FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	ERP System Integration Tool
Student:	Bc. Ondřej Štauda
Supervisor:	Ing. Lukáš Charvát, Ph.D.
Study program:	Informatics
Branch / specialization:	Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

The aim of the thesis is to design and implement a modular tool for data integration between ERP systems. Based on a mapping, the tool should primarily be able to transfer data exported from a source ERP system to a target system. The tool should be designed in a modular way so that the source data can be, for instance, a local file, a file stored on an FTP server, or even an online document (e.g., Google Sheets). The integration with the target system will be based on the representation of a data model captured using a description language for HTTP-based APIs (such as OpenAPI, RAML, or Apiary Blueprint).

Goals of the thesis:

* Learn about HTTP-based web services and languages used to describe their APIs.
* Familiarize with existing ERP systems and web services they provide.
* Propose a method/format for ERP data mapping.
* Design a modular core of the tool that will allow easy addition of data sources and connectors to target ERP systems.
* Implement at least one data source and one target data connector for a selected ERP system.
* Test the tool with real data.

Electronically approved by Ing. Michal Valenta, Ph.D. on 22 December 2022 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

ERP System Integration Tool

Bc. Ondřej Štauda

Department of Software Engineering
Supervisor: Ing. Lukáš Charvát, Ph.D.

June 29, 2023

Acknowledgements

I would like to thank my supervisor, Ing. Lukáš Charvát, Ph.D., for his patience with me and the huge help I received when writing this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on June 29, 2023 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Štauda, Ondřej. *ERP System Integration Tool*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstrakt

Roustoucí společnosti mají vždy zájem o robustnější systémy oproti jejich stávajícím. Takový přechod mezi systémy je málokdy bezbolestný, obvzlášť pokud stávající systém používá rozdílný formát dat od toho nového, nebo nový systém neumožňuje importování dat z toho stávajícího. V takovém případě potřebují tyto společnosti nástroj, který jim umožní importovat jejich data do nového systému a ulehčit tak přechod k novému systému.

Cílem této práce je tvorba modulární aplikace, která umožňuje načíst data exportovaná z ERP systému, transformuje je za pomoci uživatelem vytvořeného mapperu a následně je exportuje do cílového ERP systému. Tato práce pokrývá celý proces od návrhu formátu vstupních dat, formátu mapování dat mezi systémy přes modularizaci nástroje až po následnou implementaci těchto návrhů se zaměřením na načítání dat z CSV formátu a exportování dat do cílového systému. Fungování aplikace je nakonec ověřeno jejím otestováním nad daty exportovanými z reálného ERP systému.

Klíčová slova ERP, integrace, REST

Abstract

Growing businesses are always in need of a more robust software than what they currently have. But such a transition to better systems can be painful, especially if the current system uses different data formats than the new one or the new system does not have the ability to import the data from the current one. In such cases, the companies need some sort of tool that can import their data to the new system to ease the transition to the new systems.

The aim of this work is to create a modular application that can load the data exported from an ERP system, transform it with the help of a user-created mapper, and export it to a target ERP system. It covers the whole process, from designing the input format specification, the data mapping format, the modularization, and the subsequent implementation of these designs, with primary focus on loading data from CSV format and exporting to a target system. Finally, we tested the capabilities of the application with data exported from an actual ERP system.

Keywords ERP, Integration, REST

Contents

List of Figures

List of Tables

Introduction

For small businesses with just a few employees, the list of utilized software is often limited to essential bookkeeping and basic customer relationship management (CRM) systems. However, as companies start to grow, they require more robust software solutions that can integrate their existing software and data into a single, efficient system. In such scenarios, an enterprise resource planning (ERP) system is often the most suitable solution. To effectively utilize all data in an ERP system, companies require a tool to export their existing data to the new system. Although manual data transfer is an option, it is time-consuming and prone to errors, which can cause additional delays in the process.

Moreover, we can also consider the perspective of a more stable and larger company instead of a small-scale business. The majority of such companies already utilize an ERP system to manage their customers, accounting, and invoices. However, there comes a point where the current ERP software may not suffice for the company's requirements. In such cases, companies are obliged to transfer all their data to a new system.

Most current ERP systems have some sort of function to export customers' data to CSV or XML files. But most of them do not even have an import function. If they have an import function, the imported file needs to be in a certain format to work. In such scenarios, a tool, that is the goal of this thesis to design and develop, comes in. The tool should take the data from a CSV file, provide options to map the source system properties to properties in the target system, and then import them to that system, for example, by using their REST API. The tool should be modular in order to allow future extensions, such as adding more data sources and/or target systems.

But what exactly is an ERP system? As mentioned above, the acronym ERP stands for Enterprise Resource Planning. It refers to a system that an organization uses to manage its day-to-day business activities, tying together multitude of business processes and integrating the data between them. Doing

so eliminates duplicate data and serves as a central hub for all organization information, providing data integrity with a single source of truth.

The ERP system usually consists of multiple modules, where each one is specialized in one aspect of business processes. Some of the most common modules focus on accounting, customers, inventory management, order processing, procurement, or human resources. With these modules, the ERP system can be customized to suit the business processes of many organizations.

Many ERP systems offer a wide degree of customization, allowing them to meet almost any organization's needs. Typical characteristics of ERP system include being an integrated system operating in (or near) real time with consistent look and feel across its modules.

ERP systems can be local-based or cloud-based. Cloud-based solutions has significantly rose in popularity, with information being readily available at any location thanks to internet access, which became more of a necessity than a privilege. Traditional on-premises ERP systems are now considered legacy technology.

Thesis Organization. This thesis is organized into chapters as follows. In the chapter 1, the so-called Interface Definition Languages (IDLs) are briefly introduced. The chapter 2 then discusses selected ERP systems. Next, the chapter 3 provides an overview of the contemporary tools that aim at integration between various ERP systems. Based on the initial analysis, the chapter 4 sets up major goals of this thesis. Furthermore, the chapter 5 focuses on designing the REST API using the OpenAPI specification, internal data/mapping formats, and the modularization of the tool. The chapter 6 describes its implementation in detail, with a focus on the implementation of a local CSV file as data source and NetSuite as a target application. The chapter 7 provide a basic user guide and chapter 8 shows how the tool can be used with real data. Finally, in the last chapter, we summarize the content of this thesis and discuss possible future improvements.

Chapter 1

Interface Definition Languages for HTTP-based APIs

Interface Definition Language (IDL) is a term for a language that enables a program written in one language to communicate with another program regardless of the language in which it was written. It describes the interface in a language-independent way, enabling communication between software or its components that do not share one language. The description of the interface written in IDL can also be used as documentation itself for the said interface, or it can be generated from the description. In this chapter, we take a short look at some of the most used and important IDLs that are used for HTTP-based APIs, specifically the ones that can be used for describing REST APIs. Some degree of understanding of these languages is important, because most ERP systems use them to specify the format of transferred data. Furthermore, one of the languages (described in the section 1.2) is also used for an input API description of the developed tool.

1.1 API Blueprint

API blueprint enables you to quickly design and prototype APIs that have yet to be created or document and test already deployed APIs [1]. It is based on Markdown syntax with a set of semantic assumptions laid on top. API Blueprint is built to encourage collaboration at all points in the API lifecycle between project stakeholders, developers, and customers. API Blueprint is all about design-first philosophy. Comparable to tests in test-driven development, the API Blueprint represents a contract for an API. Analyzing your API and settling on the contract before its development tends to produce better API designs. There is a plethora of tools built for API Blueprint, thanks to its broad adoption.

1.2 Swagger/OpenAPI

The OpenAPI specification, formerly known as the Swagger specification, is a specification for a machine-readable interface definition language used for the description of REST APIs [2]. It can be written in both JSON or YAML. It has an extensive suite of open source tools built around it, called Swagger. These tools can help design, build, document, and consume REST APIs. Swagger is a set of open source tools built around the OpenAPI Specification that can help you design, build, document, and consume REST APIs. The most notable are the Swagger editor and Swagger Codegen. Swagger editor is a browser-based editor that can be used to write the OpenAPI Specification. The OpenAPI Specification can be converted to both YAML or JSON and downloaded right from the editor. Swagger Codegen tool can generate server stubs or API clients from given OpenAPI Specification in various languages, such as Java, C#, Python or many others.

1.3 RAML

The RESTful API Modeling Language (RAML) is an YAML-based language for describing static APIs [3]. Although it was designed with RESTful APIs in mind, it is not capable of describing APIs that obey all constraints of REST, specifically APIs that obey the HATEOAS (Hypermedia as the Engine of Application State). The HATEOAS enables the server to send not only the data to the client but also the actions or operations that can be performed on those data. This way, the client can navigate the application state by following links provided in the response, without requiring prior knowledge of the application's behavior. It encourages reuse, enables discovery and pattern sharing, and aims at the merit-based emergence of best practices.

1.4 WSDL

Web Services Description Language (WSDL) is a standard specification used to describe networked, XML-based services, usually SOAP web services. WSDL defines an XML format to describe network services as a set of endpoints that operate on messages that contain document-oriented or procedure-oriented information. WSDL is extensible to allow endpoints and their messages to be described, regardless of the message format or the network protocol used for communication. This means that interfaces are abstractly defined using the XML schema and then tied to specific representations appropriate to the protocol. WSDL allows the service provider to specify the operations, parameters, and data types comprising the interface of the web service, as well as the protocol and encoding to be used when accessing public operations of the web service [4].

WSDL documents allow developers to expose their applications as network accessible on the Internet. Client programs connected to web services can read the WSDL file to determine the available operation on the server. The client can then use SOAP to call one of the operations listed in the file, for example, using XML over HTTP. Since WSDL 2.0 it offers better support for RESTful web services and is much simpler to implement.

1.5 WADL

Web Application Description Language (WADL) is a machine-readable XML description of HTTP-based web services [5]. It models the resources of the service and the relationship between them. It is platform and language independent, and it aims to promote reuse of applications beyond the basic use in the web browser. WADL was supposed to be the REST equivalent of SOAP's WSDL. It was submitted to the World Wide Web Consortium (W3C), but the consortium currently does not have any plans to standardize it. Each resource has param elements and method elements. Param elements describe that describes inputs and method elements describe the request and response of a resource.

Chapter 2

Overview of Selected ERP Systems

This chapter focuses on the analysis of four ERP systems: NetSuite, Oracle Fusion ERP, QuickBooks, and Xero. Each of these systems has a significant traction in the global market and offers comprehensive functionalities and unique features tailored to different organizational needs. Along with the basic description of these systems, special attention is given to their integration options.

2.1 NetSuite

The NetSuite ERP system, developed by Oracle Corporation, is a cloud-based solution that integrates various business processes within an organization. It offers a comprehensive suite of applications designed to manage key functions such as financial management, inventory management, order management, procurement, human capital management, and customer relationship management [6].

One of the key features of the NetSuite ERP system is its modular approach. Organizations can choose the specific modules they need based on their requirements, allowing for a customized implementation that aligns with their unique business processes. These modules integrate with each other, enabling smooth data flow and real-time visibility throughout the organization.

From the point of view of this thesis, the most interesting modules are CRM, inventory, and order management, which form the basis for order-to-cash process. The CRM module enables businesses to effectively manage customer interactions, track sales opportunities, and improve customer satisfaction. The inventory management module offers comprehensive inventory tracking capabilities, enabling organizations to effectively manage stock levels, monitor supply chain activities, and optimize inventory.

```
POST https://webservices.netsuite.com/services/
  NetSuitePort_2023_1
Header: SOAPAction: add

<soap-env:Envelope
  xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/">
  <soap-env:Header>
    <tokenPassport>
      <- ... Code omitted ... -->
    </tokenPassport>
  </soap-env:Header>
  <soap-env:Body>
    <add xmlns="urn:messages_2023_1.platform.webservices.
      netsuite.com">
      <record xmlns:q1="urn:relationships_2023_1.lists.
        webservices.netsuite.com" xsi:type="q1:Customer">
        <q1:companyName>
          Glenrock General Hospital
        </q1:companyName>
        <q1:email>
          alan.smith@example.com
        </q1:email>
      </record>
    </add>
  </soap-env:Body>
</soap-env:Envelope>
```

Figure 2.1: Example of SuiteTalk SOAP request.

Finally, the order management module facilitates the entire transaction processing, from order capture to fulfillment and invoicing. It automates order processing, improves order accuracy, and enhances customer satisfaction through efficient order tracking and delivery management.

Because the thesis aims at integration options between various ERP tools, let's also discuss NetSuite's APIs that can fulfill this task. There are three major HTTP-based options available: (1) SuiteTalk SOAP, (2) SuiteTalk REST, and (3) RESTlets.

The SOAP web services option is based on the so-called Web Services Description Language (WSDL), which provides schema for various ERP objects and allows developers to use SOAP to exchange data (discussed in section 1.4). The SOAP-based integration option is ideal for customers who require real-time data exchange and transaction processing with robust error handling

```
POST https://webservices.netsuite.com/services/rest/record/v1/
    customer
Authorization: ... Code omitted ...
Header: Content-Type: application/json

{
  "companyName": "Glenrock General Hospital",
  "email": "alan.smith@example.com"
}
```

Figure 2.2: Example of SuiteTalk REST request.

and logging functionality [7]. Compared to the REST web services, this solution might require extensive development work to implement — specialized frameworks like Apache Axis [8] or CXF [9] are typically needed. The SOAP solution can be slow and difficult to debug due to its XML-based data format and complexity. An example of a SuiteTalk SOAP request that creates a new customer is given in Figure 2.1.

On the other hand, integrations based on the SuiteTalk REST tend to be more lightweight, flexible, and easy to implement [10]. In the simplest cases a plain JavaScript available in the modern browsers can suffice. In NetSuite, a JSON format following Oracle custom media type [11] specification is utilized for data exchange with the schema being provided in Swagger/OpenAPI (discussed in section 1.2). Due to its simplicity and support by modern frameworks, this option is also used in the tool implemented in chapter 6. An example of a SuiteTalk REST request (equivalent to the SOAP one from Figure 2.1) is given in Figure 2.2.

Finally, RESTlets allow developers to build custom REST APIs. Therefore, they are ideal for customers who require custom data exchange and transaction processing. However, they require extensive development work to implement and can be difficult to maintain over time due to custom code.

2.2 Oracle Fusion Cloud ERP

Oracle Fusion Cloud ERP is a comprehensive cloud-based system offered by Oracle Corporation. It is designed to automate core business processes, enhance productivity, and provide real-time visibility into the financial, operational, and human resources data of an organization [12]. Similarly to other ERP systems, it consists of various modules such as financial management, procurement, project management, supply chain management, and others into a unified data model.

This integration enables fluent flow of information and eliminates data silos, allowing for better decision-making throughout the organization.

Oracle Fusion Cloud ERP offers multiple integration options to enable connectivity with external systems, allowing businesses to exchange data and automate processes across different applications. The major integration options include: (1) RESTful APIs, (2) SOAP Web Services, and (3) pre-built adapters.

Oracle Fusion Cloud ERP exposes a comprehensive set of RESTful APIs that allow developers to integrate and interact with various functionalities of the system programmatically [13]. These APIs enable actions such as retrieving data, creating transactions, updating records, and executing business processes. Similarly as in the case of NetSuite, the RESTful APIs provided by the Oracle Fusion Cloud ERP follow standard protocols and data formats (namely, Oracle Mime Type documented in [11] and OpenAPI described in the section 1.2), making them easily consumable by external systems.

In addition to RESTful APIs, Oracle Fusion Cloud ERP also provides SOAP-based web services that offer programmatic access to system functionalities [14]. These services provide a robust integration option for systems that rely on SOAP-based communication protocols (see the section 1.4 for details).

Finally, Oracle Fusion Cloud ERP offers a range of pre-built integration adapters, also known as the so-called Application Adapters, which enable connectivity with third-party systems [15]. These adapters provide standardized methods for integrating with popular applications and technologies, such as Salesforce, SAP, Microsoft Dynamics, etc.

2.3 QuickBooks

QuickBooks developed by Intuit company is an accounting software package designed for small and medium-sized businesses. It provides a comprehensive suite of tools for managing financial tasks, including invoicing, expense tracking, payroll management, inventory management, and financial reporting. QuickBooks offers both cloud-based and desktop versions, providing flexibility and accessibility for businesses operating in various environments [16].

QuickBooks recognizes the importance of seamless integration with other business systems and offers a robust set of APIs to facilitate data exchange and integration with external applications. Notable QuickBooks APIs include: (1) Online API and (2) Payments API.

The online API enables developers to programmatically interact with QuickBooks Online data. It provides methods for accessing and modifying a wide range of QuickBooks entities, such as customers, invoices, and expenses. This API supports RESTful principles and OAuth 2.0 authentication for secure integration [17].

The Payments API enables integration with QuickBooks Payments, Intuit's payment processing service. It allows developers to securely process credit card and other forms of payments within their applications [18].

2.4 Xero

Xero is a cloud-based accounting software platform designed for small and medium-sized businesses. With its user-friendly interface and comprehensive set of features, Xero has gained popularity as a reliable solution for managing financial tasks efficiently. Xero offers a wide range of features to streamline various financial processes, including invoicing, bank reconciliation, expense tracking, payroll management, and reporting. These features provide businesses with the tools they need to maintain accurate financial records, make informed decisions, and improve overall operational efficiency [19].

The Xero exposes the following notable APIs that can be utilized for integration: (1) Xero API, (2) Xero Webhooks.

The Xero API allows developers to build custom integrations and applications that interact with data. It provides access to a wide range of resources, including contacts, invoices, payments, and bank transactions. Similarly to the other competitors, the API uses a RESTful architecture, making it easy to integrate with a variety of programming languages and platforms [20].

The Xero's Webhooks allow developers to receive real-time notifications about specific events and changes that occur within a Xero organization. By setting up webhooks, businesses can build applications that respond to specific triggers, such as invoice creation, payment updates, or contact modifications. This enables timely and automated responses to critical financial events, facilitating proactive decision-making and reducing manual monitoring efforts [21].

Chapter 3

Contemporary ERP Integration Tools

With the rise of cloud computing and the adoption of Software-as-a-Service (SaaS) applications, there has been a growing need for businesses to integrate their disparate systems and applications to streamline workflows and improve productivity.

The Integration Platform as a Service (IPaaS) is a cloud-based solution that enables organizations to connect and integrate their disparate systems, applications, services, or databases into one centralized platform. This makes it possible to streamline their workflows and business processes and reduce the possibility of manual or human error, which improves productivity and efficiency and subsequently makes it easier to manage their digital ecosystems. It is of no significance whether these various systems are cloud-based or running on-premises.

IPaaS typically offer a wide range of features, such as prebuilt connectors for popular applications, data mapping and transformation, workflow automation, API management, real-time monitoring, reporting, and other security or compliance features. Many of these features eliminate the need for custom development and manual processes and make it easier for organizations to automate their workflows and data-sharing processes, both inside the organization and outside, between different organizations.

In this section, we will look at some of iPaaS providers, to see how they do integrations on a larger scale. We will take a look at AtomSphere, Anypoint, and Integrator.io.

3.1 AtomSphere

Boomi AtomSphere, formerly known as Dell Boomi, provides an iPaaS platform, which enables integration for over 300k applications and data sources.

Boomi provides a cloud-based integration platform that supports both the in-the-cloud deployment model, when all the integration endpoints are cloud-based, and the on-premise deployment model, when any of the integration endpoints are within the corporate network. It is a low-code development platform. This means that it provides a development environment to create software through a graphical user interface (GUI) by connecting application components together and requires little to no coding. The low-code development platform generally features drag-and-drop interfaces to help users visualize the application they are building. It features a large library of pre-built application connectors or provides guides to make integrations easier. We will focus on the most important, relevant, or interesting features of the Boomi platform, since it has countless features and some of these are not relevant to this work.

3.1.1 Integration

The integration feature is the central part of the Boomi platform. It includes tools and connections to connect apps, data, or devices and automate workflows across distributed environments. The Boomi platform supports two types of integrations. First is *B2B Integration*, which as its name indicates, integrates data or systems between two or more organizations. The second is *Application Integration*, which integrates data or systems within a single organization. In integration, the profile represents the structure and format of the source or destination data. There are many different types of profiles. The available types of profiles are `Database`, `EDI`, `Flat File`, `XML`, or `JSON` [22]. As mentioned above, Boomi supports both cloud-based and on-premise deployment models. The cloud-based model is called `Atom Cloud` and the on-premise model is called either *Atom* or *Molecule*, where Molecule is a clustered Atom capable of concurrently running multiple Atom processes. The Molecule is the enterprise-grade version of Atom that can be deployed on multiple servers to ensure load balancing and high availability for mission critical integration processes [23]. If the integration includes connecting to applications or resources behind a firewall, such as databases or other on-premises applications, it is necessary to use an on-premises model and deploy the Atom locally. As long as the integration includes only connecting to applications or resources available through the internet, it is possible to deploy the Atom to the cloud, i.e. to use Atom cloud. In this case is the integration "zero-footprint" solution that does not include any software or hardware installation, because all computing is done on Boomi's side in a data center.

3.1.2 Process

The central component within the integration is *Process*. It is a graphical representation of the path the document must take from the time it is received

to the point it is sent to a destination. The process component includes an inbound connector that retrieves data from the source. The source can be a web or on-premise application or a data source such as a disk, FTP, or database. It also includes an outbound connector to send data to one or more destinations. Destinations can be applications or data sources, the same as sources. The process also contains a series of various steps that will be performed on the data. These actions are represented as shapes. These shapes can be connected in endless combinations to build simple or complex integration workflows. There are four types of shapes: *Special*, *Execute*, *Logic*, and *Connector* shapes. There is only one *Special* shape, which is the so-called *Start shape*, which indicates the starting point of the process. *Connector* shapes load the data into the process or send them out of the process. *Execute* shapes, manipulate, or transform the data. They can transform documents, convert them to different formats, send messages or notifications, execute database commands or command-line scripts, or execute another process within the current process. *Logic* shapes direct the flow of documents through the process. They can divide the flow into multiple separate branches or they can send the document through different paths based on defined conditions. The condition value can be static or pulled from the document properties or data [24]. Integrations can run when they are scheduled or in response to the occurrence of specific events. One such event-driven integration is when we turn the integration process into a web service that can be deployed on-premise or in the cloud. It can accept both HTTP and SOAP requests. These requests are the events upon which the integration is based [25].

3.1.3 Data Mapping

One of the features of Boomi is data mapping, which uses maps to convert data from one format to another. Each mapping consists of a source profile, a destination profile, and the map itself. The *Source profile* describes the layout of the input and the *Destination profile* describes the layout of the output. The *Map* is a graphical representation of how the fields in the source profile need to be mapped to the destination profile. The user can drag and drop fields from the source profile into the destination profile to define how to move the data. It can also contain functions to allow for more complex transformation of data when moving from source to destination. These functions can use multiple fields as input and connect the output to multiple fields. There are two main types of functions. The first is *Standard*, which performs a single step, such as converting the value to uppercase or performing a mathematical operation. The second is *User defined*, which can perform complex transformations by linking together multiple standard function steps in a defined sequence. The user can determine the order in which the functions will be executed. Each field in the source profile can be connected to multiple functions, to be used as input, or to multiple destination fields.

Each destination field can be connected to only one source field or to one function output. We can even assign default values to destination fields. These default values are only used if the destination field is not connected to any source field or the value of the connected field is null or blank. The mapping is fundamentally a 1:m mapping we know from databases [26].

Figure 3.1: Example of mapping in AtomSphere. Source: Boomi forums [27].

3.1.4 API Management

Another feature of Boomi is API management. To use this feature, the user needs to create API components in Integration. It consists of two main components: *API Service* components and *API Proxy* components.

API service components are used to expose sets of REST, SOAP, or OData endpoints. We can use them to expose a different set of endpoints that will be used by different customers or partners. Each defined endpoint has a listener process configured to listen for requests for a specified operation on a particular object. The default settings for an operation on an endpoint are derived from the linked process. These default settings can be overridden. For OData endpoints, you can set an entity name and you can override the number of documents returned. For SOAP endpoints, you can override the operation name. For REST endpoints, you can override the object name and the HTTP method. For both REST and SOAP endpoints, you can override the operation's input type and output type, and for structured input and output, we can override the request and response profiles. Integration automatically generates WSDL for each SOAP API deployed and OpenAPI specification for each deployed REST API [28].

API Proxy components allow for proxying requests through an API gateway to a service that is not served through an Atom, Molecule, or Atom cloud. This means that these services are outside the Boomi platform environment [29].

3.2 Anypoint

Anypoint is an iPaaS developed by Mulesoft. The main methodology of Mulesoft is API-connectivity, which they use to connect data to applications through reusable APIs. The main parts of the Anypoint platform are *API Management* and *Integration*.

Mulesoft has its own programming language designed for transforming data, called DataWeave. It is also an expression language used to configure components and connectors. It allows users to easily perform common use cases used in application integrations like read and parse data from one format, transform it, and write it out as a different format. It allows developers to focus on transformation logic, instead of worrying about the specifics of each data format they use. The typical flow is that data go through the reader, who parses the data into a canonical model. It is then passed to the DataWeave script, where it is used to generate output, which is also a canonical model. This model is then passed to the writer, who parses the data into the desired output data format [30].

Mulesoft also has its own IDE that enables users to start building APIs and integrations quickly with pre-built connectors, templates, and examples. It is named *Anypoint Studio*. Users can use it to design and edit Mule configuration files. API specifications or properties files.

Anypoint API Manager is a component of the Anypoint platform that enables users to manage, govern, and secure their APIs. API Manager is used to enforce policies, collect and track analytical data, manage proxies and applications, and provide encryption and authentication.

There are two types of gateways in Anypoint. *Flex Gateway* is an ultrafast and lightweight gateway. It is designed to manage and secure APIs running anywhere. It is built to integrate with DevOps and CI/CD workflows. *Mule Gateway* can apply a basic authentication policy on top of a Mule application or add a complex capability to an API without having to write any code. The difference between Flex Gateway and Mule Gateway is that the flex gateway can manage and secure any API, both Mule and non-Mule. Mule Gateway protects a single Mule API.

The *AnyPoint DataGraph* is a tool that allows users to unify all the data within their application into a unified schema. You can also dynamically query data from a unified schema or explore the application network from a single UI. It can also reuse and serve information from an application without writing new code. The Anypoint platform stores all APIs as graphs of metadata. Anypoint DataGraph can connect those graph into one unified schema that runs as a single GraphQl endpoint, that contains and links all of the fields from all the APIs. The result is that the user can query across the underlying APIs without having to understand all the relations or capabilities that exist within them [31].

The design center is a development environment that includes *API De-*

signer and *Flow Designer*. API Designer is a web UI tool to design API specifications in either RAML, OAS, or AsyncAPI. The Flow Designer is used to create Mule applications to integrate systems into workflows. A flow consists of cards, each representing a core component, connertor, module, or API. Each card receives input data, performs a specific task using the input data, and then sends the output data to the next car in the flow. The last card usually sends a notification or sends the data to the target [32].

3.3 Integrator.io

Celigo company focuses mostly on the automation of workflows, with pre-built connectors for multiple applications, with easy configuration.

Users can create such flows in their iPaaS Integrator.io. One of the main parts of Integrator.io is *Flow Builder* [33]. It allows users to create custom flows in the Celigo platform using a drag-and-drop interface. The platform has a variety of connectors that can be used as a source or as a destination. If the user wants to use an application that does not have a connector, he needs to use a universal connector. Universal connectors can also be used for a customized connection. You can add mappings to the flow. Mappings can be added at several points in the flow. You can add mapping when importing data, after a look-up, or between multiple imports. The mapping allows users to customize how the data are transformed and allows them to customize which source field is mapped to which destination field [34]. The flow builder allows users to set up conditional logic inside the flows, so the system can make decisions based on certain criteria or conditions.

Chapter 4

Goals of Thesis

The main goal of this thesis is the creation of a modular tool that enables exportation of data that was exported from a source ERP system to a target ERP system. But we should divide this goal into smaller separate goals. The first goal is to thoroughly design the tool. This means designing its REST API, various internal data formats. One thing that particularly needs attention during the design phase is the modularity. Without proper design, the modularity could be only partial or non-existent at all. The next goal would be the implementation of the tool, according to the design. This means implementing at least one data source and at least one target application. The last goal would be to ensure that all the capabilities and features of the application are working correctly. To do that, a proper testing is needed. We chose to test it with data exported from Xero, because this way we can mimic a real-life scenario of a company using our tool to export their data to a new system.

The main target group of this tool is small or starting businesses. One of the main reasons this target group was selected is on account of the tool not being robust enough to handle large batches of data. It would need to handle such batches of data to be of any use to medium or large businesses, due to the sheer amount of data they have and can generate every day, mostly in terms of invoices or inventory tracking. With the target group in mind, the best data source was deemed a CSV file that is stored locally. Regarding the target system, the NetSuite ERP system was chosen, mainly because of prior experience with this system.

Chapter 5

Analysis and Design

The application was designed as a REST web service that is in principle similar to the concept of Enterprise Service Bus (ESB) [35]. The difference in our applications is that the one that initializes the connection is always our application, not one of the connected applications. In fact, the connected applications are not really connected because they have no way to initialize the connection and they do not even know about our application before we send them some requests.

Another difference is that the connections can only be a data source or only a target application, they do not have to be able to do both, send data, and receive them. That does not mean that you cannot have a connection that can handle both and have it as a data source as well as a target application.

Most of the terminology, especially with regard to data, is adopted from NetSuite (described in section 2.1) and can be different in other applications. For example, individual items saved within the application are called `records`, and the properties of the records are called `fields`. The so-called `sublist` is then a list of records inside a record, for example, a list of items on a particular invoice.

5.1 API Specification

For the development of this application, we decided to use a design-first approach when creating the API. This means that we created detailed API definitions before writing any code. Such definitions contain the structure of the API, such as endpoints and methods available on these endpoints. It can also contain the data structures that are sent in request and response bodies. This definition can be utilized as documentation for the API and can as well be used to generate server or client code. We used the OpenAPI specification while designing the API, because it is easy to use and because of previous experience with it. We also used this definition to generate the server code

that was used as a starting point for the implementation of the application.

5.1.1 OpenAPI Specification Format and Structure

The Open API document that follows the OAS [2] is itself a JSON object, which may be written in JSON or YAML format. All field names in the specification are case-sensitive. This includes all fields used as keys in a map.

There are two types of fields: *Fixed* fields, which have a specified name, and *Patterned* fields, which have a regex, which specifies the field name. All patterned fields must have a unique name inside the containing object. The data types in OAS are based on those supported by the JSON schema specification. These types are: `null`, `boolean`, `object`, `array`, `number`, and `string` [36].

The integer as a type is also supported and is defined as a JSON number without an exponent or fraction part. JSON data types can have an optional modifier property `format`. OAS defined additional formats to further expand the primitive types. The defined formats are `int32` and `int64` for integers, which are signed 32-bit and 64-bit integers. The defined formats for number are `float` and `double`, and for string is a password, which serves as a hint to UI to hide the input.

The whole document is divided into several JSON objects. Each object has several fields, where most of them are optional, but some have a few required fields.

5.1.2 OpenAPI Schema Objects

In this subsection, we will describe some of the important types of objects.

OpenAPI Object This is the root document object of the OpenAPI document. All other objects are part of this one. The required fields are *Info Object*, *Paths Object*, and the OpenAPI field, which contains the semantic version number of the OpenAPI specification.

Info Object This object provides metadata about the API. The only required fields are the title of the API and the version of the API document. It can also contain the contact object or license object. The example below demonstrates a possible structure of the Info Object.

```
info:
  title: ERP Tool
  description: This tool is used to export data
    from one ERP system to another.
  contact:
    email: staudond@fit.cvut.cz
  version: 1.0.1
```

Contact Object This object should contain the contact information for the person responsible for the API.

Server Object This object represents a server, its required field is a server URL.

Components Object This objects holds the re-usable objects for different aspects of the OAS. These objects have no effect on the API, unless they are referenced outside of the components object. It can hold `schemas`, `requestsBodies` or responses or any other type of reusable objects. In our OAS document, it holds schema of the objects that represents the body of a REST request.

Paths Object This object holds the relative paths to the individual endpoints and their operations. To construct the full URL of the endpoint, the path is appended to the URL from the server object. Each element of this object represents a different REST endpoint.

Path Item Object This object describes the operations available on a single path. It usually only contains multiple *Operation Objects*.

Operation Object This object describes a single HTTP operation on a path. It contains all the information needed about that operation, like the schema of the request body or how the responses look.

```yaml
paths:
  /v1/local:
    post:
      tags:
        - local
      summary: Export data from local CSV file
      description: Export data from local
      CSV file as a data source
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Local'
        required: true
      responses:
        '204':
          description: Successful operation
        '400':
```

```
      description: Bad request
    '500':
      description: Internal server error
```

In this code example, we can see the Paths object. The `/v1/local` indicates the beginning of the Path Item Object and the `post` indicates the beginning of the operation object.

5.2 Generated Code

As a starting point for the implementation, the code, generated by Swagger Codegen from OpenAPI specification, was used. The generated code can be separated into three main sections. These are the *API Section*, *Models of Entities*, and *Configuration Files*.

API Section: This section includes the interface that defines the REST endpoints with all available HTTP operations on them. It also includes the REST controller that implements this interface, thereby implementing the HTTP operations available on the endpoints. This means handling the incoming requests and returning the appropriate response. In addition, the generated code contains two bare bones exceptions and a filter. Moreover, it included a basic API response message.

Models of Entities: This section contains Plain Old Java Objects (POJO) classes, where each class represents the entities used by the application. Each model corresponds to a schema defined in the OpenAPI specification.

Configuration Files: This part contains configuration classes for the Swagger UI and the Swagger documentation. It also includes a converter for local date and local date time, which can convert string to corresponding time format. The last thing in this section is a controller for the root endpoint.

From these generated files, we used all the models and configuration classes. From the API part, we took the API controller and divided it into separate controllers for each endpoint.

5.3 Data Model

When we work with data, we want to organize the data into some predefined structures. Working with such structures is easier, because we know how the data is structured inside. Such structures can be called data models. In this section, we will discuss the design of data models used throughout the

ApiException	Not Used
ApiOriginFilter	Not Used
ApiResponseMessage	Not Used
NotFoundException	Not Used
V1Api	Modified
V1ApiController	Modified
CustomInstantDeserializer	Used
HomeController	Used
JacksonConfiguration	Used
LocalDateConventer	Used
LocalDateTimeConventer	Used
SwaggerDocumentationConfig	Used
SwaggerUiConfiguration	Used
Local	Used
Mappings	Used
TargetApplication	Used
RFC3339DateFormat	Used
Swagger2SpringBoot	Renamed

Table 5.1: The list of generated java files and if they were used.

application, from models of request bodies, the internal format of data to structure of mapper.

5.3.1 Request Body Format

Since the application is designed to be a REST web service, we need to define the format of the body of incoming requests to our application. Every endpoint should correspond to a data source, and the format of request body for each endpoint could be different, while still containing few of the same parameters that are required in all types of request bodies.

In Table 5.2, we can see a list of all the parameters in a request to the local endpoint, which is used when the data source is a local file, and whether they are required in the request or are optional. Furthermore, you can see if these parameters should be present in requests to other endpoints or if they are endpoint-specific.

The only parameter specific to the request body to the local endpoint is `path`, which is the path to the local CSV file. The path can be relative to the folder where the application is located or absolute. The `target` parameter determines to which target application the data should be sent. Its value is a string of the name of the application, all in lowercase. The `type` determines what type of record is the data we are sending transformed into and specifies the endpoint where the data will be sent. The `mapper` is used to transform the

Name	Is Required	Endpoints
`path`	Required	Local Only
`target`	Required	All Endpoints
`mapper`	Required	All Endpoints
`type`	Required	All Endpoints
`groupId`	Optional	All Endpoints
`sublistName`	Optional	All Endpoints
`sublistMapper`	Optional	All Endpoints

Table 5.2: The list of parameters in a request to the local endpoint.

data before sending them to the target application. The `sublistMapper` is also used to transform the data before sending them to the target application, but is only used when we are utilizing the sublist feature. It also determines which fields are placed in the sublist. More about the mapper and the sublist mapper, and their internal format can be found in a subsection 5.3.4. The `groupId` is used to group data that will be used to create sublists. The `sublistName` specifies the name of the field where the sublist is placed. The `groupId`, `sublistMapper` and `sublistName` are only used when we are utilizing the sublist feature.

5.3.2 RecordField

During the development of the application, we became aware of the fact that for some ERP applications, having the value of a field be only string is not enough. Many of these applications have complex records, which can have complex and intricate structures that involve nested objects or nested arrays of objects. A simple string is not enough to capture these structures, which is especially true for the nested objects.

For that reason, we created a new entity named `RecordField`, which, as the name suggests, is used to represent each field in the record. It can act as either a simple string value or as an object. It is designed to have two separate states where the state corresponds with what it contains inside. It can act as either a value or an object at a time, and once created, it cannot be transformed from its value state to its object state and vice versa. The original idea was that the entity would always start in its value state and if necessary would be transformed into its object state. However, that idea had a few certain issues that we needed to solve before it could work. The first issue is that when an entity contains a value of a field, it does not know the name of that field. Considering this issue, if we wanted to transform this entity from its value state to the object state, we would need to either store its name inside the entity or pass it its name upon its transformation into object. But both of these approaches have the same problem, the object has

in general a different name than the fields inside of it. We could have it saved under the name of the object and store the name of the field it contains inside of it. That would create a variable in it that would be used only in this one use case, and it would only cause problems down the line when handling this entity. It is far less complicated to decide that its going to be in object state upon its creation and put the field inside the object with its name and value.

The entity contains two variables, a field, and an object. The field is a string, and the object is a map, where both the key and the value are strings. To ensure that the entity is always in either of these two states and only in one of them, we made the field variable read only. The only way to add fields inside the object is by using method `addValueToObject`. Furthermore, we made two separate constructors, each for the construction of the entity in one state. One constructor accepts a `string` as a parameter, creating the entity in `value` state with the `string` value inside the field variable. The second accepts `two strings` as a parameter, creating the entity in the `object` state, with one field inside the object.

5.3.3 Internal Data Format

When communicating with multiple applications, it can often be the case that each application represents its data in a slightly different format. In that case, it can be pretty bothersome to have converters for converting the data between each of these formats. The usual strategy to prevent this is to always convert the data into a certain internal format and then have converters between the formats of each application and this format, which can significantly reduce the number of individual data converters.

As an internal data format, we chose to store each record as a map, where the key is the name of a field, and the value is the value of the said field. This value can be a `string`, in case of a simple `field`, or a `map`, where key and value are both `strings`, if the field is an `object`. This is possible due to using a custom class as a value that can act as either a simple string or a map. All of these records are then stored in a list.

In case we want to utilize the feature of sublists, we can group the records by a `groupId`, which results in a map, where the key is the aforementioned `groupId`, which is used to group the records together, and the value is the list of maps, where the maps have the same format as if we do not use sublists, which was mentioned earlier.

We could use the same format and group the records by `groupId` for both scenarios, but it would just make it unnecessarily complex, make parsing the data into a JSON body more difficult, and use more memory, in the case where we do not use the sublists feature.

5.3.4 Mapping component

Because we want our application to have the ability to transform the data before it is sent to the target application, we need some sort of mapper. The transformation is handled by the target class and can greatly differ between each target application. Furthermore, because we do not want to limit the complexity of the transformation, we decided to make the format a simple map, where both the key and the value are a string. With this format, the transformation can be just a simple renaming of fields, where the key is the name of the field in the loaded data and the value is the name of the field in the transformed data. We can also use the string to describe structures of various complexity. We can for example use symbol to signify that this field will be inside an object whose name is the string before the symbol. For example `item>price`, means that the field will be named `price` and it will be inside the object named `item` With this rule in place we could make multiple levels of nested objects, we would just have to add the same symbol for each level of nested objects we want. As of now, we did not find a use case for more than one level of nested objects. Consequently, this rule will be applied only once for each field, therefore only one level of nested objects is available.

If we want to have arrays in our records, we can use a feature called `sublists`. We decided to call this feature sublists because that is what it is called inside NetSuite. It works by grouping multiple records together by a `groupId`, which can be any field in the record currently used. Then, we group records with the same `groupId` and use `sublistMapper` to transform the data. The `sublistMapper` is the same as the ordinary mapper, its only used to determine which fields are to be placed inside the sublist. The number of items in the list is equal to the number of records with the same `groupId`.

5.4 Application Flow

The data flow of the application is very simple. The flow is started by sending a `POST request` to one of the endpoints of our application. The endpoint to which we decide to send a request determines what data source will be used because each endpoint represents one data source that can be utilized. The application then loads the data from the said source and transforms them into the appropriate format used by our application. The program determines which application is the target by a parameter named `target` in the request it received. The tool then uses the `target` class for this application to transform the loaded data. The transformation can be as simple as renaming fields or it can be a complex transformation of data. Finally, it sends the transformed data to the target application.

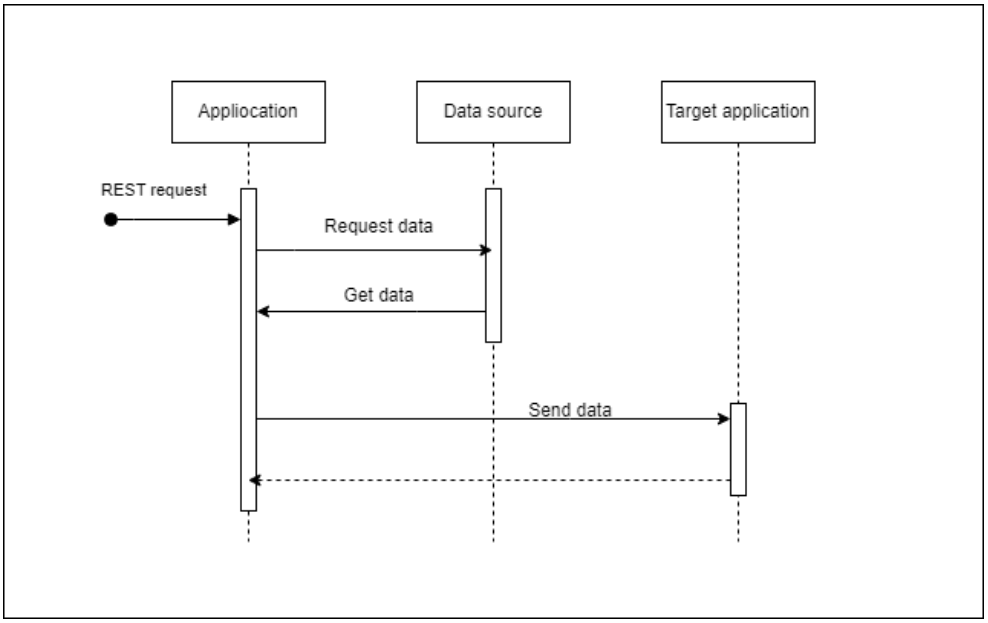

Figure 5.1: Sequence diagram of application flow.

5.5 Modularization

This application was designed with modularization in mind, so when a need arises for a new source of data or a new target application to which we can export the data, it would be fairly easy to add this functionality. We will describe how to add a new source of data and how to add a new target application.

5.5.1 Adding New Data Source

As was mentioned earlier, the application was designed with modularization in mind, so adding new sources for data or new target applications is fairly easy, apart from the actual implementation of their functionality, and can be done in a few steps.

The first step is to add a new POJO class that will represent the body of incoming REST requests to the endpoint that is to be created in the following steps. This POJO class should contain everything the application needs to load the data from the said source. It should also contain a mapper that maps data from the source to how they should be represented when they are sent to the target. For example, the field that is named `ContactName` in the source data should be named `companyName` in the data sent to the target. The mapper is a map (dictionary in JSON) where the key is the name of the field in the source data, and the value is the name of the field in the data sent to the target. It should also contain to which source should the data be

sent and what kind of data are we loading, which determines exactly where to send them. The target application usually has different endpoints to receive, for example, customer data and data about their orders. It can contain other things, but the ones mentioned above are the most important.

The second step is to add a new Source class. This class should load the data from the said source and return it as a list of maps, where the key is a `string`, and the value is either a string or map, where the key and value are both strings. Each map should represent one record that was loaded. For example, in the case of a CSV file, each map represents one line that was loaded, where the key is the header and the value is the value of said header on that line.

The third step is to add a new REST endpoint that the new source will use and create an appropriate controller for this endpoint with the method for HTTP POST operation that accepts the aforementioned POJO class as the request body. This controller should also contain bean `TargetBeans` that can return the appropriate service of the desired target. The method for `HTTP POST` operation should use the `source` class to load the data, get the Service of the target from `TargetBeans` and transform it by using the `mapProperties` method of the `target service`. Finally, it should use the `target service` to send the mapped data. We use the `POST method` because this operation is not idempotent and we also need to receive data in the body of the request.

5.5.2 Adding New Target Application

Adding a new target application is also fairly easy and can be done in a few steps, the same as was the case with adding a new data source. The first step is to add a new target class that should extend `abstractTarget` or any of its subclasses. This class should handle everything that is needed to send the loaded data to the target application. If the application needs some sort of authentication to access its API, this class should take care of that authentication. If we need to transform the data in a way different from the standard method, which is implemented in `abstractTarget`, this class can override the `mapProperties` method and implement its own way to transform the data. This class should be annotated with spring annotation @Service, so it is a spring bean, which enables dependency injection with this class, which will be needed shortly. It could be annotated with annotation `@Component`, but `@Service` better describes the intended behavior of this class.

The next step is to add a new entry in the `TargetApplications` enum for this new target application. The entry should be a string containing the name of the target application in lowercase with underscores as word separators. An example of how it should look is `new_target_application`.

The last step is to add the aforementioned target class to `TargetBeans` class via `@Autowired` annotation and edit the `getTarget` method by adding a case where the added `target class` can be returned.

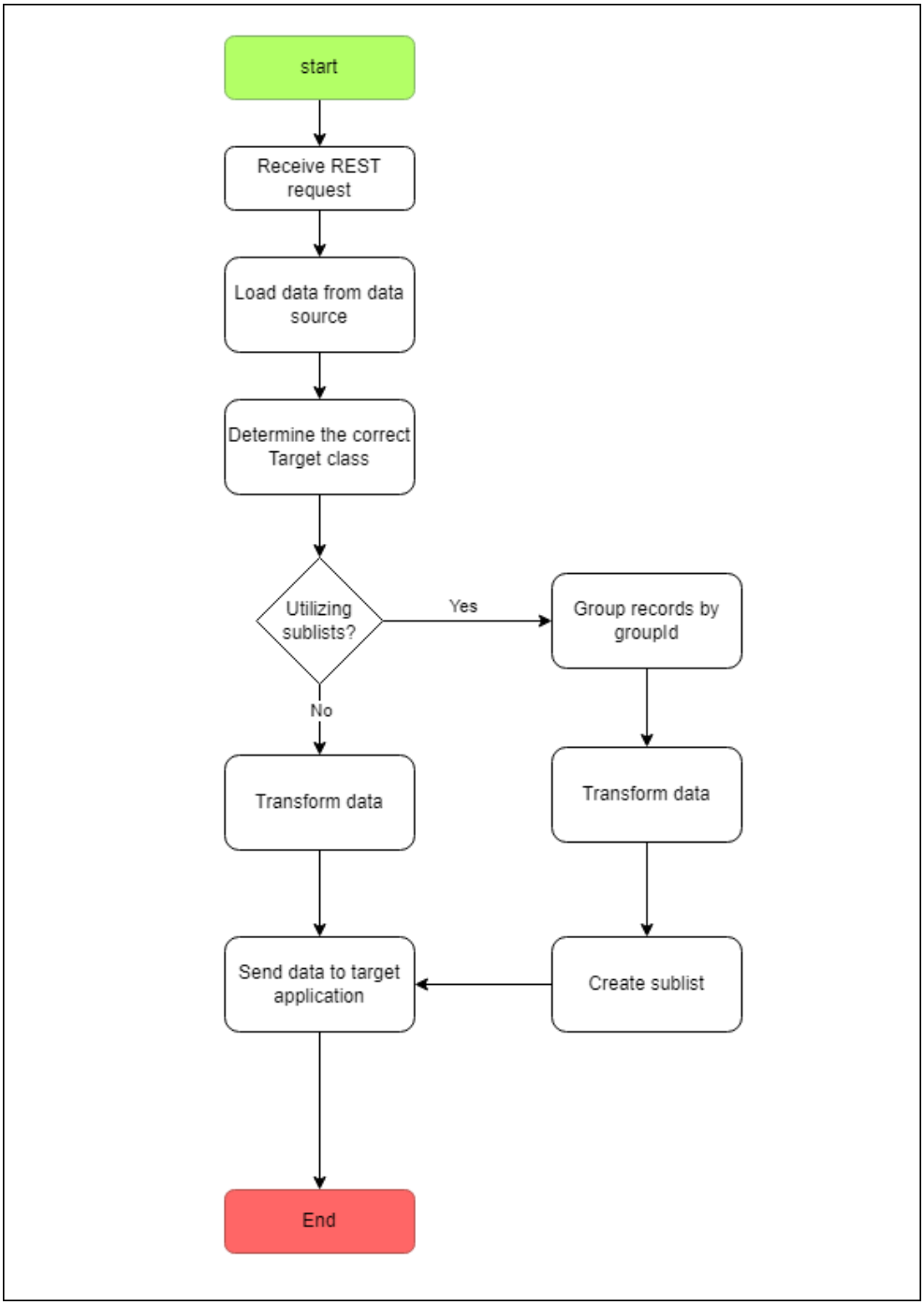

Figure 5.2: Application flow diagram.

Chapter 6

Implementation

After we have carefully and thoroughly designed all aspects of the application, we need to implement it. To implement the application, we chose to use Spring Boot as the main technology. We chose Spring Boot, because we wanted to implement the application in Java and Spring Boot is the most popular Java framework with Spring MVC being the second [37].

We used Spring Boot instead of standard Spring, because it is easier to deploy a Spring Boot application, because it comes with an embedded and pre-configured web server so we can deploy it as-is and do not need to have a separate webserver server, where we will deploy it as war. The other reason we chose Spring Boot was because the Swagger Codegen can generate the server stub in Spring Boot. To load the data from CSV files, we used the OpenCSV library. [38] To send REST requests, we used the Spring Web client.

Next, to construct the request JSON body, we used Jackson tree model. It allows one to easily add fields, objects, or arrays to one node, and convert it easily to JSON string.

The main focus of implementation can be separated into two main components, the data sources, from which the application can load data, and the target applications, where the data can be sent. Both of these components were designed to be modular, so new ones can be easily added, as is described in section 5.5. First, we will focus on the general part of implementation, and then we will take a closer look at specific implementations of both components, which will be local CSV file as data source and NetSuite as target application.

6.1 Data Sources

The first component are the data sources that we can load the data from. We will first focus on the general part of the implementation and then look at the specific implementation, which is a local CSV file as a data source. The data

sources component can be divided into a few sections. The first section would be the POJO classes that represent the incoming REST requests or some of their parts. The second would be the controllers for the REST endpoints, where one controller should control all the endpoints that belong to one data source. The last part is the `Source` class, which is responsible for loading the data from the data source and returning it as a list of maps, where the key is a string and the value can be either string or a map of strings. Each map should represent one record that was loaded, where the key represents the name of the parameter and the value is its value.

6.1.1 Local CSV File

This data source is fairly simple because it is not difficult to load data from a CSV file, especially with an abundance of libraries that can do most of the tedious work, especially parsing the data from various formats. Nevertheless, the first implementation of this data source was done without using any library. We simply loaded the data line by line and separated each line into individual fields by splitting the line with a comma as a separator. If the loaded field is empty, it is omitted and not saved, to make the data smaller in size, and because the empty fields have no use. We then put these fields into a map with their appropriate header. This method has a few problems. According to RFC 4180, which is a specification that includes the definition of CSV format, if the field is enclosed in double quotation marks, it can contain commas, line breaks, or double quotation marks [39]. We even encountered data where there was a comma inside a field, which made a mess of the loaded data. Because of this, we decided to use a library that will handle this and other future problems that could arise from the previous naive implementation. The library we decided to use is `OpenCSV` because it is easy to setup and use. At first, we wanted to use `CSVReaderHeaderAware`, because it directly converts the line in CSV into a map, where the keys are the CSV headers. That is exactly the same format to which we converted the CSV lines, so it would save us some work. Unfortunately, the CSV file, which is directly exported from Xero, does not work with this reader because it has more headers than fields, which causes an error. Moreover, since we want to use the exported CSV files without changing them, if we can help it, we decided to use a different method. We load the CSV line as if from a normal file and then parse it with a parser that adheres to all of the CSV specifications, which is appropriately named `RFC4180Parser`. Fundamentally, the only difference between our first implementation and this one is the parsing of the line into individual fields. The rest of the implementation remained the same.

6.2 Target Applications

The second component is the target applications. We will first briefly focus on the general implementation of the target, and after that we will take a closer look at a specific implementation for a target application, which in this case will be NetSuite, which is for now the only target application.

The target application component is for the most part comprised of `Target` classes, where each target class is basically a service that processes the loaded data and sends them to the target application that this class represents. All target classes must extend the `AbstractTarget` class or any of its subclasses. The other parts are some other classes needed to send the data, like entity classes representing objects to send as REST body or receive from calls to other APIs. There is also `TargetBeans` class, which is a simple class that returns a bean of an appropriate service based on the passed parameter, which is an entry from an enum of available target applications. This enum entry is received as a `target` parameter in the REST request body. `Target` classes are also responsible for the transformation of the loaded data to data that will be sent to the target application. The `AbstractTarget` class includes basic transformation, which basically means renaming the properties. If the target application requires some form of more complex transformation, the `Target` class for that target application should implement it.

6.2.1 Default Transformation

As we said before, the transformation is the responsibility of the target class. We also said that there is a default transformation that the `Target` classes can use, since they should be a subclass of the `AbstractTarget` class, which implements this default transformation. This transformation does two things.

The first is simple renaming of fields, where the key in the mapping is the name of the field in loaded data, and the value is the name it will be renamed to.

The second is placing fields into nested objects. If we want to send a JSON object as a parameter, we can use this to choose which fields will be inside the object. When we are placing fields into an object, they are at the same time also renamed. To determine whether we want to only rename the object or also place it in an object, we use the character `>`. So, if the value inside the mapper is `Item>price`, the field would be renamed to price and placed inside a JSON object with the name item. The default transformation takes each loaded record and then iterates through the mapping, checking if any of the fields we want to rename are present in the loaded data. If the field is present, we rename it, and eventually we can place it into an object, and place it into a new map that will then be returned as transformed data. This means that only fields that are in the loaded data and that are transformed are present in the data that will be sent to the target application. We chose this behavior for

multiple reasons. The first is for the user to have complete control over what fields are sent to the target application. The second is that we can have data where only some of the records have some fields that we want to transform and then use, and we do not want to send empty values for the records that do not have these fields, they are omitted from the data.

6.2.2 NetSuite

Now we will look more closely at the actual implementation of the target class. We chose NetSuite as the main target application due to our previous experience with this system and its API.

Before we can send the data to the NetSuite SuiteTalk REST API, we need some sort of authentication so that we can access the resources through the API. NetSuite offers two types of authentication for web services: Token-Based Authentication (TBA) and OAuth 2.0. We decided to use OAuth 2.0 because it is more straightforward and easier than TBA. There are two flows available for OAuth 2.0, the Authorization Code Grant Flow and the Client Credentials Flow. We use the Client Credentials Flow because it is machine-to-machine and does not require any user interaction.

To use this authentication, we need to set up the Integration and the OAuth 2.0 Client Credentials in NetSuite UI. The description of the setup of the so-called *Integration Application* and the related OAuth 2.0 flow is not captured here, because it is not essential to the thesis and it is already well-described in NetSuite documentation. Integration Set Up and OAuth 2.0 Flow for. If we set up this integration, we will get clientId when setting up the integration and keyId when setting up the OAuth 2.0 flow. These values among others are sent to NetSuite inside a request to get an authentication token that can then be used to access the resources through the API.

To get the token, we first load the private key for the certificate we assigned to the integration we are using and decode it from `base64` format. After this, we construct a JSON Web Token (JWT). This token has a predefined structure that it must adhere to, to successfully get back the authentication token.

The token header includes three parameters: `typ`, `alg`, and `kid`, where the value of `typ` is always `JWT`, the value of `alg` is the algorithm used for signing of the token and the value of `kid` parameter is the `keyId` we got when setting up the OAuth 2.0 flow.

The token payload includes five parameters: `iss`, `scope`, `aud`, `exp`, and `iat`. The value of the `iss` parameter is the `clientId` we obtained when setting up the integration. The value of the `scope` parameter should in this case be `rest_webservices`. The value of the `aud` parameter is the NetSuite token endpoint, where we will send the request. The `exp` parameter is the number of seconds from January 1, 1970, until the token expiration, and `iat` is the time when the token was issued in seconds from January 1, 1970. When we construct the JWT, we can send an HTTP

POST request to the same URL as the `aud` parameter. The POST request includes three paramaters: `grant_type`, `client_assertion_type` and `client_assertion`. The values of `grant_type` and `client_assertion_type` are always the same and are `client_credentials` for the `grant_type` parameter and `urn:ietf:params:oauth:client-assertion-type:jwt-bearer` for the `client_assertion_type` parameter. The value of the `client_assertion` parameter is the value of the JWT signed by the private part of the certificate assigned to the integration. The response we receive for sending this request contains the type of token, the time in seconds it takes for the token to expire, and the value of the access token itself in JWT format. The expiration is always the same value, that is, one hour in seconds, which is 3600. Because we always get the time it takes for the token to expire, even though it is always the same, we can request a new access token only after the old one expires, instead of requesting a new access token for each request the application receives.

As was mentioned earlier, NetSuite has a feature in which we can put a list of records inside a record. This feature is called a sublist. This feature is used if the request body contains the parameters `groupId`. It should also contain parameters `sublistName` and `sublistMapper` to function properly. When we utilize this feature, the handling of the data is a little different, so we divide the application flow between when we are not using sublists and when we are. The part that is different is the processing of the data and the construction of the request body. The sending of the requests is the same in both cases.

6.2.2.1 Standard Flow

The first thing we need to do is to transform the loaded data. We can transform the data before sending them to the target class, or send them along with the mapper to the target class and transform them there. But because we do not have a need for the untransformed data in the target class, we could transform them in the target class to separate the transformation from the controller. Another option is to do it before we send it to the target class because transformed data are generally smaller than the untransformed data and we do not have to send the mapper as well. We decided to transform it before, to send fewer data to the target class.

To create the JSON body, we use Jackson and its tree model. We create an object node, which represents the JSON object, and that will be the main object node, that will be transformed into the request body. We add all data fields by field to the main object node. If the field is a value, we first determine whether the type of value is integer, double, or something else. We do this because NetSuite requires fields, containing rates, amounts, or internal identifiers, to be sent as numbers, not as strings. If we determine that the value is integer or double, we add it as such to the object node. If not, we simply add the field with its header to the object node. If it is an object, we create an object node, adding all the fields to it, one by one, with the same

principle as we added fields that were a value to the main node. We add this object node to the main object node.

6.2.2.2 Working with Sublists

When we want to utilize the sublists, we need to group the records by the groupId. After grouping the records by `groupId`, we create the sublist. The sublist is a list of records forming a one-to-many relationship. Therefore, each element in the sublist is created by transforming data from one record. After creating the sublist, we transform the data that all grouped records had in common. These common data make up the body of the request we send to NetSuite, except for the sublist, which we need to insert into the body later. We create the JSON body almost identically to the standard flow. First, we add all the shared transformed data, field by field, to the main object node with the same principle as before. After we have added all the shared data, we need to add the sublist. We create an array node that represents a JSON array. We create an object node from all items in the sublist by the same principle we used to create the main object node. We then add all these object nodes to the array node, and finally add the array node to the main object node.

6.2.2.3 Sending the Data

After constructing the node, we can send it as a request body to the target application by using `Spring WebClient`. In older versions of Spring, `restTemplate` was used to send rest requests, but since Spring 5.0 it is recommended to use `Spring WebClient` [40]. In the following code, we see an example of how to construct a REST request using `Spring WebClient`. First, we create the instance of `WebClient` with the URL of the endpoint where we want to send the request, as a parameter. We also specify that it is a POST request. Then, we set the headers. The first header specifies that the content of the body is in JSON format and the second one contains the access token. Right after that, we set the accept header with the `accept()` method, so the response we get is in JSON. We did not set this header in the headers block because inside this block it only accepted a list of media types, so it was easier to do it like this. Finally, we set the request body, which is the object node constructed earlier.

```
WebClient
  .create(authProperties.getBaseUrl() + restUri + type)
  .post()
  .headers(httpHeaders -> {
    httpHeaders.setContentType(MediaType.APPLICATION_JSON);
    httpHeaders.setBearerAuth(token);
  })
  .accept(MediaType.APPLICATION_JSON)
  .body(BodyInserters.fromValue(body));
```

After preparing the request, we can send it by calling either the `retrieve()` method or the exchange methods. We use the `exchangeToMono()` method if the request returns a single object and we use the `exchangeToFlux()` method if it returns collection of objects. The difference between exchange and retrieve is that retrieve only returns body information, whereas exchange methods offer more control over the response by giving access to the `ClientResponse` object [41].

Chapter 7

User Guide

This application was designed as a REST Web service. The user tells the application what to do through a REST request, and it is the only way for the user to interact with the application, apart from the data the application loads from a data source.

The examples in this chapter will focus on local CSV files as the data source and NetSuite as the target application, since these are the only data source and target applications available right now.

First, the user needs to choose which data source he wants to use. Each data source has its own endpoint, and the request body, although similar in many parameters, has differences between each endpoint. After the user has chosen the data source, he needs to construct the appropriate request body. Each request has some parameters that are mandatory and can have some parameters that are optional. In the case of a local CSV file as a data source, the mandatory parameters are path, target, type, and mapper. The path is the path to the file and the type is the type of data we are sending. The target is the target application to which we send the data. Its value is a lowercase string. In the case of NetSuite, its value is `netsuite`. One of the most important parameters is the mapper, which tells us how to transform the data before sending them to the target application. It is a map (dictionary in JSON), where the key is the name of the field in loaded data, and the value is what it is transformed into. The transformation is the responsibility of the target class and can be different for each target application. In the case of NetSuite, the transformation is mostly renaming the fields or placing them inside nested objects. The parameters `groupId`, `sublistName`, and `sublistMapper` are used in case we want to utilize the feature of sublists. This feature is used if the groupId parameter is present in the request. It specifies the field that is used to group the records. More information on the parameters can be found in subsection 5.3.1 and information about the structure and functionality of mapper in subsection 5.3.4.

Here is the whole process summarized in few steps:

1. Choose the data source you want to use and its appropriate endpoint.
2. Construct the request body.
    a) Choose which target you want to use by setting the `target` parameter.
    b) Specify the type of data that is used by setting the `type` parameter.
    c) Construct the `mapper` parameter.
    d) If you want to use the feature of sublists (if it is available for the chosen target), set `groupId`, `sublistName`, and `sublistMapper` parameters.
    e) Set all remaining endpoint-specific parameters.
3. Send the request to the endpoint selected in step 1.
4. Wait until the data are processed and you receive the REST response from the application.

7.1 Running the Application

As we mentioned earlier, the application is using Spring Boot, which has an embedded web server, so it can be run as-is without any difficult deployment. The jar file of the application can be run simply with a command `java -jar Thesis-1.0.jar`. Inside the application files, there is an `application.properties` file with the configuration needed to connect to a NetSuite account. If we want to run the application with different configuration, for example, to connect to a different NetSuite account, we can run the application with an external configuration file. Spring has multiple ways to use the external configuration with a priority order to allow reasonable overriding of the configuration. If we place an application.properties file in the same directory as the jar file, this configuration has higher priority than the application file inside the jar.

We can also use a command line argument `--spring.config.location` with the value of `file:<path/to/application/properties/file>` when running the application to specify the location of the properties file.

Chapter 8

Tests

We tested our application with data exported from Xero. We tried to use the data as is, without changing anything in it. The records we wanted to test were customer, item, and sales order. These records are essential for each ERP system, so it was important that we could successfully import them into NetSuite. The application should be able to export any type of record to NetSuite, given the right mapping. We did not test all records, because NetSuite has a huge variety of records, and you can even create new custom records. Furthermore, a lot of records do not have counterpart directly in Xero, so we do not have data to import into NetSuite. Thus, we focused on the essential ones, namely customer, item, and sales order. At first, we tried to import them with the bare minimum of data possible to create each record.

After we were successful in importing them with the bare minimum of data, we wanted to import them with all the data we had. We first needed to construct the mapping, to correctly map the fields, because they are named differently in Xero and NetSuite. After constructing the mapping, we tested to import them with all the data that has equivalent fields in NetSuite and can be set through REST API.

The importing of customers went smoothly without any problems, we imported all fields that had their equivalent in NetSuite. We managed to import the address, with the use of sublists.

The exported data of item records had an issue that one record did not export correctly, because it was exported into 3 lines instead of one, and the headers did not correspond with the data. It turned out that the only problem was that the data was on multiple lines. If we made one line out of these 3, the record was without issues and the headers had the correct values. When we examined the data, we found that the items could be divided into two separate categories of items, items for sale, and items for resale. The difference is that items for sale have no purchase price, but items for resale do. NetSuite has different endpoints for each of these types of items, so we divided them into two separate files and exported them separately.

When we wanted to export the items with their price, which is important for items we encountered a problem. Because when setting price for an item, NetSuite need two separate parameters, quantity and price level.These values are not present in the files of items,so we needed to add two columns to them. The value of price level for setting base price is 1 and the items did not specify different price based on quantity. That means that the value of both columns was set to one.

The import of sales order also had a problem. Because sales orders in NetSuite need the internal identifiers of customer and item, when creating the sales order. Because these values are not present in the CSV file of sales orders, we need to edit it and add two columns, one with the internal id of customer and one with the internal id of the item. We can get the values of these identifiers in the NetSuite web GUI. There was also problem, that some items in orders did not specify which item it is.Because of this, we created item with the name of `Unspecified item`, and assigned it to the ones that did not specify the item. After adding the appropriate identifiers of the imported customers and items to the sales orders CSV, we could finally import them. We also needed to delete three orders, because the were meant as refunds, and NetSuite does not support sales orders with negative total.

With this we successfully imported the three most important types of records into NetSuite.

Conclusion

The goal of this work was to create a modular tool that allows the export of data that was exported from a source ERP system to a target ERP system. We divided this goal into three smaller ones: Design of the tool, its implementation, and subsequent testing of the tool. The design part in chapter 5 is where we designed various aspects of the application, from the REST API, multiple internal data formats, and the mapping functionality to modularization. The implementation part in chapter 6 first discussed some general aspects of the implementation and then focused on the CSV file as a data source and NetSuite as the target application. Testing was carried out with data exported from Xero to a CSV file. We tested importing these data into NetSuite. We only tested importing three different records, but the application is capable of importing any type of record, when given the right mapping. In our analysis, we first compared a few of the most notable Interface Definition Languages in chapter 1. Subsequently, we discussed selected ERP systems in chapter 2. At last, we discussed few of the iPaaS providers, and what functionality their platforms offer in chapter 3.

Future Improvements

We tested the application and imported the real data that were exported from the Xero accounting system into the CSV format. We only tested importing three types of records, however it should be possible to import any type of record when given the data and the correctly configured mapper. This is because of the general design of the application. This means that the process of loading and importing data is independent of the type of records the data represent. It depends only on the input data format, the data format in which the target application accepts the data, and whether the mapping is correct or not. The dependence on data format at both input and output is resolved by the modularity of the application and the possibility of having multiple

data sources and target applications. This means that one of the obvious improvements of this application is the addition of more data sources and target applications. An interesting data source would be to take the data from an ERP system by means of REST requests or load the data from a Google Sheets document. The target applications could also be other types of application, not exclusively ERP systems.

The application is now targeted at businesses that do not have large amounts of data to import. This is due to the application not using multithreading and the requests to the application being synchronous. This means that only one request is being processed at a time. Therefore, a useful improvement would be to make the application asynchronous and to use multiple threads to process requests faster and to be able to import larger amounts of data. Right now, the mapper implementation allows us to map the field in the loaded data to only one field in the target data, a 1:1 mapping. The limitation is due to mapper being a map, where the name of the field in loaded data is used as a key. Since keys are unique in a map, it is impossible to have more than one entry for each field in the loaded data. A possible improvement would be to improve the mapper implementation to allow for a 1:m mapping, same as it is in Boomi iPaaS. If we look at the implementation of NetSuite as the target application, there are some aspects that could be improved. Right now, we only support one sublist in a record. We could improve it to support multiple sublists. Another improvement is in the case of nested objects. Currently, we support one level of nested objects, so we could improve it to support more levels of nesting.

Bibliography

1. APIARY. *API Blueprint Specification* [online]. [visited on 2023-06-27]. Available from: https://apiblueprint.org/documentation/specification.html.
2. SMARTBEAR SOFTWARE. *OpenAPI Specification* [online]. [visited on 2023-06-25]. Available from: https://swagger.io/specification/.
3. STOIKOVITCH, Jonathan. *RAML Version 1.0: RESTful API Modeling Language* [online]. [visited on 2023-06-27]. Available from: https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/.
4. CHRISTENSEN, Erik; CURBERA, Francisco; MEREDITH, Greg; WEERAWARANA, Sanjiva. *Web Services Description Language (WSDL) 1.1* [online]. [visited on 2023-06-27]. Available from: https://www.w3.org/TR/wsdl.html.
5. HADLEY, Marc. *Web Application Description Language* [online]. [visited on 2023-06-27]. Available from: https://www.w3.org/Submission/wadl/.
6. ORACLE CORPORATION. *NetSuite Enterprise Resource Planning (ERP) System* [online]. [visited on 2023-04-06]. Available from: https://www.netsuite.com/portal/products/erp.shtml.
7. ORACLE CORPORATION. *SuiteTalk SOAP Web Services Platform Overview* [online]. [visited on 2023-05-12]. Available from: https://docs.oracle.com/en/cloud/saas/netsuite/ns-online-help/chapter_N3412777.html#SuiteTalk-SOAP-Web-Services-Platform-Overview.
8. APACHE SOFTWARE FOUNDATION. *Apache Axis* [online]. [visited on 2022-07-22]. Available from: https://axis.apache.org.

9. APACHE SOFTWARE FOUNDATION. *Apache CXF* [online]. [visited on 2023-06-12]. Available from: https://cxf.apache.org.
10. ORACLE CORPORATION. *SuiteTalk REST Web Services Overview and Setup* [online]. [visited on 2023-05-12]. Available from: https://docs.oracle.com/en/cloud/saas/netsuite/ns-online-help/chapter_1540391670.html#SuiteTalk-REST-Web-Services-Overview-and-Setup.
11. *New Media Type for Oracle REST Services to Support Specialized Resource Types*. 2015-03. Tech. rep. Oracle Corporation. Available also from: https://www.oracle.com/webfolder/technetwork/tutorials/appdevinfo/New%5C%20REST%5C%20Media%5C%20Type.pdf.
12. ORACLE CORPORATION. *Oracle Fusion Cloud ERP* [online]. [visited on 2023-01-22]. Available from: https://www.oracle.com/erp/.
13. ORACLE CORPORATION. *REST API for Oracle Fusion Cloud Financials* [online]. [visited on 2023-03-16]. Available from: https://docs.oracle.com/en/cloud/saas/financials/23b/farfa/rest-endpoints.html.
14. ORACLE CORPORATION. *SOAP Web Services for Financials* [online]. [visited on 2023-03-18]. Available from: https://docs.oracle.com/en/cloud/saas/financials/23b/oeswf/index.html.
15. ORACLE CORPORATION. *Prebuilt Application Adapters* [online]. [visited on 2023-03-18]. Available from: https://www.oracle.com/integration/application-adapters/.
16. INTUIT. *QuickBooks Online* [online]. [visited on 2023-04-09]. Available from: https://quickbooks.intuit.com.
17. INTUIT. *QuickBooks Online API Reference* [online]. [visited on 2023-04-09]. Available from: https://developer.intuit.com/app/developer/qbo/docs/api/accounting/all-entities/account.
18. INTUIT. *QuickBooks Payments API Reference* [online]. [visited on 2023-04-09]. Available from: https://developer.intuit.com/app/developer/qbpayments/docs/api/resources/all-entities/bankaccounts.
19. XERO LIMITED. *Xero Accounting Software* [online]. [visited on 2023-05-26]. Available from: https://www.xero.com.
20. XERO LIMITED. *Xero API Overview* [online]. [visited on 2023-05-26]. Available from: https://developer.xero.com/documentation/api/accounting/overview.
21. XERO LIMITED. *Xero Webhooks Overview* [online]. [visited on 2023-05-26]. Available from: https://developer.xero.com/documentation/guides/webhooks/overview.

22. BOOMI. *Boomi Documentation - Integration* [online]. [visited on 2023-06-27]. Available from: https://help.boomi.com/bundle/integration/page/c-atm-Integration_and_iPaaS.html.

23. BOOMI. *Boomi Documentation - Molecule* [online]. [visited on 2023-06-27]. Available from: https://help.boomi.com/bundle/integration/page/c-atm-Molecules.html.

24. BOOMI. *Boomi Documentation - Process Building* [online]. [visited on 2023-06-27]. Available from: https://help.boomi.com/bundle/integration/page/c-atm-Process_building.html.

25. BOOMI. *Boomi Documentation - Event-based integration* [online]. [visited on 2023-06-27]. Available from: https://help.boomi.com/bundle/integration/page/c-atm-Event-Based_Integration.html.

26. BOOMI. *Boomi Documentation - Map component* [online]. [visited on 2023-06-27]. Available from: https://help.boomi.com/bundle/integration/page/c-atm-Map_components.html.

27. YADAV, Bhagyashree. *Boomi map shape* [online]. [N.d.]. [visited on 2023-06-28]. Available from: https://community.boomi.com/s/question/0D51W00007yl47uSAA/how-to-do-a-many-to-one-mapping-from-custom-scripting-functions-to-destination-profile.

28. BOOMI. *Boomi Documentation - API Service components* [online]. [visited on 2023-06-27]. Available from: https://help.boomi.com/bundle/api_management/page/int-API_Service_components.html.

29. BOOMI. *Boomi Documentation - APIProxy components* [online]. [visited on 2023-06-27]. Available from: https://help.boomi.com/bundle/api_management/page/int-API_Proxy_components.html.

30. ERNEY, Joshua. *What is DataWeave?* [online]. [visited on 2023-06-27]. Available from: https://developer.mulesoft.com/tutorials-and-howtos/dataweave/what-is-dataweave-getting-started-tutorial/.

31. MULESOFT, LLC. *Anypoint DataGraph Overview* [online]. [visited on 2023-06-27]. Available from: https://docs.mulesoft.com/datagraph/.

32. MULESOFT, LLC. *About Design Center* [online]. [visited on 2023-06-27]. Available from: https://docs.mulesoft.com/design-center/.

33. SANTIAGO, Tom. *Create custom flows* [online]. [visited on 2023-06-27]. Available from: https://docs.celigo.com/hc/en-us/articles/360025919171-Create-custom-flows.

34. SANTIAGO, Tom. *Response mapping, results mapping, and failed record flow behavior* [online]. [visited on 2023-06-27]. Available from: https://docs.celigo.com/hc/en-us/articles/4414777521307-Response-mapping-results-mapping-and-failed-record-flow-behavior.

35. GARTNER, INC. *Enterprise Service Bus: A Definition* [online]. [visited on 2023-06-27]. Available from: https://www.gartner.com/en/documents/1405237.

36. WRIGHT, Austin; ANDREWS, Henry; HUTTON, Ben; DENNIS, Greg. *JSON Specification* [online]. [visited on 2023-06-25]. Available from: https://datatracker.ietf.org/doc/html/draft-bhutton-json-schema-00#section-4.2.1.

37. JETBRAINS S.R.O. *The State of Java Developer Ecosystem 2022* [online]. [visited on 2023-06-24]. Available from: https://www.jetbrains.com/lp/devecosystem-2022/java/.

38. *Opencsv* [online]. [N.d.]. [visited on 2023-06-28]. Available from: https://sourceforge.net/projects/opencsv/.

39. SHAFRANOVICH, Yakov. *Common Format and MIME Type for Comma-Separated Values (CSV) Files* [online]. [visited on 2023-06-17]. Available from: https://www.ietf.org/rfc/rfc4180.txt.

40. NICOLL, Stéphane; HOELLER, Juergen. *Spring Framework 5 FAQ* [online]. [visited on 2023-06-24]. Available from: https://github.com/spring-projects/spring-framework/wiki/Spring-Framework-5-FAQ.

41. VMWARE, INC. *Spring Framework 5.3.13 API Documentation* [online]. [visited on 2023-06-24]. Available from: https://docs.spring.io/spring-framework/docs/5.3.13/javadoc-api/org/springframework/web/reactive/function/client/WebClient.RequestHeadersSpec.html#exchangeToMono-java.util.function.Function-.

Appendix A

Acronyms

API Application Programming Interface

CRM Customer Relationship Management

CSV Comma-Separated Values

ESB Enterprise Service Bus

ERP Enterprise Resource Planning

GUI Graphical user interface

HTTP Hypertext Transfer Protocol

IPaaS Integration Platform as a Service

JSON JavaScript Object Notation

POJO Plain Old Java Object

RAML RESTful API Modeling Language

REST Representational state transfer

SaaS Software as a service

SOAP Simple Object Access Protocol

UI User Interface

URL Uniform Resource Locator

W3C World Wide Web Consortium

WADL Web Application Description Language

WSDL Web Services Description Language

XML Extensible markup language

YAML YAML Ain't Markup Language

Appendix B

OpenAPI Specification

```yaml
openapi: 3.0.3
info:
  title: ERP Tool
  description: This tool is used for exporting
  data from one ERP system to another.
  contact:
    email: staudond@fit.cvut.cz
  version: 1.0.1

servers:
  - url: http://localhost:8080/

tags:
  - name: local
    description: Local CSV file as data source

paths:
  /v1/local:
    post:
      tags:
        - local
      summary: Export data from local CSV file
      description: Export data from local
      CSV file as a data source
      requestBody:
        content:
          application/json:
            schema:
```

```
              $ref: '#/components/schemas/Local'
        required: true
      responses:
        '204':
          description: Successful operation
        '400':
          description: Bad request
        '500':
          description: Internal server error

components:
  schemas:
    Local:
      type: object
      properties:
        path:
          type: string
          description: Absolute path to the local file
        type:
          type: string
          description: Type of record we are exporting
          e.g. Customer, SalesOrder etc.
        groupId:
          type: string
          description: Name of property that is used to group
          records together. Used for sublists.
          If this property is set, it signalizes
          that we want to use sublist functionality.
        sublistName:
          type: string
          description: Name of the sublist needs to be specified,
          because in different types of records,
          they can be named differently.
        target:
          $ref: '#/components/schemas/TargetApplications'
        mapper:
          $ref: '#/components/schemas/Mappings'
        sublistMapper:
          $ref: '#/components/schemas/Mappings'
      required:
        - path
        - type
        - target
        - mapper
```

```yaml
Mappings:
  type: object
  additionalProperties:
    type: string


TargetApplications:
  type: string
  enum:
    - netsuite
```

Appendix C

Contents of enclosed media

- `readme.txt` ..... the file with CD contents description
- `exe` ..... the directory with executables
- `src` ..... the directory of source codes
  - `app` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `resources` ..... the directory with csv files used for import
  - `Thesis.postman.json` ..... Postman collection with sample requests
- `thesis.pdf` ..... the thesis text in PDF format