Email subscription utility for updates in Dyntaxa.

Jesper Andersson
Abstract

Email subscription utility for updates in Dyntaxa.

Jesper Andersson

Dyntaxa is a database system where one can find information about the Swedish animals and plants such as name(s), taxonomic hierarchy, and Swedish occurrence. One of the Dyntaxa’s requirements is that anyone should be able to subscribe and receive information about taxon’s updates by email. This is the task given by Artdatabanken at SLU that this thesis solves.

The report describes the developed email subscription utility for the Dyntaxa system. In the developed utility one can subscribe to any of the animals, organisms or plants existing in the database, and receive an email when data is updated. Furthermore, a study of three different message queue libraries was carried out in order to see the positive and negative features, and to choose the most well suited queue library for Dyntaxa.

The email subscription utility is now a working prototype that needs some more testing and minor changes to be put into production.
## Contents

1. Introduction ........................................................................................................................................... 2

2. Background ............................................................................................................................................... 3

3. Problem definition .................................................................................................................................. 5
   4.1 Publish revision extension .................................................................................................................. 11
   4.2 Gathering information and sending emails to subscribers ............................................................... 13

5. Related work .......................................................................................................................................... 24

6. Conclusion and Future work .................................................................................................................. 25

7. References .............................................................................................................................................. 26

Appendix A - The part of the TaxonDB that is used in this thesis ............................................................. 27
Appendix B - Stored Procedure to collect subscribers that should receive an update. ............................ 28
Appendix C - Code comparison of message queue libraries ..................................................................... 29
Appendix D - Two Code examples of the functions that creates the strings for the emails. ....................... 30
Appendix E - Stored procedure collecting the taxonid and name by revisionEventId. .............................. 31
Appendix F - Function compiling and sending all the emails .................................................................... 32
Appendix G – NServiceBus Receiver used in the application. .................................................................... 33
1. Introduction

Dyntaxa is a taxonomic database over Sweden’s naturally occurring organisms. Using this database one can get information about the organisms’ Swedish names, scientific names, synonyms, classification etc. Dyntaxa is developed and administered by the company ArtDatabanken at Swedish University of Agricultural Sciences (SLU). The content of the database is updated continuously with the help of taxonomic experts called taxonomists. These are the experts who describe and name species [2]. Many taxonomists work with regional fauna and flora descriptions, which includes identification names and information about the investigation of a species, status etc. “Nationalnyckeln till Sveriges flora och fauna” [3] is an example of projects taxonomists work with. The goal of Dyntaxa is to contain taxonomic information about all Swedish wildlife.

Dyntaxa is running in a production environment. In this system one can find information about any multicellular organism. If a person for example is interested in mushrooms he/she can look for what scientific name a specific mushroom has or what family it belongs to. In Dyntaxa one can also look up the Swedish occurrence, or if it is red listed. The Red list is a list over animals, mushrooms or plants that risks extinction from a region, a nation or the world [5].

There is a problem with the Dyntaxa system though. For example, if a scientist is doing research on some species and he/she wants to be sure that he is using inside the publications the correct up to date data about the species then he has to look it up every time when he wants to publish. This is the problem that this thesis solves by developing an email subscription utility which helps scientist or others interested to update themselves on data concerning species. The utility gives the users an opportunity to subscribe to a group of species or a single species. In this way it is guaranteed that the users receive emails about what has been changed in the database for the specie(s) they are subscribed to so they can be sure that they have the latest updated information.
2. Background

Artdatabanken or Swedish Species Information Centre started out as a project in 1980 at Swedish University of Agriculture Sciences - SLU under the name: "Databanken för hotade arter" in Swedish or Databank for endangered species in English. It was meant to contain data about endangered species. In July 1990 the work on the project was turned into a permanent unit at SLU. It was first some years ago in 2007 this unit changed its name into Artdatabanken (Database for species). Artdatabanken has continued to grow and a database system called Dyntaxa which contains information about most of the Swedish wildlife has been developed. Dyntaxa consists of many layers which can be seen in figure 1.

![Figure 1: Dyntaxa overview](image)

The top layer GUI ASPX is the User Interface, organized as a website. It sends requests to the application layer, which is called Business logic. Then the business logic sends requests to Taxon Service, User Service or some other service. The two important services involved in this thesis work are the User and the Taxon services. There are a couple of other services existing but they are not used here.

The User service and the Taxon services are the layers closest to the databases. Both User Service and Taxon Service are central services that are used by many applications, not only by Dyntaxa. The User Service is a service for user handling and user permissions for ArtDatabankens IT-department. The Taxon Service is a service for Taxonomic information. Both
services contain methods for reading and writing data to the separate databases UserDB and TaxonDB. The UserDB contains information about the users of Dyntaxa, i.e. their names, email addresses, ages, etc. The TaxonDB contains information about the animals, plants and organisms.

Artdatabanken is an important link between scientists and the public. Most of Artdatabankens activities falls within SLU:s environmental monitoring and assessment (EMA), which focuses on the needs of the society as expressed in the Swedish Parliament’s environmental quality objectives, international commitments and the overall objective for long-term sustainable development.
3. Problem definition

First we need to define some terms that are going to be used in the rest of this thesis:

- **Taxon** [1] (singular, Taxa plural) is a group of organisms with a scientific name. The expression is mostly used when one talks about a systematic group without giving its name or specifying its systematic rank.
- The term **revision** [13] refers to a set of changes made to any taxon in Dyntaxa.
- A **revision event** is one specific change in a revision.

Before the email utility was developed when a user of Dyntaxa wanted to know if an update had occurred for any of the many species, he/she had to remember all the latest information and then look it up each time to see if it had been changed. The reason was that there was no utility in Dyntaxa that showed if there had been an update. This thesis solves this problem by implementing a subscription utility where a person can subscribe to any taxon, and get information about the updates via email.

The utility would be very useful for scientists who are doing research about species or just persons who want to be updated on the new details about species. For example, if a scientist specializes in crustaceans and is currently writing an article about some special crabs he would like to be sure that he uses the correct name for those crabs. Before the proposed solution for email subscription utility he could never be sure what name to use without looking it up in the database. If this scientist would have had the feature of a subscription utility he would know that if he had not gotten an email the crabs data has not been updated. On the contrary, if he had received an email it would contain the newly updated crabs data.

A revision is built up of revision events. These events are the individual changes included in a revision. The revision changes in Dyntaxa are put directly into the Taxon database, (TaxonDB) in Figure 1. Then a flag that defines if the information is published is marked false. In that way no one but the taxonomist can see the changes. Before the proposed solution when a revision was published, Dyntaxa had to go through 3 major steps which can be seen in Figure 2.
The first step is to delete the redundant information from the revision, i.e. the information that has been changed more than once. This is done since Dyntaxa only cares about the valid and most up to date information. The second step is to remove the revision reference from the old data that now is a history data. This means that we can no longer see in what revision this data was changed. The last step is to set the old data to be invalid and the new to be valid and show that it is published. Now after a revision when everything has been changed the taxonomist who published the revision can get feedback that it is done.

Figur 2 Data flow showing how a revision was published.

The proposed solution for an email subscription utility solves three main problems. The first problem is that when a revision is published there is a lot that needs to happen before the taxonomist get any feedback, as explained in Figure 2. This is troublesome for the taxonomist who publishes since he does not want to wait for a long time to get feedback that the revision is published. Furthermore, the taxonomist does not care if emails were actually sent to the subscribers and thus he does not want to wait for this. If the application were to send all the emails to the subscribers sequentially with all the steps in Figure 2 it would take a long time before the taxonomist would get a feedback. That is why this should be done asynchronously to the rest of the processing. This problem is solved using a library called NServiceBus which sends messages to a program that handles the emails.

The second problem, which this thesis solves, is a problem concerning the fact that the taxon tree is a very complex tree since taxon can have multiple parents from different taxon groups. For example, in Figure 3 we can see an overview of the taxon Stictis populorum which is lichen. The family tree chosen to get to this lichen is the fungi family even though it is mainly lichen. Further down in the figure one can see under “Other parents” that this species aside from a fungi can be specified as two different kinds of lichens and a Saprophyte as well [4].
Since the taxon tree is so complex it is troublesome to figure out if a person should get information about an update or not. For example, if a person is subscribed to both Lichens and Fungi (see Fig.3), he would get two emails for the same taxon if, for example, some information about stictis populorum was changes in a revision. This is because this species is both a child to the Lichen’s sub-tree and the Fungi’s sub-tree. Furthermore, another problem with the tree structure is that for each subscription we need to find what children a taxon has that the person is subscribed to. If this is not done correctly it can take a long time to send all the emails if there are many subscribers.
Table 1 Specification on revision event type

<table>
<thead>
<tr>
<th>ID</th>
<th>Identifier</th>
<th>Changes included</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AddTaxon</td>
<td>A new species has been added.</td>
</tr>
<tr>
<td>2</td>
<td>RemoveTaxon</td>
<td>A taxon has been removed.</td>
</tr>
<tr>
<td>3</td>
<td>ChangeTaxonParent</td>
<td>A taxon has gotten a parent replaced.</td>
</tr>
<tr>
<td>4</td>
<td>AddTaxonParent</td>
<td>A taxon has gotten another parent.</td>
</tr>
<tr>
<td>5</td>
<td>RemoveTaxonParent</td>
<td>A taxon's parent has been removed.</td>
</tr>
<tr>
<td>6</td>
<td>ChangeTaxonSortOrder</td>
<td>The taxons in a family has changed sort order.</td>
</tr>
<tr>
<td>7</td>
<td>ChangeTaxonCategory</td>
<td>A taxon has been moved to another category.</td>
</tr>
<tr>
<td>8</td>
<td>LumpTaxa</td>
<td>Several taxons has become one.</td>
</tr>
<tr>
<td>9</td>
<td>SplitTaxon</td>
<td>One taxon has become several.</td>
</tr>
<tr>
<td>10</td>
<td>AddTaxonName</td>
<td>A taxon has gotten a new name.</td>
</tr>
<tr>
<td>11</td>
<td>EditTaxonName</td>
<td>A name of a taxon has been edited.</td>
</tr>
<tr>
<td>12</td>
<td>DeleteTaxonName</td>
<td>A name of a taxon has been deleted.</td>
</tr>
<tr>
<td>13</td>
<td>ChangeRecommendedName</td>
<td>Not used anymore.</td>
</tr>
<tr>
<td>14</td>
<td>ChangeTaxonNameCategory</td>
<td>Not used anymore.</td>
</tr>
<tr>
<td>15</td>
<td>ChangeTaxonNameUsageType</td>
<td>Not used anymore.</td>
</tr>
</tbody>
</table>

The third problem, solved by this thesis concerns the fact that the information about what is included in a revision, i.e. the information that is being updated/changed is spread out over several tables in the Dyntaxas database shown as Taxon DB in Figure 1. This information is also cryptic in the sense that it can be hard to see what actually happens in a revision event. Some of the tables in the TaxonDB database can be seen in Figure 4. In Dyntaxa a revision event has a type, represented by an integer value. This integer is used to determine exactly which update has been done. One can see the in Table 1 all the revision event types. So let’s say that a taxonomist finds that a group of taxa that were believed to be different species is really one and the same, so he merges them into one. To get this information out of the database it must first check in what revision event this is done. Then the revision event type can be obtained, showing what a single update is for. This is made clear in table 1. In the case of a merge of taxa, as explained in the example, the event type is the integer number 8 which is called LumpTaxa.

Knowing a revision event and type we can find what tables in Taxon DB have been changed. In these tables there is not always only one record updated for a revision event, it may be one record for each taxon or sometimes even multiple records per taxon updated. For example, if a search is done in the table LumpSplitEvent (in Appendix A) for a revision event merge of taxa, one record per each taxon included in the merge is obtained.
Figure 2 A part of Taxon Database, TaxonDB about species.

Figure 4 shows a part of Dyntaxas database called Taxon DB. These tables are a small part of the tables where one can find information about updates. These tables are filled when a taxonomist adds new changes to a revision. There are two things that changes in the tables when a revision is published. The first is that the Boolean attribute isPublished in tables TaxonTree, TaxonName, TaxonProperties and Taxon is changed to true. Before this is done no one can see that these changes exist except for the taxonomists. The second change is that the data updated several times in the revision can be removed, except for the data that specifies the new information that should be seen as the correct one. This data and the data that was the previously correct data is the only thing left in this database. The most troublesome about this is that the old data reference to what revision changed is removed. This is done because previously Artdatabanken never used this information. In order to be able to use the information before removing it we propose to copy it from TaxonName, TaxonProperties and TaxonTree tables to temporary tables (shown in figure 6), which will keep it for a certain amount of time.

In Appendix A there is an overview of the part of the Dyntaxa database, and the TaxonDB that is used for storing all this data.
4. Implementation of the Email utility

Figure 3. Dataflow showing what happens now when a revision is published.
In this chapter the developed Email subscription utility, together with the publish revision extension will be described. In Figure 5 one can see the data flow for the developed email utility. It is divided into two units, one that handles publishing of revisions and one that sends emails to the subscribed users.

4.1 Publish revision extension
The publish revision extension unit is an extension of the Dyntaxa system that handles revisions in the TaxonDB. It starts running, as shown in Figure 2, when a taxonomist presses the publish revision button. During the first step (see Fig. 2) it saves the data that will be removed in step 3 into a temporary table, so that the data can be used to gather information about what has happened in the revision. This addresses the third problem explained in the problem definition chapter where some of the data was deleted before because it was not used. Thus three tables, called Temporary data tables have been designed to store temporary data during a revision process.

![Fig. 4. The temporary data tables.](image)
The design of the temporary data tables can be seen in Figure 6. In these tables is stored only the necessary information needed to see what has been changed in the TaxonDB during a revision.

The temporary table *SubscriptionTaxonName* stores temporary data about the names of updated taxa in a revision. In the top of table *SubscriptionTaxonName* there is the primary key attribute ID which is an automatically generated unique number to identify a taxon. Then there is the attribute *TaxonID* which is the identity of an updated taxon and is a foreign key to the table *Taxon in TaxonDB* shown in Figure 4. The *Name* attribute represents the name of the taxon. The *Name* can be the scientific name, the Swedish name or name in any other language. The way the name is presented, is specified by the next attribute *NameCategory* which is a foreign key to a table called NameCategory in TaxonDB having two attributes, id and category name. The attribute *Author* in Fig. 6 consists of the name of the person who invented the taxon’s name as well as the year it was done. For example, a possible value for the Author attribute can be the text “Adam Andersson, 1990”. The attribute *Isrecommended* is of the type Boolean and shows if the taxon’s name is the recommended one to use. The last two columns *RevisionEventId* and *ChangeRevisionEventId* are foreign keys to the Taxon DB’s table *RevisionEvent* in Figure 4. They specify in which revisionEvent the data was added and in which revision it was changed in.

The temporary table *SubscriptionTaxonProperties* stores temporary data about the properties of the updated taxa in a revision. The table *SubscriptionTaxonProperties* has as a primary key, the attribute *ID* that is an automatically generated unique number to identify a property. Then the second attribute is *TaxonID* which is the id of the taxon and a foreign key to the table Taxon in TaconDB. The next attribute *TaxonCategory* is a foreign key to a table called TaxonCategory that contains Id and CategoryString. TaxonCategory specifies if the taxon is a class, a family or a single species stored as an integer. This integer is taken from the Taxon DB table called *TaxonCategory*. The attribute *ConceptDefinitionPartStringId* is referencing another TaxonDB table that stores comments from the taxonomist who made the revision. The last two attributes *RevisionEventId* and *ChangeRevisionEventId* are the same as in the previously explained table. They are needed to locate in what revision the properties where changed.
The third temporary table *SubscriptionTaxonTree* stores temporary data about the taxon tree of updated taxa in a revision. The primary key of the table SubscriptionTaxonTree is the ID attribute which is automatically generated unique number to identify a relation between two taxa in the taxon tree. The attribute *TaxonIDParent*, specifies a taxon id that is a parent node to the taxon with id *TaxonIDChild*. Both of these taxon ids are foreign keys to the TaxonDB’s table Taxon. The attribute *isMainRelation* shows if this is the main relation for the child taxon or there exists another branch that goes to this child taxon and it is the main relation. A main relation can be explained as follows: a person can have many parents, but only one can be the biological parent, so one can see the parent of the main relation as the biological parent.

Step 2 in the publish revision extension unit addresses the problem that the taxonomist does not need to wait for the emails to be sent in order to get feedback that the revision is published. So here the unit sends a message containing the revision ID to a message queue using NServicebus library [10]. The revision ID is used to find the revision itself from the table Revision shown in Appendix A. An explanation of how NServiceBus works and why the unit uses NServiceBus can be found further down in the subchapter “Comparison of message queuing frameworks”.

After the second step, during the third and the fourth step the unit deletes the redundant information from the revision, i.e. the information that has been changed more than once, and then removes the revision reference from the old data that now is a history data.

The fifth, the last step in the publish revision extension is to set the old data to be invalid and the new to become valid and show that it is published.

### 4.2 Gathering information and sending emails to subscribers

The gathering and sending information unit sends emails to the users. It is triggered whenever there is a message in the queue generated by the publish revision extension unit. It retrieves the revision id that exists in the triggering message sent by the publish revision extension unit. Then the processing goes to step 1 (the lower part in Fig.5) where the receiver class in C# is used to look and see if there is a message in the specified queue. In Appendix G one can see the code of the receiver. The receiver writes the revision id to a log file and to the console, then it passes it as an argument to another method that starts to
collect the taxons involved in the update and all the necessary information needed to send all the e-mails.

In the second step of getting all subscriptions the problem concerning the difficult tree structure described in the “Problem definition” is addressed since it is needed to collect all the taxon ids involved in a revision.

The example below explains the collection of all the taxon ids that are concerned, the concerned taxon tree, including both the updated taxa and the parents of the updated taxa.

![Taxon Tree Diagram](image)

In Figure 7 one can see an example of a taxon tree where the numbers 1-17 are taxon ids. We suppose that there were some changes in a revision made in taxa 4, 5, 8-12 marked dark grey. Although, the taxon id that can be found in the TaxonDB’s Revision table (shown in Fig. 4) as TaxonId is the taxon with id 2 since this is the highest taxon covering all involved taxa. We call this node the top node, which can be found in the revision table in TaxonDB. The part of the stored procedure (Appendix B) that collects the top node is:

```sql
Declare @taxonId int
Select @taxonId = TaxonId
from Revision
Where id = @revisionId
```
So one can see that the collecting starts by taking the revision Id from the triggering message and querying the Revision table for the TaxonId where the id is equal to @revisionId. Then the collected taxon ids of the updated taxa, i.e. the children and grandchildren and so forth to the top taxon, are inserted in a temporary table by the query shown below.

\[
\text{INSERT INTO } \#\text{TmpTaxonTree (TaxonId)} \\
\text{SELECT taxonId} \\
\text{FROM TaxonInRevision} \\
\text{WHERE RevisionId = } @\text{revisionId}
\]

To get the parents of the top node, one needs to search through the taxonTree table (seen in Figure 4) starting from top taxon as shown in the following query:

\[
\text{SELECT t.taxonidparent, t.taxonidchild} \\
\text{FROM TaxonTree t} \\
\text{WHERE t.TaxonIdChild = } @\text{taxonId} \\
\text{AND t.IsPublished = 1} \\
\text{AND t.ValidToDate > GETDATE()}
\]

The first part collects the first parent to the top taxon as @taxonId. To make sure that it is the correct relation it finds it checks if the relation is published (ISPublished=1), and if the valid date is bigger than the current date (ValidToDate > GETDATE() ). To collect all the parents the result of the previous query is in UNION ALL with the result of the following query:

\[
\text{SELECT t.TaxonIdParent, t.TaxonIdChild} \\
\text{FROM TaxonTree t} \\
\text{INNER JOIN TaxonTreeUpperCTE cte ON t.TaxonIdChild = cte.TaxonIdParent} \\
\text{AND t.IsPublished = 1} \\
\text{AND t.ValidToDate > GETDATE()}
\]
When all the parents are collected they are put into the temporary table \#TmpTaxonTree with this query:

```
INSERT INTO \#TmpTaxonTree (TaxonId)
SELECT cte.TaxonIdParent
FROM TaxonTreeUpperCTE cte
```

Then all the collected nodes stored in \#TmpTaxonTree table would form the concerned taxon tree. In Figure 8 it is shown how the concerned taxon tree looks like for the example in Fig. 7.

To store the subscribers data the table UserSubscriptions (Figure 9) was created. It connects the users with the taxa. The first attribute ID is the primary key and is an automatically generated unique number to identify a user. The attribute UserId is an id that refers to a user stored in the table user in the User DB in Dyntaxa (seen in Fig.1).. The attribute TaxonId is a foreign key that refers to ID in table Taxon in TaxonDB.

Once the concerned taxon tree has been obtained (as shown in Fig. 8) the gathering and sending information unit continues with step 2. Knowing all the taxa' ids a a search in the table UserSubscriptions (Figure 9) is done in order to get all the UserIds of the users subscribed to a taxon in the concerned taxon tree or a taxon in the tree that is a child of a subscribed taxon. The query for searching the users is the following:
Both step 1 and 2 in this unit are implemented as one stored procedure in MS SQL. The procedure can be seen in Appendix B.

In step 3 The UserId’s obtained from step 2 are used to search in Dyntaxa’s user DB, i.e. the table Email in order to get the email addresses corresponding to the UserId’s.

Now it is known who to send emails to, but it is not yet known what taxon’s each subscriber cares about. It is only known what taxa a subscriber is subscribed to. Step 4 (the lower part in Fig.5) is responsible for collecting the tree of taxon each subscriber cares about. The reason is that if a subscriber is subscribed to a taxon he is not only interested in that taxon but in all the children and grandchildren of it. The way this is done is by searching for all the children of a subscribed taxon in the TaxonTree table in TaxonDB (Figure 4) by the following C# function:

```csharp
public static List<List<int>> GetTaxonTreePerTaxon(List<int> taxonList)
{
    int previousTaxon = -1;
    List<int> previousTree = null;
    List<List<int>> taxonsForUserList = new List<List<int>>();

    foreach (var taxon in taxonList)
    {
        if (previousTaxon == taxon)
        {
            taxonsForUserList.Add(previousTree);
        }
        else
        {
            SqlDataReader rdr = DatabaseManager.GetTaxaTreeByTaxonId(taxon);
            List<int> tempList = new List<int>();
            tempList.Add(taxon);
            while (rdr.Read())
            {
                tempList.Add(rdr.GetInt32(0));
            }
            rdr.Close();
            previousTree = tempList;
        }
    }
    return taxonsForUserList;
}
```
previousTaxon = taxon;

taxonsForUserList.Add(tempList);
}
}

In this function called GetTaxonTreePerTaxon a list of all the subscribed taxon is supplied. For each taxon in this list the function checks if the taxon is the same as the previously checked taxon. If it is, it sets the tree of the taxon to be the same as the previously checked taxon, which is done to minimize the stress to the database and to optimize the program. If the taxon is not the same as the previously checked it calls the method DatabaseManager.GetTaxaTreeByTaxonId (taxon) that calls a stored procedure in Dyntaxa collecting all the children to a given taxon.

An example of how this function works can be described with a subscriber subscribed to taxon 1 in figure 7. Then the function would search for all the children of taxon 1 and it would find out information about all the nodes seen in figure 7.

For each subscription we now have a list of taxonid’s that subscribed users want to get information about. To gather the information of what has happened in the revision (step 6) one first needs to get all the revisionEvents’ Ids and types from RevisionEvent table in Figure 4. This is done in step 5 by the query seen below:

```sql
SELECT RE.Id, RE.RevisionId, RE.EventType
FROM RevisionEvent RE
WHERE RE.RevisionId = @RevisionId
Order by RE.Id
```

In step 6 the program of the unit starts to collect data that is needed to create the strings composing the emails. This is done by first looking at each revision event in a revision, more specifically looking at its event type. When the program knows the event type it goes through the function below to decide what function for composing email string is supposed to be used. The function GetAllStringForEmail in C# shown below takes the revision event list as input argument. Then it loops over the list and finds which function to call regarding to the type of an event. If an event has an event type 1, i.e. a taxon has been added, the function GetAddTaxonInfo will be called. The code of GetAddTaxonInfo together with the function GetAddTaxonNameInfo called from GetAllStringForEmail can be seen in Appendix D.

```csharp
public static List<EmailSubstring> GetAllStringForEmail(List<RevisionEvent> revEventList)
{
    TextList = new List<EmailSubstring>();
    foreach (RevisionEvent rEvent in revEventList)
    {
```
switch (rEvent.eventType)
{
    case 1:
        GetAddTaxonInfo(rEvent.id);
        break;
    case 2:
        GetRemoveTaxonInfo(rEvent.id);
        break;
    case 3:
        GetChangeTaxonParentInfo(rEvent.id);
        break;
    case 4:
        GetAddTaxonParentInfo(rEvent.id);
        break;
    case 5:
        GetRemoveTaxonParentInfo(rEvent.id);
        break;
    case 7:
        GetEditTaskInfo(rEvent.id);
        break;
    case 8:
        GetLumpTaxaInfo(rEvent.id);
        break;
    case 9:
        GetSplitTaxaInfo(rEvent.id);
        break;
    case 10:
        GetAddTaxonNameInfo(rEvent.id);
        break;
    case 11:
        GetEditTaxonNameInfo(rEvent.id);
        break;
    case 12:
        GetDeleteTaxonNameInfo(rEvent.id);
        break;
}
return TextList;
The function `GetAddTaxonInfo` begins with calling the function

```
DatabaseManager.GetTaxaByRevisionEventId(revisionEventId)
```

which in turn calls a stored procedure searching the table Taxon using the revision event id. This stored procedure can be seen in Appendix E. When this is done we will get 1 row back containing the taxon added and both its Swedish name and its scientific name. Once the information is extracted from the query, a String builder 16 in C# is called to optimize the creation of the string. This is needed since the string has to be manipulated many times. When the string is created it is added to a C# object together with the taxon id concerned. Then that object is added to the final list of objects containing the strings for the emails. The class for this object can be seen below:

```
public class EmailSubstring
{
    public int revisionEventId { get; set; }
    public int taxonId { get; set; }
    // the text string that will be included in the emails.
    public String text { get; set; }
}
```

When the data concerning a revision has been constructed into a string and saved together with the taxon id’s, the last step is carried out to compile the individual emails for every subscription and send them. The function written in C# which sends the emails can be seen in Appendix F. The function compiles each email for each subscription in turn. When compiling one of the emails the function first extracts all the `EmailSubstring` Objects from a list called messageContents. Inside an `EmailSubstring` Object it exists taxon ids of the list of taxa a user wants updates on. When the function knows all the strings it can start putting the email message together. It does that by using a string builder and loops over all the strings and appends them to a single string. When this is done a call to the `TaxonDB` database table `TaxonName` is done to get the subscribed taxon’s name for the subject of the email.

When an email is created it is sent and the function continues with the next subscription in the list `userList`. An example of the content of such an email can be seen in Figure 10. This email contains information about a revision where the subscription is for the top node (Life).
Each of the rows in the email in Figure 10 is created by a single function similar to the function in Appendix D.

4.3 Comparison of message queuing frameworks

To solve the first problem explained in Problem definition which is that the taxonomist does not want to wait for the emails to be sent in order to get a feedback from a revision amessage queuing framework is used to send a message to a separate program that can run independently to the other processing explained in Problem definition. To solve this problem the most basic feature of the queing framework is needed which is send and receive. The sender i.e. the one who publishes a revision only wants to send a command, he does not need to know when the sending is finished.
Send and receive is illustrated by Figure 11: A sender sends a message to a queue and the receiver reads from this queue. The send/receive pattern is ideal for creating 'command pipelines', where one wants a buffered channel to a single command processor [11].

To decide what message queue framework would fit best in the application a comparison of three different variants of message queuing frameworks was carried out. The tested frameworks were: NServiceBus[10], EasyNetQ[12] and RabbitMQ[8].

NServiceBus is a library for message queues developed by the company Particular software which one need a license to use in a live production. NServiceBus can be used in a number of ways using features like publish and subscribe, send receive and many more advanced features.

RabbitMQ is a messaging broker library. It gives an application a common platform to send and receive messages, and the messages a safe place to live until they are received[8]. RabbitMQ is written in Erlang programming language. The messaging broker is available for all the biggest programming languages such as c#, python and PHP.

Another framework that was studied is EasyNetQ. EasyNetQ is an open source library that works on top of RabbitMQ, it strives to make it easier to work with RabbitMQ in .NET. EasyNetQ is not as flexible as NServiceBus and RabbitMQ. In order to make it as easy as possible some simple conventions have been enforced, i.e.

- Messages should be represented by .NET types.
- Messages should be routed by their .NET type.

The meaning of this is that each specific message is represented by its own class. This Message class should not contain any functionality but it should be seen as a simple data container or Data Transfer Object (DTO). A form of DTO is also used in NServiceBus.

In Appendix C can be found code for a receiver for each of the compared queue frameworks. There we can see that RabbitMQ has the most cluttered and hard to understand code. One can also see likenesses between NServiceBus and EasyNetQ. The difference between NServiceBus and EasyNetQ in the receiver is that EasyNetQ uses a main function to state what messages it should listen too. This is done much cleaner in NServiceBus where you use the interface IHandleMessages<> and send in the type.
How easy the code is to use or how clean the code is not the only thing one want to consider when deciding which library to use. One should consider how one may want to use it in the future or if one can get help if one gets stuck. Also how well used the library is may be in favor because many of the questions one might have might already be answered. Another thing to consider is how the maintenance is done.

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy To use, Nice clean code.</td>
<td>Need Licen to use in Production/Commercial support.</td>
</tr>
<tr>
<td>Well used, many to get help from.</td>
<td></td>
</tr>
<tr>
<td>Good documentation.</td>
<td></td>
</tr>
<tr>
<td>Many features. Can be used in many ways.</td>
<td></td>
</tr>
<tr>
<td>NServiceBus</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good Performance tools</td>
<td>Hard to read, messy code.</td>
</tr>
<tr>
<td>Many features. Can be used in many ways.</td>
<td>Need licen for commercial support.</td>
</tr>
<tr>
<td>Okay documentation</td>
<td></td>
</tr>
<tr>
<td>RabbitMQ</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy to use, Clean code.</td>
<td>Not that many features.</td>
</tr>
<tr>
<td>Okay documentation</td>
<td>No commercial support.</td>
</tr>
<tr>
<td>Licen free.</td>
<td></td>
</tr>
<tr>
<td>EasyNetQ</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10 Pros and cons for the libraries.

What the proposed solution for the email subscription utility ended up using was NServiceBus because of the reasons seen in Figure 12. It shows the pros and cons for each of the compared libraries.

NServiceBus was much easier to use. It was also better documented than the other two. A bonus that both RabbitMQ and EasyNetQ was their monitoring UI [9]. There one can see how a queue and receivers are coping, how many messages are sent etc. This though is not that big advantage since NServiceBus can use RabbitMQ as well and utilize this monitoring UI if needed.

When it comes to help if one gets stuck it was found that NServiceBus has more users than the other systems. This produces more help among forums, etc. Only NServiceBus and RabbitMQ provide commercial support.
5. Related work

Nowadays there are loads of forums, social networks and other homepages that have a function where one can subscribe to some feature on a site or a thread on a forum or someone’s page on Facebook etc. Although, since there is no other system like Dyntaxa it is quite hard to find features similar to the features of the developed email subscription utility.

Twitter which is a social media does not support the feature of subscriptions but there is a site called Twilert [6] that does this. Twillert is a tool to search for specific tweets on twitter. One can define keywords and filters and set it up so that you will get all the tweets that contain these keywords or are in line with the filters by email. This is different from the purpose of the developed email subscription utility for the Dyntaxa database

Forums like Sweclockers[14] often use email subscription so one can get updates when someone writes in a specific thread.

There is also a Bird watching site called bird alarm [7]. It allows one to subscribe to a certain spot on the map and when someone else posts a sighting inside this area one gets an update in a phone app. This service is quite different from the developed email subscription utility since there are only bird sightings stored in the bird alarm’s database. On the contrary Dyntaxas database is much bigger which is more difficult to handle. Another thing is that in bird alarm anyone who pays for the service can post a sighting. In Dyntaxa one needs to be certified to be able to post some information. Bird alarm has a yearly cost of 40 euro, Dyntaxa is free to use. Anyone can use the Dyntaxa database and get updates by the developed email subscription utility which is the contribution of this thesis work.
6. Conclusion and Future work

The work in this project resulted in an email subscription utility for the Dyntaxa database. This utility makes it possible for a user to subscribe to a taxon which can be a family of species, a class or even a single species and receive updates on this taxon made by taxonomist employed by SLU.

In general the developed utility works as intended but needs some more testing to be able to be released to the public. Although the goal explained in the problem definition chapter was achieved, and the developed utility works in a test environment and could with some minor tweaks and testing be used in production.

I have also spent a lot of time comparing three libraries for message queues. The result of the comparison can be seen in the “Comparison of message queuing frameworks” chapter. It would be beneficial for people trying to pick one of these libraries,

One improvement to the developed utility might be to specify what level of details a user wants to get updates on a specific taxon, i.e. if one only wants to know when a new taxon is added or if a new name has been added etc. This can be seen as an add-on to the main feature.

Another improvement suggested during the projects time is to make the message sending via NServiceBus easily expandable so that some other site or program can be notified when a revision is published.
7. References


Appendix A - The part of the TaxonDB that is used in this thesis.
Appendix B - Stored Procedure to collect subscribers that should receive an update.

CREATE PROCEDURE GetSubscriptionsByRevisionId
@revisionId int
AS
BEGIN
-- SET NOCOUNT ON added to prevent extra result sets from interfering with SELECT statements.
SET NOCOUNT ON

Declare @taxonId int
Select @taxonId = TaxonId from Revision Where id = @revisionId

-- create a temp worktable
CREATE TABLE #TmpTaxonTree (TaxonId int);

BEGIN
WITH TaxonTreeUpperCTE (TaxonIdParent, TaxonIdChild) AS
(
    -- get the anchor
    SELECT t.TaxonIdParent, t.TaxonIdChild
    FROM TaxonTree t
    WHERE t.TaxonIdChild = @taxonId
    AND t.IsPublished = 1
    AND t.ValidToDate > GETDATE()

    -- recursively union upper levels
    UNION ALL

    SELECT t.TaxonIdParent, t.TaxonIdChild
    FROM TaxonTree t
    INNER JOIN TaxonTreeUpperCTE cte ON t.TaxonIdChild = cte.TaxonIdParent
    AND t.IsPublished = 1
    AND t.ValidToDate > GETDATE()
)

-- insert ids into tmp table
INSERT INTO #TmpTaxonTree (TaxonId)
SELECT cte.TaxonIdParent FROM TaxonTreeUpperCTE cte
END

-- insert the bottom half of the tree
INSERT INTO #TmpTaxonTree (TaxonId)
SELECT taxonId FROM TaxonInRevision
WHERE RevisionId = @revisionId
Appendix C - Code comparison of message queue libraries.

RabbitMQ Receiver:

```csharp
public class RabbitMQReceiver
{
    public static void Main()
    {
        var factory = new ConnectionFactory() { HostName = "localhost" }; using (var connection = factory.CreateConnection())
        {
            using (var channel = connection.CreateModel())
            {
                channel.QueueDeclare("RabbitMQQueue", false, false, false, null);
                var consumer = new QueueingBasicConsumer(channel);
                channel.BasicConsume("RabbitMQMessage", true, consumer);
                while (true)
                {
                    var ea = (BasicDeliverEventArgs)consumer.Queue.Dequeue();
                    var body = ea.Body;
                    var message = Encoding.UTF8.GetString(body);
                    Console.WriteLine("[x] Received {0}", message);
                }
            }
        }
    }
}
```

EasyNetQ Receiver:

```csharp
public class EasyNetQReceiver
{
    static void Main(string[] args)
    {
        using (var bus = RabbitHutch.CreateBus("host=localhost"))
        {
            bus.Subscribe<EasyNetQMessage>("messages", Handle);
        }
    }
    static void Handle(TextMessage textMessage)
    {
        Console.WriteLine("ID: ", textMessage.Id);
        Console.WriteLine("Description : ", textMessage.description);
    }
}
```
NServiceBus Receiver:

```csharp
public class NServiceBusReceiver : IHandleMessages<NServiceBusMessage>
{
    public IBus Bus { get; set; }
    public void Handle(NServiceBusMessage message)
    {
        Console.WriteLine(@"ID: ", message.Id);
        Console.WriteLine(@"Description : ", message.description);
    }
}
```

Appendix D - Two Code examples of the functions that creates the strings for the emails.

```csharp
public static void GetAddTaxonInfo(int revisionEventId)
{
    SqlDataReader rdr = DatabaseManager.GetTaxaByRevisionEventId(revisionEventId);
    while (rdr.Read())
    {
        EmailSubstring temp = new EmailSubstring { taxonId = rdr.GetInt32(0),
            revisionEventId = revisionEventId }
        StringBuilder sb = new StringBuilder();
        sb.Append("A new taxon was added with ID = ");
        sb.Append(temp.taxonId);
        if (!rdr.IsDBNull(1))
        {
            sb.Append(" with Scientific Name = ");
            sb.Append(rdr.GetString(1));
        }
        if (!rdr.IsDBNull(2))
        {
            sb.Append(" and swedish: ");
            sb.Append(rdr.GetString(2));
        }
        sb.Append(". Author: ");
        sb.Append(rdr.GetString(3));
        sb.Append(". Link to taxon: http://dyntaxa.se/Taxon/Info/";)
        sb.Append(*temp.taxonId);
        temp.text = sb.ToString();
        TextList.Add(temp);
    }
    rdr.Close();
}
```
Appendix E - Stored procedure collecting the taxonid and name by revisionEventId.

```
-- Author: Jesper Andersson
-- Change date: 2014-04-10
-- Description: Get Taxon identified by revisionEventId and ChangedInRevisionEventId.
CREATE PROCEDURE GetTaxaByRevisionEventId
    @RevisionEventId int
AS
BEGIN
    SET NOCOUNT ON
    SELECT T.Id, ScientificName.Name, CommonName.Name, ScientificName.Author
    FROM Taxon as T
    LEFT OUTER JOIN SubscriptionTaxonName AS ScientificName
    ```
Appendix F - Function compiling and sending all the emails.

```csharp
public static void SendEmail(List<string> emailList, List<int> userList, List<List<int>> taxonList, List<EmailSubstring> messageContents)
{
    for (int i = 0; i < userList.Count(); i++)
    {
        List<EmailSubstring> tempList = messageContents.Where(contents => taxonList[i].Contains(contents.taxonId)).ToList();
        tempList = tempList.GroupBy(substring => substring.text).Select(g => g.First()).ToList();
        List<EmailSubstring> sortedTempList = tempList.OrderBy(o => o.revisionEventId).ToList();

        if (tempList.Count != 0)
        {
            String emailToSendTo = emailList[userList.Distinct().ToList().IndexOf(userList[i])];
            StringBuilder sb = new StringBuilder();

            foreach (EmailSubstring eSubstring in tempList)
            {
                if (eSubstring.text != "")
                {
                    sb.Append(Environment.NewLine);
                    sb.Append(eSubstring.text);
                }
            }
            message.From = new System.Net.Mail.MailAddress("jesper.andersson@slu.se");
            message.To.Add(emailToSendTo);
            String taxonName = "";
            SqlDataReader rdr2 = DatabaseManager.GetTaxonNameByTaxonId(taxonList[i][0]);
            while (rdr2.Read())
            {
                taxonName = rdr2.GetString(0);
            }
        }
    }
}```
Appendix G – NServiceBus Receiver used in the application.

```csharp
public class RevisionCheckInHandler : IHandleMessages<RevisionCheckIn>
{
    public IBus Bus { get; set; }

    private static readonly ILog log = LogManager.GetLogger(System.Reflection.MethodBase.GetCurrentMethod().DeclaringType);

    public void Handle(RevisionCheckIn message)
    {
        log4net.Config.XmlConfigurator.Configure();

        Console.WriteLine(@"Starting the operation of sending emails to the subscribers of taxons that are in revision:{0}", message.RevId);

        SubscriptionManager.SendEmailToSubscribersByRevisionId(message.RevId);

        Console.WriteLine(@"Sent the emails that needed to be sent for revision:{0}" , message.RevId);
    }
}
```