EPIM Sync
Verifying information on inRiver and Episerver

EPIM Synk
Verifiera information på inRiver och Episerver

Adam Hellberg and Tim Hemdal

Department of Computer Science
Dissertation
15hp
Donald F. Ross
Lothar Fritsch
7th June 2017
When inRiver pushes to Episerver through a connector, there is no sure way to guarantee that all information is transferred without corruption. This project aims to create a service which pulls all product information from inRiver and Episerver and does a field by field comparison and creates a checksum for the values. The result of the comparison will be saved to a database and can be displayed graphically as a tree structure on a website. The project successfully achieved all the goals defined in the specification.
Acknowledgements

The authors would like to extend their thanks to Donald F. Ross at Karlstad University for his guidance throughout the writing of the dissertation. Your advice in the writing of this dissertation has been invaluable and we are forever grateful for it.

We would also like to thank Magnus Odell at Ninetech for giving us the opportunity to do our project there and the guidance provided throughout the project. Thanks also to all employees at Ninetech for making us feel welcome.

Adam Hellberg and Tim Hemdal

I would like to thank Bjørn Tore Håvie and Brandon Scott for all your programming advice and support over the years.

Thanks also to Pontus Gagnero and Samuel Häggmyr for providing support and listening ears.

Adam Hellberg

I would like to extend my thanks to my family who have supported me during my education.

A special thanks to Adam Hellberg who have worked with me and helped me throughout the years at Karlstad University.

Tim Hemdal

This document is compiled with \LaTeX{} and biber, using \LaTeX{} for reference management.

The template is based on KOMA-Script with a custom title page adapted from the official KAU template.
# Contents

1. **Introduction**  
   1.1. Goal .................................................. 1  
   1.2. Customer & Motivation ................................. 1  
   1.3. Project results ....................................... 1  
   1.4. Dissertation layout .................................. 2  

2. **Background**  
   2.1. Introduction ........................................... 3  
   2.2. Chapter layout ......................................... 3  
   2.3. PIM ...................................................... 4  
   2.4. inRiver as a PIM provider ............................ 5  
   2.5. Episerver as an e-commerce and CMS provider ....... 5  
   2.6. Checksums ............................................ 6  
   2.7. Job scheduler ......................................... 7  
   2.8. Tools and languages ................................. 7  
      2.8.1. VS .................................................. 7  
      2.8.2. C# ................................................ 7  
      2.8.3. .NET ............................................. 7  
   2.9. Problem .............................................. 8  
   2.10. Summary .............................................. 9  

3. **Project design**  
   3.1. Introduction .......................................... 10  
   3.2. Chapter outline ....................................... 11  
   3.3. inRiver ............................................... 11  
      3.3.1. inRiver bridge .................................. 11  
   3.4. Episerver ............................................ 12  
      3.4.1. Episerver bridge ................................ 12  
   3.5. Checksums ............................................ 13  
   3.6. EPIM Core ............................................ 14  
      3.6.1. Client interface ................................ 14  
      3.6.2. Data formats ................................... 14  
   3.7. Website and database ................................ 15  
   3.8. Job scheduler ........................................ 15
Contents

3.9. Summary .......................................................... 16

4. Project implementation ........................................... 17
  4.1. Introduction .................................................... 17
  4.2. Chapter outline ............................................... 17
  4.3. inRiver bridge ............................................... 17
    4.3.1. Data classes ........................................... 18
  4.4. Episerver .................................................... 19
    4.4.1. Episerver bridge .................................... 20
  4.5. Checksums ................................................... 20
    4.5.1. AlgorithmChecksumBuilder ............................ 22
    4.5.2. Creating the CRC32 builder ............................ 22
  4.6. EPIM Core .................................................. 23
    4.6.1. Client interface ...................................... 23
    4.6.2. Data formats .......................................... 23
    4.6.3. Comparison classes .................................. 25
  4.7. Website .................................................... 26
    4.7.1. Historical view ....................................... 28
  4.8. Job scheduler ............................................... 29
    4.8.1. Comparison job ....................................... 29
  4.9. Summary .................................................... 30

5. Evaluation .......................................................... 31
  5.1. Introduction .................................................. 31
  5.2. Chapter outline ............................................... 31
  5.3. inRiver bridge ............................................... 31
  5.4. Episerver bridge ............................................ 31
  5.5. Checksums ................................................... 32
  5.6. EPIM Core .................................................. 33
  5.7. Website .................................................... 33
  5.8. Summary .................................................... 34

6. Conclusion ........................................................ 35
  6.1. Introduction .................................................. 35
  6.2. Chapter overview ............................................ 35
  6.3. inRiver ..................................................... 35
  6.4. Episerver ................................................... 35
  6.5. Future work ................................................ 36
    6.5.1. Icons on tree nodes .................................. 37
    6.5.2. Comparison statistics ................................ 37
    6.5.3. Automatic triggering .................................. 38
List of Figures

2.1. Illustration of a PIM system. ................................. 4
2.2. Generating a checksum from product information. ................................. 6
2.3. Illustration of the .NET stack. ................................... 8
2.4. Flowchart describing the finished project. ................................... 9

3.1. Main project components and their relations with each other. ... 10
3.2. A visual mapping of properties. ................................... 13
3.3. Example of a EPIM Core data tree. ............................... 15

4.1. inRiver client implementation. ................................... 18
4.2. The channel helper class. .................................... 18
4.3. Episerver client implementation. ................................ 19
4.4. Diagrams for checksum classes. ................................ 21
4.5. Client interface in core library. ................................. 23
4.6. EPIM Core node interface. .................................... 24
4.7. EPIM Core abstract node class. ................................ 24
4.8. Interfaces for information comparison. .......................... 25
4.9. Comparison implementation classes. ............................ 25
4.10. Screenshot of comparison results on website. ................. 27
4.11. Example of the historical view showing past results. ........... 28

E.1. inRiver data class helpers. .................................... 49
List of Tables

4.1. Database table schema for comparison results. . . . . . . . . . . . . . . 29
B.1. Checksum examples using the MD5 hash algorithm. . . . . . . . . . . 45
B.2. Checksum examples using the CRC32 algorithm. . . . . . . . . . . . . 45
D.1. Comparison of string encoding techniques. . . . . . . . . . . . . . . . . 48
List of Listings

3.1. Example of JSON object. ........................................ 12
3.2. Example of XML object. ........................................ 12

4.1. Episerver data (JavaScript Object Notation (JSON)). .... 20
4.2. Episerver C♯ classes. ........................................... 20
4.3. Creating a CRC32 checksum builder. ......................... 22

C.1. Example program using scheduler to write messages. .... 47
1. Introduction

1.1. Goal

The goal of this project is to create an integration application for an e-commerce system. The system consists of two servers, one Product Information Management (PIM) server that uses the PIM system inRiver made by and named after the company inRiver, and an e-commerce platform called Episerver made by the company Episerver.

The purpose of this application is to create a graphically displayable result of a comparison of the product information stored on the inRiver server and the product information stored on Episerver.

In order to easily compare the product information, checksums are created for the information. Using checksums also provides an efficient way of storing the comparison result.

1.2. Customer & Motivation

Ninetech is an Information and Communication Technologies (ICT) firm that works with a wide spectrum of technologies. They are the customer who specified the product that was developed in this project. The main motivation behind the project is to provide a selling point for customers of Ninetech’s inRiver and Episerver solutions. Ninetech does not currently have any such system between inRiver and Episerver that can indicate to a user that all information was transferred successfully.

1.3. Project results

The project was successful as it provided all the functionality requested in the original project specification. Additional features were also implemented in the project.
1. Introduction

1.4. Dissertation layout

Chapter 2 (Background)  The background chapter will serve to describe tools used in the project. The chapter will also provide explanations of the concepts required to understand the design and implementation, as well as a description of the core problem this project aims to solve.

Chapter 3 (Design)  The design chapter describes the planned solution to the problem. The chapter explains how the implementation is expected to be built and why these decisions were made.

Chapter 4 (Implementation)  The implementation chapter describes how the solution was actually implemented in practice. This chapter will provide the details of how each component of the design was realised in the final application.

Chapter 5 (Evaluation)  The evaluation chapter evaluates the implementation of the project, comparing it to the original specification to see how well it matches the requirements.

Chapter 6 (Conclusion)  The conclusion chapter will discuss the project itself and summarise the experience of working on it. Tools used in the project that were not directly used in the implementation will also be discussed here.
2. Background

2.1. Introduction

The basic system consists of a collection of product information on one side contained in a database (the provider) and other systems using this information (the consumers), by way of having the provider push the information to the consumers. One such consumer which will be handled in this project takes care of displaying product information to customers via an e-commerce system for selling products.

The transferred information from provider to consumer is assumed to be correct, no verification is done by an intermediate system (for example by creating and comparing checksums, see section 2.6, to verify that the information is not corrupt when it arrives at the destination system).

The customer wants a solution that does verify that information has been transferred in its entirety, without corruption. Ninetech suggested using checksums for verification of information integrity. The solution should also be responsive, meaning any long-running tasks should be performed in the background (using a job scheduler).

2.2. Chapter layout

Section 2.3 Description of PIM systems (providers) and what purpose they serve, section 2.4 further explains the inRiver implementation of PIM.

Section 2.5 Description of Episerver (a consumer) and its functionalities.

Section 2.6 Basics and purpose of checksum values.

Section 2.7 Describes what a job scheduler is.

Section 2.8 Description of tools and languages.

Section 2.9 Discusses the core problem.

Section 2.10 Summarises the chapter.
2. Background

2.3. Product Information Management (PIM)

PIM describes a system to store information about products in a central database for use in other applications. Scott Cairney and Dominic Citino describe it as “PIM, as a concept, seeks to enable true item management outside the dependent application environment.” [1], where “dependent application environment” refers to an application in which the data is actually used and displayed to a customer. Item management in a PIM system refers to updating the product data with regular Create, Read, Update, and Delete (CRUD) operations.

Different applications use product data in different ways and present it in application dependent formats. PIM is a way to centralise the data to unify it and provide a streamlined process to modify existing product data, create new data, and delete existing data. Updates will then seamlessly propagate to all dependent systems. Item management in a PIM system refers to updating the product data with regular CRUD operations.

![Diagram of PIM system](image)

**Figure 2.1.** Illustration of a PIM system.

Figure 2.1 is an example of a PIM system with several inputs and outputs going through a central server that implements the PIM system. The inputs at the top are the various data sources providing data into the PIM system which then converts it into a common format. The data is later *published* to the output systems (in the bottom of the figure), with the exported data having a format suitable for the target system.
The *sales optimisation* node might be a system for employees in the sales department to review the product data and take decisions on how to improve sales in the future.

Take a mobile phone as an example of a product. When creating an e-catalogue, it might be fitting to have a long description with many details about the product, such as battery type and life, screen size, panel type, operating system, display resolution, display pixel density, and more. However, when presenting this data in a printed catalogue there is usually less space available. In this case some details are left out, for example less vital details such as the display pixel density, as well as using a shorter description.

### 2.4. inRiver as a PIM provider

inRiver is a leading provider of PIM services for businesses. They focus on providing Business-to-Business (B2B) and Business-to-Consumer (B2C) services to provide unified product information across different sales channels [4].

They aim to provide a platform to create, handle, and distribute product information in multiple languages, their service is used worldwide.

inRiver (the company) has named their product after themselves. inRiver (the product) can import data from external systems using so-called “inbound connectors”, and export data with “outbound connectors”. Connectors are developed in C# as class libraries. The resulting Dynamic Link Libraries (DLLs) are placed in directories where inRiver will be able to detect them.

A number of client applications are included with inRiver to provide customers or administrators with the ability to review and update the contents of the database, as well as a website portal providing similar functionality. The web portal is also the place where connectors can be enabled or disabled for the inRiver installation.

In inRiver, product information is stored as named fields, where the field names are defined by the user. For example, a field for storing the name of a product may be called “Title”, but it may also be called “ProductName” in another implementation. It is the customer who decides the name of fields, and which fields should be included in which categories.

### 2.5. Episerver as an e-commerce and Content Management System (CMS) provider

Episerver is a company providing CMS, commerce, and marketing for .NET applications. It focuses on eCommerce and providing solutions for businesses to establish an online presence with websites and shopping [2].
2. Background

Analytics tools are provided by Episerver, allowing businesses to investigate how well the solution is working.

A Representational State Transfer (REST) Application Programming Interface (API) is provided by Episerver that can be used to extract its list of products for use in other applications, or perform other common operations dealing with the CMS content, such as CRUD operations on the product information.

Episerver also stores the product information as fields, however the user may choose to name these fields differently than those of inRiver. As an example of this, consider the user configuring the name of a product to be “ProductName” on the inRiver side, while using the name “Title” on the Episerver side.

2.6. Checksums

A checksum is a value produced from hashing data (see figure 2.2). The value is compared to another checksum that is generated from a similar set of data as the first one. The reason for doing this is to verify integrity between two versions of the same type of data (such as two versions of product information that are expected to represent the same actual product), usually when transferring data from one place to another.

![Checksum Example](image)

Figure 2.2.: Generating a checksum from product information.

In a general case, the data provider (provider in this case does not refer to Episerver or inRiver in the project implementation) will generate a hash from its data, called a “checksum”. The recipient of the data will, after downloading the data, compute a new checksum from their instance of the data and compare it to the provider’s checksum. If the two checksums match, the data is considered to have been transferred successfully without corruption.

Checksums can also be used on two or more already obtained sets of data to verify that they are indeed the same data, in which case the recipient already in possession of all data and does not need to obtain it from a provider.
2. Background

2.7. Job scheduler

A job scheduler is a component that runs in the background and executes configured jobs at regular intervals. An example of such a job could be fetching data every hour, doing some work with it, and then saving the results in a database.

The job itself is like any other piece of code in a program, with the difference that it is handled and executed by the scheduler instead of manually by a user. Manual execution would for example be the user explicitly launching an executable file or visiting a web page to trigger code execution.

2.8. Tools and languages

2.8.1. Visual Studio

Visual Studio (VS) is an Integrated Development Environment (IDE) created by Microsoft. It includes support for a variety of languages and frameworks, many of which are also created by Microsoft. The language support includes C#, Visual Basic (VB), HyperText Markup Language (HTML), JavaScript (JS), and several others through updates and plugins. The main framework supported by VS is the .NET framework, but support for other technologies is available either through native functionality or via third-party plugins. [19]

VS includes several tools to aid the development process like IntelliSense [18] for autocompletion when writing code, and IntelliTrace [15] for debugging.

2.8.2. C#

“C# [is a] modern object-oriented programming language with functional programming capabilities for building an application on the .NET platform.” [20]

It is a general-purpose language that has developed several new features over the years, like null-conditionals, the “nameof” operator, and expression bodies. [9]

2.8.3. .NET

.NET is a general purpose development platform developed by Microsoft. .NET comes with a variety of features, two of which being modern programming languages (such as C# for this project) and automatic memory management. [7]

The platform provides low-level access whilst being a high-level programming environment. The main languages targeting .NET are C#, F#, and VB.

.NET is built upon the idea of the Common Language Runtime (CLR), enabling interoperability between languages, meaning that it should be possible to send objects between languages without the risk of loss in semantics or capabilities.
2. Background

Figure 2.3.: Illustration of the .NET stack.

2.9. Problem

Given that different technologies are interacting with each other, product information from the Product Information Management (PIM) system (inRiver) risks not being transferred completely to the Content Management System (CMS) or commerce system (Episerver).

Ninetech has expressed their desire to create a solution, preferably with a Graphical User Interface (GUI), that can highlight any issues in the data exchange from inRiver PIM to the Episerver CMS and commerce system. The issues in this case being loss of data, through network problems or corruption, but also old data not being updated. Ninetech’s reason for wanting this comparison tool is to have an extra feature for their customers to improve their product and solutions.

The flowchart in figure 2.4 describes the process of how the finished project will work.

1. A user visits the website with their web browser of choice.
2. On the website, the user triggers an event to make the server collect all product information from both Episerver and inRiver PIM.
3. Checksums are calculated for the information and then compared to each other.
2. Background

4. The result of the comparison is stored in a database.

5. Transfer the result to display methods.

6. The result is displayed graphically to the user in the browser.

If there are already one or more comparison results stored in the database, the most recent will be displayed directly when visiting the site (at step two in figure 2.4). A new comparison can however be initiated on the website.

2.10. Summary

This chapter has explained PIM (with inRiver’s implementation), Episerver, and how they work in a customer’s environment to consolidate and display product information.

Checksums have been explained and how they can be used to simplify comparison of complex data structures.

Relevant tools for developing the project have been discussed to give the reader an understanding of their functions and how they aid in project development. Certain platforms are discussed to explain their purpose and why they are involved in the development process.

Finally, the problem related to data loss when synchronising product information was discussed, and Ninetech’s reasoning for wanting a solution to this problem.
3. Project design

3.1. Introduction

Due to the many components involved in the project, a modular design would be preferable, so that components can be substituted or individually upgraded to suit future needs. Each module in the project will deal with one aspect or functionality in order to follow the single responsibility principle as part of the core principles in object oriented programming (single responsibility, open-closed, Liskov substitution, interface segregation and dependency inversion (SOLID)) [8].

The single responsibility principle states that a class should only be responsible for one aspect of an application, and no other class should deal with that aspect.

The dashed arrows represent components that depend on others (the arrow points at the dependency). Solid arrows represent a component making use of another component (the arrow points at the component that is being used). Components inside the yellow area (“Project Space”) are the ones being developed in this project, components outside the area (elliptical) are external and maintained by third parties.

Figure 3.1 shows an overview of what the overall project is expected to look like.

Figure 3.1.: Main project components and their relations with each other.
3. Project design

3.2. Chapter outline

Section 3.3 Description of the inRiver API and how a bridge project will be designed to interface with it.

Section 3.4 Description of the Episerver REST API and how a bridge project will be designed to interface with it.

Section 3.7 Description of the website and its planned functionality.

Section 3.5 Description of the planned design for how checksums will be used to generate a simple comparable value for product information.

Section 3.6 Description of EPIM Core and the data types defined in it.

Section 3.8 Describes the scheduler planned to be used for job scheduling (Quartz [6]) and how it will be incorporated in the design.

Section 3.9 Chapter summary.

3.3. inRiver

inRiver supplies a .NET API library called “inRiver Remoting API” for applications to connect to a running inRiver server and perform operations (collecting, creating, and updating information). The library allows fetching data from inRiver and storing it directly in a defined data class provided by the library.

Using the library, data will be fetched from inRiver and stored in the classes provided. A subset of the available data will be extracted and put in the core data classes defined in the project itself, which will then be used for comparison.

3.3.1. inRiver bridge

A bridge library will be created to obtain data from inRiver and store it in a format usable by all other components in the project. The bridge project will make use of the inRiver Remoting API to fetch product information and store it in the data classes provided by the core project. The core project can then construct checksums from the information to be used in comparisons later on.

Details required to connect to inRiver (server hostname, server port, username, password) will need to be supplied in some way, either from a configuration file or as direct input from a user.
3. Project design

3.4. Episerver

To access product information from the Episerver instance, the Episerver REST Application Programming Interface (API) will be used. The received data can be formatted in one of two ways, Extensible Markup Language (XML) or JSON. For this project JSON will be used since it is a more compact format.

```json
{
  "actors": [
    {
      "firstName": "Arnold",
      "lastName": "Schwarzenegger"
    },
    {
      "firstName": "Steven",
      "lastName": "Seagal"
    },
    {
      "firstName": "Sylvester",
      "lastName": "Stallone"
    }
  ]
}
```

Listing 3.1.: Example of JSON object.

```xml
<actors>
  <actor>
    <firstName>Aarnold</firstName>
    <lastName>Schwarzenegger</lastName>
  </actor>
  <actor>
    <firstName>Steven</firstName>
    <lastName>Seagal</lastName>
  </actor>
  <actor>
    <firstName>Sylvester</firstName>
    <lastName>Stallone</lastName>
  </actor>
</actors>
```

Listing 3.2.: Example of XML object.

Listings 3.1 and 3.2 show examples of JSON and XML code describing the same object. It is clear that JSON is both more readable for the human eye and takes up less space than XML. This is an advantage for this project since a large number of products, sometimes thousands, will be fetched and the more compact JSON format will make transfer times shorter, making the project more efficient.

3.4.1. Episerver bridge

In order to get the product information from the Episerver instance to the core project, a bridge library will be created. This bridge contains the code for fetching the data from a specified Episerver instance via Episervers REST API and converts it to a data format that is common for all project components. This data will then be used by the core project to create checksums as a base for comparing the data to the data from the inRiver server.

Data needs to be fetched from a server, but to specify an Episerver data source, some form of input will be needed. This input can be in the form of a configuration
3. Project design

file or user input.

3.5. Checksums

To compare different versions of product information it can become laborious to keep track of multiple kinds of properties and data types. It would be easier to have a single string that could be compared to easily decide if two versions of product information are identical or not.

By using checksums, the various properties and data types will be stored as a single hash value. This value will provide a way of more easily comparing different versions of information, in contrast to comparing each property verbatim.

In order to provide an accurate comparison, the checksum will need to be composed of all properties that are shared between product information on inRiver and Episerver servers. This requires additional operations, such as configuring which properties to include and if these properties have the same name in both systems.

As an example of shared properties, consider the fictional property “Name”. On inRiver, the property is stored as “Name”, on the Episerver side however, the property is stored as “Title”, but represents the same information. Some kind of configuration file will need to be provided that will map inRiver properties to Episerver properties. This configuration then also specifies which properties will be involved in creating the checksum.

<table>
<thead>
<tr>
<th>inRiver</th>
<th>Episerver</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Code</td>
</tr>
<tr>
<td>Name</td>
<td>Title</td>
</tr>
<tr>
<td>Description</td>
<td>Description</td>
</tr>
<tr>
<td>Price</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.2.: A visual mapping of properties.

In figure 3.2, the arrows show how properties on inRiver are mapped to Episerver properties. Properties listed in grey only exist on one side and will not be included when the checksum is calculated (as that would lead to information never being identical even when all relevant properties have the same values). This kind of mapping will be provided by the customer through a configuration file.

A common method when computing checksums is the MD5 hash function, which also has built-in support in the .NET framework [24, 16]. CRC32 is another common method to use for checksums which is not based on a hash, but requires usage of third party libraries for use in .NET [23].
3. Project design

For an example of MD5 and CRC32 checksums and what they output, see tables B.1 and B.2 in appendix B.

The checksum generation functionality will be provided through interfaces to ease the switching of algorithms. This way, if a company wants to use a specific hashing algorithm, it will be easy to provide.

Which method to use for generating the checksums will be left as an implementation decision and discussed in chapter 4.

3.6. EPIM Core

EPIM Core ("Core" in figure 3.1) is the central part of the project where the key functionality for the system will lie. The EPIM Core will contain the checksum functionality, central data structures for storing the product information within the project, and data comparison functionality.

3.6.1. Client interface

Each bridge project will need a central client class that will be responsible for providing the product information. In order to have a common way to obtain this data, the clients need to follow a common specification defined in the EPIM Core library.

3.6.2. Data formats

The Episerver bridge and InRiver bridge pull the product information from their respective server and format the information received according to the data structures specified in EPIM Core. This way both the product information from Episerver and InRiver are stored in the same format and can therefore more easily be used and compared. Since the name of fields can differ between Episerver and InRiver (see figure 3.2), a solution to this problem is discussed in section 3.5.

Data structures for storing product information are defined in EPIM Core as classes. A single Node interface and abstract class will represent all possible node types, and can contain sub-nodes. The data is stored as a tree structure where Node is a node in the tree which contains the product information. See figure 3.3 for an example of the tree layout.

The Node class is, for all intents and purposes, abstract and must be extended in bridge projects to represent the actual data. A node is not limited to any specific information other than what is defined in the interface, and can be used to represent both categories and products.
3. Project design

![Diagram of a data tree structure](image)

Figure 3.3.: Example of an EPIM Core data tree.

### 3.7. Website and database

In order to present the results of the comparison of the product information to the customer, a website will be created. Through the website, the customer should also be able to trigger a new comparison event, meaning there should be a button for the user to press that will activate a comparison.

Since the project is implemented using VS and C\#\), the most natural framework to use for creating a website is Microsoft’s ASP.NET [10]. The website project can then be constructed as part of the whole solution and can access data directly or from other parts of the project.

After completing a comparison, either through a scheduled job or a user triggered event, the result of the comparison should be stored in a database. Data from this database should be accessible through the website which then displays the most current comparison result. Through the website, the user should also be able to access a list of previous comparison results.

The comparison results will be displayed on the website as a tree structure. Each node may be expanded to show more information and also to show and hide child nodes. Nodes that contain incorrect information will be shown in red whilst correct nodes are green. All parents to a node with incorrect information will also show as red so that the user can easily find its way down to the node where the information differs.

### 3.8. Job scheduler

To handle job scheduling in the background, Quartz.NET [6] will be used. Quartz.NET designed around a central scheduler to which jobs are added. The JobScheduler will then handle executing the jobs at the interval configured in the job itself.

An example of a simple job that prints text to the console can be seen in appendix C.
3. Project design

The scheduler will be used to handle fetching product information from both inRiver and Episerver, followed by comparing the information and storing the result of the comparison in a database.

The interval at which the job will run will be a setting in the configuration file for the website. By running the scheduler in the same context as the website, it also benefits from being able to more easily run operations on the database.

3.9. Summary

In this chapter the general design of the project has been explained. The project is divided into smaller parts to make it more modular and allowing parts to be exchanged.

The main parts and the dependencies between them may be seen in figure 3.1 which shows the general outline of the finished project. Methods for extracting data from Episerver and inRiver servers through their respective API and how the bridges are designed to interface with the APIs are discussed.

EPIM Core design and the data types for working with data from both Episerver and inRiver are explained.

The parts not shown in the diagram are checksums and the job scheduler (Quartz.NET) which are both submodules to EPIM Core.

Finally, to perform the comparisons and to see the results, a website will be used as a GUI for the customer. Comparisons will also be run at regular intervals and stored in the database where all previous results of comparisons will be stored.
4. Project implementation

4.1. Introduction

This chapter has been written in chronological order to best describe the development process of each part of the project.

Firstly, the data was to be accessed from the Epi and inRiver servers through the bridges. Secondly, the core module of the project was made to contain the data received from the bridges. Thirdly, checksums could now be created from the data and also stored in the core data classes. Fourthly, a prototype website project was created to show only the test results. Fifthly, a job scheduler to run the comparison at regular intervals was implemented. The website was then updated to show more results and be more visually appealing. Lastly, optimisation and refactoring and improvement of the project code was carried out.

4.2. Chapter outline

Section 4.3 Presents the implementation of the inRiver bridge project.

Section 4.4 Describes the Episerver implementation and its bridge project.

Section 4.5 Presents details of checksums and how they are built.

Section 4.6 Describes the core library and the implementation of its various classes.

Section 4.7 Describes the website part of the project.

Section 4.8 Discusses the scheduler and how it is used in the website project.

4.3. inRiver bridge

The version of inRiver used in the implementation was 6.3.0 SP2. inRiver data is accessed through PIM.Client (see figure 4.1), a class implementing the client interface defined in the core library described in section 4.6.1. The constructor
4. Project implementation

takes care of connecting to an inRiver instance and setting up the \texttt{RemoteManager} for further communication with inRiver.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{inRiver_client_implementation.png}
\caption{\textit{inRiver} client implementation.}
\end{figure}

\texttt{RemoteManager} is a static class provided by the \textit{inRiver} .NET API. It provides methods for creating \textit{inRiver} connections and obtaining product information.

The private \texttt{GetChannels} method on \texttt{Client} uses the \texttt{Channel} helper class to obtain available channels from \textit{inRiver} (see figure 4.2). The \texttt{Channel} class implements the \texttt{INode} interface (via the abstract \texttt{Node} class), meaning they can be exposed directly in the \texttt{Categories} property on \texttt{Client}.

4.3.1. Data classes

The bridge project defines several helper classes matching the overall structure of data on \textit{inRiver}. These classes all extend the abstract \texttt{Node} class from the core library and have a private constructor matching one or more constructors on the \texttt{Node} class. Their definitions are similar to the \texttt{Channel} class described earlier. Full definitions for all helper classes may be found in appendix E.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{channel_helper_class.png}
\caption{The \textit{channel} helper class.}
\end{figure}
4. Project implementation

The private constructor enables the bridge project to control instantiation via the static `Get` methods available on each helper class.

Each `Get` method takes an integer ID that specifies which entity of a matching type to load from inRiver. The channel helper class also has a method to obtain all available channels from inRiver.

In the inRiver API, all data is represented in an `Entity` class. Extension methods [11] are available in the bridge project to extract specialized data objects from entities and convert them to lists and strings.

4.4. Episerver

The version of Episerver used in the implementation was 10.4.0. Episerver data is accessed through `Epi.Client` (see figure 4.3), a class implementing the client interface defined in the core library described in section 4.6.1. The constructor takes care of connecting to Episerver and obtaining an API token to be used for further access to endpoints.

```
public class EpiserverClient
{
    private readonly HttpClient _client;

    public EpiserverClient(string address, string username, string password)
    {
        _client = new HttpClient()
        {
            BaseAddress = new Uri(address),
            Credentials = new NetworkCredential(username, password)
        };
    }

    public async Task<string> GetAsync(string path)
    {
        return await _client.GetAsync(path);
    }
}
```

All communication with the Episerver REST API is done via an instance of an `HttpClient` pointed at the address of the Episerver (configured in the configuration file).

The method `GetAsync(string)` on `Client` wraps the `GetAsync` method on the internal `HttpClient` `_client`. This is an internal method, meaning only callers inside the Episerver bridge project can access it. Data classes will use the method to retrieve product information from the Episerver instance running in a user’s system.

The framework-provided `HttpClient` contains methods for performing Hypertext Transfer Protocol (HTTP) requests to a server and obtaining the response [14].
4. Project implementation

This makes the implementation of the Episerver interface relatively simple, only needing to call endpoints via the `HttpClient` and parse the results.

### 4.4.1. Episerver bridge

The whole bridge project is implemented as a simple data fetcher that uses the API endpoints to fetch raw JSON data and parse it into data container classes defined in the bridge project (the definitions of which are provided in Episerver documentation).

```csharp
public class Entity
{
    public string Title { get; set; }

    public MetaData[] MetaData
    {
        get;
        set;
    }
}

public class MetaData
{
    public string Name { get; set; }

    public string Value { get; set; }
}
```

Listing 4.1.: Episerver data (JSON). Listing 4.2.: Episerver C# classes.

Listing 4.1 shows an example of the data as returned from Episerver. Listing 4.2 shows how a class could be implemented to contain the data in C#. Each set of braces ({}) in JSON denotes an object, and brackets ([]) denote arrays. The structure of the class on the C# side then has to match the JSON structure, by using C# classes for JSON objects and C# arrays for JSON arrays.

### 4.5. Checksums

CRC32 was chosen as the method to use for generating checksums. The main reasoning for choosing CRC32 over MD5 was that it is the more compact format of the two, thus making it faster to transmit and taking up less space in storage.
4. Project implementation

Since the product information may contain multiple properties, there must be a way to construct a checksum by building it up from a sequence of string values. Checksum builders are implemented to achieve this, as shown in figure 4.4.

![Checksum Class Diagram](image)

Figure 4.4.: Diagrams for checksum classes.

A checksum builder must implement the `IChecksumBuilder` interface, which declares a set of standard properties and methods that must be present on a checksum builder.

**ByteValue** The checksum value as a sequence of byte values.

**Value** The checksum value as a string representation of the byte sequence.

**Add** Takes a string value and adds it to the list of strings that are to be used in the checksum calculation.

**CreateInstance** A method that creates a new instance of the same type of checksum builder the method is defined in. This method can be used when it is necessary to obtain a new instance of a checksum builder with a matching checksum generation method.

In order to be modular, it was decided to implement checksum builders as generic classes, where the underlying algorithm can be switched out without changing any of the actual code.

The .NET framework has a base class for hash algorithms, called `HashAlgorithm`, which specific algorithms may derive from to provide a unified interface to calculate hash values from [13].
4. Project implementation

4.5.1. AlgorithmChecksumBuilder

Given that any algorithm that could be used will always inherit from the base HashAlgorithm class, a generic class can be constructed to handle any kind of implementation.

The AlgorithmChecksumBuilder takes, at instantiation, a type parameter [12] that tells the class which hash algorithm to instantiate and use for checksum building.

The private Build method takes care of constructing the checksum and storing it in the _byteValue field, exposed to callers via the ByteValue property. The _value field and corresponding property are automatically generated on access from the previously generated byte sequence.

The string values added via the Add method are stored in a list of strings (the _data field) to later be used in the Build method.

For the sake of compactness, the string representation of the byte sequence is generated using Base64 [5]. A simple hexadecimal representation of the bytes was considered at first, but would need two characters per byte (in hexadecimal representation, one character is 4 bits of information). Base64 is able to represent two and a half bytes in two characters (one character of Base64 represents six bits of information). In other words, Base64 is able to store 150% as much information in the same amount of space as a hexadecimal representation.

A comparison between hexadecimal string encoding and Base64 may be seen in appendix D.

4.5.2. Creating the CRC32 builder

CRC32 does not have built-in support in the .NET framework, but there exists a NuGet package that adds a CRC32 class [3], which also implements HashAlgorithm.

Given this, to get a checksum builder that uses CRC32 to generate values, AlgorithmChecksumBuilder needs only be passed the Crc32 class as type parameter. An example of creating a new AlgorithmChecksumBuilder instance is shown in listing 4.3.

```csharp
var builder = new AlgorithmChecksumBuilder<Crc32>();
```

Listing 4.3.: Creating a CRC32 checksum builder.
4. Project implementation

4.6. EPIM Core

The EPIM Core project consists of a set of classes and structures (the structures are defined as C\(^\#\) structs) that define custom data types and functions. To be able to work with data from Episerver and inRiver which store data differently, a centralised format for storing that data is required. In EPIM Core the interface INode and accompanying abstract class Node are defined to satisfy these requirements.

4.6.1. Client interface

Figure 4.5 shows the definition for the client interface.

![Client interface in core library.](image)

The client interface is minimal and only contains a declaration for a property to obtain an IEnumerable for all categories available on the server. The inRiver and Episerver bridges provide concrete classes implementing this interface as described previously in sections 4.3 and 4.4.

4.6.2. Data formats

Figure 4.6 shows the interface that defines the data format. This interface has a name, ID, checksum, and a set of properties (keys with associated values). It also declares a collection of sub-nodes (in the Nodes property), meaning any node can have any number of sub-nodes attached to it.

The InternalType property is used to support comparing different properties depending on the source data type from inRiver or Episerver. An IComparer instance may use this information to only compare relevant properties.

The methods provided by the abstract Node class may be seen in figure 4.7.

Equals(INode other) takes another INode as input and compares the previously created checksums of both nodes. If the checksums do not match, Equals will return false.

Each bridge project (for inRiver and Episerver) has one or more classes extending Node to provide methods for obtaining implementation-specific data and convert it to the common format supported by Node.
4. Project implementation

![EPIM Core node interface](image1)

Figure 4.6.: EPIM Core node interface.

![EPIM Core abstract node class](image2)

Figure 4.7.: EPIM Core abstract node class.
4. Project implementation

4.6.3. Comparison classes

One of the customer’s wishes was the ability to show the results of comparing products, including which properties actually differ between two instances of product information.

In order to facilitate this, and keep everything modular, classes were designed that would accept products as parameters and perform a comparison on a property-by-property basis. The results are then stored in a list, with ways of obtaining the sub-results (comparison of each child node).

(a) Interface for comparer.
(b) Interface for comparison result.

Figure 4.8.: Interfaces for information comparison.

Figure 4.8 shows the interfaces which implementations of comparison functions must follow. An instance of IComparer is responsible for taking two INode instances and comparing them in a way specified by implementing classes (RecursiveComparer and RecursiveComparison) and returning an instance of IComparison detailing the result of the comparison.

(a) A recursive comparer implementation.
(b) A recursive comparison result.

Figure 4.9.: Comparison implementation classes.

A recursive comparison implementation is provided in the core project. Figure 4.9 describes these implementation classes. The main comparer class must be
4. Project implementation

constructed by supplying a `Config` instance, specifying the property configuration. With this config, the comparer is able to determine how to compare properties on inRiver and Episerver.

The `Compare` method on `RecursiveComparer` will, in accordance with the `IComparer` interface, take two nodes and return a comparison result. `RecursiveComparer` will return an instance of `RecursiveComparison` as its result class. The comparison result class will also do most of the actual comparison work.

`RecursiveComparison` is constructed by supplying two nodes to compare along with a configuration object. The `RecursiveComparison` class will then check for basic equality with the `Equals` method on the nodes, and compare each property by iterating over each property mapping available in the configuration and calling the private `PropertiesMatch` method.

Any properties that do not match have their mapping stored in a list, later exposed via the `DifferingProperties` property.

After the nodes have been compared, `RecursiveComparison` will recursively go through the children of the nodes and create new instances of `RecursiveComparison` for each pair of children. The sub-results are stored in a list exposed by the `Children` property.

Basic boolean properties `Equal` and `ChildrenEqual` are provided to quickly assert whether the compared nodes and their children are equal.

Using this information, the website project can build a tree view of the comparison and let a user browse it to get an overview of the system.

4.7. Website

After completing the checksum builders, a graphical way to display the results was required. Since Episerver and inRiver both have web based client tools, a web page for EPIM also seemed reasonable.

The service for creating web pages was chosen to be Microsoft’s ASP.NET framework in order to easily communicate with other parts of the project as well as to keep all parts of the project inside of VS.

At first a very basic website was set up displaying only the checksums for the Episerver and inRiver data and whether the comparison found any mismatching data. The website uses Microsoft’s implementation of Model View Controller (MVC) which automatically generates some controllers and a layout. Updates to the site made the comparison results more structured and visually appealing.

The website was updated to use the `IComparison` data class from `EPIM.Core` to display all the desired information about the result. This includes checksums for each node, the children count for each parent node, ID’s for each node and more. An example of an output with some errors may be seen in figure 4.10.
A configuration file was created for the website project where settings for which fields in the Episerver and inRiver product data were to be included in the checksum and comparison. The contents of this file are then sent to IComparison in the core project when a comparison is made.

After a comparison is complete, the results are displayed as a tree structure where, with the help of Cascading Style Sheet (CSS) and JS, the user can click each node to show more information about the node. If the node contains sub-nodes (also called children or child nodes), there is an arrow to the left that can be clicked to show the child nodes.

To build the tree structure in the front-end view, a recursive pattern had to be used. This was achieved with the help of view partials which are part of the Razor templating engine. Razor in turn is part of ASP.NET and is the default templating engine in ASP.NET MVC. To collapse information, HTML elements need a unique ID that can be used to specify which element is to be expanded or collapsed. A function for generating Globally Unique Identifiers (GUIDs) was created to be used in the partials to set the ID to the element.

At the top of the page, there are buttons that can toggle showing only erroneous nodes or all nodes, and to expand and collapse all parent nodes (nodes with children). Parent nodes of nodes that contain errors (non-matching product in-
4. Project implementation

formation), are also flagged as erroneous, making them display in the colour red on the page. Nodes with valid product information are displayed in green.

If there are any errors in the product information, the top node and all sub-nodes leading to the node with non-matching product information will be shown in red. The user can then easily find their way down the tree to the specific erroneous node.

The button to expand or collapse all nodes can be pressed to display or not display sub-nodes.

A button was also added to the page that, when pressed, will start a new comparison job in the background. The user can then visit the page again when the job completes to see a newly generated comparison result. Pressing the button directs the user to a separate page where the controller will use the JobScheduler to trigger the job via Quartz.NET.

4.7.1. Historical view

Since each result from running the comparison job is saved to the database, the website can display a log of previous comparison results (see figure 4.11). A separate page was created to display this log of comparison entries.

![List of results](image)

Figure 4.11.: Example of the historical view showing past results.
4. Project implementation

Each entry has links to view details, delete the entry, and to visit the main page using the requested data to build the comparison result view.

4.8. Job scheduler

As noted in the design chapter (chapter 3), the scheduling code is placed in the website project to facilitate easier connections with the underlying database. A helper class is also created to encapsulate the actual job scheduler from Quartz.NET. This allows for Quartz.NET to be more easily replaced by a different framework in the future.

When the website starts up, the job scheduler is initialised, and the comparison job is created and added to the scheduler. The time interval at which to run the job is obtained from the website configuration file (`Web.config`).

4.8.1. Comparison job

The comparison job will, in its `IJob.Execute(IJobExecutionContext)` method, collect product information and compare it using the methods from the Core library.

Care must be taken to guard against possible failures by using `try-catch` statements around the job code. Any errors that occur during the job will be logged to the configured logging output (by default the trace logger [17] in VS).

The comparison job is largely decoupled from the website code, and loads its configuration from files on disk (the JSON config file for matching inRiver and Episerver property names when comparing data). Connection details to inRiver and Episerver are not loaded in the job, but rather provided by the website at job initialisation. The only point at which the job uses website code is when inserting the comparison results in the database.

The results are stored in the database following the schema described in table 4.1.

<table>
<thead>
<tr>
<th>Id</th>
<th>Entry ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Whether comparison was successful.</td>
</tr>
<tr>
<td>PimData</td>
<td>Serialised PIM product information.</td>
</tr>
<tr>
<td>EpiData</td>
<td>Serialised Episerver product information</td>
</tr>
</tbody>
</table>

Table 4.1.: Database table schema for comparison results.

Rather than storing the serialised `IComparison` object, the raw product information from each platform is serialised to JSON strings. The comparison page can then load the data and perform a comparison on the cached data. Initially it
4. Project implementation

was planned to serialise the actual comparison result, but since this contains a reference to a node in each tree node it would generate a considerable amount of duplicate data. Storing the product information instead ensures that each node is only included as many times as it appears in the original source data.

4.9. Summary

The Episerver bridge fetches data through a REST client via HTTP requests. inRiver bridge uses inRiver’s provided .NET API to fetch data from the inRiver server. Both the Episerver and inRiver bridge contain data classes that represent data types that exist on the respective server. On the inRiver bridge, these data classes extends the EPIM Core Node class, and can then be stored as nodes in the data tree.

In the Episerver bridge, when fetching data through the REST client, the data is sent to the Node constructor to adjust to the data format to that of EPIM.

When the data is being gathered and converted into nodes through constructors, checksums are calculated with the help of checksum builders for the data and saved to the checksum property of the node.

When starting up the project, a job scheduler is instantiated to run a comparison at an interval configured by the user and stores the result in a database.
5. Evaluation

5.1. Introduction

This chapter evaluates the results, and discuss how well the implementation matched the project expectations.

The first prototype of the solution (that was demonstrated to the customer) fulfilled the requirements listed in the specification (see appendix A).

Further work was done to improve the design and performance of the solution.

5.2. Chapter outline

Section 5.3 Evaluates the inRiver bridge project.

Section 5.4 Evaluates the Episerver bridge project.

Section 5.5 Evaluates the checksum implementation.

Section 5.6 Evaluates the EPIM core project.

Section 5.7 Evaluates how well the website implementation matches what was asked for in the specification.

5.3. inRiver bridge

With the native .NET API available for inRiver, fetching products was as straightforward as the specification implied. Several calls were still required to retrieve all products, but compared to the Episerver API it performed quite well, only taking a few seconds to retrieve all products from the test system.

5.4. Episerver bridge

The implemented Episerver bridge project works well considering the limitations. It was expected that it would be possible to export the entire list of products from Episerver with a single “export command”, but instead multiple API calls were
5. Evaluation

required. Each call to retrieve a category contains links to further sub-items that need one API call each to obtain. This continues recursively until all products are obtained.

Because of the inability to retrieve all data in one go, the Episerver bridge turned out to perform less efficiently than expected. In local tests with about 50 products, it was still reasonably fast (between 10 and 20 seconds to fetch products). However, in a bigger production system with thousands of products the Episerver API for fetching product information could pose a problem.

Episerver does have support for a full export where all product information is put into an XML file on the server that runs Episerver. However, it does not have built-in support for sending this file over the network. If an Episerver instance has a large product list, it might take an unacceptable amount of time to transfer the product information through the REST API.

One solution for the file transfer issue would be to implement a directory listener on the server that runs Episerver. When activated by a change in the directory where a new XML file is created, this listener would connect to the EPIM server and send the file. In the EPIM project there would have to be classes to handle this connection and converting the contents of the XML file to the EPIM format. Due to lack of time, this functionality was not implemented.

5.5. Checksums

The checksum implementation was expected to consist of a class that takes a list of data objects and create checksums for each of these objects, then create a single checksum value from this list of checksums. In accordance with the expectations, the class contains an add method to add data, along with a generation method that would generate the checksum value from the added data when called. The end result did have an interface (IChecksumBuilder, discussed in section 4.5) following these expectations, but required more code to support it than expected.

Due to how the HashAlgorithm [13] is designed, a different method needed to be called when adding the final piece of data before calculating the hash value to use for the checksum. The checksum abstraction classes do this by not calculating the checksum value until requested by a caller, making it possible to do the separate method call when the final data item is detected in the checksum generation process.

CRC32 turned out to be a better algorithm to use in the checksum generation method compared to MD5, not only for storage space reasons but also for debugging reasons. It is easier to visually compare two eight-character strings than to compare two 32-character strings. The detailed view on the website also gets easier for the user to digest if shorter checksums are used.
5. Evaluation

5.6. EPIM Core

Data classes in Core changed throughout the timespan of the project. At first, data classes were created for categories and products, which in turn would also have led to data classes for bundles and articles. But due to the lack of difference in content between the classes, it was decided that it would be easier to simply have a class Node for all types. With products now being represented by the Node class, implementing article support was simpler, as articles are child nodes of a product. Code duplication between Category and Product was no longer necessary.

The configuration handling in Core was chosen to be a class of properties serialized to file. The deserialized configuration object can then be passed to components that need it, for example the Node constructor. The configuration object would then be passed from the website project to Core when a comparison is made.

Another way to do this would be to store the configuration as a global static class, however this would not be as dynamic or modular. The implemented solution also has the advantage of being more easily tested, as mock-up configuration objects can be used in place of the real configuration implementation.

5.7. Website

In accordance with the specification, the website displays the results to the user as a tree structure. The tree is generated with the help of recursive ASP.NET MVC partial views. JS is used to let the user collapse or expand each node in the tree, as well as show or hide more detailed information about each node.

As the specification states, each node provides visual indication about the validity of itself and its children. Nodes that have mismatched information (or their children do) are shown in red, as per the specification requirements.

The visual part of the website project turned out better than expected. Thanks to the use of Bootstrap [21], many frontend problems were easily solved. These problems included collapsible information which was solved using Bootstrap’s collapse.js plugin [22]. Another problem was that in order to be able to collapse elements, each element must have a unique ID that distinguishes it from other elements. This was solved with the use of ASP.NET partial views to which an ID was sent with the help of the Razor templating engine. A function for generating GUID was made to create these ID’s.

Since the main parts of the project are backend related, very little time was spent working on the frontend part until the end of the project. It was not until towards the end that relevant results were available for display. At first, only information about whether the comparison completed successfully (true or false) was displayed along with checksums of the categories, made from only the name
5. Evaluation

of the products and categories.

5.8. Summary

In the software construction phase of this project, few problems arose. The most notable of these was the setup of the Episerver and inRiver connector, which consumed far more time and energy than all the other problems combined. Had this not been the case, more functionality than currently implemented might have been available, such as exporting XML from Episerver.

One possible solution for the efficiency problems of retrieving products from Episerver was considered, but not implemented (see section 5.4).

In some parts of the project, several solutions were considered, for example the configuration handling in Core. Checksum algorithms were also chosen with care for efficiency and readability. The EPIM Core Node was rebuilt in order to better support a variety of data types.

The website project was updated at several stages to improve ease of use and the aesthetics as well as functionality. When compared, the results of the website project matches the expectations expressed in the specification from Ninetech very well.
6. Conclusion

6.1. Introduction

EPIM was a success in the sense that all of the original specifications requirements were met. Moreover, additional features were also implemented. The major problem that arose during the development of the project was the setup of the test environment. In essence, setting up a working instance of inRiver and Episerver that could work together through the connector.

6.2. Chapter overview

Section 6.3 Discusses the setup experience of inRiver’s PIM solution.

Section 6.4 Discusses Episerver’s setup procedure and associated problems.

Section 6.5 Discusses future work to further improve the project.

6.3. inRiver

The inRiver documentation is better structured than that of Episerver, making the setup easier. inRiver supplied sample databases to use for setting up a test environment, something which Episerver did not.

The only problem encountered was during initial setup of the test databases. Some formats did not quite match, and there were issues with permissions. The main issue being that the server runs as a specific user (in Windows), and the database may or may not run under the same user, leading to issues where they cannot read or write each other’s files. Care was taken to ensure the user running the inRiver server also was able to access the database properly.

6.4. Episerver

Setting up a development instance of Episerver was perhaps the most time consuming part of the project. Getting an Episerver instance set up and running
6. Conclusion

properly, and communicating with inRiver through the connector required almost two weeks (≈40 hours).

The documentation for setting up an Episerver instance was fairly straightforward, even though their documentation site was poorly structured.

When installing Episerver, the latest version (version 10.4.0) was used. This caused problems when using the inRiver connector since it only officially supports an older version of Episerver. An older version of Episerver (version 9.12.3) was installed to see if the problems could be fixed, but the same issues persisted. One such issue occurs when updating a published channel on inRiver, followed by an attempt to publish the updated information to Episerver. inRiver reported success in publishing the information, but when inspecting the results on Episerver no change had been made. After inspecting Episerver logs and debugging the connector code, it was discovered that it tried to create new entries instead of updating old ones. Since all entries are identified by primary keys, this behaviour led to the update failing.

Another issue probably caused by the connector was that some fields were not included for some products. For example products with the type Product_FashionProduct may contain a gender, but for some of these products the field is not copied from inRiver to Episerver, while on the remainder of these products it is included.

It was later realized that the older version of Episerver that was installed to fix the issues was still too new for the connector (the connector only supported version 9.0.0.1125 of Episerver).

Since these problems were caused by the software that was required for the project, there was nothing to be done except perhaps contact inRiver and Episerver, or look in the source code for these and find the problem.

Once Episerver had been set up, it remained stable, although rather slow to query. Some API requests could take several seconds to complete, even if only querying for a single product.

6.5. Future work

Due to time constraints, a number of features requested by the customer were not implemented. These features were requested by the customer during project development and were not part of the original specification.

Other than the customer, some features were suggested by our dissertation supervisor, friends, or by ourselves.
6. Conclusion

6.5.1. Icons on tree nodes

It was suggested to use icons to make it easier to differentiate between different nodes (categories, products, articles, and more depending on the customer-specific setup) for users. Different icons would be placed next to the name of a node in the tree view (see figure 4.10) to make it easier to discern what internal type a certain node represents in the underlying systems.

The main issue encountered when trying to implement the feature was how to reliably determine which icons to use on which nodes. All nodes use the same internal class in the EPIM project, but they do expose their internal inRiver and Episerver names. The internal names could be used in conjunction with CSS classes to give them different styles and icons.

There are many different internal names on both inRiver and Episerver. Some node types (for example product) can have multiple internal names (Product, Product_FashionProduct). To write simple CSS mappings for all these names is possible, but could generate hard to maintain code. A more portable solution would be to use a customer-specific configuration that maps internal names to standardised CSS classes for the main node types.

Using a configuration approach, the core system does not need to change for each customer. Instead, the system is configured once during the setup of a new customer installation to map internal names to the relevant CSS classes.

6.5.2. Comparison statistics

A few employees at Ninetech that were shown an early demo version of the project suggested that more statistics be shown on the website. Information such as the percentage of products that failed to transfer, and graphs to show change over time.

These ideas were put on a suggestions list but never realised due to time constraints. Implementing these ideas could provide a useful analytical tool for users of the system, and give further insight into what could be causing synchronisation issues between inRiver and Episerver.

The basic infrastructure to support a statistics system exists in the finished project. The database contains basic historical data of the results and could be used to plot failures over time. It would not take more than a couple of weeks for a developer experienced with web-oriented graph libraries to implement these features.
6. Conclusion

6.5.3. Automatic triggering

Another feature suggested by Ninetech was to trigger the comparison automatically when data changes on inRiver or Episerver. This could be implemented by creating a custom connector module for inRiver.

6.5.4. Further development

Any further development would be done by a team at Ninetech. A time estimate to finish the features is two months, given the complexity of the inRiver connector.

6.6. Concluding remarks

The EPIM project ended up successful, meeting all the requirements stated in the specification from the customer. Designing the project went smoothly and no major obstacles were encountered. The setup phase however, required considerable effort. Connecting inRiver and Episerver with the connector posed some problems. Most major problems were encountered during this phase.

Implementing EPIM after all the setup was completed was carried out steadily with new features being implemented often. Some minor changes to the design were made at this stage in order to better please the customer and making the implementation easier to develop further.

There are still extra features to be implemented. Ideas for these new features were suggested during implementation by us or the customer, but it was not feasible to complete these suggestions within the time span of this project.

This project has provided experience of working at an ICT consultant company. The work environment and staff at the company have been welcoming. The work was challenging but educational.

From this project, experience was gained in ASP.NET, and e-commerce. These skills are useful when working at firms in Karlstad, since the .NET framework is dominant here from what we have gathered when speaking with companies.
References


References


Glossary

.NET is a software framework created by Microsoft to provide common libraries for applications and enable language interoperability. vii, 5, 7, 8, 11, 13, 18, 21, 22, 30, 31, 38
Acronyms

API Application Programming Interface. 6, 11, 12, 16, 18–20, 30–32, 36
B2B Business-to-Business. 5
B2C Business-to-Consumer. 5
CLR Common Language Runtime. 7
CMS Content Management System. 5, 6, 8
CRUD Create, Read, Update, and Delete. 4, 6
CSS Cascading Style Sheet. 27, 37
DLL Dynamic Link Library. 5
GUI Graphical User Interface. 8, 16
GUID Globally Unique Identifier. 27, 33
HTML HyperText Markup Language. 7, 27
HTTP Hypertext Transfer Protocol. 19, 30
ICT Information and Communication Technologies. 1, 38
IDE Integrated Development Environment. 7
JS JavaScript. 7, 27, 33
JSON JavaScript Object Notation. ix, 12, 20, 29
MVC Model View Controller. 26, 27
PIM Product Information Management. vii, 1, 3–5, 8, 29, 35
REST Representational State Transfer. 6, 11, 12, 19, 30, 32
SOLID single responsibility, open-closed, Liskov substitution, interface segregation and dependency inversion. 10
VB Visual Basic. 7
VS Visual Studio. 7, 15, 26, 29
XML Extensible Markup Language. 12, 32, 34
**A. Original project specification**

**Exjobb Ninetech VT2017**

**Bakgrund**

Ninetech utvecklar och förvaltar ett antal e-handelslösningar på uppdrag av kunder. Två av byggensterna i dessa lösningar är EPiServer och inRiver PIM. Båda dessa produkter är etablerade på marknaden och används i många lösningar. Ett problem som ibland upplevs i lösningarna är en osäkerhet kring om integrationen mellan dessa produkter fungerar till 100%. Är produktträdet i inRiver PIM exporterat i sin helhet till EPiServer? Trädstrukturerna är ofta stora och svåröverskådliga för en manuell granskning.

Integrationen är av typen one way push (se bild). Det finns ingen återkoppling som verifierar att allt kommit över som det skall. Exjobbet går ut på att skapa en parallell lösning som kan verifiera att allt fungerar och om det inte fungerar kan peka ut vad som inte fungerar.

**Figur 1:** Integrationer mot EPiServer. Ny funktionalitet med integration mot både EPI och PIM.
Uppgift

Utveckla en tjänst med en webbapplikation i företrädesvis MVC.NET. Systemet ska visualisera ett produktträd på en e-handel. Varje nod (produktkategori, produkt eller artikel) i trädet ska indikera om den noden och dess undernoder har synkroniserats från inRiver PIM till EPIserver Commerce på ett framgångsrikt sätt.

Funktioner

- Integration mot **inRiver Connect**.
- Integration mot **EPIserver Commerce**.
- Beräkning av en checksumma för samtliga katalog-noder och ingående produkter och artiklar.
- Matchning av checksummor för katalogträden i inRiver och EPIserver.
- Presentation av resultatet i en trädstruktur på ett överskådligt sätt där användaren enkelt kan borra sig ner i trädet för att hitta eventuella fel.

Insag

Största delen av arbetet innefattar programmering och webbutveckling i Microsoft-miljö. Integrationerna mot API:er i inRiver och EPIserver kommer att vara centralet. GUI och design för hög användbarhet kommer vara önskvärt, läsning och skrivning mot databas, samt planering och strategi för att göra applikationen lätt att vidareutveckla i ett senare skede ingår också. Javascriptramverk så som Twitter Bootstrap och AngularJS eller liknande kommer att behövas i lösningen.

Rimlighetsbedömning

Det är rimligt att anta att arbetet kan klaras av inom ramen för två studenter, 20 veckors halvtidsjobb. Vi ser gärna att studenterna drivna .NET utvecklare och inte är främmande för webutveckling. Naturligtvis är vi på Ninetech med och stöttar och agerar bollplank!

Kontakt

Magnus Odell
Tel: 0708-132879
magnus.odell@ninetech.com

Ninetech

Telefon vv: 010-470 64 00 Fax: 010-470 64 99
www.ninetech.se
## B. Checksum examples

### B.1. Checksums with MD5

Table B.1.: Checksum examples using the MD5 hash algorithm.

<table>
<thead>
<tr>
<th>Source data</th>
<th>MD5 checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello world</td>
<td>5eb63bbbe01eeed093cb22bb8f5acdc3</td>
</tr>
<tr>
<td>longer data with more words</td>
<td>6e5c3220586e61f84558bc3b9a29d431</td>
</tr>
</tbody>
</table>

### B.2. Checksums with CRC32

Table B.2.: Checksum examples using the CRC32 algorithm.

<table>
<thead>
<tr>
<th>Source data</th>
<th>CRC32 checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello world</td>
<td>7813f744</td>
</tr>
<tr>
<td>longer data with more words</td>
<td>2332a2ca</td>
</tr>
</tbody>
</table>

45
C. Scheduled job example

The code in listing C.1 will create a job that outputs a string to console (in the `Execute` method of `WriterJob`). The trigger created in `Main` configures the job to run every minute.
namespace JobExample
{
    public class WriterJob : IJob
    {
        public void Execute(IJobExecutionContext context)
        {
            Console.WriteLine("This is WriterJob");
        }
    }

    public static class Program
    {
        public static void Main()
        {
            IJobDetail job = JobBuilder.Create<WriterJob>()
                .WithIdentity("writer", "example")
                .Build();

            ITrigger trigger = TriggerBuilder.Create()
                .WithIdentity("trigger", "example")
                .StartNow()
                .WithSimpleSchedule(x => x.WithIntervalInSeconds(60)
                                    .RepeatForever())
                .Build();

            ISchedulerFactory factory = new StdSchedulerFactory();

            IScheduler scheduler = factory.GetScheduler();
            scheduler.Start();

            scheduler.ScheduleJob(job, trigger);
        }
    }
}

Listing C.1.: Example program using scheduler to write messages.
D. Hexadecimal and Base64 string encodings

Table D.1 shows a string encoded using two different techniques: regular hexadecimal representation and Base64. The string is encoded in ANSI (one byte per character).

Table D.1.: Comparison of string encoding techniques.

<table>
<thead>
<tr>
<th></th>
<th>The quick brown fox jumps over the lazy dog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td></td>
</tr>
<tr>
<td>HEX</td>
<td>54686520717569636b2062726f776e206666f78206a7</td>
</tr>
<tr>
<td></td>
<td>56d7073206f76657220746865206c617a7920646f67</td>
</tr>
<tr>
<td>Base64</td>
<td>VGr1IHF1aWNrIGJyb3duIGZveCBqmdWlucyBvdmVyIHRoZSBsYXp5IGRvZw==</td>
</tr>
</tbody>
</table>
E. inRiver data classes

(a) Node helper class for inRiver.

(b) Bundle helper class for inRiver.

(c) Product helper class for inRiver.

(d) Item helper class for inRiver.

Figure E.1.: inRiver data class helpers.