Degree Thesis, Åland University of Applied Sciences, Degree Programme in Information Technology

# DEVELOPMENT OF A BACK OFFICE CUSTOMER INFO APPLICATION

Mathias Brandtberg, William Söderlund

40:2016

Publishing date: 19.12.2016
Supervisor: Joakim Isaksson

# EXAMENSARBETE
# Högskolan på Åland

| Utbildningsprogram: | Informationsteknik |
|---|---|
| Författare: | Mathias Brandtberg, William Söderlund |
| Arbetets namn: | Development of a Back Office Customer Information Application |
| Handledare: | Joakim Isaksson |
| Uppdragsgivare: | Crosskey Banking Solutions |

**Abstrakt**

Denna uppsats fungerar som dokumentation för utvecklingen av en backofficeapplikation gjord för Crosskey Banking Solutions.
Uppgiften var att skapa en ny applikation som visade tidigare dold information om slutkunden.

Applikationen är skriven i Java och använder teknologier som Spring, Javascript och JSP. Applikationen följer strukturmässigt Domain Driven Design (DDD) mönstret och använder sig av Model View Controller (MVC) och Inversion of Control (IoC).

Resultatet var en fullt funktionerande applikation där backofficeanvändare kan söka efter slutkunder och se detaljer om deras profil.

**Nyckelord (sökord)**

Spring, Java, MVC, Domain Driven Design

| Högskolans serienummer: | ISSN: | Språk: | Sidantal: |
|---|---|---|---|
| 40:2016 | 1458-1531 | Engelska | 32 sidor |

| Inlämningsdatum: | Presentationsdatum: | Datum för godkännande: |
|---|---|---|
| 19.12.2016 | 02.12.2016 | 19.12.2016 |

# DEGREE THESIS
# Åland University of Applied Sciences

| | |
|---|---|
| **Study program:** | Information Technology |
| **Author:** | Mathias Brandtberg, William Söderlund |
| **Title:** | Development of a Back Office Customer Information Application |
| **Academic Supervisor:** | Joakim Isaksson |
| **Technical Supervisor:** | Crosskey Banking Solutions |

**Abstract**

This thesis serves as a documentation of the development of a backoffice application made for Crosskey Banking Solutions.
The task was to create a new application that provided previously hidden information about end users.

The application was written in Java and uses technologies like Spring, Javascript and JSP. Architecturally the application follows the Domain Driven Design (DDD) pattern and applies design principles like Model View Controller (MVC) and Inversion of Control (IoC).

The result was a fully functional application where backoffice users can search for end users and see details about their profile.

**Keywords**

Spring, Java, MVC, Domain Driven Design

| Serial number: | ISSN: | Language: | Number of pages: |
|---|---|---|---|
| 40:2016 | 1458-1531 | English | 32 pages |

| Handed in: | Date of presentation: | Approved on: |
|---|---|---|
| 19.12.2016 | 02.12.2016 | 19.12.2016 |

# Table of contents


**1. INTRODUCTION** **5**
1.1 Purpose 5
1.2 Method 5
1.3 Limitations 6
1.4 Definitions 7
**2. ARCHITECTURAL PATTERNS AND PROCESSES** **8**
2.1 Domain driven design 8
2.2 Facade pattern 10
2.3 Model view controller 11
2.4 Data Transfer Object 12
2.5 Inversion of Control 12
2.6 Dependency Injection 12
**3. FRAMEWORKS** **13**
3.1 Java EE 13
3.2 Spring 13
3.3 Tiles 15
3.4 iBATIS 16
3.4.1 Typehandlers 17
**4. DEVELOPMENT OF APPLICATION** **19**
4.1 Specification 19
4.2 Preparation 19
4.3 Development 20
4.3.1 Successful logins 22
4.3.2 Balance alerts 24
4.3.3 Payee register 26
4.3.4 Services 26
4.3.5 Account settings 27
4.3.6 User settings 27
4.3.7 Failed logins 28
4.3.8 Validation 28
4.3.9 Error and info handling 29
4.4 Configuring the application for all customers 29

4.5 User limitations 30

**5 CONCLUSION** **30**

5.1 Result 30

5.2 Reflections 31

**REFERENCES** **32**

# 1. INTRODUCTION

## 1.1 Purpose

The purpose of this thesis is to describe the development process of an application for a back office system. This paper will bring forth details and any issues encountered during the development and planning phases.

The back office system consists of many sub applications and is used to administrate Internet banks. The new development will feature an interface for the application user to be able to search for end users in the Internet bank and to view different details such as login information.

This application was commissioned by our external supervisor Crosskey Banking Solutions in order to help their employees to solve customer specific cases. Crosskey is a Finnish company that develops and maintains systems for Nordic banks and capital markets. They provide services such as internet banking and card management systems to their customers (Crosskey, 2016).

Before this application was created the customers had to ask Crosskey for information that is now displayed in a web interface. This will enable the customers to get the information they want faster and easier. Also, Crosskey's employees do not have to bother with fetching the information and can instead focus on other tasks.

## 1.2 Method

The first step was to research the the already existing back office system. Research was done by analysing the code of other applications that is structurally and functionally similar to the new application. Analysing the code provided ideas on the structure of the new application, and showed if any code that could be reused.

The development was done in Java using frameworks such as the JSP standard tag library (JSTL)(Walls, 2015) to create the view and iBatis (Begin, Goodin, & Meadors, 2007) to fetch the data. Spring (Pivotal Software, 2016) was used to control the web flow and handle security.

The implementation also used Inversion of Control (IoC), in the form of Dependency Injection (DI) to handle dependencies. The structure of the application followed the Domain Driven Design (DDD) software development concept.The design of the user interface followed the specifications provided by the external supervisor from the start but were changed later on.

## 1.3 Limitations

As the project is part of the back office system, it’s crucial to use the same frameworks and libraries as the base system as long as possible. Because of this, the planning phase did not examine what frameworks to use and just settle with the frameworks used in the other sub applications.

The application uses a persistence framework called iBatis that is no longer actively supported. This is not ideal, but development is simply stuck with the older framework instead of the currently supported framework MyBatis. As data tables are already in place for the needed data no database design was done during the development.

Dependency configuration and Spring configuration is already fixed for the application.
As the application does not have a mobile view the user interface does not need a separate mobile view.

## 1.4 Definitions

Here are a short description of common terms and abbreviations that are used throughout this paper.

- **Backoffice user:** Bank personnel that has access to the back office application.
- **Backoffice:** Application used to administrate Internet banks.
- **CLOB:** Character Large Object. An object containing many characters in a database.
- **Customer:** Crosskey's customer, a banking company that uses Crosskey's services.
- **End user:** The customer's customer, internet bank user, has access to the services the bank provides.
- **Enterprise JavaBeans:** A server-side Java component that encapsulates business logic of an application.
- **iBATIS:** A persistence framework for mapping SQL data to objects.
- **JIRA:** Software used to track workflow for software development teams.
- **JSON:** Javascript Object Notation. A syntax used to store and transmit data.
- **JSP:** JavaServer Pages is a technology used to develop dynamically generated web pages.
- **JSTL:** JavaServer Pages Standard Tag Library extends the JSP functionality by adding tags for tasks like iteration and conditions.
- **LDAP:** Lightweight Directory Access Protocol (LDAP) is a protocol used to permit access to directories on a server.
- **MyBatis:** Persistence framework, predecessor to iBATIS.
- **Spring:** An application framework used for Java. It provides loads of useful functionality that helps development.
- **XML:** Extensible Markup Language is a universal markup language that defines the encoding rules for a document that is both machine and human readable

# 2. ARCHITECTURAL PATTERNS AND PROCESSES

## 2.1 Domain driven design

Following the Crosskey company standard, the application will be developed using the Domain Driven Design (DDD) software architecture (Evans, 2015). DDD provides ideas for developers on how to create applications that are easily extendable and follow a coherent structure. The term “supple design” is used to describe code that is extendable, elastic and is often mentioned along with DDD.

This may seem like an odd choice for an application that is supposed to simply show data. Before starting a project like this you can not know what issues you might encounter, so it’s a good idea to choose a solid architectural approach to be able to deal with them. A DDD application is built of four core layers (figure 1):

- **Interface layer.** Used to present information to the user and reading input. This layer should only contain very lightweight logic related to the data presentation. No domain objects should be accessible here, instead DTOs (3.4 Data Transfer Object) are used as data holders.

- **Application layer.** This is the functionality of the application that is exposed to an interface to use. It should be lightweight and mostly act as an controller for the underlying domain and infrastructure layers. A routine like bookTicket(...) would be located in this layer, which involves calling operations on several domain and infrastructure objects. This application will not contain functionality like this so the layer will remain empty for now.

- **Domain Layer.** This is where all the business logic and rules are. The domain describes the area of business the application is supposed to work in. For instance, a

domain for a shipping company would contain models for cargo, ships and so on. In this layer we also find the model, representing objects designed to solve domain related problems. There are also services located here that work as a layer in between the interface and infrastructure layers for domain object retrieval. This is the core of the application and should not be visible to the interface layer.

- **Infrastructure layer.** Everything that is independent of the application, like an application server or a database service are in this layer. Persistence and retrieval of domain objects are done in this layer with the help of repository interfaces. This is where the Ibatis integration is applied in the new application.

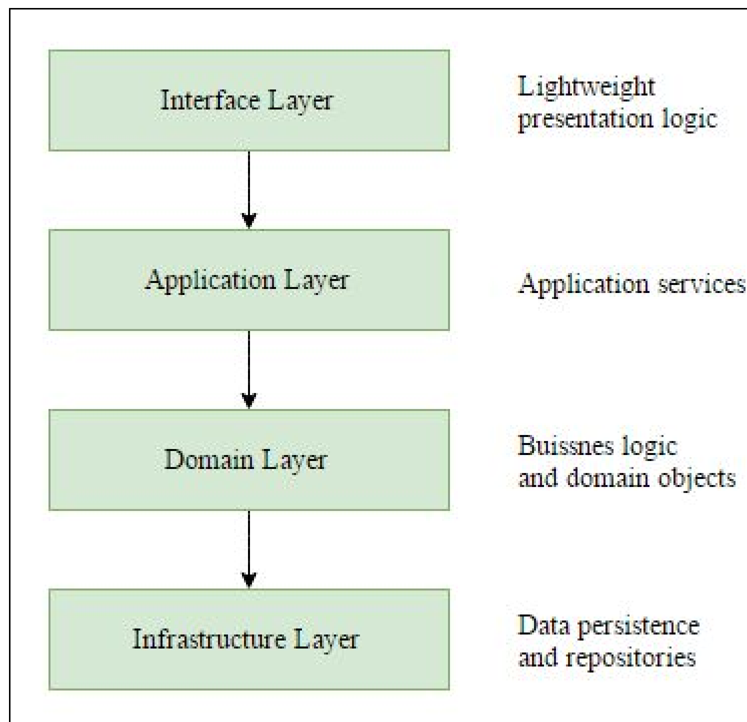


*Figure 1. How the Domain driven design layers interact with each other*

## 2.2 Facade pattern

The facade (Yener & Theedom, 2014) is a part of the interface layer and works with the controller. All services created in the domain service gets injected into the facade, which provides a top layer interface for all domain services. This is also where the logging is triggered if anything goes wrong with data retrieval. The DTOs (see 3.4) are also created in this layer by a mapper that maps domain objects into their DTO counterparts. Figure 2 shows the final package structure of the application. The controller is located under the web package.

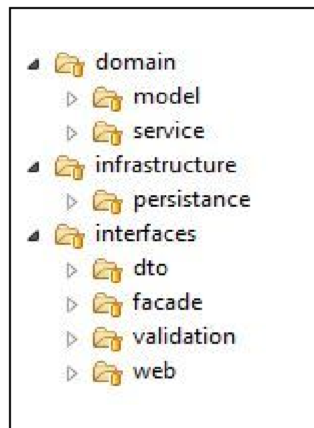


*Figure 2. Package structure of the application*

## 2.3 Model view controller

Model view controller (MVC) (Yener & Theedom, 2014) is a design pattern used to separate the data model from the presentation layer (figure 3). This makes changes to the presentation layer easy, without having to change the model.

The model or "domain layer" represents business logic layer. Here you can find business related functionality. For example, this is where the stock decreases when buying products from a web store.

The view displays the data managed in the domain layer. This layer should only contain data formatting related to the display of data and overall be lightweight. The last part is the controller. Here user input gets managed and sent to the model as commands describing what the model should retrieve. Changes in the model also changes the view.

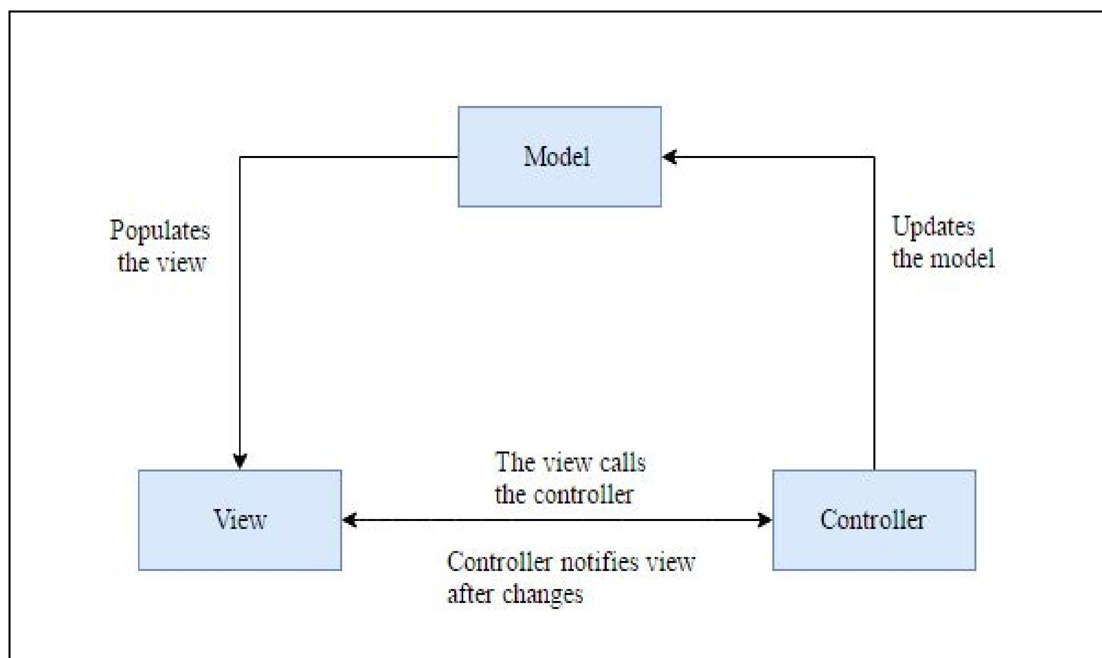


*Figure 3. The model view controller structure*

## 2.4 Data Transfer Object

A Data Transfer Object (DTO) (Martin Fowler, 2002) is a lightweight representation of a domain object. It is used to restrict access to domain objects in the interface layer. Mapping classes are used to map fields from the model into their DTO counterpart. DTO objects should not inherit any behaviour from the domain object. It serves only as a container for the data that should be displayed.

## 2.5 Inversion of Control

Inversion of Control (IoC) (Yener & Theedom, 2014) is a design pattern that handles dependencies for an object. This means that the class does not know about the implementation, only about the interface that the implementation is supposed to fulfill. This makes the implementation decoupled from the functionality and can easily be replaced by other implementations.

## 2.6 Dependency Injection

Dependency Injection (DI) (Yener & Theedom, 2014) is a form of Inversion of Control. Instead of having the objects creating its own dependencies the dependencies are injected through the constructor or via property setters. This makes the application loosely coupled and by injecting different implementations the behaviour of the application changes.

An example of this would be the SSN strategy implementations in this application. The format of the SSN were different for some customers, so separate implementations were required to handle the difference properly.

# 3. FRAMEWORKS

## 3.1 Java EE

Java EE (Java Enterprise Edition)(Oracle, 2016) is a collection of APIs and other technologies for the Java platform to help create large enterprise applications. An enterprise application is a software that is often hosted on a server and provides a business with its mission-critical applications.
Common technologies that makes up Java EE are JSP, JSTL, etc. A brief description of these technologies can be found in chapter 1.4.

## 3.2 Spring

Spring is an Open Source framework and Inversion of Control container for Java. It is often used together with Java EE as an addition or a replacement to Enterprise JavaBeans. In this application Spring is used instead of Enterprise JavaBeans to create beans that are defined in an XML document and later used to make dependency injections. A Spring bean is a POJO (Plain Old Java Object) that is managed in a Spring container (Walls, 2015).

An example of a bean definition can be found in Figure 4. The `constructor-arg` tag defines the implementations that the constructor of the `class` takes in as parameters. The `index` defines what position the constructor argument should use. The `ref` property indicates what bean to inject into the class.

Spring features a framework for handling the MVC architecture (Figure 3) called Spring MVC (Walls, 2015). This framework provides an annotation to define controller classes and a model to transfer data to the view.

```xml
<bean id="inet.customerinfo.backoffice.CustomerInfoFacade"
    class="fi.crosskey.inet.customerinfo.backoffice.interfaces.facade.impl.DefaultCustomerInfoFacade">
    <constructor-arg index="0" ref="inet.customerinfo.backoffice.CustomerService"/>
    <constructor-arg index="1" ref="inet.customerinfo.backoffice.LoginService"/>
    <constructor-arg index="2" ref="inet.customerinfo.backoffice.BalanceNotificationService"/>
    <constructor-arg index="3" ref="inet.customerinfo.backoffice.PayeeRegisterService"/>
    <constructor-arg index="4" ref="inet.customerinfo.backoffice.ServiceService"/>
    <constructor-arg index="5" ref="inet.customerinfo.backoffice.SettingsService"/>
    <constructor-arg index="6" ref="inet.customerinfo.backoffice.SearchCustomerRequestValidator"/>
    <constructor-arg index="7" ref="inet.customerinfo.backoffice.LoginDataRequestValidator" />
</bean>
```

*Figure 4. Example of how the facade bean in this application is defined.*

Data binding is what makes it possible to automatically bind user input to the request object. In Figure 5 a date interval that the user has entered is bound to the `loginDataRequest` object. The `bindingResult` object contains errors if the data binding fails. The `model` is a data container that contains the only data the view has access to. The controller returns a tiles definition.

```java
@RequestMapping(value = "/successfulLogins", method = RequestMethod.POST)
public String succesfulLogins(final Model model, final LoginDataRequest loginDataRequest,
        final BindingResult bindingResult) {

    if (!bindingResult.hasFieldErrors()) {
        final List<SuccessfulLoginDTO> data =
                (List<SuccessfulLoginDTO>) customerInfoFacade.getSuccessfulLogins(loginDataRequest);
        model.addAttribute("successfulLogins", data);
        model.addAttribute("fromDate", loginDataRequest.getFromDate().getTime());
        model.addAttribute("toDate", loginDataRequest.getToDate().getTime());
    }

    return "customerinfo.successfullogins.def";
}
```

*Figure 5. Controller handling the POST call for successful logins.*

```
<c:if test="${not empty successfulLogins}">
    <table id="successfulLoginsTable" class="data">
        <thead>
            <tr class="">
                <th><fmt:message bundle="${bundle_labels}" key="lbl_timestamp" /></th>
                <th><fmt:message bundle="${bundle_labels}" key="lbl_loglin_type" /></th>
                <th><fmt:message bundle="${bundle_labels}" key="lbl_ip_address" /></th>
                <th><fmt:message bundle="${bundle_labels}" key="lbl_identity_provider" /></th>
                <th><fmt:message bundle="${bundle_labels}" key="lbl_jsession_id" /></th>
                <th><fmt:message bundle="${bundle_labels}" key="lbl_authentication_level" /></th>
            </tr>
        </thead>
        <tbody>
            <c:forEach items="${successfulLogins}" var="successfulLogin">
                <c:url value="/customerinfo/successfulLoginDetails.do" var="successfulLoginDetails">
                    <c:param name="timeStamp" value="${successfulLogin.timeStamp}" />
                </c:url>
                <tr>
                    <td><a href="${fn:escapeXml(successfulLoginDetails)}">
                        <fmt:formatDate value="${successfulLogin.timeStamp}" type="date" pattern="dd.MM.yyyy HH:mm:ss,SSS" />
                    </a></td>

                    <td><c:out value="${successfulLogin.loginType}" /></td>
                    <td><c:out value="${successfulLogin.ipAddress}" /></td>
                    <td><c:out value="${successfulLogin.identityProvider}" /></td>
                    <td><c:out value="${successfulLogin.jsessionId}" /></td>
                    <td><c:out value="${successfulLogin.authenticationLevel}" /></td>
                </tr>
            </c:forEach>
        </tbody>
    </table>
</c:if>
```

*Figure 6. JSP displaying the successful login table*

## 3.3 Tiles

Apache Tiles (The Apache Software Foundation, 2016) is a template composition framework that allows developers to define templates that are used in the layout of the page. A tiles definition contains attributes specific to the page, such as the body (table in figure 7). These definitions are what Spring utilizes to know which interface (UI) to show to the user after a controller routine is completed.

```
<definition name="customerinfo.successfulLogins.datepicker.def" extends="customerinfo.main.def">
    <put-attribute name="table" value="/WEB-INF/customerinfo/jsp/datepicker.jsp" cascade="true"/>
    <put-attribute name="activeTab" value="successfulLogins" cascade="true"/>
</definition>

<definition name="customerinfo.successfulLogins.def" extends="customerinfo.main.def">
    <put-attribute name="table" value="/WEB-INF/customerinfo/jsp/successfulLogins.jsp" cascade="true"/>
    <put-attribute name="activeTab" value="successfulLogins" cascade="true"/>
</definition>
```

*Figure 7. Example of a tiles definition.*

## 3.4 iBATIS

iBATIS is a persistence framework that is used to automate the mapping between Java objects and SQL databases. The mapping is done by writing SQL statements in an XML-file. In order to get the most out of iBATIS it is recommended to use result maps. With the result maps it is easy to map the results from the database to a Java object (Begin et al., 2007).

An example is given in Figure 8. The class which the data is sent to is specified in the declaration of the result map tag. Inside the result map tags the variables that are used in the class is specified as result tags. The property in the result tag is the name of a variable in the class. Column is the name of the database column which the data is retrieved from and javaType is the datatype of the variable.

The queries are written using standard SQL syntax but must be written inside tags like `select, update or insert`. Inside the tag declaration an id, a parameter class and result map needs to be specified for it to work. The `id` property is just the name of the query and it is used when the query is executed. The `parameterClass property` defines the datatype of the input parameter in this case userId. The `resultMap` property specifies which result map that should be used.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN" "http://ibatis.apache.org/dtd/sql-map-2.dtd" >

<sqlMap namespace="inet.backoffice.customerinfo">

    <resultMap id="serviceMap" class="fi.crosskey.inet.backoffice.customerinfo.domain.model.Service">
        <result property="code" column="SERVICE" javaType="java.lang.Integer" />
        <result property="owner" column="SERVICE_OWNER" javaType="java.lang.String" />
        <result property="accountNumber" column="ACCOUNT_NR" javaType="java.lang.String" />
        <result property="customerNumber" column="CUSTOMER_NR" javaType="java.lang.String" />
        <result property="openingDate" column="OPENING_DATE" javaType="java.util.Date"/>
        <result property="lastUpdated" column="UPDATE_DATE" javaType="java.util.Date" />
    </resultMap>

    <select id="getServices" parameterClass="java.lang.String" resultMap="serviceMap">
        SELECT
            SERVICE,
            SERVICE_OWNER,
            ACCOUNT_NR,
            CUSTOMER_NR,
            OPENING_DATE,
            UPDATE_DATE
        FROM
            AAB_USER_SERVICE
        WHERE
            USER_ID = #userId#
    </select>

</sqlMap>
```

*Figure 8. Example of a sqlmap & resultmap.*

### 3.4.1 Typehandlers

iBATIS provides an interface to implement your own typehandlers that can be used in the result maps. If the standard iBATIS mapping is not enough for the object in question a typehandler can be used to change the behavior as you want it. In the application some enumerations were saved as strings. To easily map them directly to the enums custom type handlers were made to get the required behavior. An example of a typehandler created for this application can be found in figure 9.

```java
public class DeliveryEndpointTypeHandler implements TypeHandlerCallback {

    @Override
    public void setParameter(final ParameterSetter setter, final Object parameter) throws SQLException {
        final DeliveryEndpoint deliveryEndpoint = (DeliveryEndpoint) parameter;
        setter.setString(deliveryEndpoint.getValue());
    }

    @Override
    public Object getResult(final ResultGetter getter) throws SQLException {
        final String stringParameter = getter.getString();
        return DeliveryEndpoint.getDeliveryEndpoint(stringParameter);
    }

    @Override
    public Object valueOf(final String s) {
        return DeliveryEndpoint.getDeliveryEndpoint(s);
    }

}
```

*Figure 9. Example of a typehandler in this application.*

# 4. DEVELOPMENT OF APPLICATION

## 4.1 Specification

When the project started a specification was provided to the developers. This document serves as guidelines for the developers and also describes the requirements.

The core points found in the specification are:

- The back office user should get information about an end user by searching with its ID.
- There should be separate tabs that display successful logins, failed logins, balance alerts, payee register, services, account settings and user settings for the end user. Also details pages should be available for the successful logins and balance alerts.
- The application should work for all of Crosskey's customers.
- Support of three languages: Finnish, Swedish and English.
- You should be able to regulate which users have access to the application. This should be handled by the LDAP permission system that is already in place for back office.
- The application should be feature toggleable. This is to make it easier to turn on the service for customers that has ordered it. There were no details on how to achieve this in the specification so it started as an open question.

## 4.2 Preparation

Before the application development could begin the specification was broken down into smaller pieces, by creating JIRA cases that describe the tasks. JIRA was also used as a tool to document our progress and to keep track on who does what.

To be able to store all needed data, some adjustments had to be made to the database. The datatable for successful logins needed to be extended with two additional fields. This led to some minor changes in the code where these logins were written to the database.
The existing code was analysed and it was determined what functionality could be reused and what should be developed from scratch.

## 4.3 Development

One of the first things that was done was to make a basic page where the back office user could search for a end user to get some info that needed to be displayed. In the process of making that page many of the key components like the controller, dataflow, the mapping of the SQL queries and how the data from the database should be handled were implemented.

When this was all done the tabs needed to be implemented. At this stage the method for implementing the tabs needed to be decided, the choices being an implementation of tabs in the Internet bank that had to be ported to the back office or some third party tabs library. The bootstrap framework was considered but issues occurred with the back office layout when trying to implement it. In the end the tabs from the internet bank were chosen since it was already in use and this would make it easier to get a consistent layout (figure 10).

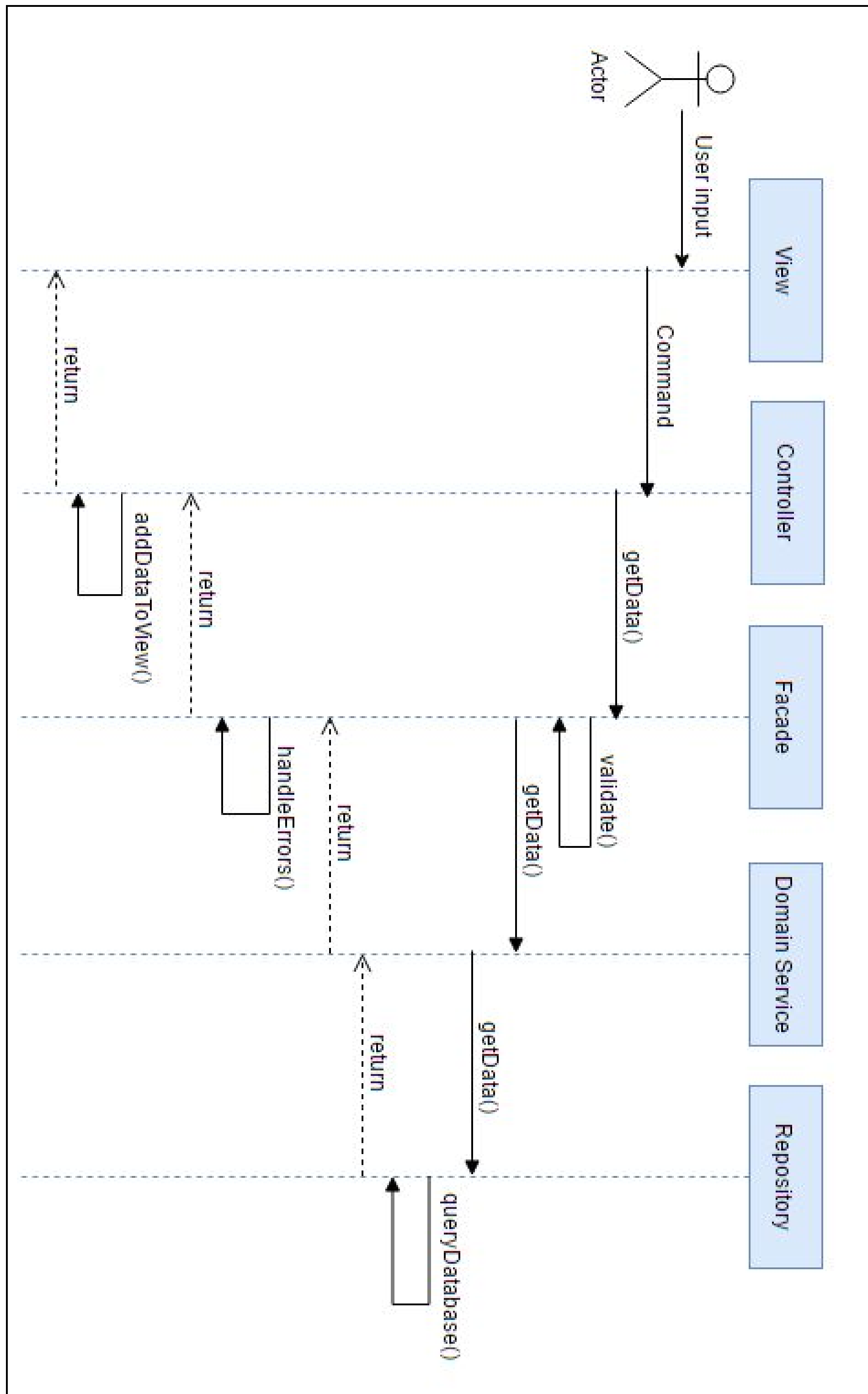


*Figure 10. Sequence diagram of a use case*

The tabs are predefined in a JSP file with URLpaths that define what data each tab has access to, see Figure 11. These URLs are picked up by the Spring controller that prepares the data and returns a tiles definition corresponding to the tab. The tiles definitions control what tab to show as active and what JSP to display the data in. After this the development for each individual tab could begin, using the methods described in chapters 3 and 4.

```
<cbs:cbsTab key="successfulLogins" first="true" >
    <jsp:attribute name="title">
        <fmt:message bundle="${bundle_labels}" key="lbl_successful_logins" />
    </jsp:attribute>
    <jsp:attribute name="link">${pageContext.request.contextPath}/customerinfo/successfulLogins.do</jsp:attribute>
</cbs:cbsTab>
```

*Figure 11. Example of how the tabs are predefined in a JSP.*

In Figure 12 you can see that customer details like their name and SSN always has to be displayed in the application after the backoffice user has searched for them. This was solved by storing a reference to a DTO object in the session to be able to access the data without having to query the database each time. This also helped to access the end user's user ID when switching through tabs.

### 4.3.1 Successful logins

The first tab that was developed was the successful logins tab. The main purpose of this tab is to display a list of successful login attempts by the end user in question (Figure 12). This tab features a details page that displays more details about a specific login.

This tab proved to be the most difficult one since the successful logins did not have a unique key in the datatable, which made fetching data for the details page troublesome. At first all properties of the successful login entry were posted directly from the list that is displayed in Figure 12. This was only temporary until a better solution was found.

The solution was to use a built in unique id of a datatable row. The row id combined with the user id made a reliable combination that successful logins can be retrieved with.

Another thing that needed to be done was to parse the values displayed in the parameters box on the details page. These were stored as a CLOB in the database in JSON format. These fields were added to the database in the preparation phase of the project.

This tab features a date picker for choosing date intervals of the logins and a details page that shows the details of a specific login when the backoffice user clicks on a login date. When the form is posted the dates in the input fields are bound to a request object using Spring.

*Figure 12. The successful logins tab with its datepicker*

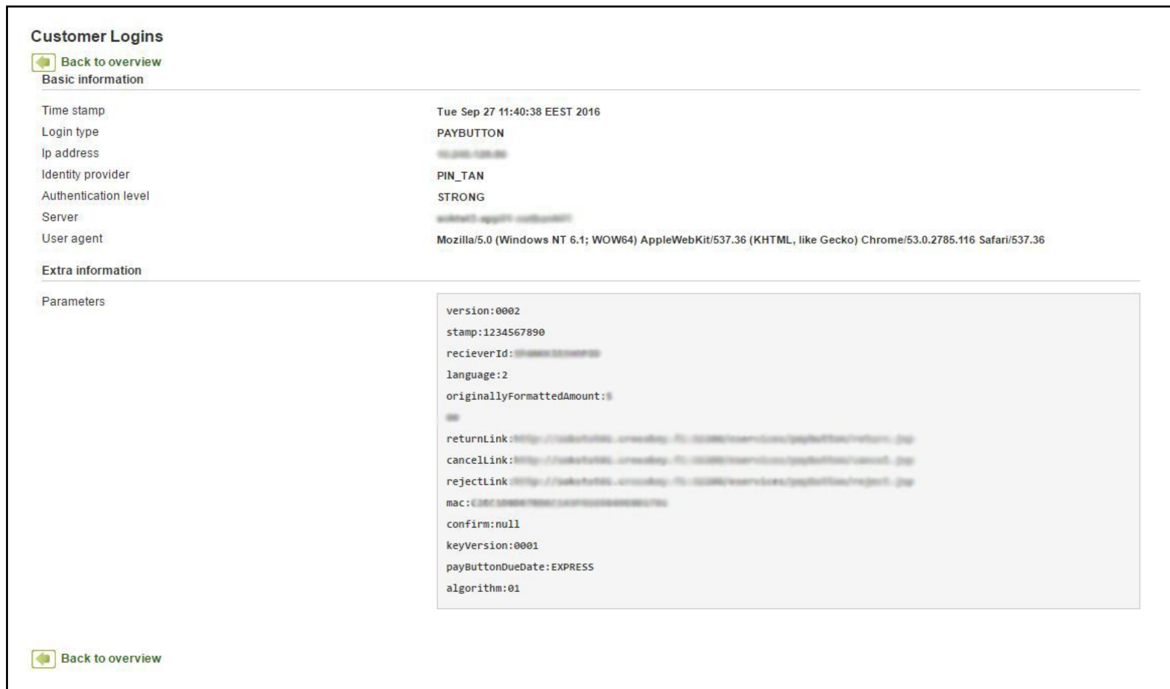


*Figure 13. Successful login details page*

### 4.3.2 Balance alerts

Next up was the balance alerts tab. This is where balance alerts and active balance reports to the end user should be displayed (Figure 14). The data had to be retrieved from two different datatables. Due to their similarities, a shared model object could be used, representing both balance alerts and balance reports with an enumeration separating what type it was.

This tab has two details pages one for balance reports and one for balance alerts (Figure 15 and 16). Because these pages needed to display different information depending on if it was a report or alert separate models were created for the details pages. The creation date contains a link that contains the id of the notification and what type it is, so the controller knows what tile definition to return.

The tab was later renamed to balance notifications to better represent that it does not only contain balance alerts.

Successful logins | Balance alerts | Payee register | Services | Account settings | Settings

Visa 10 rader                                                            Sök:

| Creation date | Account number | Type | Delivery endpoint | Available amount | Email | Last triggered | Last modified |
|---|---|---|---|---|---|---|---|
| 25.11.2011 12:38:11,000 | [illegible] | REPORT | EMAIL | true | WORK | 21.00.2016 09:00:02 | |
| 10.11.2011 13:36:27,000 | [illegible] | ALERT | EMAIL | true | HOME | 10.45.2011 13:45:01 | |
| 10.11.2011 13:08:02,000 | [illegible] | ALERT | EMAIL | true | HOME | 10.15.2011 13:15:01 | |

Visar 1 till 3 av totalt 3 rader

Previous 1 Next

*Figure 14. Balance notifications.*

**Balance alert details**

Back to overview

**Basic information**

| | |
|---|---|
| Account number | [illegible] |
| Delivery endpoint | EMAIL |
| Amount | [illegible] |
| Trigger condition | LESS_THAN_OR_EQUAL |
| Already triggered | true |
| Use available amount | true |
| Send only once | false |
| Delivery address | HOME |
| Creation date | 29.52.2011 08:52:19 |
| Last modified | 15.05.2011 10:05:09 |
| Last triggered | 15.20.2011 10:20:01 |

Back to overview

*Figure 15. Detailed view for a balance alert.*

**Balance report details**

Back to overview

**Basic information**

| | |
|---|---|
| Account number | [illegible] |
| Delivery endpoint | EMAIL |
| Use available amount | true |
| Time of day | MIDDLE_OF_DAY |
| Day pattern | everyday |
| Active time | ANY_DAY |
| Delivery address | HOME |
| Creation date | 14.02.2012 09:03:38 |
| Last modified | 04.12.2012 08:23:21 |
| Last triggered | 22.11.2016 14:00:01 |

Back to overview

*Figure 16. Detailed view for a balance report.*

### 4.3.3 Payee register

This tab shows the payee register of the end user (Figure 17). The development of this tab was pretty straightforward since it had all of its data in one table. The information displayed here is information that one usually want to store in a payee register like account number, name, nickname and amount.

Successful logins | Balance alerts | Payee register | Services | Account settings | Settings

Visa 10 rader Sök:

| Creation date | IBAN | Nick name | Name | Reference | Amount | Fast process | Last modified |
|---|---|---|---|---|---|---|---|
| 27.01.2010 17:15:03,000 | [illegible] | [illegible] | [illegible] | | | false | 27.15.2010 17:15:03 |
| 27.01.2010 17:12:43,000 | [illegible] | [illegible] | [illegible] | | | false | 27.12.2010 17:12:43 |
| 27.01.2010 17:06:01,000 | [illegible] | [illegible] | [illegible] | | | false | 27.06.2010 17:06:01 |
| 27.01.2010 17:02:41,000 | [illegible] | [illegible] | [illegible] | | | false | 27.02.2010 17:02:41 |

Visar 1 till 4 av totalt 4 rader

Previous 1 Next

*Figure 17. The end users payee register.*

### 4.3.4 Services

In the service tab you can see what services the end user has currently active. It displays info such as the name of the service and when it was activated. Implementationwise this tab was pretty unremarkable; the services could be retrieved directly from the database without trouble (Figure 18).

Successful logins | Balance alerts | Payee register | Services | Account settings | Settings

Visa 10 rader Sök:

| Service | Owner | Account number | Customer number | Opening date | Last updated |
|---|---|---|---|---|---|
| 880 | CORE_REPLICATION | [illegible] | | 24.01.2011 | 24.01.2011 |
| 460 | CORE_REPLICATION | | | 21.08.2015 | 31.08.2015 |
| 420 | CORE_REPLICATION | [illegible] | | 04.05.2015 | 04.05.2015 |
| 291 | CORE_REPLICATION | [illegible] | [illegible] | 23.04.2015 | 23.04.2015 |

Visar 1 till 4 av totalt 4 rader

Previous 1 Next

*Figure 18. Services the end user has activated.*

### 4.3.5 Account settings

This tab was responsible for showing all account settings the end user has activated. The tab features account related information such as if the account is default or if it is displayed (Figure 19).

Successful logins | Balance alerts | Payee register | Services | Account settings | Settings

Visa 10 rader  Sök:

| Creation date | Account number | Display account | Default account | Default invoice account | Soft block | Last modified |
| --- | --- | --- | --- | --- | --- | --- |
| 29.01.2015 | [illegible] | true | false | false | false | 03.03.2016 |
| 29.01.2015 | [illegible] | true | false | false | false | 03.03.2016 |
| 29.01.2015 | [illegible] | true | false | false | false | 03.03.2016 |
| 29.01.2015 | [illegible] | true | false | false | false | 03.03.2016 |
| 29.01.2015 | [illegible] | true | false | false | false | 03.03.2016 |
| 29.01.2015 | [illegible] | true | false | false | false | 03.03.2016 |
| 29.01.2015 | [illegible] | false | false | false | false | 03.03.2016 |
| 29.01.2015 | [illegible] | true | false | false | false | 03.03.2016 |
| 29.01.2015 | [illegible] | true | false | false | false | 03.03.2016 |
| 29.01.2015 | [illegible] | true | false | false | false | 03.03.2016 |

Visar 1 till 10 av totalt 21 rader

Previous 1 2 3 Next

*Figure 19. Account settings the end user has applied,*

### 4.3.6 User settings

The settings tab was renamed to user settings to better describe what data it contained. These settings are pretty cryptic and cannot really be understood without access to the system. The settings displayed here are applied all over the internet bank (Figure 20).

Successful logins | Balance alerts | Payee register | Services | Account settings | Settings

Visa 10 rader  Sök:

| Property | Value | Last modified |
| --- | --- | --- |
| tab:payments | payments_mnu_debit | |
| tab:messages | messages_mnu_messages | |
| tab:cards | cards_mnu_cardapplication | |
| tab:accounts | accounts_mnu_accountoverview | |
| stylesheet | 99.05_css_classic | |
| overview | lbl_accounts\|lbl_s_cards\|lbl_s_cards2\|lbl_depositaccounts | |

Visar 1 till 6 av totalt 6 rader

Previous 1 Next

*Figure 20. User settings the user has activated.*

### 4.3.7 Failed logins

A few models were created for the domain but that was as far as it got. Issues were encountered when trying to fetch failed logins for a specific user. These issues were never solved and the failed logins tab remains undone. It was later decided that the tab should not be developed.

### 4.3.8 Validation

Fields where the back office user is able to enter free text in a request needs to be validated to stop harmful and incorrect input reaching the infrastructure layer. For example the backoffice user gets to enter a keyword in the application to search for an end user (Figure 21). This is usually some kind of customer identifier such as SSN, except for the '-' character in a SSN this keyword should not contain any special characters. If these characters are encountered the validation will fail and inform the backoffice user that the input was invalid. There is also a date validator that checks that the start date is not before the end date.

```
public boolean validate(final SearchCustomerRequest searchCustomerRequest) {
    boolean valid = true;

    if (searchCustomerRequest != null) {
        final String searchString = searchCustomerRequest.getSearchString();

        if (StringUtils.isEmpty(searchString)) {
            valid = false;
        }
        else if (!searchString.matches("[a-zA-Z0-9- ]*")) {
            valid = false;
        }
    }
    else {
        valid = false;
    }

    return valid;
}
```

*Figure 21. Example of a simple validation routine*

### 4.3.9 Error and info handling

To handle error and info messages in the application a framework developed internally by Crosskey was used. This library is basically a wrapper that wraps the retrieval result with error and info messages depending on what has occurred. These messages are later added to the model and displayed to the user.

## 4.4 Configuring the application for all customers

All of the development was first done for one customer and then moved to the other customers. The first customer that was tested had an issue with the date picker, the issue being that the customer had a different date format which made Spring unable to databind the date to a Java date object. A missing Spring configuration caused this, so the configuration was added and the application was able to databind the date.

The last customer that was tested stored the SSN in a different format in the database. This caused problems when trying to search for a user since the application formats the search string using a different standard. The solution was to create a separate version of the SSN strategy class that was injected into the system. This was later also done for the rest of the customers since they all had minor variations of the SSN format.

Another issue was discovered when configuring the application for the Swedish customers. The payee register was stored in a different table in the database which meant that the query did not work any more. The solution was to create a new SQL map for the Swedish banks and then inject it into the affected banks in the same way as the SSN strategy.

Another issue that was discovered with the payee register was that the account number was stored in a way that needed to be parsed for the Swedish customers. The solution was to

create a new service for the Swedish banks and inject a dependency to a method that handles the parsing.

Additionally there were some minor CSS fixes that needed to be done for each individual customer.

### 4.5 User limitations

Since the customer information application  handles sensitive information like IP addresses there had to be some sort of limitation that allows only some back office users to access the application. This was not a problem since backoffice user permissions were already handled by LDAP authentication. New permission levels were added for both read and write access to the application. The Spring controller checks for these permissions before granting a backoffice user access.

## 5 CONCLUSION

### 5.1 Result

The application was developed in 2016 following the specification given by an external supervisor.  All of the specification requirements were successfully implemented in the customer information application except for one. A decision on how to develop the feature toggle was never reached, so that remains incomplete.

There are also a few layout issues present that should be polished before the official release. Support for three languages is implemented but the translations of the applications labels are not done yet, everything is currently in english.

## 5.2 Reflections

We learned a lot during the development and we managed to complete the application during the scheduled time. We were provided with everything we needed from Crosskey and got guidance from our fellow co-workers if we encountered any issues.

In the end the application became bigger than we initially thought. We came to the conclusion that we did not have to use the same technologies as back office, for instance MyBatis instead of iBATIS. The library we created had its own dependencies and that is why it would have been possible.

In the future this application could be extended to retrieve even more information about end users. The failed logins tab, that was scrapped, is an example of an extension you could to to this application. This application should be able to handle any kind of information Crosskey's customers want to see about their end users.

Proper unit and integration tests could probably be implemented. Small changes to the database should not break the implementation, but you can never be too sure.
Since a MyBatis implementation was possible it could be implemented instead of iBATIS. It would require change to the repository implementations and typehandlers so it can be run with MyBatis.

# REFERENCES


Begin, C., Goodin, B., & Meadors, L. (2007). *iBATIS in Action*. New York: Manning Publ.

Crosskey. (2016). Retrieved from https://www.crosskey.fi/our-story/

Martin Fowler. (2002). *Patterns of Enterprise Application Architecture* Addison-Wesley Professional.

Oracle. (2016). Java EE at a Glance. Retrieved from http://www.oracle.com/technetwork/java/javaee/overview/index.html

Pivotal Software. (2016). Spring, Let's build a better Enterprise. Retrieved from http://spring.io/

The Apache Software Foundation. (2016). Apache Tiles documentation. Retrieved from https://tiles.apache.org/

Walls, C. (2015). *Spring in Action* (4th ed.). Shelter Island: Manning.

Yener, M., & Theedom, A. (2014). *Professional Java EE Design Patterns*. Hoboken: Wrox.