Cloud Based System Integration

System Integration between Salesforce.com and Web-based ERP System using Apache Camel

Molnbaserad systemintegration
Systemintegration mellan Salesforce.com och ett webb-baserat ERP-system med Apache Camel

Henrik Johansson
Mikael Söder
Abstract

In an era of technological growth, cloud computing is one of the hottest topics on the market. This, along with the overall increased use of digital systems, requires solid integration options to be developed. Redpill Linpro recognizes this and has developed a cloud-based Integration Platform as a Service (IPaaS) solution called Connectivity Engine. New techniques like this can however seem very abstract to a customer, something which a demo application could help substantiate.

To aid in this issue we have developed a web-based Enterprise Resource Planning (ERP) system as well as an Integration Application to connect the ERP system with Salesforce.com in a bidirectional integration. With the use of Connectivity Engine this can be hosted in the cloud and be easily accessible.

The project has been a success for Redpill Linpro as well as the authors. A solid way to demonstrate the abilities of Connectivity Engine has been developed along with descriptive documentation for any sales representative assigned to pitch the platform.

Keywords: Apache Camel, Camel, Spring Framework, Spring Boot, Integration, DevOps, OpenShift, Open Source, Salesforce, API, Continuous Integration, Integration Application, Rest-Templates, JSON, Java, ERP System, CRM, ERP, REST
Acknowledgements

Special thanks to:
Redpill Linpro
Fredrik Svensson - Co-founder and regional manager, Redpill Linpro
Pontus Ullgren - Software architect and developer, Redpill Linpro
Hans Hedborn - Supervisor, Karlstad University

Karlstad, June 12, 2017
Henrik Johansson and Mikael Söder
1 Project Implementation 23

1.1 ERP System Implementation ........................................... 23
1.1.1 Database and JPA Repositories ....................................... 23
1.1.2 Data from Salesforce .................................................. 25
   1.1.2.1 Insert or Update .............................................. 25
   1.1.2.2 Bulk Insert .................................................. 28
   1.1.2.3 Delete .................................................... 28
1.1.3 Local Data Operations and Salesforce Transmission ............... 29
   1.1.3.1 Get All .................................................... 29
   1.1.3.2 Create .................................................... 29
   1.1.3.3 Edit ..................................................... 30
   1.1.3.4 Delete .................................................... 31
   1.1.3.5 Salesforce Requests ....................................... 32
1.2 Integration Application .................................................. 33
   1.2.1 The Salesforce Camel Component Configuration ................. 33
   1.2.2 Camel Routes .................................................. 34
   1.2.3 JOLI Mapping .................................................. 40
   1.2.4 Processors .................................................... 41
1.3 Deployment .............................................................. 42
1.4 Use Cases ........................................................................ 46
   1.4.1 Use Case 1 - Creating a New Contact on Salesforce ............ 46
   1.4.2 Use Case 2 - Edit a Case in the ERP System ................... 47
   1.4.3 Use Case 3 - Sync All Accounts from Salesforce ............. 48
1.5 Choice of Tools and Framework ......................................... 48
1.6 Summary ................................................................. 49

5 Results and Evaluation ..................................................... 50

5.1 Project Results ............................................................ 50
5.2 Evaluation of the ERP system ........................................... 50
5.3 Evaluation of the Integration Application ............................. 51

6 Conclusion ................................................................. 52

6.1 Project Conclusion ....................................................... 52
6.2 Problems ....................................................................... 52
   6.2.1 Streaming APT Random Disconnect ............................ 52
   6.2.2 TableName With MariaDB ........................................ 52
   6.2.3 Testing ........................................................... 53
# Final Remarks

## Appendix A  Database tables and enums

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 Account</td>
<td>57</td>
</tr>
<tr>
<td>A.1.1 Account table</td>
<td>57</td>
</tr>
<tr>
<td>A.1.2 Account enumerators</td>
<td>57</td>
</tr>
<tr>
<td>A.2 Contact</td>
<td>58</td>
</tr>
<tr>
<td>A.2.1 Contact table</td>
<td>58</td>
</tr>
<tr>
<td>A.3 Case</td>
<td>58</td>
</tr>
<tr>
<td>A.3.1 Case table</td>
<td>58</td>
</tr>
<tr>
<td>A.3.2 Case enumerators</td>
<td>58</td>
</tr>
</tbody>
</table>

## Appendix B  UML Diagrams

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1 Integration Application</td>
<td>60</td>
</tr>
<tr>
<td>B.2 ERP System</td>
<td>61</td>
</tr>
</tbody>
</table>
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Project design overview</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>DevOps process overview. Image from SUSE, Thinking DevOps [1]</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>CP</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>MVC Architecture</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>Project design overview</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>Diagram depicting classes and dependencies between them</td>
<td>13</td>
</tr>
<tr>
<td>7</td>
<td>ERP system database model</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>Listing of Contacts (some fields have been cut for the image to fit on screen)</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>Listing of Contacts belonging to an Account (Fields have been cut for the image to fit on screen)</td>
<td>17</td>
</tr>
<tr>
<td>10</td>
<td>Edit form with format error</td>
<td>18</td>
</tr>
<tr>
<td>11</td>
<td>Diagram depicting classes and dependencies</td>
<td>20</td>
</tr>
<tr>
<td>13</td>
<td>Conditional routing in Apache Camel. The message router directs the data based on its content. Image from Apache Camel documentation [3]</td>
<td>37</td>
</tr>
<tr>
<td>14</td>
<td>Visual representation of the route by Hawthio</td>
<td>39</td>
</tr>
<tr>
<td>15</td>
<td>Screenshot of the OpenShift management application</td>
<td>43</td>
</tr>
<tr>
<td>16</td>
<td>OpenShift environment variables</td>
<td>44</td>
</tr>
<tr>
<td>18</td>
<td>OpenShift</td>
<td>45</td>
</tr>
</tbody>
</table>
1 Introduction

Cloud computing is one of the, if not the, most growing technology used in IT. Redpill Linpro recognizes this and have developed a cloud-based Integration Platform as a Service (IPaaS) called Connectivity Engine \cite{connectivity_engine}. More about Connectivity Engine in section 2.2.3. This platform provides over 180 different connectors to popular applications and services as well as allow users to build their own custom components for specific requirements. We are set out to build another connector to help distinctively demonstrate the capabilities of Connectivity Engine.

1.1 Motivation

A big problem with the growing use of cloud-based services and integration between different applications is the ability to substantiate it to a customer. The process can be hard to grasp for someone who is less technically equipped. This can scare away some customers from the approach that is cloud based integration. Redpill Linpro felt that a form of demo application to show the prospect what is actually happening would improve the sales pitch by a great deal.

The idea is to connect the cloud-based Customer Relationship Management (CRM) system Salesforce (more on Salesforce follow in section 2.2.1) through Redpill Linpro’s own Connectivity Engine with a simple demo Enterprise Resource Planning (ERP) system as a web application. This way the user can see that the synchronization went smoothly.

The project presented interested us right away. It allows for work with many of the current tools and techniques used in the business and will build relevant knowledge for the future. With the growth of the Internet of Things (IoT), integration and APIs in particular are becoming a big part of the market \cite{iot_integration}.

1.2 Specification Requirements

The desired ERP system should hold three of the entities present on Salesforce.com – Accounts, Contacts and Cases. A Case should be connectable to both a Contact and an Account and a Contact in turn should be connectable to an Account. There should be functionality for creating, editing and deleting these entities as well as create and break connections between them.

---

\footnote{The Salesforce logo is a trademark of Salesforce.com, inc.}
An Integration Application (IA) for Connectivity Engine should be developed as well, to sync data. An overview of Connectivity Engine as well as an explanation of what an Integration Applications is will be given in greater detail later on. As entities gets added, updated or deleted on Salesforce.com, the IA should sync automatically with the ERP system to show the changes. If possible, and within the time-frame, the same synchronization should be implemented in the other direction as well, meaning changes made in the ERP system should also be made on Salesforce. All the interactions should be clearly logged to be able to easily demonstrate each step in the process.

Since the applications are built for demo purposes, solid documentation was emphasized. This should include things like available functionality, route diagrams and directions to show relevant log entries.

1.3 Results

1.3.1 Expected

Not being knowledgeable in the tools and techniques relevant to the project the assignment felt very abstract. It was hard to estimate the workload and whether it was doable at all. After initial briefings with the technical supervisor appointed to us, it cleared up a bit however.

The initial requirement was to build an ERP system and a unidirectional integration from Salesforce with a stretch goal of it being bidirectional. We knew from the start that we wanted to complete the bidirectional integration.

1.3.2 Actual

The project went as desired with the bidirectional integration being implemented well within the margin of time. Every aspect of the project started out slow but as we caught on to the techniques, proceeded to accelerate exponentially.

1.4 Disposition

In Chapter 2 the reader is introduced to the company behind the project as well as given an overview of the tools and techniques used to complete it. This covers already implemented systems such as Salesforce, as well as frameworks like Spring Boot which is used to build the applications.

Chapter 3 will introduce the design of the applications. How the database, UI and data handling is designed.

Next, in Chapter 4, we cover in a more technical way how we implement this design using tool and techniques described in Chapter 2. This includes implementation of the database, how data is received, handled and transmitted both in the ERP system and the Integration Application as well as deployment of the applications. Moreover, three use cases are described to give a clearer picture of the entire chain of events.
Chapter 5 goes over the general results of the project and evaluation of both the ERP system and the Integration Application. Finally, chapter 6 presents the project conclusion with managed problems and final remarks.
2 Background

This section is set to describe the project as well as the company behind it. The reason for the project along with tools and techniques used are also discussed.

2.1 Open Source and Redpill Linpro

Redpill Linpro is a company founded in Karlstad, Sweden, with offices throughout the Nordic region who specialize in Open Source services and products [3]. With Open Source the source code used to create the software is, as the name suggests, open and easily accessible.

As the source code is available to everyone the possibilities of debugging, implementing changes on the basis of feedback and further development increase drastically. Feedback given to developers is not necessarily limited to what is wrong or what should be added generally, but can go as far as to specify on a code level what should be done.

Since the software is, in a majority of cases, available for free, the cost for acquiring it goes down considerably. The money usually spent on licenses could instead be spent on buying additional services such as installation, support, adjustments and consultation.

Open Source code also result in a significant increase in flexibility for the user. The ability to see how a program is built opens up the potential of knowing exactly how to approach it to e.g. share and exchange data, integrate it with other services or adjust it to fit your needs.

This type of product require Redpill Linpro to provide their absolute best service to their customers as the flexibility of Open Source allows them to easily switch providers as they please.

2.2 Tools and Techniques

2.2.1 Salesforce

Salesforce is a cloud based customer relationship management (CRM) system provided by the American company Salesforce.com as a Software-as-a-Service (SaaS) platform [8]. The main purpose of a CRM system such as Salesforce is to improve and maintain customer relationships for its users which usually consists of medium and large sized companies.

- **Accounts** on Salesforce represent a company or an organization and usually have one or more associated **Contacts** and **Cases**.

- A **Contact** is a person who, as the name states, acts as a Contact person for an Account. The Contact is usually an employee of the company he or she represents. Prior to the Salesforce summer ’16 release one Contact may only have one associated Account. As of the summer ’16 release Salesforce provides an option to let a Contact have multiple associated Accounts [3]. However, this is considered to be an advanced option and thus is disabled by default.
• Although not required, a Case usually has an associated Contact and an associated Account. This Account and Contact does not have to be associated to each other, although it is most common that they are. A typical Case may be a Contact who asks for repairs or installation instructions for a product. A Case can never have more than one associated Account or Contact. Accounts and Contacts may however have multiple Cases assigned to them.

A major selling point for Salesforce is that it runs entirely in the cloud and is accessed through a web interface as well as a mobile application called Salesforce1. Because of its SaaS design, no local installation is required which, in theory, reduces costs for implementation and maintenance for those who adopt it.

Salesforce is easily integrated with external systems. Facebook integration is provided by default as well as the possibility to send emails from within the system. Salesforce allows its users to specify so called workflow rules to automate processes. A common task is to notify users by email if a certain action occurs, e.g., if a new customer is added.

2.2.2 DevOps and Continuous Integration

DevOps is a fairly new concept in the world of software engineering [11] whereas the term continuous integration has been around for awhile [14]. The Connectivity Engine platform is designed with the philosophies of continuous integration and DevOps in mind.

![DevOps process overview](image)

Figure 2: DevOps process overview. Image from SUSE, Thinking DevOps [14]

Contrary to earlier approaches where developers and system administrators were working in somewhat more isolated teams the DevOps philosophy propagates for a tighter collaboration between the two to create a more unified process which is seen in Figure 2. The delivery of software should happen rapidly, frequently and be automated.
Continuous integration simply states that new contributions to the software should be implemented in small chunks while continuously running automated tests to ensure quality and stability. This ensures that there’s always an updated and stable version of the software available which can be deployed into production.
2.2.3 Connectivity Engine

Provided as an Integration Platform as a Service (IPaaS) by Redpill Linpro, Connectivity Engine allows for more than 180 different applications and services to be integrated through its so called connectors[2]. Connectors for Salesforce and cloud based file storage service Dropbox as well as more basic technologies such as HTTP, REST and FTP are available[3].

![Connectivity Engine architectural overview](image)

Connectivity Engine consists of a Devtime and a Runtime environment which both are built out of various open source tools which helps to practice the ideas of DevOps and continuous integration. Below follows a detailed description for each component seen in Figure 3.

2.2.3.1 Integration Applications

When a new system is to be integrated a new Integration Application needs to be developed. The Integration Application contains all the logic needed to complete the integration and is further described in section 2.2.3 below.

2.2.3.2 Devtime

- **Template Archetypes** provides a set of templates used as a starting point to begin developing new Integration Applications.

- **Source Control Management** is provided using Git which is one the most common industry tools for version control. The usage of a strict branching strategy where new features are added to applications in separate branches and merged into a master branch when finished allows for new versions of applications to be deployed continuously.

- **Continuous Integration and Delivery** is provided using Jenkins which is an open source automation tool used for building, testing and deploying software. Jenkins ensures that unit tests are being run on each commit and release of an Integration Application before being deployed to production[12].

---

[2]Figure provided by Redpill Linpro.
• **Code Quality Management** to ensure consistent usage of indentation style and detection of unreachable code is provided using code inspection tool *SonarQube*. Inaccuracies such as unreachable code and not following coding conventions will result in decreased readability and complicates maintenance later on. This is known as *technical debt*.

### 2.2.3.3 Runtime

When ready for production the application is deployed to runtime as a *Docker* container which ensures strict separation between applications. The usage of container technologies such as Docker improves security through isolation but also facilitates load balancing. In Connectivity Engine, Red Hat product *OpenShift* (based on Google’s *Kubernetes*) is used for management of Docker containers. OpenShift and Docker containers are explained in section 4.3 Deployment.

• Applications are exposed to the outside world through APIs using **API Management** tool *WSO2* [13]. WSO2’s web front end allows for easy creation and monitoring of APIs.

• The **connectivity** layer is built on top of the open source *Apache Camel* paragraph 2.2.4.1 project which is an integration framework for the Java programming language that implements the Enterprise Integration Patterns (EIP) as stated by Gregor Hohpe and Bobby Woolf in *Enterprise integration patterns : designing, building, and deploying messaging solutions* [14].

• **Monitoring/Logging** is provided using *Kibana* for logging and *Hautio* for real time monitoring of integrations [17]. In this context it can be used to give a visual representation of the Camel routes we’ll implement later on and also provide us with live monitoring of the data flowing through them. This is particularly useful in a demo case since the integration and the data it processes can be shown to a potential customer when it happens in real time.

### 2.2.4 Integration Application

#### 2.2.4.1 Apache Camel

Apache Camel is, as previously mentioned, an open source integration framework [18]. It works in conjunction with many Java-based domain-specific languages, one of them being Spring Boot which will be explained below. To use it, add the Camel dependency to the project.

Camel works with any kind of transport or messaging model, for example HTTP, using Uniform Resource Identifiers (URIs). Using the API the developer can set up routes to handle incoming and outgoing data. Routes typically receive data from some source, like a queue, processes it and sends it to an endpoint. The endpoint can be another queue, a Java message service or an HTTP endpoint among others.

**Salesforce Component**

Being open source, Camel also has a Salesforce component available. With this component the
developer can query data from Salesforce, send request to update data and subscribe to different topics, such as when an Account has been updated [13].

Jackson
Jackson is a JSON processor for Java. It can be used to convert JSON data to Plain Old Java Objects (POJO) and the other way around as well. It supports integration with Camel for marshalling and unmarshalling objects among other things [20].

JOLT
Jolt is a JSON to JSON transformation library which also supports integration with Apache Camel. The transformation follows user-written specifications also written in JSON [21].

2.2.5 ERP System as a Web Application

2.2.5.1 MVC Architecture
Model-View-Controller (MVC) architecture is a common design approach for web application development. It focuses on the three components: model, view and controller (Figure 2). In this paragraph the MVC design with database storage will be described.

Figure 4: MVC Architecture

The **model** part is what represents the database tables. Each attribute in a model refer to a column in the table. It is typically possible to set restrictions and properties for these by using language specific annotations defined in provided libraries.

The **view** is the component responsible for displaying a user interface (UI). Generally the views are used to represent the models. Views are used to show data and enable the user to perform operations on it. This can for example be a create form where the user can input new values for model attributes and pass it on to the controller for handling.

**Controllers** are in charge of handling all user interaction. This includes working with the models to perform database operations as well as rendering the correct view back to the user. An
example would be if a user edits a tuple through a view, the controller is responsible for taking care of the input by querying the database, saving the changes and selecting the right view to display.

2.2.5.2 Spring Boot

Spring boot is a convention over configuration solution for creating stand-alone Spring-based web applications. Convention over configuration is a design paradigm used to reduce the amount of decisions for the developer. This basically means that everything will follow the implemented convention unless the developer explicitly specifies that it should not.

Spring is on its own a popular Java-based framework used to build enterprise applications deployable on any kind of platform. It is entirely open source and has been around since 2002 [22]. It provides dependency injection, aspect-oriented programming, an MVC web application framework, foundational support for database access through JDBC, JPA, Hibernate and all the other commonly used data access frameworks in Java, as well as multiple more modules [23]. Spring also provides support classes to facilitate the process of writing unit tests and integration tests.

Spring Boot simplifies the Spring framework by reducing the complexity of configuring the web application. Dependency management is more straightforward and easier to handle. By using starter dependencies e.g spring-boot-starter-<dependency name> provided by Spring Boot, version control and possible missing related dependencies are taken care of by the dependency manager, whether it is Maven or Gradle.

Auto configuration is another big component of Spring Boot. When using regular Spring to build a web application, configuration of the DispatcherServlet (responsible for mapping requests to the right handlers), resource handlers, data source properties, entity management among other things, is necessary for the application to work. With Spring Boot all of this is done under the hood but is still writable if there is a need for special configuration.

It also comes with support for an embedded web container so that deployment to an external server becomes redundant. Which container to use is simply decided by which dependency is included, Tomcat or Jetty being prime candidates. This results in the possibility to package the application as a jar, making it runnable from command line by simply typing java -jar <package name>.

2.2.5.3 Spring Data Java Persistence API (JPA)

JPA includes the API, the Java Persistence Query Language (JQL) and object-relational mapping. It uses entities to build database tables. Attributes in an entity corresponds to columns in a database table. Entities can also have relationships with other entities, expressed through object-relational mapping. This mapping can be implemented by simply using annotations in the classes. What Spring Data JPA does beyond JPA is add another layer of abstraction, defining a standard as to how to implement repositories [24].
Repositories for each entity is created to perform operations on them. These repositories implement the JpaRepository interface which holds standard functions such as save, delete and findAll among others. In the repositories it is also possible to define methods by using JPA named queries which then translates to JPQL queries. There are defined keywords such as And, Or, LessThan, GreaterThan, OrderBy etc. A named keyword example would be findByNameAndLastName(String firstname, String lastname), which will create a query that selects the entry where firstname and lastname match the input [20].

2.2.5.4 RestTemplate
RestTemplate is a Java-based Representational state transfer (REST) web service client developed by Spring. It inherits from an HTTP client and provides higher level methods for data transfer with the HTTP methods [23].

2.2.5.5 H2
H2 is a lightweight relational database written in Java. Commonly used during development because of its small footprint and its ability to run in-memory even though persistent storage on disk is supported. It supports standard SQL as well as a JDBC API [27].

2.2.5.6 MariaDB
The MariaDB project is a community-developed fork of, and fully compatible with, the well proven MySQL database [28]. It was made by the original developers of MySQL after Oracle’s acquisition and is guaranteed to stay open source [29].

2.2.5.7 Thymeleaf
Thymeleaf is a commonly used server-side Java template engine. Its main purpose is to bring natural templates to development. The meaning of this is that it works just as well in both web and standalone environments. Thymeleaf comes with full Spring integration, meaning it is effortless to integrate with other Spring libraries [31].

Evaluation and iteration of objects with Thymeleaf is done by linking them to template tags inside HTML code, for example:

```html
<span th:if="${error}">Error text</span>
```

Listing 1: Evaluation

```html
<tr th:each="account : ${accounts}"
     th:if="${account.name}"
     th:text="${account.name}">
  <td th:text="${account.city}"
```

Listing 2: Iteration

11
2.3 Summary

In this section, the company behind the project, Redpill Linpro, and their modus operandi is discussed, along with their motivation for conducting it as well as our own interest in it. The growth of integration and APIs, primarily cloud-based, is also touched upon. Moreover, the tools, techniques and components worked with to carry out the task are described, with Salesforce, Connectivity Engine, Apache Camel and Spring Boot being the main ones.
3 Project Design

In this section the overall design of the systems are discussed. As the project contains two parts, the ERP system and the Integration Application, the design section is broken down into two parts as well. Section 3.2 describes the design of the ERP system. Section 3.3 in turn, goes over the design of the Integration Application - the so called Connector in Connectivity Engine.

3.1 Design Overview

Figure 5: Project design overview.\(^3\)

Figure 5 shows a simple overview of the whole project with Salesforce, the ERP system and the Integration Application which is implemented using Apache Camel.

Any changes in either system triggers a notification to the Integration Application containing the data that have been altered. The data coming in at one end of the Integration Application may not be compatible with the system on the other end and may therefore require some modification before being fed into it (see Apache Camel routes in section 3.3.2).

3.2 Design of ERP System

Figure 6: Diagram depicting classes and dependencies between them

\(^3\)The Salesforce logo is a trademark of Salesforce.com, inc.
The entities **Account**, **Contact** and **Case** each have repositories (implemented as interfaces) in which available operations on the database are specified. These are then used in controllers to query the database and update it according to the operations performed by the user (*Figure 6*). Each entity also has views corresponding to these operations, where the user can make his desired changes e.g create, edit and delete. A bigger version of the UML diagram can be found in Appendix E.

### 3.2.1 Database

The ERP system consists of the **Account**, **Contact** and **Case** entities with relationships defined between them in a relational database. Below follows an overview of the design of this database.

![ERD diagram](image)

*Figure 7: ERP system database model*

As seen in *Figure 7* one entity, or table, may have either a **one-to-many** or a **many-to-one** relationship with the other two. Although being very alike in design, the tables in the ERP system are not a mirroring of the tables in Salesforce since they contain only a portion of the fields present there. Below is a list of all tables and their relationships explained together with their **primary** and **foreign keys**. In relational databases, a **primary key** is used as a unique
identifier for a tuple (row) whereas a foreign key refers to another table and thereby specifies a relationship to that table.

- The **Account** table has a one-to-many relationship defined with the Contact as well as the Case table. This simply means that an Account may have multiple associated Contacts and Cases. The `ACCOUNT_ID` field is the primary key of the Account table. No foreign keys are present here.

- The **Contact** table has a many-to-one relationship with the Account table and a one-to-many relationship with the Case table. This corresponds with the earlier stated fact that a Contact may only have one associated Account but multiple associated Cases. Here, the `CONTACT_ID` field is used as the primary key whereas the `ACCOUNT_ID` field acts as a foreign key.

- The **Case** table has a many-to-one relationship with the other two. Because of this a single Case can only have one associated Contact and Account. The primary key here is the `CASE_ID` field whereas `CONTACT_ID` and `ACCOUNT_ID` are foreign keys.

When assigning a Contact to a Case it will automatically be assigned to the Contact’s Account, if it has one and if the Account property of the Case is unassigned. However, it is possible to assign the Case to another Account than that of the Contact.

### 3.2.2 User Interface

When designing the user interface for the ERP system, simplicity was a cornerstone. The point of the system is to easily distinguish between database entries and the possible operations. During a demonstration for a potential customer it should be easy to identify when something is added or changed. To achieve this, tables are used to display all the data available for the given entities.

A menu at the top of the screen in the form of a nav bar is used to navigate to whichever part of the website the user wants to view. There are also buttons to sync all Account, Contacts and Cases from Salesforce as well as a empty the local database to start a fresh demo.

The different entities available to view and operate on are **Accounts**, **Contacts** and **Cases**. Clicking on any of these buttons will take the user to a table listing information about the desired entity in table form. There are buttons for the user to edit or delete each entity [Figure 8].
Figure 8: Listing of Contacts (some fields have been cut for the image to fit on screen)

For Account it is also possible to view the Contacts and Cases connected to that Account. Similarly for Contact, a listing of connected Cases is available. When navigating to these connected elements it is made clear by the title and header that you are now viewing a list of the connected tuples. The delete button is also changed to a different colored "Remove from <Entity navigated from>" button to be extra clear that the element’s connection to the Account/Contact is the only thing being removed (Figure 9).
Figure 9: Listing of Contacts belonging to an Account (Fields have been cut for the image to fit on screen)

The editing of elements is done by a simple form where each property available for editing is listed. If an input error is made by the user, a message is listed under the miswritten property when trying to save. If correctly edited, the page redirects back to the listing of the elements (Figure 10).
3.2.3 Data from Salesforce

The application is built to handle and correctly map data from Salesforce. Incoming data is handled by different methods depending on which entity is being sent from the Integration Application and how it should be handled. When receiving, the incoming JSON properties are directly mapped to its corresponding attributes in the specified entity. In the case of a bulk request, the reception is handled using the batch-classes containing lists of entities which are then iterated through.

Salesforce uses ids as strings to identify their entities. Account, Contact and Case all have unique externalIds. A Contact also has an accountId to specify its possible connection to an Account. Cases have both an accountId and a contactId which are used to handle its connections on their end. These ids correspond to that entity’s externalId, e.g account Id on a Case is that Account’s externalId.

Add or Edit

If the action requested from the Integration Application to the ERP system is to add or edit, all the connections between entities have to be handled first. This is to make sure that no entity-connections on Salesforce are lost, no matter the order, or approach, in which the data is added.
When adding or editing an entity, the ids previously mentioned on Salesforce are used to identify the existing connections between the entities. Using these, the local foreign key connections are made according to the database design of the ERP system. Once the correct connection is bound the database is updated.

**Delete**

When a *delete* request is received by the ERP system, a query to the database is made first, to verify if the entity exists locally. If this is the case, all the connections between entities are also evaluated and subsequently removed before deleting from the database. The connection-ids used by Salesforce are not needed in this case, since the link is already specified according to the local database design.

3.2.4 Local Data Handling and Salesforce Transmission

Handling data locally with database transactions requires less evaluation since the connection between entities is specified according to the local design choices. However, when this data is to be sent to Salesforce through the Integration Application, the ids used by Salesforce are necessary again for them to be able to handle the data. For instance if a Case has a connection to an Account and a Contact locally, the account Id and contact Id of these has to be updated for Salesforce to be able to make the correct connection on their end.

Creating new entities with connections as well as adding new links to already existing ones requires both the foreign key connection to be made, and the correct ids to be updated. The foreign key connection is for local use, while the ids are for Salesforce’s handling of the objects.

When deleting entries, the local connections are removed for internal management purposes and the ids used by Salesforce are removed so Salesforce knows to break the connections.

The local entity operated on is then converted to JSON data before being sent to the Integration Application. This is to make the later transformation to Salesforce-compatible properties possible.

There is also functionality for emptying the database completely. Since the application is built to be demo-friendly, this is a needed feature when starting a new demonstration.
3.3 Integration Application

![Diagram](image)

Figure 11: Diagram depicting classes and dependencies

In Figure 11 the SalesforceCamelComponent class uses the SalesforceCamelConfig classes to set up configuration and authorization. The main application class then initializes the Component to be able to access Salesforce data, and the four RouterBuilders to set up the endpoints.

The RouterBuilders contains all the defined routes for data handling and makes use of the processors to manage incoming data. The Account, Contact and Case Data Transfer Objects (dto), with their accompanied enums, are used to map incoming data and the QueryRecords classes, containing list of their corresponding dtos, enables handling of bulk data. A bigger version of the figure is available in Appendix E.

3.3.1 Salesforce APIs

Salesforce.com provides a set of APIs that may be used to integrate the Salesforce platform with external systems such as our ERP system. The APIs that are interesting for this project are the following.

- The REST API which is used to fetch as well as to insert data into Salesforce using the HTTP REST standard. The API supports the execution of queries in the Salesforce Object Query Language (SOQL) which is a SQL-like query language and thereby allows for a large amount of data to be fetched in a single API call. When using the REST API the applications that make use of it have to initialize the API request towards Salesforce. The REST API is duplex since it can not only send data from, but also insert data into, Salesforce. For more information see the SalesForce REST API pages [31].

- The Streaming API is used to receive live update notifications when an object, such as an Account, is created, updated or deleted in Salesforce. When using the streaming API the application that chooses to implement it creates a subscription which specifies which
fields, entities, and actions that should trigger a notification. The main difference between the REST API and the streaming API is that the streaming API uses push technology, meaning that calls are triggered from within Salesforce and then pushed to the clients as notifications. This means that applications that implements it will receive almost instant updates of what is happening in Salesforce. The streaming API is simplex in the sense that it only allows for data being sent from Salesforce to various clients. Insertion of data into Salesforce is not possible with the streaming API. For more information see the Salesforce Streaming API pages [32].

3.3.2 Apache Camel routes

The APIs explained in Salesforce APIs (section 3.3.1) are being interacted with by specifying endpoints which will receive the data from Salesforce. The endpoints in this case are set up in Apache Camel. The data will eventually be passed on to another endpoint in the ERP system, after being processed and transformed.

In Apache Camel, the concept of an endpoint receiving data which may be modified or processed and then passed on to another endpoint is called a route. Routes are the core functionality of Apache Camel [33].

![Figure 12: Simple model of a Camel route. Image from Eclipse documentation](image)

In Figure 12 is a simple model of a Camel route. Data is sent from a system to an endpoint on the route and then processed in one or more processors. The function of a processor may vary wildly but does usually involve modification of data and/or logging. After being processed, Camel sends the data into another system through the endpoint at the other end of the route and thereby completes the integration.

Apache Camel provides a Salesforce component which supports both the REST and streaming APIs which makes the integration possible.

For the integration between Salesforce and our ERP system we make use of a total of fifteen routes involving both of the APIs mentioned earlier. Routes exists to handle creation and updates as well as deletion of the Account, Contact and Case entities respectively. These routes implement the streaming API and thus detects changes made on Salesforce and sends the new data to the ERP system.

The REST API is used to perform an initial synchronization by bulk fetching data from Salesforce as well as for insertion of data into Salesforce when changes occur in the ERP system.
When feeding data into the ERP system a simple HTTP POST call is made to an endpoint and the data needed is provided in the body as JSON.

3.4 Summary

In the Project Design section, the ERP system and Integration Application have been discussed. Both applications have been broken down into subsections where the ERP system section contains the database design, the user interface and the way local data is handled as well as how data is sent to, and received from Salesforce. The Integration Application section, in turn, describes the APIs used to get data from, and send data to, Salesforce and the ERP system. The Apache Camel routes used to implement it and process the data are also described.
4 Project Implementation

This section goes more into detail on how each entity and operation, is managed by the systems - both the ERP system and the Integration Application.

4.1 ERP System Implementation

4.1.1 Database and JPA Repositories

During the development phase the H2 (paragraph 2.2.3.3) database engine with its in-memory features was used. In-memory databases only live during execution and all data is lost when the application is terminated. Not very usable in production but very much so during development.

Since the database is temporary it has to be populated every time the application is launched. This is what the DatabaseSeeder class is for. The CommandLineRunner interface is implemented which ensures that the run method is executed once the application has started. The class will instantiate a predefined set of Accounts, Contacts and Cases and store them to H2.

All database queries are managed by Spring Data JPA using repositories which are classic Java interfaces that are annotated with the @Repository annotation. These repositories extends the JpaRepository which takes the entity class and the primary key data type as arguments. A repository is implemented for each entity respectively, AccountRepository (Listing 3), ContactRepository (Listing 4) and CaseRepository (Listing 5) and provides create, read, update and delete (CRUD) operations for these entities in the database.

```
@Repository
public interface accountRepository extends JpaRepository<Account,
Long> {
    Account findById(Long id);
    Account findByExternalId(String id);
    Long countByExternalId(String id);
    Long countById(Long id);
}
```

Listing 3: Repository interface for the Account entity
@Repository
public interface contactRepository extends JpaRepository<Contact, Long> {
    Contact findById(Long id);
    Set<Contact> findByAccount(Account account);
    Contact findByExternalId(String id);
    Set<Contact> findByAccountId(String id);
    Long countByExternalId(String id);
}

Listing 4: Repository interface for the Contact entity

@Repository
public interface caseRepository extends JpaRepository<Case, Long> {
    Case findById(Long id);
    Set<Case> findByAccount(Account account);
    Set<Case> findByContact(Contact contact);
    Case findByExternalId(String id);
    Set<Case> findByAccountId(String id);
    Set<Case> findByContactId(String id);
    Long countByExternalId(String id);
}

Listing 5: Repository interface for the Case entity

By declaring methods that follows the Spring Data JPA naming convention we can use these methods to perform CRUD operations on the database. There’s no need to define and instantiate classes that implement these interfaces, Spring does this automatically upon compilation. By doing this Spring removes the time consuming task to write the boilerplate code usually needed to get a database connection up and running.\footnote{From Wikipedia - The Free Encyclopedia: In computer programming, boilerplate code or boilerplate refers to sections of code that has to be included in many places with little or no alteration.}  

When the system is ready for production the H2\footnote{paragraph 2.2.5.3} database is dropped in favor of MariaDB\footnote{paragraph 2.2.5.6} which stores data permanently to disk. For more information on deployment see section 4.4.3. 

Although it may seem as a drastic move to change the database back end completely when deploying to production it actually is extremely straightforward thanks to JPA. As explained in paragraph 2.2.5.3 JPA uses repositories and named queries and thus abstracts all database
operations completely. This means that we can change the database software to any database supported by JPA without modifying a single line of code.

A detailed listing of the entity tables and their enumerators is available in appendix A.

4.1.2 Data from Salesforce

The incoming data is handled according to which operation is requested by the Integration Application (IA). Different methods are used for bulk insert, insert or update and delete. These methods in turn differ depending on which entity is being operated on, although they are similar.

```java
@RequestBody
public void fromSalesforceCreateOrUpdate(@RequestBody Account account) {
    // Method for receiving an Account entity from the IA
}
```

Common for all methods handling data from Salesforce, or rather from Salesforce through the Integration Application, is that they all use the `@RequestBody` annotation to specify which URI they listen to for incoming data (line 1, listing 6). This means that the Integration Application can send the data to the specific URI mapped to handle the desired entity and operation.

Moreover, all methods receive and map data through the use of the `@RequestBody` annotation (line 3, listing 6). What this does is basically map the incoming JSON properties directly to the corresponding attribute names in the specified entity e.g. `@RequestBody Account account` will map JSON data to the Account entity. In this case properties like `name` and `accountNumber` will match directly to their identically named attributes in the Account class. The reason it is this simple is because the incoming JSON data has already been transformed in the Integration Application to match the ERP system’s attributes.

The methods also send an HTTP status OK back to the IA on completion as seen on line 2 in listing 6.

4.1.2.1 Insert or Update

The methods for insert or update vary depending on which entity is handled, but all of them have the same fundamentals. First check if the entity already exists in the database, then look over connections to other entities and set or remove them depending on the data coming in from Salesforce i.e. if a connection exists on Salesforce.

As Salesforce uses a 18 character autogenerated string as an id, which is not allowed to be set externally, we have to check against that id when data is coming in. The database is queried using the `findById` defined in the repositories and if the incoming entity exists, its
internal id (autogenerated Long value) has to be set. Coming from Salesforce this id is null, but if the entity exists locally with the same externalId, it has a regular id that can be used.

```java
if (accountRepository.countByExternalId(account.getExternalId()) > 0) {
    account.setId(accountRepository.findByExternalId(account.getExternalId()).getId());
}
```

Listing 7: Check if entity exists

In essence, if the incoming Account exists, set its internal id (null when coming in) to its own id which it has in the database – ensuring that when the save operation is called later on, the existing entity will be updated instead of creating a new identical one.

**Account**

Aside from the aforementioned operations, the method in charge of handling incoming Accounts has to check Contacts and Cases which has, or is supposed to have, a connection to it. To do this the findByAccountId query defined in the contactRepository and the caseRepository is used.

```java
if (!contactSet.isEmpty()) {
    for (Contact contact : contactSet) {
        contact.setAccount(account);
        accountRepository.save(account);
        contactRepository.save(contact);
    }
    contactRepository.save(contact);
}
```

Listing 8: Connect Account to Contact

The set of Contacts is then evaluated and if it is not empty, it is iterated through with a foreach loop. For each Contact in the set, its Account is set to the incoming one and both the Account and Contact is saved (Listing 8). The Account is saved first so that when the Contact is saved it does not have a reference to a non-saved entity, which is not allowed by the database implementation.

The same exact process is then carried out for Case, only difference being that the Case’s Account is set instead.

**Contact**

Since a lot of different Cases can be connected to a Contact, a set of Cases is initially extracted here as well, with the use of findByContactId in the caseRepository (Listing 9). All the Cases,
if any, are then looped through, connected to the Contact and saved along with the incoming Contact. Same procedure as when Contacts and Cases are connected to an Account.

```java
if (contact.getAccountID() != null) {
    Account account = accountRepository.findByExternalId(contact.getAccountID());

    if (account != null) {
        contact.setAccount(account);
    }
}
else {
    Contact dbContact = contactRepository.findByExternalId(contact.getExternalId());

    if (dbContact != null && dbContact.getAccount() != null) {
        contact.setAccount(null);
    }
}
```

Listing 9: Setting Account connection

When handling the possible connection to an Account, first we take care of the scenario where the Contact has an accountId (and therefore an Account on Salesforce). If this is the case, an attempt to retrieve the Account in question from the database is made. This is done using `findByExternalId` in the `accountRepository` where accountId of the Contact is passed as argument. If the attempt is successful, and the Account exists in the database, the connection to the Account is made (Listing 9).

There is also the situation where the Contact does not have an accountId. This has to be handled as well since the underlying reason can be that the connection has been removed on Salesforce. In this case it also needs to be removed in the ERP system.

As also seen in Listing 9 this is handled by an else-statement corresponding to the previous check if the Contact has an accountId. If it does not have one, we have to check if the incoming Contact exists in the database. This is to make sure that an attempt to break a non-existing connection is not incorrectly made. Beyond that, the Contact needs to have a connected Account in the database as well if the connection is to be broken. If these evaluations are true the Contact’s Account is set to null. Lastly the incoming Contact is saved using the `contactRepository`.

Case

A Case has the possibility of being connected to one Account and/or one Contact. This means that no iteration through possible Accounts or Contacts connected to the case is needed.
The possible connection to an Account and a Contact does however have to be evaluated. This is done in the exact same way as when an Account is connected to a Contact as described in the previous paragraph. The only difference being that the Case and its accountId and contactId are used to evaluate connections.

4.1.2.2 Bulk Insert

The bulk insert works the same way for all three entities. Below follows the method for Account as an example [Listing 10].

```java
public void fromSalesforceGetAll(@RequestBody AccountBatch accounts) {
    accounts.getRecords().forEach(account ->
        fromSalesforceCreateOrUpdate(account));
}
```

Listing 10: Bulk insert Account

The AccountBatch class holds a list of Accounts called records. When the Integration application forwards bulk data it does so in a JSON array of (in this case) Accounts, where all attributes have been transformed to match those of the ERP. The JSON array is then mapped to the list of Accounts which is iterated through and entered into the system using the method for insert or update explained above in paragraph 4.1.2.1.

4.1.2.3 Delete

All the methods handling delete requests from Salesforce are almost the same, no matter which entity is deleted. They all check if the incoming entity exists in the database and if it does, deletes it. The only difference between deleting an Account, Contact or Case is that they are handled in their own respective controller.

The reason it can be done this simple is because Salesforce does not allow for deletion of entities which others have a connection to. This means that an Account with Contacts or Cases which are connected to it cannot be deleted until the connections are removed. The exact same thing applies to a Contact with Cases connected to it. A Case however can be deleted freely since it (possibly) holds a connection to an Account and a Contact, but has no other entities connected to itself.

What this results in is that whenever a delete request is received by the ERP system, all the possible connections that entity had have been removed beforehand, since this is a requirement in Salesforce’s system. Because of this, the streaming API has already triggered on those connection removals and the ERP has broken them as well, making the incoming entity free to delete.
4.1.3  Local Data Operations and Salesforce Transmission

The operations **create**, **edit** and **delete** are available for all entities, along with the possibility to **getAll** and view them. Depending on if the request to create, edit or delete is of type GET or POST different methods handle the call.

4.1.3.1  Get All

The methods for **getAll** are the same for all three entities with the sole exception being that they use their own corresponding **repository** to get data. The entities are then added in a model attribute and later on extracted in the view.

Account also has **getAllContacts** and **getAllCases** methods for Contacts and Cases connected to it and Contact, a **getAllCases** method for connected Cases. These methods take the id of the desired entity to view connections to as a **@PathVariable** as seen in [Listing 11]

```java
@RequestMapping(value = "/account/contacts/{id}", method = RequestMethod.GET)
public String getAllContacts(@PathVariable Long id, Model model) {

Listing 11: Method Declaration getAllContacts
```

Using the id, the entity is extracted and subsequently its connected entities are as well, using **contactRepository.findByAccount** or **caseRepository.findByContact**, depending on which are requested. Same as before, all connected Contacts or Cases are extracted and added in a model attribute.

Each entity has a matching **listAll** view which iterates through all the data added in the model attribute and prints it in a table. The **listAllContacts** view is also used to list all Contacts connected to a certain Account and the **listAllCases** view to list Cases connected to an Account or a Contact.

With the use of the request URI we can determine if for example all Contacts should be listed or if just the Contacts connected to a specified Account should. If the request URI is "/contact/all" all Contacts should be shown but if it is "/account/contacts" only Contacts connected to the chosen Account should. See section 3.2.2 for the UI difference.

The URI is also used to modify the view, and the available delete operation, further to decide if the delete trashcan icon or the "Remove From Account" button, along with their mapped operations, should appear.

4.1.3.2  Create

**Account**

The GET request for **createAccount** adds a new empty Account to the model and returns the correct view. The attributes of this Account are then set by the user in a form, and later on
submitted. When submitting, a POST request is sent to the same URI as the GET, but is mapped to another method. This method receives the Account with the @Valid annotation along with a BindingResult to make sure that all attribute restrictions are met. This includes things like input type of a field as well as restrictions set, like for example @NotBlank.

The BindingResult will contain errors if the input form data is invalid. This is evaluated at the start of the method and if it has errors the user is returned to the create page once again, but this time with error messages printed for what went wrong. If the form data is valid however, the Account is saved and passed to the sendRequestToSalesforce method, with create as operation, which then sends it to the Integration Application for transformation and transmission to Salesforce. Finally the user is redirected back to the listAllAccounts view.

Contact
For the createContact GET request all the available Accounts are added to the model in excess of the new empty Contact. This is because when creating a Contact, the user is able to connect it to an Account.

The Contact POST request works similar to the one for Account with the difference being that createContact also, possibly, connects an Account. This Account is chosen by the user from a dropdown list where the available ones are listed along with an unassigned option. The latter results in null being passed to the controller. If an Account is chosen, it is connected to the Contact and the accountId of the Contact is set to the externalId of the Account.

Lastly the Contact is saved, sent to the Integration Application and the user is redirected back to listAllContacts.

Case
The method createCase works just like createContact with the addition of one more possible connection. Since Case can have a connection to a Contact as well as an Account, both lists have to be added to the model. Just as for Contact, the user chooses the connections through dropdown lists in the create view. Any Account and Contact combination can be chosen.

The selected connections, if any, are set along with the Case’s accountId and contactId which are set to the incoming Account’s and Contact’s externalId respectively. If the user has chosen a Contact but no Account, the Case is automatically assigned to the Account of the selected Contact, assuming it has one.

4.1.3.3 Edit

Account
The editAccount method handling the GET requests takes the id of the desired Account as a parameter. This is passed as a @PathVariable from the listAllAccounts view in the same way as in Listing 11. Using this id the Account is fetched from the database and added to the model, and the view for Account editing is returned.
The POST method is exactly the same as the one for `createAccount` but with the `edit` operation passed to the method handling Salesforce transmission (`sendRequestToSalesforce`).

**Contact**
The GET request for `edit` Contact adds a list of all available Accounts to the model in addition to the Contact itself, same as `createContact`.

The POST request is close to the same as for `createContact`, with the `bindingResult` and setting of connection to an Account. The only differences are that if the user changed Account to connect to, the `accountId` has to be updated to match it, or `null` if `unassigned` is chosen. Also that an `edit` operation is passed to `sendRequestToSalesforce`.

**Case**
Same as for the GET request to `createCase`, all available Accounts and Contacts are added as separate lists to the model along with the Case.

The POST method has a small addition to the `createCase`. It evaluates the `status_enum` as well as the value of `dateTimeClosed` to determine if a closed date should be set or removed from the Case. If no closed date exists and the status is changed to `CLOSED`, `dateTimeClosed` is set to the current time. If the Case's status is changed from `CLOSED` to anything else, the closed date is removed.

### 4.1.3.4 Delete
All delete methods are similar except for a few differences. Same for all methods is that the entity to be deleted is extracted from the database using its id and then deleted using the `delete` method in the corresponding repository e.g `accountRepository.delete(account)`.

The `delete` operation is then passed to the `sendRequestToSalesforce` method of the active controller and the user is redirected back to the `listAll` view matching the entity.

What differs between them is the evaluation of connections in the methods to delete Account and Contact. For Account it is assessed if any Contact or Case has a connection to it. If this is the case the Account may not be deleted and a warning message is passed to the view and displayed for the user ([Listing 12]).

```java
if (caseRepository.findByAccount(account).size() > 0 ||
    contactRepository.findByAccount(account).size() > 0) {
    redirectAttributes.addFlashAttribute("notAllowed", "Not allowed,
                                      Account still has associated Contacts and/or Cases");
    return "redirect:/account/all";
}
```

Listing 12: Check if entities has connection to Account

For Contact it is just the possibility of connected Cases that has to be checked.
4.1.3.5 Salesforce Requests

Each of the three `sendRequestToSalesforce` methods takes the entity to be operated on as well as the operation to perform (as a string) as parameters.

The `RestTemplate` class described in paragraph 2.23.3 is then used to pass all forms of requests to the Integration Application, which in turn processes and transforms the data and forwards it to Salesforce.

**Single Entity Operation**

```java
1. restTemplate.postForObject(url, entity, String.class);
2. restTemplate.put(url, entity);
3. restTemplate.exchange(url, HttpMethod.DELETE, entity, String.class);
```

Listing 13: RestTemplate methods used

A switch statement with the operation as parameter is used to evaluate which request is to be sent, POST, PUT or DELETE.

The `RestTemplate` methods in Listing 13 all take the desired url for the request as well as an instance of the `HttpEntity` class as arguments. The url is set to `iaDomain + "<entity>"`, where `iaDomain` is mapped to an environment variable in `application.properties` which can be changed easily if the application host were to change. The entity, in these cases, contain the desired headers – `Content-Type` and `Authorization` – and the object to operate on. `Content-Type` is set to `json/application` and `Authorization` to the authorization type, which is `basic`, followed by the base64 encoded username and password. This type of encoding means that an `HTTPS` connections is required for it to be secure.

In Listing 13 `postForObject` and `put` automatically sets the `HttpMethod` to `POST` and `PUT` respectively. For the `exchange` method on line 3 to be a `DELETE` request however it needs to be passed as the second argument. `postForObject` and `exchange` also needs the response type as a final argument.

**Get Bulk Requests**

```java
1. restTemplate.exchange(url, HttpMethod.GET, new HttpHeaders(), String.class);
```

Listing 14: RestTemplate methods used

Different for the bulk request is the uri as well as the entity. The `HttpEntity` only contain headers for authorization as well as the self-defined header `dtoClass`, which is set to the entity’s name.
i.e. ‘account’, ‘contact’ or ‘case’. Directly is set so the IA can distinguish which bulk request to send to Salesforce. The `HttpMethod` header is set as the second argument of `exchange`.

### 4.2 Integration Application

This section is set to give a detailed explanation of the implementation of the Integration Application. As earlier mentioned in the background [section 2.2.3], the Integration Application is implemented using Apache Camel. The configuration of the Apache Camel Salesforce component is explained as well as all Camel routes used for the integration.

#### 4.2.1 The Salesforce Camel Component Configuration

As mentioned in [section 2.2.3], Apache Camel provides us with a component to simplify integration towards Salesforce. The Salesforce component needs to be configured before it can be used, which is done in three different Java classes. Below follows an explanation of these classes and the configuration of the Salesforce component.

**SalesforceCamelLoginConfig**

The sole purpose of the `SalesforceCamelLoginConfig` class is to store values used for authentication against the Salesforce system. The class contains a constructor which when called will create a Java object with the credentials as attributes. These credentials are set using environment variables which are declared in the `application.yml` configuration file. In Table 1 the ones used by the Integration Application are listed.

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>security:user.password</td>
<td>Sets the password for the Integration Application</td>
</tr>
<tr>
<td>sf:username</td>
<td>Salesforce username</td>
</tr>
<tr>
<td>sf:password</td>
<td>Salesforce password</td>
</tr>
<tr>
<td>sf:clientId</td>
<td>Salesforce client Id</td>
</tr>
<tr>
<td>sf:clientSecret</td>
<td>Salesforce client secret</td>
</tr>
<tr>
<td>sf:loginurl</td>
<td>URL for the Salesforce login endpoint</td>
</tr>
<tr>
<td>sf:securityToken</td>
<td>Unique Salesforce security token</td>
</tr>
<tr>
<td>erp:domain</td>
<td>Domain for the ERP System</td>
</tr>
<tr>
<td>erp:username</td>
<td>ERP system username</td>
</tr>
<tr>
<td>erp:password</td>
<td>ERP System password</td>
</tr>
</tbody>
</table>

Since we provide the end users with the opportunity to set their Salesforce login credentials via environment variables it becomes a hassle free task to point the Integration Application towards a completely different Salesforce Account. The environment variables can later be set in OpenShift when the application is deployed (see Deployment [section 4.3] and Figure 10).
**SalesforceCamelEndpointConfig**

The SalesforceCamelEndpointConfig class is only a few lines long. The only configurable option is to specify the Salesforce API version which at the time of writing is set to 33.0. The constructor returns a SalesforceCamelEndpointConfig Java object.

**SalesforceCamelComponent**

The SalesforceCamelComponent class instantiates the other two configuration classes mentioned in this section and sets them in its constructor. The component is then added to the CamelContext, which holds configuration for the application, on startup.

### 4.2.2 Camel Routes

As explained in section 3.3 IntegrationApplication we define routes in Apache Camel. Below follows an explanation of the classes in the Integration Application where these routes are implemented.

**StreamingRouteBuilder**

The StreamingRouteBuilder class contains all routes which receives updates from the Salesforce Streaming API. They are thus responsible for detecting changes in Salesforce and sending them to the ERP System.

```java
from("salesforce:CamelTopicAccount?" +
"notifyForFields=ALL&" +
"notifyForOperationCreate=true&" +
"notifyForOperationUpdate=true&" +
"notifyForOperationUndelete=true&" +
"notifyForOperationDelete=false&" +
"sObjectName=Account&" +
"updateTopic=true&" +
"sObject Query=SELECT Id ...FROM Account")

.log (...)

.convertBodyTo(String.class)

.to("jol:joltMapping/accountMappingToERP ?...")

.to("direct:setHeaders")

.to("http4://{{erp.domain}}/account/fromSalesforceCreateOrUpdate?
    authUsername={{erp.username}}&authPassword={{erp.password}}")

.to("direct:cleanUpHeaders")

.log (...)
```

Listing 15: A route receiving updates from the Salesforce Streaming API (parts of the code has been left out)
The route seen in the [Listing 10](#) code receives updates from Salesforce when an Account has been created or updated. Similar routes are implemented for Contacts and Cases respectively.

Line 1-8 specifies the routes subscription to the Streaming API. Note that all properties except `notifyForOperationDelete` are set to true. This means that the route will only be triggered when an Account has been created or updated but not upon deletion. The content of the notification is specified by the SOQL query at line 9. In the route listed above we ask for all fields present in the ERP system since we don’t know what has been altered, only that it has been altered.

Deletion of entities is handled by routes with minor differences to the one listed above. The only difference between the update/creation routes and the deletion ones are the `notifyForOperation` values (which are inverted) and the SOQL queries. The deletion handling routes will thus only be triggered upon deletion and will only receive the unique identifier for an entity since that is the only field needed to delete the entry from the database.

Before the data is sent to the ERP system endpoint the properties are remapped by JOLT at line 12 to match those of the receiving system. See [section 4.2.3](#) for more information about JOLT and the mapping of properties. There is also some altering of the headers done by the `setHeaders` and `cleanUpHeaders` before and after the data is sent to the ERP system at line 14. These routes are explained in [HelpersRouteBuilder](#) (section 4.2.2).
**ToSalesforceRouteBuilder**

The *ToSalesforceRouteBuilder* class is responsible for all data coming from the ERP system and into Salesforce. A typical use case may be a newly created Account in the ERP system which now has to be added in Salesforce. As explained in [section 3.3.1 Salesforce API](#), this requires us to make use of the Salesforce REST API.

```java
from("/contact")
  .log(...)  
  .convertBodyTo(String.class)
  .to("jolt:joltMapping/contactMappingToSalesforce?...")
  .unmarshal().json(Contact.class, String.class)
  .process(new NameConverterProcessor())
  .process(new ReferenceProcessor())
  .choice()
    .when(header("CamelHttpMethod").isEqualTo("POST"))
      .recipientList(simple("salesforce:createSObject?ObjectName=Contact"))
      .log(...)
    .endChoice()
    .when(header("CamelHttpMethod").isEqualTo("PUT"))
      .recipientList(simple("salesforce:updateSObject?ObjectName=Contact&$object.Id=${body.id}"))
      .log(...)
    .endChoice()
    .when(header("CamelHttpMethod").isEqualTo("DELETE"))
      .recipientList(simple("salesforce:deleteSObject?ObjectName=Contact&$object.Id=${body.id}"))
      .log(...);
```

Listing 16: A route handling Contact operations in the *ToSalesforceRouteBuilder* class (parts of the code has been left out)

Listing 16 shows a route which handles creation, update and deletion of Accounts. When any of these actions occur in the ERP system the Integration Application will receive a HTTP request at the `/contact` path which will trigger this route to execute (as seen on line 1). After logging of the request has been done on line 2 the JSON data in the HTTP body will be transformed and then converted to a Salesforce Contact object.

At line 6-7 the data is processed by two processors which makes it compatible with Salesforce (see [section 3.2.6](#)).

Next follows a choice operation on line 8 which is the Apache Camel equivalent of a switch-case
statement. The when operations on line 9, 13 and 17 checks whether the 'Camel.RequestMethod' header is a POST, PUT or DELETE. The path that the data will take thus depends on its content which in Apache Camel is known as conditional routing (see Figure 13). By using conditional routing we can keep the amount of routes to a minimum which leads to less, and thereby more maintainable, code.

![Figure 13: Conditional routing in Apache Camel. The message router directs the data based on its content. Image from Apache Camel documentation](image)

We implement similar routes for the other two entities: Contacts and Cases, which gives us a total amount of three routes in the ToSalesforceRouteBuilder class.

A different approach would be to create a route for each action (which in turn would have to be implemented for each entity) which would bring the total amount of routes in this class up to nine.
BulkRequestRouteBuilder

The BulkRequestRouteBuilder class contains the routes used for bulk fetching data from Salesforce into the ERP system. These routes are commonly used immediately after startup to perform an initial synchronization. All routes in this class make use of the Salesforce REST API.

```java
from("servlet:/getEntity")
  .to("direct:cleanUpHeaders")
  .choice()
    .when(header("dtoClass").isEqualTo("Account"))
      .to("direct:accountBulk")
    .when(header("dtoClass").isEqualTo("Contact"))
      .to("direct:contactBulk")
    .when(header("dtoClass").isEqualTo("Case"))
      .to("direct:caseBulk")
    .otherwise()
      .multicast()
      .to("direct:accountBulk", "direct:contactBulk", "direct:caseBulk");
```

Listing 17: The entity selection and the Account bulk fetching routes (parts of the code have been left out)

Just as in the ToSalesforceRouteBuilder class this one uses a choice operation. The first route just looks at the dtoClass header property (set in the ERP) to determine whether the request is for an Account, Contact or a Case and redirects the request to their respective fetching route. There is one bulk fetching route for each entity, all very similar which is why only the Account route is listed above. If none of the when statements evaluates to true the execution will reach the otherwise statement at line 10 and retrieve all Accounts, Contacts and Cases.
Figure 14: Visual representation of the route by Hawkio.

In Figure 14 is a visual representation of the route in Listing 17 by the monitoring tool Hawkio which was mentioned in Section 2.3.3. The green numbers to the right indicates how many times that part of the route has been used. In the example above we see that the route has processed two bulk requests: one to fetch all Accounts, to the left after the choice statement, and one to fetch all entities at once, to the far right.
### HelpersRouteBuilder

The `HelpersRouteBuilder` class contains two routes used for handling of HTTP headers which are used throughout the Integration Application.

```java
from("direct: setHeaders")
  .setHeader(
    Exchange.HTTP_METHOD,
    constant(HttpMethods.POST)
  )
  .setHeader(
    Exchange.CONTENT_TYPE,
    constant("application/json")
  )

from("direct: cleanUpHeaders")
  .removeHeaders(
    "x",
    "breadcrumbId | Content-Type | CamelHttpResponseCode | content-type | dtoClass"
  )
```

Listing 18: The `HelpersRouteBuilder` class

The first of the two routes seen in the listing code section adds two headers to the HTTP request. One sets the request method to POST (line 2-4) and the other states that the requests payload is provided as JSON (line 5-7).

The second route removes all headers other than those specified on line 12.

### 4.2.3 JOLT Mapping

Since the name of the properties in the JSON data that we retrieve from the Salesforce API isn’t the same as the ones needed for the ERP system they need to be mapped. E.g. the `Account.Id` property in Salesforce corresponds to the `Account.externalId` property in the ERP system. JOLT mapping is used to ensure that the property names are altered correctly before being sent to either system.
In listing 19 we see the Salesforce properties of the Account entity to the left and their corresponding properties in the ERP system to the right. This particular file is used when an Account goes from Salesforce into the ERP system.

Unfortunately, a JOLT file can’t be used both ways, neither can it be used when entities are being bulk fetched. Therefore each entity requires three different JOLT files: one from Salesforce to the ERP system, one from the ERP system to Salesforce and one used for bulk fetching. This gives us a total of nine JOLT files.

4.2.4 Processors

By creating so called processors we can perform more advanced operations on the data. A Camel processor is a class which implements the org.apache.camel.Processor interface which only contains one method: void process. A processor is always a part of a Camel route (see Figure 12). Our Integration Application contains three different processors.

ReferenceProcessor

Since Salesforce uses empty strings whereas the ERP system uses null values to indicate that there is no relationship, for instance between a Case and a Contact, this needs to be taken care of when data is transferred between the two. This is done in the ReferenceProcessor class. If it is a Case we are dealing with, for example, the transformation is just a simple if-statement checking whether the Case.getContactId property is null. If the Case.getContactId == null statement evaluates to true we just assign it an empty string (""").
NameConverterProcessor

The naming of Contacts is handled differently in Salesforce and the ERP system. As seen in Figure 7 in the Design of ERP System section the ERP system uses a single name field to store a Contacts name. Salesforce, on the other hand, uses a first name and a last name field which requires us to do some transformation on this property.

The NameConverterProcessor splits the name string at the first occurrence of a whitespace which is a fully acceptable solution in this case.

ClosedDateProcessor

Salesforce does not allow for the closed date of a Case to be set via the API. Therefore, if a Case property is coming from the ERP system, the dateTimeClosed field needs to be nulled before being sent to Salesforce. This is done by the ClosedDateProcessor class by calling the Case.setClosedDate method with a null parameter. The closedDate will still be set by Salesforce, since the Status of the Case is still CLOSED.

4.3 Deployment

As explained briefly in the Connectivity Engine background section applications are being deployed using the Docker container technology which ensures isolation between applications and improves scalability. A Docker container in its simplest form is a minimal Linux system containing just the tools needed for an application to function and provides it with its own file system structure and network interfaces. Applications running in such containers may only communicate with each other over a network.

The OpenShift Origin software made by the American company Red Hat is used for the management of docker containers [55]. It is able to pull and build the latest committed code from a Git repository and deploy it automatically with as little downtime as possible. OpenShift in turn is built on top of the open source Kubernetes project originally founded by Google [52]. What OpenShift provides on top of Kubernetes is a graphical management tool accessed through a web browser along with tools used for monitoring and continuous integration (See Figure 13). It is also possible to set environment variables via the web interface as seen in Figure 16.

Below follows an explanation of the various parts of Kubernetes which also applies to OpenShift.
Figure 15: Screenshot of the OpenShift management application.
OpenShift works with pods which is a set of one or more containers running in a shared context. The set of containers within a pod shares the same IP address and port space while still being isolated from each other, they may find each other and communicate via localhost. They also shares storage volumes.

Figure 16: OpenShift environment variables

Figure 17: Kubernetes/OpenShift containers and pods. Image from the official Kubernetes documentation [3].
The official Kubernetes documentation which also applies to OpenShift clearly states when containers should be coupled together in a pod and when they should not [1]:

*Containers should only be scheduled together in a single Pod if they are tightly coupled and need to share resources such as disk.*

One prime example of coupling would be a database and the application using it.

Pods, in turn, runs on *nodes* which simply is a machine, may it be virtual or physical. Multiple nodes can be provided to create a cluster of nodes. A single node can have multiple pods. When a cluster of nodes is present, Kubernetes will automatically schedule the pods across the nodes and thus provide load balancing. The management of pods is handled by the Master (see Figure 18).

![Figure 18: OpenShift overview with nodes, pods and a master present](image)

For our ERP system we deploy a pod consisting of two containers with one being the system itself hosted on a Tomcat web server (named demoerp) and the other a MariaDB database back end (named demoerpdb). See Figure 19 for a graphic view.

The Integration Application runs in a single container in its own pod since it doesn’t need to share resources with the ERP system.

---

5Figure provided by Redpill Linpro.
4.4 Use Cases

Use case 1 and 2 will be explained with the assumption that all data has been synced from Salesforce beforehand, while for use case 3 all Contacts and Cases have been synced.

4.4.1 Use Case 1 - Creating a New Contact on Salesforce

When creating a new Contact on Salesforce there are around 25 fields available, with "Last Name" being the only required field. In this case we choose to enter information as specified in **Listing 20**

| First Name: Gustavo | Last Name: Fring | Account Name: Los Pollos Hermanos | Title: CEO | Fax: (070) 412 3456 |

Listing 20: Contact Information

On saving the Contact, the Streaming API listening to Contact changes in `StreamingRouteBuilder` will trigger. The API only collects the fields interesting to the ERP system, in this case meaning the "Fax" field is simply ignored. The hidden `id` field is also collected along with `accountId` which is set to the id of "Los Pollos Hermanos" by Salesforce. Which fields are to be collected is configured by us.

The incoming Contact is logged as an incoming request and the the body is converted to a string the same way as seen on line 11 in **Listing 19** When converted it is passed to a JOLT endpoint with the mapping file for Contact, which will convert the property names to match those of the ERP system. Next, the headers for `HttpMethod` and `Content-Type` are set to `POST` and `application/json` respectively.

The Contact is now transformed and the http request configured, thus ready to be sent to the ERP system. This works the same way as shown on line 14 in **Listing 15** but with "/contact" instead of "/account". The username and password are set in `application.yml` to the credentials needed to authenticate against the ERP system. Finally, the response headers are cleaned up using the `cleanUpHeaders` route in **Listing 15** and the confirmation that the Contact has been received is logged.

The JSON body is received and mapped in `ContactController` in the same manner as in **Listing 8**. The Contact is now logged in the ERP before it is evaluated if it already exists in the database. This works just as in **Listing 8** but with the `contactRepository`. Since it is a new Contact it will not exist in the ERP system database and the evaluation is false.

In the next step where we check if any Cases should be connected to the Contact. This evaluation will also yield false, since the Contact is brand new.
Now the possible connection to an Account is checked. The statement on line 1 in listing 5 will pass since the Contact has been connected to an Account. "Los Pollos Hermanos" is now attempted to be retrieved, as seen on line 2. The attempt will succeed since the Account exists and the Contact’s Account is set (line 5). Finally the Contact is saved and given a local id by JPA.

4.4.2 Use Case 2 - Edit a Case in the ERP System

Below follows the information edited for a typical Case in the ERP system (Table 2).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Current Value</th>
<th>New Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject</td>
<td>Electrical circuit malfunctioning</td>
<td>NC</td>
</tr>
<tr>
<td>Type</td>
<td>Electrical</td>
<td>NC</td>
</tr>
<tr>
<td>Reason</td>
<td>Breakdown</td>
<td>NC</td>
</tr>
<tr>
<td>Priority</td>
<td>Medium</td>
<td>NC</td>
</tr>
<tr>
<td>Status</td>
<td>Working</td>
<td>Closed</td>
</tr>
<tr>
<td>Contact</td>
<td>Josh Davis</td>
<td>Tom Ripley</td>
</tr>
<tr>
<td>Account</td>
<td>Express Logistics and Transportation</td>
<td>GenePoint</td>
</tr>
</tbody>
</table>

These changes does not contain any bindingResult errors and thus will be allowed when the user presses submit. The new connections are set for Account and Contact and the accountId and contactId are updated to their respective ids. The status also changed to closed - automatically setting a dateTimeClosed for the Case. The Case is saved locally and passed to sendRequestToSalesforce (see the Single Entity Operation paragraph) with the Case and "edit" as arguments.

The headers for Content-Type and Authorization are set and added to a new HttpHeaders along with the Case. The url is set to inDomain, as explained in Single Entity Operation followed by "case". Next, the switch statement evaluates the operation which is "edit", meaning a put (line 2, listing 13) request will be sent. The request is then logged.

The route consuming from "servlet:/case" in the Integration Application, similar to the one in listing 10 will receive the Case, log it, and start to transform and process it. The body is converted to a string and then passed to a JOLT endpoint for property transformation. When it is transformed to match the Case dto object’s properties it is unmarshalled from a JSON string to a Case object (line 5) with the use of Jackson (section 22.3.1). The object is now processed by two processors - ClosedDateProcessor and ReferenceProcessor. The first one will still do its job as mentioned in the section 22.3 while the second one will have no effect since neither accountId or contactId is null.

The closedDate will still be set to the same one as in the ERP system by Salesforce, as the status of the Case is still CLOSED.

47
The Case route also has a choice evaluation based on the `HTTPMethod` header, same as the one on line 8-19 in Listing 13. In our case the header is `PUT` meaning we will send an update request to Salesforce just as on line 14, but with Case as `ObjectName`. Finally, the response is logged.

4.4.3 Use Case 3 - Sync All Accounts from Salesforce

When executing a bulk request for Account the url is set to `/aDomain` followed by "getEntity" and the headers – `Authorization` and `dtoClass` – are set to the needed credentials and "account" respectively. The request is then logged and an exchange like the one in Listing 13 is sent.

The route shown in Listing 14 receives the request and directs it to `cleanUpHeaders` (line 9, Listing 18) which removes unnecessary headers. The request is then process by a choice with the `dtoClass` header as evaluation factor. In our case the header is equal to "account" and the request will be directed to `accountBulk` which is also depicted in Listing 17. Here, the incoming request is logged before a query is made to Salesforce for all available Accounts. The response is converted to a JSON string before being remapped with JOLT to fit the ERP system properties (line 18-19). We then set the headers exactly the same as in the streaming API route and proceed to send the data to the url mapped to handle bulk inserts of Accounts, described in paragraph 4.1.2.2 `Bulk Insert`. Finally the response headers are cleaned up and the response is logged.

In the ERP system the method in Listing 15 receives the list of JSON objects and automatically maps them to `AccountBatch`’s list of Accounts called `records`. After the received request is logged, the list is looped through and each object is handled with `fromSalesforceCreateOrUpdate` (Listing 6). The process is described in paragraph 4.1.2.3 and works the same way as when a single entity is processed. All the Accounts are now inserted and their connections are set.

4.5 Choice of Tools and Frameworks

The choice of some of the tools used in the project has been made on the recommendation of the technical supervisor at Redpill Linpro. This includes the choice of the Spring framework as well as Apache Camel since they’re both frequently used in Redpill Linpro’s daily business. An alternative approach would have been to use the integration solution provided by MuleSoft instead of Apache Camel [32]. The major drawbacks is, however, that the free community version lacks many of the features available in Apache Camel and that the premium version is expensive.

Regarding the choice of Spring MVC the first approach was to use Apache Struts which is another MVC framework for the Java platform [33]. However, when educated about Spring the decision to drop Struts in favor of Spring’s own MVC framework was quickly made. The framework enabled a more homogeneous design. Struts was really only considered before the development process began.

The choice of the MariaDB database was purely made on earlier experience although other
alternatives like MySQL or PostgreSQL probably would have worked just as well. The H2 memory database was used locally during development because of it being lightweight and fast thanks to it’s small footprint. It also run without any configuration when running in-memory as opposed to a fully fledged relational database such as MariaDB.

Some choices of tools have been made as the result of research or try and error often with suggestions from our technical supervisor. This includes JOLT and Jackson used in the Integration Application for the transformation of data. Although other tools may exist no alternatives where further explored since both JOLT and Jackson works satisfactorily.

The same thing applies to Spring’s RestTemplate, which turned out to be more than adequate for our requirements after trying it out.

4.6 Summary

The implementation section has given a detailed explanation of the various components that make up the ERP system as well as the Integration Application.

As for the ERP system, explanations have been given on how it implements and communicates with a database and how it exchanges data with Salesforce via the Integration Application endpoints. Explanations have also been given on how the ERP system works internally when entities are listed, created, edited and deleted.

Regarding the Integration Application the configuration and usage of the Apache Camel Salesforce component has been described as well as the various Camel routes used to accomplish the integration. The usage of JOLT mapping files, Jackson marshalling and unmarshalling and custom processors has also been explained. It has also been shown how the applications are deployed to the OpenShift infrastructure and how they are operated in this environment.

Moreover, three different use cases have been given to show how the different parts work in conjunction. The first one covers the entire process for when a Contact is created on Salesforce – the trigger of the streaming API, the transformation and ERP system’s handling of the Contact. In the second scenario a Case is edited in the ERP system and the whole chain of events until it is updated on Salesforce is described. The last use case depicts the process of bulk fetching all Accounts from Salesforce and adding them to the ERP system.

Finally, a motivation has been given regarding the choice of the tools used in the project and alternatives to these have been discussed.
5 Results and Evaluation

In this section we discuss the results of the project as a whole and evaluate both of the applications. The results and evaluations are matched against the requirements specified by Redpil Linpro.

5.1 Project Results

The results of this project are two applications hosted in the cloud. One is a simple ERP system implemented in the Java programming language as a web application following the MVC architectural pattern. The other is an associated Integration Application implemented using the Apache Camel integration framework which allows for the ERP system to be integrated with Salesforce. Both applications are hosted on OpenShift.

All the functionality requested by Redpil Linpro for the ERP system and the Integration Application, including additional stretch goal features, have been implemented. Documentation for the applications in both a more technical form and documentation for functionality and usage directed to sales representatives have been completed as well.

During the development phase of these two applications the philosophies of continuous integrations and DevOps have been practiced. This modern approach to software development and the technologies used to implement it are explained in section 2.2. A more detailed explanation of the environment and infrastructure used for deployment is described in section 4.3.

Extensive usage of open source technologies is a common denominator throughout the project since it is a cornerstone and a business idea at Redpil Linpro. This project has shown that various open source technologies, when used together, is a suitable approach for building web applications as well as integration solutions which meets the requirements of the industry.

Documentation has been written in the form of a generic user guide as well as more technically oriented markdown files. All documentation has been done after the finalization of the project. The user guide is aimed towards salespeople at Redpil Linpro and provides them with step-by-step instructions on how to use the systems and parts that may be of interest during a demo with potential customers. The markdown files contains all technical information necessary to setup the ERP system as well as the Integration Application. Information on various options that needs to be configured in Salesforce is also provided.

5.2 Evaluation of the ERP system

As explained in section 1.1, Redpil Linpro lacks a way to demonstrate the possibilities of its integration platform Connectivity Engine to potential customers. A quite simple but functional web application which could be integrated with the Salesforce CRM system via Connectivity Engine was requested.

The ERP system, which is a result of this project, provides most of the operations present in Salesforce and is able to clearly show that integration with Salesforce by using Connectivity
Engine works and is possible, as was the goal stated in Section 2.3.1.

The Spring MVC framework has proven to be a mature and fully functional framework for developing a web application following the well-proven MVC architectural pattern. When used in conjunction with the other technologies mentioned in Section 2.2.3 the process of developing web applications for the Java platform becomes a fairly hassle-free task fully on par with other frameworks and languages.

Although being limited in amount of features, the ERP system fulfills its requirements, as requested by Redpill Linpro, and provides them with an easy to understand way to demonstrate the capabilities of Connectivity Engine.

5.3 Evaluation of the Integration Application

The Integration Application, which implements the Apache Camel integration framework, is the most essential part of the project. This is where all transformation between the two systems, our ERP and Salesforce, is being made. By communicating with the endpoints of these systems, and performing transformation of data, it ensures that data can be synchronized between the two.

The Integration Application is able to perform two-way integration between the two systems using the Apache Camel framework with the additions described in Paragraph 2.2.3.1. This setup provides solid foundations for transformation of data. Integration towards Salesforce is also simplified since a dedicated component for Apache Camel is available which is able to seamlessly interact with the Salesforce APIs.

As newcomers to Apache Camel as well as system integration as a whole we’ve proven that integration is a fairly simple task to get started with. The fact that more Software-as-a-Service (SaaS) providers such as Salesforce allows for developers to access their services through APIs also helps [32].
6 Conclusion

This section covers general conclusions, problems encountered as well as gives our final remarks on the projects.

6.1 Project Conclusion

By carrying out this project we’ve provided Redpill Linpro with a way to demonstrate the possibilities of their iPaaS solution Connectivity Engine to their potential customers which, hopefully, will increase interest in their product. In return we got to experience some of the most current tools and techniques used in the fast-changing world of open source, APIs and system integration. Not only did we get to experience the pure programming part of a modern developer’s duties but also the philosophies of DevOps with increased focus on the deployment and hosting of software. Furthermore we’ve received great lessons regarding good practices on coding and documentation which is valuable for every developer.

6.2 Problems

6.2.1 Streaming API Random Disconnect

During long connection times to Salesforce, the streaming API subscriptions can sometimes disconnect. On these occasions there is a reconnect handshake in place designed to resubscribe to the topic, this can however fail. The only exception presented on these occurrences is Error subscribing to CamelTopic<topic name>: null and is a problem which is out of our hands to fix.

This becomes an issue when the Integration Application is hosted on OpenShift since it runs continuously for an indefinite amount of time. To work around the problem the best practice as of now is simply to reboot the application before a demo, which reboots the subscriptions as well. The issue of subscription connection failure does not present itself for several hours after the initial subscription, meaning a reboot will suffice for conducting a demo.

6.2.2 Table Name With MariaDB

When rebuilding the ERP system on OpenShift after the MariaDB instance was added to the docker, the pod simply failed to start with the new build. The build went through without issues but when the pod was supposed to start up it failed without error messages. When running the application locally with H2 everything worked smoothly.

The reason was a conflict with table name restrictions which MariaDB implements. The entity Case naturally generates a table with the name case. This is however a reserved word in MariaDB, causing the generation to fail. To fix the issue the name for the table was set manually using @Table(name = "tCase").
6.2.3 Testing

The automatic testing of the applications have been lacking to say the least. Going into the project the focus was on implementing the wanted functionality for both applications and testing got put aside. Testing has still been done manually, quite extensively, but this does not cover for the fact that automatic tests are needed.

The test writing has been picked up more during the last few weeks of the project, but tests should preferably have been written during the entire development process and not just after the application is finished. It is hard to say if this affected the quality of the applications, but continuous automatic testing would definitely have eased the development process.

6.3 Final Remarks

It is unfortunate that a bug in the Salesforce Streaming API makes the product a bit more high maintenance since it is out of our hands to fix. The workaround described in Streaming API Random Disconnect (Section 6.2.1) should however suffice for future usage of the product.

Overall we are still satisfied with the result of the project and the entire work process. Redpill Linpro received us well and gave us great conditions for developing a solid final product. Working with so many new and interesting techniques and frameworks have proven inspiring for the future as well as improved our knowledge by a great amount.
References


Appendix A  Database tables and enums

A.1 Account

A.1.1 Account table

Table 3: Data types for the Account table

<table>
<thead>
<tr>
<th>Field</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCOUNT_ID</td>
<td>Long</td>
</tr>
<tr>
<td>EXTERNAL_ID</td>
<td>String</td>
</tr>
<tr>
<td>NAME</td>
<td>String</td>
</tr>
<tr>
<td>ACCOUNT_NUMBER</td>
<td>String</td>
</tr>
<tr>
<td>CITY</td>
<td>String</td>
</tr>
<tr>
<td>PHONE_NUMBER</td>
<td>String</td>
</tr>
<tr>
<td>WEBSITE</td>
<td>String</td>
</tr>
<tr>
<td>OWNERSHIP</td>
<td>Enum</td>
</tr>
<tr>
<td>TYPE</td>
<td>Enum</td>
</tr>
<tr>
<td>EMPLOYEES</td>
<td>Integer</td>
</tr>
<tr>
<td>REVENUE</td>
<td>Long</td>
</tr>
</tbody>
</table>

A.1.2 Account enumerators

The OWNERSHIP and TYPE enums consists of the following enumerators:

Table 4: The TYPE enumerators

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Mapped value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TECHNOLOGY_PARTNER</td>
<td>Technology Partner</td>
</tr>
<tr>
<td>INSTALLATION_PARTNER</td>
<td>Installation Partner</td>
</tr>
<tr>
<td>CUSTOMER__DIRECT</td>
<td>Customer - Direct</td>
</tr>
<tr>
<td>CUSTOMER__CHANNEL</td>
<td>Customer - Channel</td>
</tr>
<tr>
<td>CHANNEL_PARTNER__RESELLER</td>
<td>Channel Partner / Reseller</td>
</tr>
<tr>
<td>PROSPECT</td>
<td>Prospect</td>
</tr>
</tbody>
</table>

Table 5: The OWNERSHIP enumerators

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Mapped value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUBLIC</td>
<td>Public</td>
</tr>
<tr>
<td>PRIVATE</td>
<td>Private</td>
</tr>
<tr>
<td>SUBSIDIARY</td>
<td>Subsidiary</td>
</tr>
<tr>
<td>OTHER</td>
<td>Other</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>Unknown</td>
</tr>
</tbody>
</table>
A.2 Contact

A.2.1 Contact table

Table 6: Data types for the Contact table

<table>
<thead>
<tr>
<th>Field</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTACT_ID</td>
<td>Long</td>
</tr>
<tr>
<td>EXTERNAL_ID</td>
<td>String</td>
</tr>
<tr>
<td>ACCOUNT_ID</td>
<td>String</td>
</tr>
<tr>
<td>NAME</td>
<td>String</td>
</tr>
<tr>
<td>TITLE</td>
<td>String</td>
</tr>
<tr>
<td>DOB</td>
<td>Date</td>
</tr>
<tr>
<td>DEPARTMENT</td>
<td>String</td>
</tr>
<tr>
<td>EMAIL</td>
<td>String</td>
</tr>
<tr>
<td>PHONE_NUMBER</td>
<td>String</td>
</tr>
</tbody>
</table>

A.3 Case

A.3.1 Case table

Table 7: Data types for the Case table

<table>
<thead>
<tr>
<th>Field</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASE_ID</td>
<td>Long</td>
</tr>
<tr>
<td>EXTERNAL_ID</td>
<td>String</td>
</tr>
<tr>
<td>ACCOUNT_ID</td>
<td>String</td>
</tr>
<tr>
<td>CONTACT_ID</td>
<td>String</td>
</tr>
<tr>
<td>DATE_TIME_CLOSED</td>
<td>Date</td>
</tr>
<tr>
<td>SUBJECT</td>
<td>String</td>
</tr>
<tr>
<td>TYPE</td>
<td>Enum</td>
</tr>
<tr>
<td>REASON</td>
<td>Enum</td>
</tr>
<tr>
<td>STATUS</td>
<td>Enum</td>
</tr>
<tr>
<td>PRIORITY</td>
<td>Enum</td>
</tr>
</tbody>
</table>

A.3.2 Case enumerators

The TYPE, REASON, STATUS and PRIORITY enums consists of the following enumerators:
Table 8: The **TYPE** enumerators

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Mapped value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELECTRICAL</td>
<td>Electrical</td>
</tr>
<tr>
<td>ELECTRONIC</td>
<td>Electronic</td>
</tr>
<tr>
<td>MECHANICAL</td>
<td>Mechanical</td>
</tr>
<tr>
<td>STRUCTURAL</td>
<td>Structural</td>
</tr>
<tr>
<td>OTHER</td>
<td>Other</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

Table 9: The **REASON** enumerators

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Mapped value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BREAKDOWN</td>
<td>Breakdown</td>
</tr>
<tr>
<td>EQUIPMENT_COMPLEXITY</td>
<td>Equipment Complexity</td>
</tr>
<tr>
<td>EQUIPMENT_DESIGN</td>
<td>Equipment Design</td>
</tr>
<tr>
<td>FEEDBACK</td>
<td>Feedback</td>
</tr>
<tr>
<td>INSTALLATION</td>
<td>Installation</td>
</tr>
<tr>
<td>PERFORMANCE</td>
<td>Performance</td>
</tr>
<tr>
<td>OTHER</td>
<td>Other</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

Table 10: The **STATUS** enumerators

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Mapped value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEW</td>
<td>New</td>
</tr>
<tr>
<td>WORKING</td>
<td>Working</td>
</tr>
<tr>
<td>ESCALATED</td>
<td>Escalated</td>
</tr>
<tr>
<td>CLOSED</td>
<td>Closed</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

Table 11: The **PRIORITY** enumerators

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Mapped value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HIGH</td>
<td>High</td>
</tr>
<tr>
<td>MEDIUM</td>
<td>Medium</td>
</tr>
<tr>
<td>LOW</td>
<td>Low</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>Unknown</td>
</tr>
</tbody>
</table>
Appendix B  UML Diagrams

B.1  Integration Application
B.2 ERP System