastname>
12      </workComposer>
13    </workItem>
14  </eventItem>
15
16 </DataRecordCollection>
```

Only data within the tag name specified OPASUpdate table on the database is updated, all other tags are ignored. So whether or not they are included doesn't matter.

4. Go to <http://archive.bso.org/ArchiveWebService.asmx> from the server where the site is being hosted. From the list of links at the top, select RunImport. In the textbox to the right of update, type in true to indicate that this is an update. Then press invoke. The update process may take some time depending on the number of entries to update. If the import was successful you will receive the same success message as shown above. Otherwise you will get an error message.

## Updating with New Fields

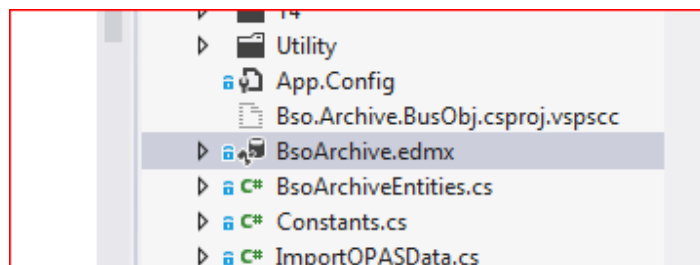
1. First add the new column to the table in the database.



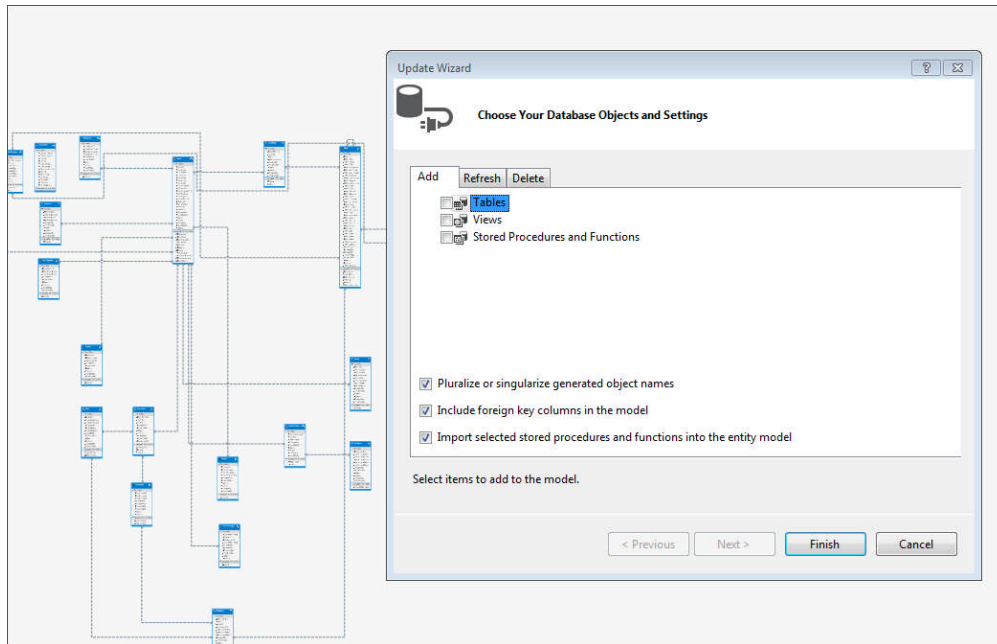
The screenshot shows a table schema in SQL Server Enterprise Manager. The table has the following columns:

Column Name	Data Type	Allow Nulls
ParticipantID	int	<input type="checkbox"/>
NewColumn	int	<input type="checkbox"/>
ParticipantLastName	nvarchar(MAX)	<input type="checkbox"/>
ParticipantFirstName	nvarchar(MAX)	<input type="checkbox"/>
ParticipantGroup	nvarchar(MAX)	<input type="checkbox"/>
ParticipantStatus	int	<input type="checkbox"/>
ParticipantStatusID	int	<input type="checkbox"/>
CreatedOn	datetime	<input type="checkbox"/>
ModifiedOn	datetime	<input type="checkbox"/>
Stamp	timestamp	<input type="checkbox"/>
Active	bit	<input type="checkbox"/>
CreatedBy	int	<input type="checkbox"/>
ModifiedBy	int	<input type="checkbox"/>

2. Go into the Bso.Archive.BusObj project in VS and open the BsoArchive.edmx model.



3. Right click in the model and select “Update Model from Database...”



4. Immediately hit finish without changing any settings. This will add the new property to the table you specified in the database.
5. You will need to add the code into the importer code to update. The files that contain the logic that parse the xml for a specified table are in the Bso.Archive.BusObj project, within the editable folder. Each file is named after the column in the database is adds data to. In this example there is a new conductor element added.

```
var conductorLastName = conductorElement.GetXElement(Constants.Conductor.conductorLastNameElement);
var conductorNote = conductorElement.GetXElement(Constants.Conductor.conductorNoteElement);
var conductorName4 = conductorElement.GetXElement(Constants.Conductor.conductorName4Element);
var conductorName5 = conductorElement.GetXElement(Constants.Conductor.conductorName5Element);
var conductorNewElement = conductorElement.GetXElement("NewXMLTagName");

conductor.newElement = conductorNewElement;

conductor = SetConductorData(conductorID, conductor, conductorFirstName, conductorLastName, conductorName4, conductorName5, conduc
return conductor;
}
```

The column within conductor was newElement, and the xml tag that contains the data is "NewXMLTagName". Once you get the value into the variable from the xml file, you can assign it to the conductor.newElement value.

6. Follow steps 1-4 in for Data Update. Make sure the new element is included in the xml file, and that its tag and the new column name you created are set in the OPASUpdate table.

## Licensing and Disclaimer

HENRY is released under GNU General Public License, version 2.

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; version 2 of the License.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.