Action Audit

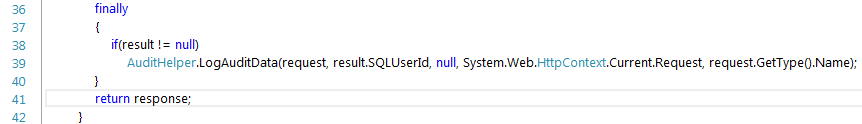
## Purpose

The purpose of the Action Audit is to track the types of actions a user takes, in the application, as well as logging the id’s of any patients who’s details a user has viewed.

The specifics of a particular action are stored in the AuditAction table of the SQL Server database that has been configured to hold all standard application data. (Currently, the database connection name for this configuration can be found in the web.config file of the Nightingale service, under the key “PhytelServicesConnName”.)

## Creating an ActionAudit Record

To add an audit record to the table, the **LogAuditData** method of the **AuditHelper** object should be called. Currently, these are called from each of the Nightingale service methods, just before the response object is returned to the caller. For example:



When this call is made, a new message queue message is created. That message is then dropped into a specific queue and will ultimately be picked up and processed by the **ASE Processor** application. That message also contains all the details for the database record, inside its message body.

The queue that holds those messages is designated by a value in the “ApplicationSettings” table of the database, under the key “AUDIT\_QUEUE”.

## ActionAudit Objects and Methods

These objects are in the Phytel.API.Common project, which is inside the API solution.

### AuditData.cs

This is the POCO data object that holds all the details about an action. This object is used to populate the body of the queue message, which in turn, gets added to the database.

#### Structure:

namespace Phytel.API.Common.Audit

{

[Serializable]

public class AuditData

{

public AuditData() { }

#region Required Properties

public string Type { get; set; }

public int AuditTypeId { get; set; }

public Guid UserId { get; set; }

public Guid ImpersonatingUserId { get; set; }

public string SourcePage { get; set; }

public string SourceIP { get; set; }

public string Browser { get; set; }

public string SessionId { get; set; }

public int ContractID { get; set; }

public DateTime EventDateTime { get; set; }

#endregion

#region Optional Properties

// patient list

[XmlArray("PatientIDList")]

[XmlArrayItem("PatientID")]

public List<string> Patients { get; set; }

public Guid EditedUserId { get; set; }

public string EnteredUserName { get; set; }

public string SearchText { get; set; }

// additional audit information

public object AdditionalAuditData { get; set; }

public string LandingPage { get; set; }

public Message Message { get; set; }

public string TOSVersion { get; set; }

public string NotificationTotal { get; set; }

public string DownloadedReport { get; set; }

#endregion

}

[Serializable]

public class Message

{

public Message()

{

}

public string Id { get; set; }

public string Text { get; set; }

public string Source { get; set; }

public string StackTrace { get; set; }

}

}

### AuditHelper.cs

* + **public static void LogAuditData(IAppDomainRequest request, string sqlUserID, List<string> patientids, HttpRequest webreq, string returnTypeName)**

This is the entry point to create an action audit record. Its purpose is to create a new thread, pass its arguments through to a new method inside that new thread to handle all the processing and immediately return control back to the original thread, while the audit is being processed.

* + - IAppDomainRequest request:

the actual request object created by the service that receives the web request

* + - string sqlUserID:

the user id for the user currently making the request

* + - List<string> patientids:

A list of id’s for any patients who’s data was viewed during this request

* + - HttpRequest webreq:

The web request object generated by the original call from the browsaer to the service endpoint

* + - string returnTypeName:

The type name of the specific request object passed to the service

* + **private static void AuditAsynch(IAppDomainRequest request, string sqlUserID, List<string> patientids, HttpRequest webreq, string returnTypeName)**

This is the starting point for processing of the actual audit. The arguments for this method are identical to the previous method and are simply passed through, from the previous message.

* + **private static void AuditAsynch(IAppDomainRequest request, string sqlUserID, List<string> patientids, HttpRequest webreq, string returnTypeName)**

This is the starting point for processing of the actual audit. The arguments for this method are identical to the previous method and are simply passed through, from the previous method to this method.

* + **public static string FindMethodType(string returnTypeName)**

This method simply parses the name of the request object that is passed into the service and returns the main name of the request type. That name is used to determine the id of the request type, in the next method, which will be used to validate the request.

* + **public static AuditData GetAuditLog(int auditTypeId, IAppDomainRequest request, string sqlUserID, List<string> patientids, HttpRequest webrequest, string methodCalledFrom, bool isError = false)**

This method creates and populates the AuditData object that becomes the primary body for the queue message, which gets processed and stored in the database.

### AuditDispatcher.cs

* + **public static void WriteAudit(AuditData auditLogToProcess)**

This method uses the AuditData passed in, to create a queue message, and then calls a helper method to send that queue message to the specified queue.

The queue is specified by looking up the key provided, from the “ApplicationSettings” table in the configured database. The connection name for that database is specified in the web.config file for the service call, under the key “PhytelServicesConnName”, and is used by the ASE Manager application to return the database connection to this method.

Data Audit

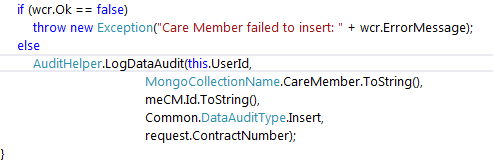
## Purpose

The purpose of the Data Audit is to track any inserts, updates and deletes processed for records in any of the specified data collections.

The specifics of a particular data audit are stored inside an “Audit” database, which matches the “live” data database, for a given customer. That “Audit” database is designated inside the configuration of an ASE Processor object. The “PhytelServicesConnName” key designates a connection reference that the Phytel Services Manager uses to return a connection to the proper database.

## Creating a DataAudit Record

To add an audit record to the correct collection in the database, the **LogAuditData** method of the **AuditHelper** object should be called. Currently, these are called from the DataAudit repository methods, just after the specific insert, update or delete action is completed. For example:



When this call is made, a new message queue message is created. That message is then dropped into a specific queue and will ultimately be picked up and processed by the **ASE Processor** application. That message also contains all the details for the database record, inside its message body.

The queue that holds those messages is designated by a value in the “ApplicationSettings” table of the database, under the key “DATA\_AUDIT\_QUEUE”.

## DataAudit Objects and Methods

These objects are in the Phytel.API.DataAudit and Phytel.API.DataAuditProcessor projects, which are inside the DataAudit solution.

### DataAudit.cs

This is the POCO data object that holds all the details about an audit. This object is used to populate the body of the queue message, which in turn, gets added to the database.

***Structure:***

namespace Phytel.API.DataAudit

{

[Serializable]

public class DataAudit

{

public const string UserIDProperty = "uid";

public const string TypeProperty = "type";

public const string EntityTypeProperty = "et";

public const string EntityIDProperty = "eid";

public const string EntityProperty = "ey";

public const string TimeStampProperty = "ts";

public const string ContractProperty = "ct";

[BsonElement(UserIDProperty)]

public string UserId { get; set; }

[BsonElement(TypeProperty)]

public string Type { get; set; } //Insert, Update, Delete

[BsonElement(EntityTypeProperty)]

public string EntityType { get; set; }

[BsonElement(EntityIDProperty)]

public string EntityID { get; set; }

[BsonElement(EntityProperty)]

public string Entity { get; set; }

[BsonElement(TimeStampProperty)]

public DateTime TimeStamp { get; set; }

[BsonElement(ContractProperty)]

public string Contract { get; set; }

}

}

### AuditHelper.cs

* + **public static void LogDataAudit(string userId, string collectionName, string entityId, DataAuditType auditType, string contractNumber)**

This is the entry point to create a data audit record. This overload is a pass-through call to the next overload, and is used to pass these arguments to the next overload, after setting that method’s “entityKeyField” argument to "\_id".

* + **public static void LogDataAudit(string userId, string collectionName, string entityId, string entityKeyField, DataAuditType auditType, string contractNumber)**

This is a pass-through method to the DataAuditAsynch method. The purpose of this method is to create a new thread and continue processing on that new thread while immediately returning control of the original thread, so that the application can continue while the audit happens in the background.

* + **private static void** **DataAuditAsynch(string userId, string collectionName, string entityId, string entityKeyField, DataAuditType auditType, string contractNumber)**

This method is responsible for acquiring a new DataAudit object and then passing that to the method that will eventually use it, to create the queue message which gets used to create the audit record.

* + **private static DataAudit GetDataAuditLog(string userId, string collectionName, string entityId, string entityKeyField, DataAuditType auditType, string contractNumber)**

This method creates the actual DataAudit object that is used to build the queue message.

* + **private static string GetMongoEntity(string contract, string collectionName, string entityId, string entityKeyField)**

This method queries the specified collection to get the entity that was just manipulated by the insert, update or delete call and returns it as a string snapshot, to be included in the audit record

### AuditDispatcher.cs

* + **public static void WriteAudit(DataAudit.DataAudit auditLogToProcess, string title)**

This method uses the AuditData passed in, to create a queue message. It converts the supplied DataAudit object to XML to be used as the message body and then calls a helper method to send that queue message to the specified queue. That queue message will eventually be processed by the ASE Process Manager and will be stored in the specified database.

* + **public static string ToXML(Object oObject)**

This method takes the oObject, which will be the DataAudit object from the method above, serializes it into XML and then returns it to the calling method above.

ASE Processor

## Purpose

The purpose of the ASE Processor is to read messages from a queue, extract the body and then process it accordingly. This application runs as a windows service and when fired, the service will invoke the “Execute” method on any processors that have been created and configured to run.

For the auditing process, there are 3 types of processors:

1. ProcessAuditLog: handles processing for ActionAudit
2. DataAuditProcessor: handles processing for DataAudit
3. DataAuditFailureProcessor: handles processing for any data audit messages that fail processing

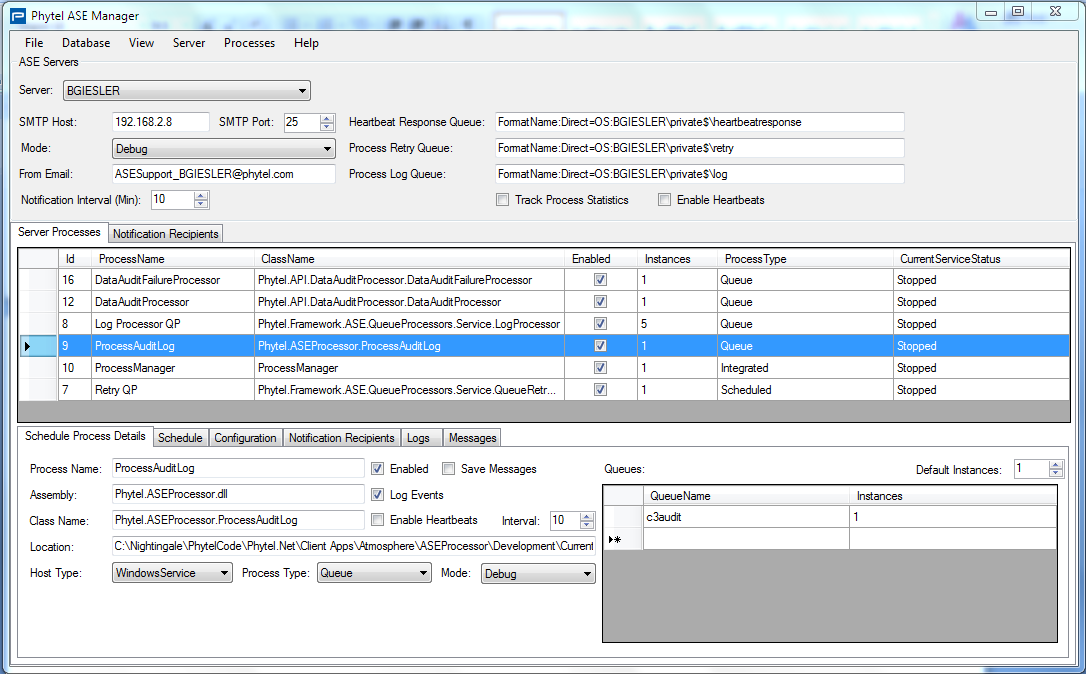
Each of these processes are configured using the “ASE Process Manager” application and contain specific details, based on their different uses.

### ProcessAuditLog.cs - This class can be found in the Phytel.ASEProcessor project

* + **public override void Execute(QueueMessage queueMessage)**

This method takes the queue message that was passed in, and based on the type of message, adds it to the appropriate table.

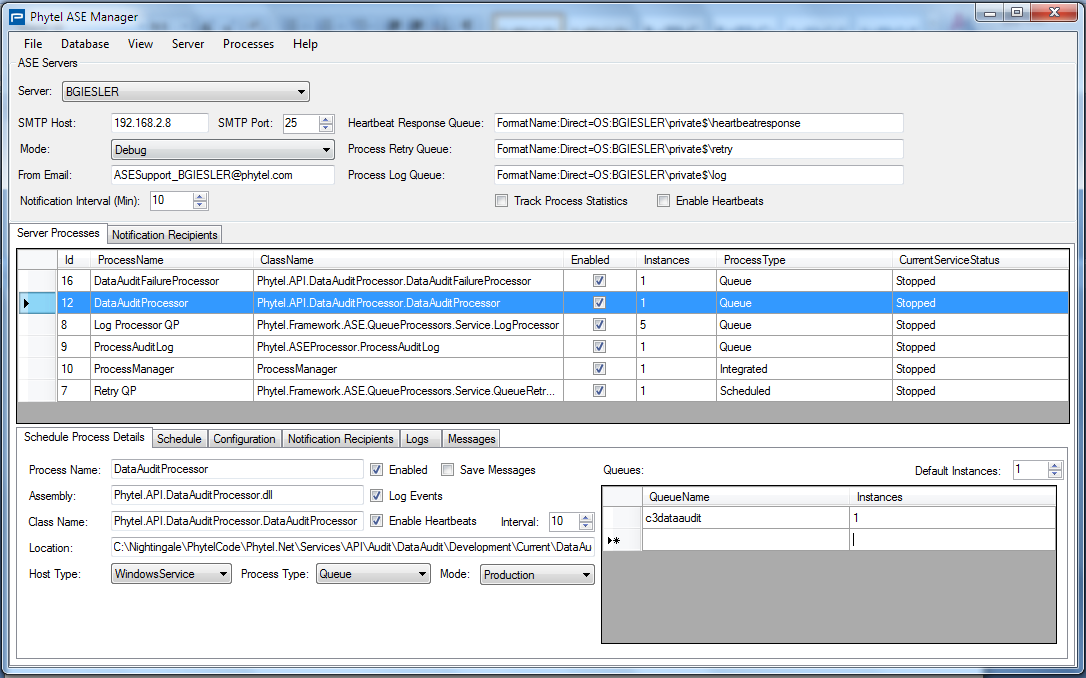
The configuration, which is set using the ASE Manager application, specifies the queue to read, to find available queue messages to use as input. The configuration also specifies what database connection to read, from the Services Manager application.



* **DataAuditProcessor.cs** – This class can be found in the Phytel.API.DataAuditProcessor project
  + **public override void Execute(QueueMessage queueMessage)**

This method takes the queue message that was passed in and adds it to the appropriate collection in the Mongo audit database.

The configuration, which is set using the ASE Manager application, specifies the queue to read, to find available queue messages to use as input. The configuration also specifies what database connection to read, from the Services Manager application.



* **DataAuditFailureProcessor.cs** – This class can be found in the Phytel.API.DataAuditProcessor project
  + **public override void Execute(QueueMessage queueMessage)**

This method takes the queue message that was passed in and writes it to the configured filepath as an XML file. Queue messages that impact this method, come from a “failure” queue. This queue is populated from messages that failed processing by either the Action Audit or the Data Audit, and must be captured to disk, for archiving.

The configuration, which is set using the ASE Manager application, specifies the queue to read, to find available queue messages to use as input. The configuration also specifies the filepath to write the queue message body information to.

