

Development of Magento 1.9 Extension for Briox ERP Integration

Joel Östman

Bachelor's Thesis
Information Technology
Vasa 2018



EXAMENSARBETE

Författare: Joel Östman

Utbildning och ort: Informationsteknik, Vasa

Inriktningsalternativ: Informationsteknik

Handledare: Kaj Wikman

Titel: Utveckling av Magento 1.9 extension för integration med Briox affärssystem

Datum: 5.4.2018

Sidantal: 44

Abstrakt

Detta examensarbete behandlar utvecklingen av en Magento extension, vars syfte är att integrera webbutiker skapade i Magento ramverket, med affärssystemet Briox.

Extensionen ska autentiseras med affärssystemet och exekvera en exportering av alla beställningar som gjorts under det senaste dygnet. Nya kunder som registrerats i webbutiken ska registreras i affärssystemet tillsammans med en kundorder och faktura.

Arbetets mål var att automatisera en tidskrävande process som hittills gjorts manuellt av anställda, samt att eliminera den mänskliga faktorn.

Extensionen utvecklades i programmeringsspråket PHP och använder sig av en RESTful API för att etablera en kommunikation med affärssystemet. Examensarbetet behandlar i högsta grad använda metoder, Magento ramverkets struktur och den utvecklade extensionens kronologiska händelseförlopp och funktionalitet.

Projektet resulterade i en konfigurerbar Magento extension som gör det möjligt för en integration mellan webbutiker och affärssystemet Briox.

Språk: engelska

Nyckelord: webbutik, PHP

OPINNÄYTETYÖ

Tekijä: Joel Östman

Koulutus ja paikkakunta: Tietotekniikka, Vasa

Suuntautumisvaihtoehto: Tietotekniikka

Ohjaaja: Kaj Wikman

Nimike: Magento 1.9 -laajennuksen kehittäminen Briox-toiminnanohjausjärjestelmän integroimiseksi

Päivämäärä 5.4.2018

Sivumäärä: 44

Tiivistelmä

Tämä opinnäytetyö käsittelee Magento-laajennuksen kehitystä. Laajennuksen tarkoitus on integroida Magento-sovelluskehitykselle luotu verkkokauppoja Brioxin toiminnanohjausjärjestelmään. Laajennus täytyy autentisoida toiminnanohjausjärjestelmän avulla ja sen on suoritettava kaikki tilaukset viimeisten 24 tunnin ajalta. Verkkokauppaan rekisteröityneet uudet asiakkaat on kirjattava kauppajärjestelmään yhdessä tilauksen ja laskun kanssa.

Työn tavoitteena oli automatisoida aiemmin manuaalisesti suoritettu aikaa vievä prosessi, sekä lisäksi poistaa inhimillisten virheiden mahdollisuus.

Laajennus kehitettiin ohjelmointikielellä PHP, ja sen viestintä toiminnanohjausjärjestelmän kanssa tapahtuu RESTful API-sovelluksen avulla. Opinnäytetyössä käsitellään käytettyjä menetelmiä, Magento-sovelluskehityksen rakennetta, kehitetyn laajennuksen toiminnollisuutta sekä sen kronologista tapahtumakulkua.

Projektin tuloksena on konfiguroitava Magento-laajennus, joka mahdollistaa integroinnin verkkokauppojen ja Briox-toiminnanohjausjärjestelmän välillä

Kieli: englanti

Avainsanat: verkkokauppa, PHP

BACHELOR'S THESIS

Author: Joel Östman

Degree Programme: Information Technology, Vaasa

Specialization: Information Technology

Supervisor: Kaj Wikman

Title: Development of Magento 1.9 Extension for Briox ERP Integration

Date: April 5, 2018

Number of pages: 44

Abstract

This Bachelor Thesis reviews the development of a Magento extension, for an integration between a Magento Ecommerce store and the Briox ERP-system. The extensions should authenticate with the ERP-system and execute an export of all orders made during the last day. All newly registered customers in the online store should be registered in the ERP-system along with a sales order and a customer invoice.

The goals of the project are to automate a time consuming process which has been manually performed to date, and eliminate the human factor. The extension was developed by use of the PHP programming language and by the use of a RESTful API it establishes a communication with the ERP-system. The Bachelor Thesis examines methods used, the Magento framework structure and the chronological workflow and functionality of the developed extension.

The project resulted in a configurable Magento extension which makes an integration between Ecommerce stores and the Briox ERP-system possible.

Language: English

Key words: PHP, Ecommerce Store

Table of Content

1	Introduction	1
1.1	Employer	1
1.2	Task.....	1
1.3	Purpose	2
1.4	Requirements.....	2
2	Tools and methods.....	2
2.1	Programming Concepts.....	3
2.1.1	PHP	3
2.1.2	PHP CURL library.....	4
2.1.3	Object Oriented Programming (OOP)	4
2.1.4	Model-View-Controller Architecture (MVC)	5
2.1.5	JSON and XML data.....	6
2.1.6	Application Programming Interface (API).....	8
2.1.7	RESTful API	9
2.2	Magento Framework.....	9
2.2.1	Magento Architecture (Model-View-Controller).....	9
2.2.2	Magento Directory Structure	10
2.2.3	Custom Module Directory Structure	11
2.3	Installation	12
2.3.1	Setup Components	13
2.3.2	Setup Resources	13
2.3.3	Installer Scripts	14
2.3.4	Upgrade Scripts	16
2.4	Briox ERP-system.....	17
2.4.1	Briox RESTful API	17
3	Implementation	19
3.1	Configuration	20
3.1.1	Config.xml.....	20
3.1.2	Adminhtml.xml	21
3.1.3	System.xml	22
3.2	Install Script	25
3.2.1	Creating the database table	25
3.2.2	Adding custom attributes	26
3.3	Authentication.....	27
3.3.1	Authentication Token	27
3.3.2	Retrieving the Access Token	28

3.4	Initiation	30
3.4.1	Token refresh	30
3.4.2	Collecting all orders	30
3.4.3	Check for customer in Briox	32
3.4.4	Creating the Customer	33
3.4.5	Preparing and posting the Customer	34
3.4.6	Creating the Salesorder and Invoice	34
3.4.7	Preparing and posting the Salesorder	35
3.4.8	Preparing and posting the Invoice	37
3.5	CURL Requests	40
3.5.2	Refreshing the Tokens	41
3.5.3	Checking for customer in Briox	41
3.5.4	Creating a new customer in Briox	42
3.5.5	Creating a new Salesorder/Invoice	42
3.6	Testing	42
4	Results and Discussion	43
4.1	Results	43
4.2	Future Work	43
4.3	Discussion	44
5	Bibliography	45

Table of Figures

Figure 1: PHP Function example.....	3
Figure 2: MVC components relationship	6
Figure 3: JSON cars object	6
Figure 4: XML cars object	7
Figure 5: API communication.....	8
Figure 6: Magento CodePool Structure	10
Figure 7: Magento Custom Module folder hierarchy	11
Figure 8: Connect Form Layout.....	23
Figure 9: Configuration Form Layout	24
Figure 10: Generating the Authentication Token.....	27
Figure 11: Authenticating in the Admin Backend	28
Figure 12: Authentication workflow	29
Figure 13: Main function workflow	30
Figure 14: The CURL Class and its Method	40

Glossary

API	"Application Programming Interface", a collection of function which allows a programmer to access application data.
Array	A collection of "things" arranged in a particular way.
Backend	A part of an application accessible only by authorized personnel.
Class	A template definition of a "category" and its properties and behavior.
Client	A computer or program, which presents the user with data.
Cronjob	A scheduled and recurring task.
ERP	"Enterprise Resource Planning", a software handling business processes and management.
Framework	An already developed computer program base for easier and faster development.
Frontend	The part of an application the user interacts with.
GUI	"Graphical User Interface", the visual composition of a computer program.
HTTP	"Hypertext Transfer Protocol", the underlying protocol used by the internet.
Instance	A creation of a class object, capturing all the properties and behaviors of the class.
Interface	A part of a software that handles inputs and outputs.
JSON	"Javascript Object Notation", a text-based language format used for sharing data.
Module	A component of a computer program that contains one or more routine.

OOP	“Object Oriented Programming”, a programming design pattern which makes use of classes.
PHP	“Hypertext Preprocessor”, a server-side language developed for web development.
SQL	“Structured Query Language”, a standardized language used for manipulating a database.
Script	A list of code commands executed by a program.
Server	A computer that stores all data and waits for a request from the client.
Token	A long string constructed of random characters and used as a key.
Variable	A named piece of computer memory used to store values.
XML	“Extensible Markup Language”, a language designed to store and transmit data.

1 Introduction

This report describes the development process of a Magento Extensions, which integrates a Magento Ecommerce store with the Briox ERP-System. The report will describe the methods and tools used to accomplish the integration and will pay attention to the structure of the Magento Framework and give the reader an understanding of the process with an in-depth explanation of the chronological workflow regarding the export of orders. Note that every word written in italic font is explained in the glossary.

1.1 Employer

Webcore is an organization in Sulva, Mustasaari. They provide their customers with smart and efficient communication solutions. Webcore offer systems for websites, e-commerce, analysis, marketing, CRM, customer service and office cloud-services. Webcore is the founder and developer of the region famous site Findit.fi.

Webcore provides their customers with online stores, developed using the Magento *Framework*. Webcore recently started collaborating with the new *ERP*-system provider, Briox.

1.2 Task

The task of this project is to automate the transaction of orders and customer details from the Webcore Ecommerce Store to their Briox cloud-based *ERP* system account. The developed Magento *Module* should run once a day and transfer the orders made during the last 24 hours. This reoccurring transfer will be executed by a *cronjob*, set up on the *server*. Currently Webcore staff do the data transfer manually. The goals of the software are to eliminate the human factor, and save precious time for the company.

1.3 Purpose

The purpose of this project is to eliminate human errors that might occur during the transfer, such as spelling errors, accidentally missed orders and duplications in the *ERP*-system. The software will likewise save precious time for the company and let the employees focus on tasks that are more important.

1.4 Requirements

The project was requested with a few specific details and requirements to achieve. These requirements are necessary for the process to be as automatic as possible.

Requirements:

- Run every 24h (Adjustable)
- Add custom attributes to registered customers and product in the online store
- Automatic *Token* Refresh
- Check if customer exists in Briox
- Create customer if it does not already exist
- Create Salesorder
- Create Invoice

2 Tools and methods

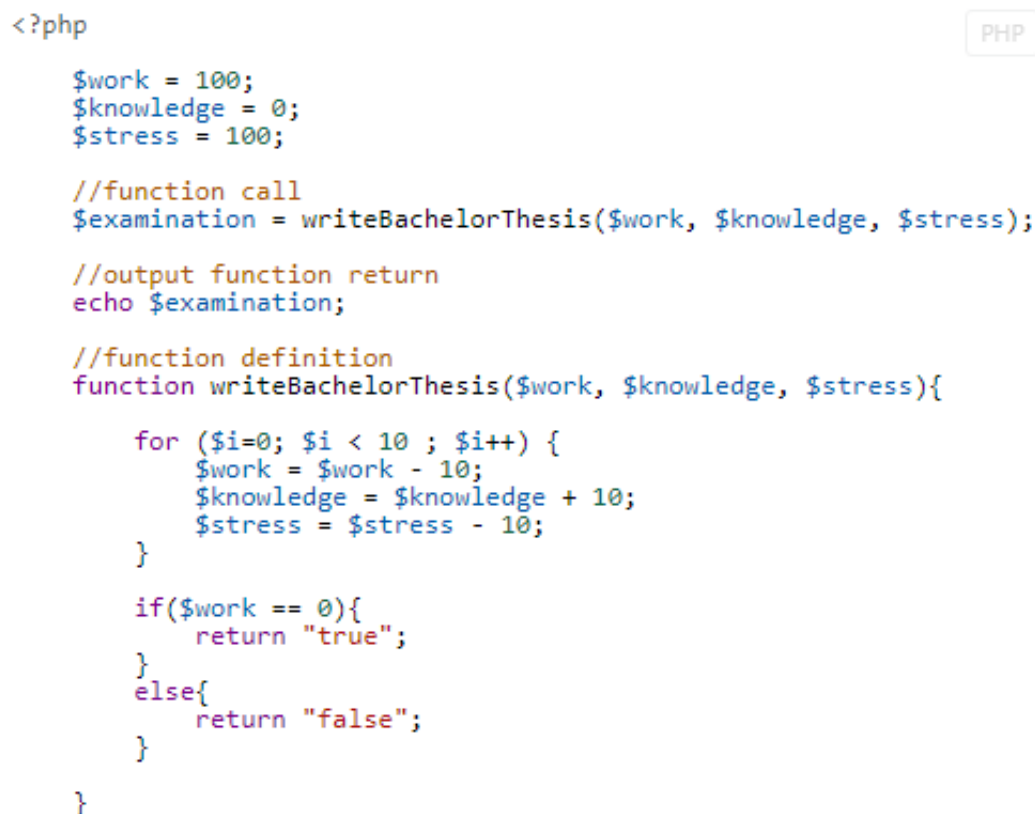
This chapter will demonstrate the tools and methods used to develop the Magento extension, and to satisfy the requirements. The chapter will educate the reader about the programming languages, design patterns and the Magento framework. The last section will discuss the Briox *ERP*-system and how the RESTful *API* works.

2.1 Programming Concepts

Computer programming have been around since the first computer. Since then it has evolved into a broad spectrum. Numerous languages, design patterns and paradigms have been featured along the way. There are currently too many components to demonstrate in one report. Therefore, this section will be demonstrating and analyzing the programming fundamentals of the Magento framework.

2.1.1 PHP

PHP (Hypertext Preprocessor) is an open source scripting language used by the Magento *framework*. The programming language is most suitable for web development and can be embedded into HTML. The *PHP* syntax is based of Java, C and Pearl with a set of unique functions implemented. The language itself is easy to learn and allows developers to write code and dynamic webpages rapidly [1] [2]. Below is an example of a basic PHP function:



```
<?php
$work = 100;
$knowledge = 0;
$stress = 100;

//function call
$examination = writeBachelorThesis($work, $knowledge, $stress);

//output function return
echo $examination;

//function definition
function writeBachelorThesis($work, $knowledge, $stress){
    for ($i=0; $i < 10 ; $i++) {
        $work = $work - 10;
        $knowledge = $knowledge + 10;
        $stress = $stress - 10;
    }
    if($work == 0){
        return "true";
    }
    else{
        return "false";
    }
}
```

Figure 1: PHP Function example

2.1.2 PHP CURL library

PHP *Client* URL library (CURL) is a library created by Daniel Stenberg. It allows for the connection and communication between *server* with numerous protocol types. The library currently supports https, *http*, ftp, telnet, file, gopher, dict and ldap protocol types. It is often used for REST *API* communication due to the support of HTTP POST, HTTP GET, HTTP PUT and HTTP DELETE actions. [2]

Example of a function performing a CURL request:

```
<?php
    // create the curl resource
    $curl = curl_init();

    // configures the URL
    curl_setopt($curl, CURLOPT_URL, "api.com");

    // setreturn the transfer as a string
    curl_setopt($curl, CURLOPT_RETURNTRANSFER, 1);

    // executes the request and stores response in output variable
    $output = curl_exec($curl);

    // closes the curl resource to free up system resources
    curl_close($curl);
?>
```

2.1.3 Object Oriented Programming (OOP)

Object Oriented Programming is a programming design pattern. The software uses *classes* to define objects. Classes contains the data and functionality of the object, for example, a Car object has a class that contain the data variables of the car (brand, model, color etc.) and the methods that provides functionality (drive, turn, reverse etc.). Objects are able to interact with each other, for example, a Human object can interact with a Car object and make the car drive.

OOP allows *classes* to inherit behavior from another class. For example, a Vehicle class may inherit the Car class and other classes, such as Boat, Bikes etc. In the Java programming language the Vehicle class would become a so called superclass. Each class is allowed to

have only one superclass while each superclass is likely to have an unlimited amount of subclasses. Many consider this an effective way to reuse code in the software.

The advantage of using OOP techniques is that they allow programmers to create *modules* that can remain unchanged while a new type of object is introduced. This makes the software more manageable and easier to modify. The most commonly used OOP languages are Java, C++, C#, Python, PHP, Ruby, Perl and Objective-C. [3]

2.1.4 Model-View-Controller Architecture (MVC)

Model-View-Controller is a software architecture, by other words the structure of the system. Its main purpose is to separate the application logic from the user interface. It accomplishes this by separating the software into three parts: the **Model**, the **View**, and the **Controller**.

The **View** is the part of the software presented to the user. It handles the design and renders data from the model into a form or something else suitable for the user interface. A more, common name for the view is *GUI* or Graphical User Interface.

The **Model** contains the applications business logic and represents the “knowledge” of the software. It is responsible for maintaining the applications data integrity. If the software is connected to a database, all the manipulations and *SQL*-queries are executed in the model.

The **Controller** is the connection between the **View** and the **Model**. It handles user input, calls model objects, and returns the desired response to the view.

The essence of a MVC application is a stand-alone Model. Which in theory means the applications should be functional without a View/Controller. This result in the Model not being aware of the View/Controller and all the communication are directed through the Models *API*. This separation enables multiple Views/Controllers to communicate with the Model without changing the core functionality of the logic. The concept of a stand-alone model causes a few problems to occur. How is the user able to get the latest data if the model is unable to send it to the View?

The answer is by notifications. Due to the Model not being aware of the View/Controller it is not able to send the latest data. Instead, it transmits a notification regarding a changed state. The Model is unaware if anyone is listening to the notification but it will transmits

anyway. A listener is set up in the View/Controller to recognize a change in the Model and to make a call to fetch the latest data. [4] [5]

The following figure illustrates the separation logic:

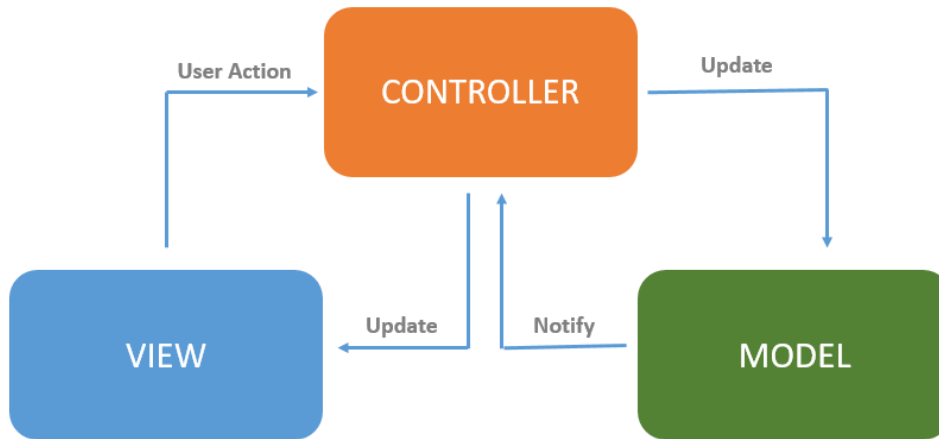


Figure 2: MVC components relationship

2.1.5 JSON and XML data

Javascript Object Notation, or **JSON**, is a syntax for exchanging and storing data. JSON data is a Javascript object converted into plain, lightweight text that can easily be stored or shared through a network connection. The data receiver can easily be parsed to a Javascript object. JSON is frequently used for *client/server* communications in web and mobile applications. [6] Below is an example of a JSON object including cars:

```

{
  "cars": {
    "car": [
      {
        "brand": "Mercedes",
        "model": "C200"
      },
      {
        "brand": "BMW",
        "model": "M5"
      },
      {
        "brand": "Audi",
        "model": "A3"
      }
    ]
  }
}
  
```

Figure 3: JSON cars object

Extensible Markup Language, or **XML**, was designed to store and transmit data. It is not as lightweight as JSON, which makes the data slower to process and transmit [7]. It is similar to HTML, with its brackets notations, and rather simple to read and understand, although JSON is often referred to as more straightforward. Below you can find the same cars object in XML format:

```
<cars>
  <car>
    <brand>Mercedes</brand> <model>C200</model>
  </car>
  <car>
    <brand>BMW</brand> <model>M5</model>
  </car>
  <car>
    <brand>Audi</brand> <model>A3</model>
  </car>
</cars>
```

Figure 4: XML cars object

JSON and XML is often debated and compared. Both languages has its pros and cons and in the end, it is all up to user preference. [8] Below is a side-by-side comparison of the two:

Javascript Object Notation	Extensible Markup Language
Simple to read/write	More difficult to read/write
Easy to learn	Easy to learn
Data-oriented	Document-oriented
No capability to display data	Capability to display data
Supports <i>arrays</i>	Does not support <i>arrays</i>
Less secure	More secure
Supports text and numbers	Supports text, numbers, images, charts , graphs etc.

Table 1: JSON vs XML [8]

2.1.6 Application Programming Interface (API)

An **API** (Application Programming Interface) is an *interface* allowing two software's to communicate. The most common communication is between *clients* and *servers*. The API interface, usually installed on the *server*-side, serves as a middle hand between communications. The API works as a security layer between the *client* and the *server* restricts the *client* from accessing the server directly.

In today's market, there are numerous types of APIs active and available. Programming languages, such as Java, uses *interfaces* (APIs) within *classes* that let objects talk to one another. APIs are used all over the web, letting users communicate with *servers* through websites and mobile applications. Every time a user logs in to an application using Facebook, an API handles the request, validates the credentials and returns the profile or an error message, e.g. "wrong password or username". An API consists of hardcoded methods which returns requested data or stores data in the *server* database. An API can be coded in multiple different language and is not tied to a specific *client* language, so every type of *client* is able to use the API. [9]

The most widely used Web APIs are Simple Object Access Protocol (SOAP) and Remote Procedure Call (RPC) and Representational State Transfer (REST), which according to Shana Pearlman [9], might be the most discussed API in the modern age. The following figure illustrates the communication:

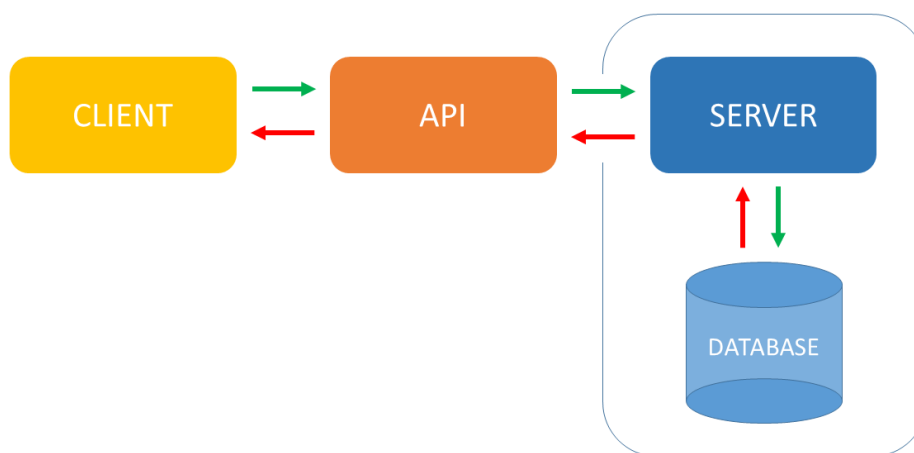


Figure 5: API communication

2.1.7 RESTful API

During the last few years, REST has evolved into a predominant Web Service design model [10]. It was first introduced in 2000 by Roy Fielding's academic dissertation, "Architectural Styles and the Design of Network-based Software Architecture", at the University of California.

The functionality of a REST API is rather simple. It uses explicit *HTTP* methods for communication. The API supports both *XML* and *JSON* and is set up to handle four different operations, depending on the protocol type.

The following example uses a Customer database for clarity:

1. GET: To retrieve a customer from the database.
2. POST: To create/add a new customer in the database
3. PUT: To edit/update an existing customer in the database
4. DELETE: To delete a customer from the database

The requests are sent by the *client*, to a specific URI and is received by the corresponding API method, which handles the database communication and executes the operations. The method return the response to the *client* who is unaware of what happened beyond the call. [10]

2.2 Magento Framework

Magento is an open-source, web application *framework* for E-commerce, developed in *PHP*. The framework was originally developed by an organization called Varen, Inc. The first version of the framework was released on March 31, 2008. Today Magento is the leading platform for open commerce innovation [11]. This section will illustrate the underlying structure of the Magento Framework and how a custom *module* can be implemented into the framework.

2.2.1 Magento Architecture (Model-View-Controller)

Model-View-Controller (MVC) is a software architecture designed to separate the applications **user interface**, **data access** and **business logic**. There are generally two types of MVC architectures: convention based and configuration based. Using the convention

based architecture, it is necessary to follow the core system instructions. While using the configuration based architecture, instructions are set up in a configuration file. Magento modules are configuration based.

The Magento *Modules* instructions are stored in a **config.xml** file, under the modules **etc** directory. The file acts like a map of the *module*, and tells the *framework* where to find all the components necessary for the *module* to work.

2.2.2 Magento Directory Structure

The object-oriented *PHP* framework is constructed by individual Modules. These Modules consists of files grouped together based on functionality. It differs from the typical *PHP* Model-View-Controller (MVC) application, where all the Controllers are collected in one folder, all the Models in another, etc.

All the Modules are divided into three different code pools, **core**, **community** and **local**.

The **core** folder holds all the Modules distributed by Magento and handles all the core functionality of the Framework.

The **community** folder holds all the Modules distributed by a third-party organization, such as installed extensions for expanded functionality.

The **local** folder holds all the Modules developed locally by an organization, such as the development of the *ERP* integration. [12]

All the Magento Modules are located under the “app” folder. As seen in the figure below:

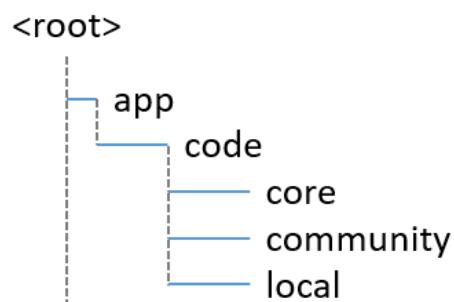


Figure 6: Magento CodePool Structure

2.2.3 Custom Module Directory Structure

The custom module developed was located under the local folder. Magento initiates the core folder first, the community folder second and the local folder last. By following this order, the framework prevents custom modules, in the developing stage, to interfere with the core modules necessary to run the framework flawlessly. By other word, a non-functioning custom module will not affect the core modules in a negative way.

In Magento, every Module is structured in the same manner, as seen in the figure below:

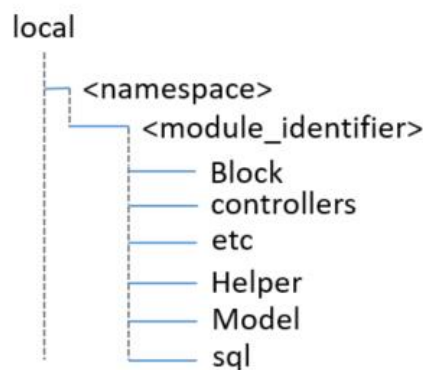


Figure 7: Magento Custom Module folder hierarchy

The Module namespace is the top level of the hierarchy. Regarding Magento development, the module namespace should refer to the founding organization, in this case Webcore. The unique namespace prevents name collisions between modules.

The Module identifier is the name of the module. The name should describe its purpose and functionality. For example, if the module handles payments the corresponding name should be Payments.

The Module itself consists of six different folders, **Block**, **controllers**, **etc**, **Helper**, **Model** and **sql**. This folder structure is a Magento standard and helps the framework recognize the different module components.

The **Block** is Magentos view. It is responsible for the rendering of content.

The **controllers** processes user input, by other words URLs. When an object in the online store is pressed, a request to the corresponding controller is sent, and the controller returns the correct view for the user. For example, a customer clicks on a product to get more in

depth information about the product, the controller processes the request and finds the correct View based on the URL.

The **etc** directory consists of configuration *XML* files for the modules. The configuration files act sort of like a map over the module, it tells the framework about every component, router, observer action, etc. [13]

The three main configuration files:

- ✓ **adminhtml.xml**: An optional file, which manages the *backend* Menu control and ACL (access control list). New system configuration tabs for the custom modules must be implemented in this file.
- ✓ **config.xml**: An obligatory file, which manages all the base configuration for the module. The file informs the framework about the modules structure and how to access components such as blocks, helpers, models, observers, layout files, etc.
- ✓ **system.xml**: An optional file, containing all configurations inserted into the *backend* configuration page, and defines all form-input fields, and its data.

The **Helper** directory contain files used adding new *classes* and overriding functions in the Module. Modifying Magento core files is not recommended, that is where helpers come in handy. Helpers are often used for translation but usually not necessary for a Module to work.

The **Model** is, according to Alan Storm, one of the most crucial Module components. The model contains the objects of data and handles the business logic of the Module and is often used for manipulating databases or sending *API* requests to distant servers.

The **sql** directory contain the setup files, which are executed on installation. A more in-depth explanation of these files will be presented in the next section. [14] [15]

2.3 Installation

This section will discuss the setup of the module. For example, how the Magento Framework finds the location of relevant files in the module and how the module is installed and updated.

2.3.1 Setup Components

As previously mentioned, the config.xml file is located in the modules **etc** directory. The *XML* file contains the configuration of the module; by other words it represents a map of the module and all of its components. Every component needs to be declared in the config.xml file, otherwise the framework will not know about their existence.

On the next page is an example declaration of a **Model**, **Resource Model** and a **Database Table**. Note that every name in bold-italic font is unique to every Module and chosen by the developer.

```
<models>
  <moduleName>
    <class>Namespace_ModuleName_Model</class>
    <resourceModel>moduleName_mysql4</resourceModel>
  </moduleName>
  <moduleName_mysql4>
    <class>Namespace_ModuleName_Model_Mysql4</class>
    <entities>
      <entityname>
        <table>table_name</table>
      </entityname>
    </entities>
  </moduleName_mysql4>
</models>
```

Other components such as, **controllers**, **Helpers**, **Blocks** and **Observers** are declared in the same manner. [14]

2.3.2 Setup Resources

Magento uses Setup Resources to keep the development and production database in sync. It is a system that uses versioned resource migration *scripts*.

The Setup Resource must be declared in the config.xml file to let the *framework* know that that the module has an installer script.

Below is an example of the setup declaration. Note that every name in bold-italic font is unique to every Module and chosen by the developer:

```
<global>
  <resources>
    <Modulename_setup>
      <setup>
        <module>Namespace_Modulename</module>
        <class>Namespace_Modulename_Model_Resource_Setup</class>
      </setup>
    </Modulename_setup>
  </resources>
</global>
```

As seen in the example above the Setup Resource *Class* is located under the following path:

Namespace\Modulename\Model\Resource\Setup.php

The *class* must be declared in the Setup.php file for the module to work, and it should extend the “Mage_Core_Model_Resource_Setup” class. [16]

Below is an example of the empty, declared *class*:

```
class Namespace_Modulename_Model_Resource_Setup
extends Mage_Core_Model_Resource_Setup {
    // code here ..
}
```

2.3.3 Installer Scripts

When the Module is installed and activated for the first time an installer script is executed on initial page refresh. The file containing the installer script is located under the following folder path:

\local\namespace\modulename\sql\modulename_setup\

and the file should be named in the following manner:

mysql4-install-0.1.0.php

The name is defined by a Magento standard naming convention to assist the *framework* identifying the install *script*. The version number must correspond to the version number set in the config.xml file for a successful installation.

The installer script consists of PHP/SQL code performing a specific task, needed for the module to work properly, for example creating a table in the database or adding an attribute for a customer/product.

Example of an install *script* creating a table for “cars” in the database:

```
// creates an object instantiated from the Setup class
$installer = $this;
// starts the setup and opens the database connection
$installer->startSetup();
// executes the SQL query
$installer->run("
    CREATE TABLE `{$installer->getTable('namespace/modulename')}` (
        `cars_id` int(11) NOT NULL auto_increment,
        `brand` text,
        `model` text,
        `yearmodel` int(4),
        `timestamp` timestamp NOT NULL default CURRENT_TIMESTAMP,
        PRIMARY KEY (`cars_id`)
    ) ENGINE=InnoDB DEFAULT CHARSET=utf8;

    INSERT INTO `{$installer->getTable('namespace/modulename')}` VALUES
    (1, 'Audi', 'A8', 2008, '2018-03-09 14:12:30');
");
// ends the setup and terminates the database connection
$installer->endSetup();
```

The install *script* is only executed once during installation and to add new features to the module or alter database tables an upgrade script must be executed. [16]

2.3.4 Upgrade Scripts

Once the installer *script* has been executed, it cannot be initialized again, but if you need to alter the module structure after installation Magento's Setup Resources supports running scripts to upgrade the module.

Upgrade scripts and installer scripts possess multiple similarities. They are located in the same folder, but named slightly differently. Below is a name example of an upgrade script for the previously described installer script:

mysql4-upgrade-0.1.0-0.1.1.php

The name informs the *framework* that it contains *SQL* queries, that it is an upgrade script and the change in versions (version 0.1.0 to 0.1.1). To execute the upgrade script the module version must be changed in the config.xml file. In this case from 0.1.0 to 0.1.1.

An example of an upgrade script making the “cars” brand column obligatory:

```
// creates an object instantiated from the Setup class
$installer = $this;
// starts the setup and opens the database connection
$installer->startSetup();
// executes the SQL query
$installer->getConnection()
    ->changeColumn($installer->getTable('namespace/modulename'),
'brand', 'brand', array(
    'nullable' => false,
    'comment' => 'this is the car brand'
))
);
// ends the setup and terminates the database connection
$installer->endSetup();
```

For the upgrade script to execute, the version number declared in the config.xml must be altered to the corresponding version number of the upgrade script. [16]

2.4 Briox ERP-system

Briox offers a web-based ERP-system for organizations, associations, accounting bureaus and schools. Due to the software being web-based, it is accessible from any device with a steady internet connection and it stores data on its *servers*. [17]

Briox is a highly flexible ERP-system offering numerous softwares such as:

- ✓ Accounting
- ✓ Billing
- ✓ Order
- ✓ Scanner/Archive
- ✓ Connected Documents
- ✓ Time Reporting
- ✓ CRM
- ✓ Purchase Order

2.4.1 Briox RESTful API

Briox provides a REST API for their customers, used for pushing and fetching data from the database. By using the API the developer can manipulate accounts, cost centers, customers, customer invoices, financial year, journals, linked documents, opening balance, payments, projects, sales orders, suppliers and tokens. The API documentation can be found at the following address:

<https://apidoc-fi.briox.services/#/>

The Briox API communicates with *JSON* data. It uses GET, POST, PUT and DELETE requests to alter the database.

To create an invoice for a customer, briox must receive a JSON object with the following structure:

```
{
  "customerinvoice": {
    "customerid": "11",
    "invoicedate": "2017-10-19",
    "paymentdate": "2017-10-19",
    "deliverydate": "2017-10-19",
    "admfee": "2234",
    "shipping": "2234",
    "orderno": "1",
    "paymentterm": "PF",
    "shippingmethod": "N",
    "shippingcondition": "CIP",
    "yourreference": "Benny Jackson",
    "ourreference": "Benny Jackson",
    "costcenter": "CC",
    "project": "1",
    "invoicetext": "A special customer invoice description",
    "customer_address": [
      {
        "addressline1": "Albertinkatu 36 B",
        "addressline2": "Albertinkatu 36 B",
        "zip": "00180",
        "city": "Helsinki",
        "country": "Finland",
        "county": "null"
      }
    ],
    "customer_deladdress": [
      {
        "name": "John Doe",
        "addressline1": "Albertinkatu 36 B",
        "addressline2": "Albertinkatu 36 B",
        "zip": "00180",
        "city": "Helsinki",
        "country": "Finland",
        "county": "null"
      }
    ]
  }
}
```

```

],
"invoice_rows": [
  {
    "itemno": "30",
    "description": "Order description",
    "unit": "h",
    "amount": "221",
    "price": "333",
    "discount": {
      "value": "0",
      "type": "1"
    },
    "account": "3000",
    "costcenter": "string",
    "project": "21"
  }
]
}
}

```

If the object differs or some of the keys are missing a value, the API returns an error message and no database manipulation is executed. To create another “form”, for example sales order or invoice, a JSON object containing different key values are posted to the API. [18]

3 Implementation

This chapter will be illustrating the implementation of the *Module*. It will explain the installation, the authentication process, how the different methods work independently and with each other, and the modules chronological workflow from execution to termination. It gives the reader a good understanding of how the extension works. Every code section will not be identical to the extensions source code due to copyright reasons, but still represent the same functionality.

3.1 Configuration

The *module* configuration files are located in the **etc** folder, according to the Magento standards. The folder consists of three different configuration files, **config.xml**, **adminhtml.xml** and **system.xml**. These three files have different tasks regarding the configuration. One of them configures the structure of the *module*, while the other two configures the *backend* GUI for dynamic admin configurations.

3.1.1 Config.xml

This file is the most important file for a *module* to work. The Magento *Framework* uses this file to establish all the numerous module components. The file is an *XML* file declaring the version number of the module and the location of every component, such as model, resources, observers etc.

An example of the observers declared in the config file:

```
<events>
  <admin_system_config_changed_section_brxintegration_connect>
    <observers>
      <brxintegration>
        <type>singleton</type>
        <class>brxintegration/observer</class>
        <method>handle_adminSystemConfigChangedConnect</method>
      </brxintegration>
    </observers>
  </admin_system_config_changed_section_brxintegration_connect>
  <admin_system_config_changed_section_brxintegration_settings>
    <observers>
      <brxintegration>
        <type>singleton</type>
        <class>brxintegration/observer</class>
        <method>handle_adminSystemConfigChangedSettings</method>
      </brxintegration>
    </observers>
  </admin_system_config_changed_section_brxintegration_settings>
</events>
```

As seen above, there are two types of observers active in the module. They are triggered when a new configuration is saved in the admin *backend* panel. The first one is used when

authenticating and the latter one is used for the custom *variables*. The class brackets specify where the observer *class* is located and the method specifies which method to be executed when the configuration is saved.

3.1.2 Adminhtml.xml

The Adminhtml file configures the tabs in the left side bar in the admin *backend* panel. The module uses two different tab views, one for authorization and one for configuration.

Below is a copy of the Adminhtml.xml content:

```
<?xml version="1.0"?>
<config>
  <acl>
    <resources>
      <admin>
        <children>
          <system>
            <children>
              <config>
                <children>
                  <brxintegration_connect>
                    <title>BrxIntegration Connect Section</title>
                    <sort_order>1</sort_order>
                  </brxintegration_connect>
                  <brxintegration_settings>
                    <title>BrxIntegration Settings Section</title>
                    <sort_order>2</sort_order>
                  </brxintegration_settings>
                </children>
              </config>
            </children>
          </system>
        </children>
      </admin>
    </resources>
  </acl>
</config>
```

The different tabs are divided inside two separate brackets, “brxintegration_connect” and “brxintegration_settings”.

The Authorization form, called “brxintegration_connect”, requests an authentication *token* and the Briox account ID.

The Configuration form, called “brxintegration_settings”, requests organization-specific configuration, dynamic *variables* that varies between organizations. For example, the company reference, shipping methods, payment terms and sales accounts.

3.1.3 System.xml

The System.xml file structures the layout of the different forms, by the use of *XML* data. It declares the sections and the labels of the different tabs.

Below is an example of a “Connect” tab declaration

```
<?xml version="1.0"?>
<sections>
  <brxintegration_connect>
    <label>Connect</label> <!-- The tab label -->
    <tab>brxintegration</tab>
    <frontend_type>text</frontend_type>
    <sort_order>1</sort_order> <!-- the sort order -->
    <show_in_default>1</show_in_default>
    <show_in_website>1</show_in_website>
    <show_in_store>1</show_in_store>
    <groups>
      <!-- configuration of form fields -->
    </groups>
  </brxintegration_connect>
</sections>
```

Inside the “groups” brackets, the structure of the form is declared. The declaration consists of a section selector, a heading for the section, the sort order its field. The fields acts as rows in the form and are produced by similar code as the section. Each row has a selector name, a label, a sort order and a frontend type. The frontend type declares what type of input to be used, whether it is a dropdown list, an image upload or a regular text input.

Below is an example of the authentication form:

```
<section_one translate="label"> <!-- Section Selector -->
<label>Connect</label> <!-- Form Heading -->
<sort_order>1</sort_order>
<show_in_default>1</show_in_default>
<show_in_website>1</show_in_website>
<show_in_store>1</show_in_store>
<fields>
  <account_id> <!-- Field Selector -->
    <label>Account ID</label> <!-- Field Heading -->
    <frontend_type>text</frontend_type> <!-- Input Type -->
    <sort_order>1</sort_order>
    <show_in_default>1</show_in_default>
    <show_in_website>1</show_in_website>
    <show_in_store>1</show_in_store>
  </account_id>
  <auth_token> <!-- Field Selector -->
    <label>Authentication Token</label> <!-- Field Heading -->
    <frontend_type>text</frontend_type> <!-- Input Type -->
    <sort_order>2</sort_order>
    <show_in_default>1</show_in_default>
    <show_in_website>1</show_in_website>
    <show_in_store>1</show_in_store>>
  </auth_token>
</fields>
</section_one>
```

The code example results in the following layout:

The screenshot displays the 'Connect' form layout in a Magento 2 admin interface. On the left, the 'Configuration' menu is visible, with 'BRIX CONNECT' expanded and 'Connect' selected. The main content area shows the 'Connect' form. The form has a heading 'Connect' and two input fields: 'Authentication Token' and 'Konto-ID'. The 'Authentication Token' field is labeled 'Field Headings' and the 'Konto-ID' field is labeled 'Input Fields'. A 'Spara konfiguration' button is visible in the top right corner.

Figure 8: Connect Form Layout

The configuration form is rendered by the same *XML*-file and constructed by using the same methods as the authentication form. It is used for configuring the extension with organization-specific data.

Inställningar

Konfigurera data export

Export orders with status:

Konfigurera försäljningskonton:

Försäljning varor, 24%:
▲ ex 3000

Försäljning varor EU, 0%:
▲ ex 3000

Varuförsäljning till Åland:
▲ ex 3000

Fakturerering

Betalningsvillkor:

Frakt

Method 1:
▲ Briox ID for shipping method "itellaEconomy"

Method 2:
▲ Briox ID for shipping method "itellaExpressBusinessDay"

Method 3:
▲ Briox ID for shipping method "itellaExpressBusinessDayParcel"

Method 4:
▲ Briox ID for shipping method "itellaPriority"

Method 5:
▲ No method activated

Övrig information

Briox referens:

Figure 9: Configuration Form Layout

The organization is able to configure which orders to be exported, the sales account, payment terms and the IDs of the shipping methods configured in the *ERP*-system. When the form is saved, the data gets stored in the *XML*-file and can then be used by the extension.

Without this configuration, the *ERP*-system would not be able to recognize which sales account the customer belongs to, or what shipping method the customer has chosen.

3.2 Install Script

For the module to work as requested it needs to store the *API* tokens in a database protected by the server, and all the registered customers must have a synchronized customer number attribute declared in both Briox and Magento. Briox already sets an increasing customer number for every new customer so the most logical way is to add a new custom attribute for every registered customer in Magento, containing the Briox customer number. To achieve this an install script was used.

3.2.1 Creating the database table

The easiest and most logical way to store the *API Tokens* was by inserting them into a database table, protected by the *server*. The table is produced by the install *script* on initial activation. The script uses a combination of *PHP* and *SQL* language to create the table.

Example of install script creating a new database table for storing the tokens:

```
// initializes the setup
$installer = $this;
$installer->startSetup();
// executes the SQL command
$installer->run("
    CREATE TABLE `{ $installer->getTable('modulename/tablename')}` (
        `id` int NOT NULL auto_increment,
        `client_id` varchar(10),
        `access_token` varchar(50),
        `expire_date` datetime,
        `refresh_token` varchar(50),
        `timestamp` timestamp NOT NULL default CURRENT_TIMESTAMP,
        PRIMARY KEY (`id`)
    ) ENGINE=InnoDB DEFAULT CHARSET=utf8;

// closes the installer
$installer->endSetup();
```

3.2.2 Adding custom attributes

For the module to operate properly, every registered customer needs a custom attribute for his or her corresponding ID in Briox. The attribute is added through the install script.

Example of *script* for adding a new attribute briox_custno:

```
// initializes the installer
$installer = $this;
$installer->startSetup();

// creates an instance to the setup class
$setup = new Mage_Eav_Model_Entity_Setup('core_setup');

// adds the new attribute description
$setup->addAttribute("customer", "briox_custno", array(
    "type"      => "text",
    "backend"   => "",
    "label"     => "Briox custno",
    "input"     => "text",
    "source"    => "",
    "visible"   => true,
    "required"  => false,
    "default"   => "",
    "frontend"  => "",
    "unique"    => true,
    "note"      => "Briox customer number"
));

//specifiec where to be used
$forms = array(
    'adminhtml_customer',
    'customer_account_edit'
);

// gets instance of the new attribute
$attribute = Mage::getSingleton('eav/config')->getAttribute('customer',
    'briox_custno');

// sets the data
```

```

$attribute->setData('used_in_forms', $forms);

// saves the attribute
$attribute->save();

// suspends the installer
$installer->endSetup();

```

The attribute is empty for every customer already registered and has to be manually transferred. Although every time a customer makes a purchase the values get synchronized, regardless if the customer is already registered in Briox or not.

3.3 Authentication

After the installation of the *module*, the administrator has to authenticate the Ecommerce store with the Briox *ERP*-system. The authentication process for the integration is achieved both manually, and programmatically by the following steps.

3.3.1 Authentication Token

The first step of the authentication is generating and manually transferring the Authentication *Token* from the *backend* of Briox to the Modules configuration panel. The token is generated by navigating to the “Lägg till licens/program”-tab in the dashboard, opening up the setting for a user and scrolling down to the bottom of the settings form. Located at the bottom is a closed row called “Applikationers token” and by clicking the heading the consumer is presented with a tokens panel. By clicking the “Generera token” button the consumer is presented with the generated token.

The following figure is a snippet of the panel with the button highlighted.

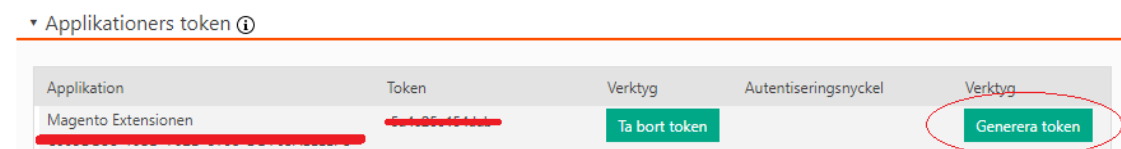


Figure 10: Generating the Authentication Token

The token is revealed to the left of the button and it will be valid until the REST API in a request receives it for exchange of the Access Token.

The token and account ID is copied and transferred into the Module configuration in the *Magento Backend* (System > Configuration > Briox Connect > Connect).

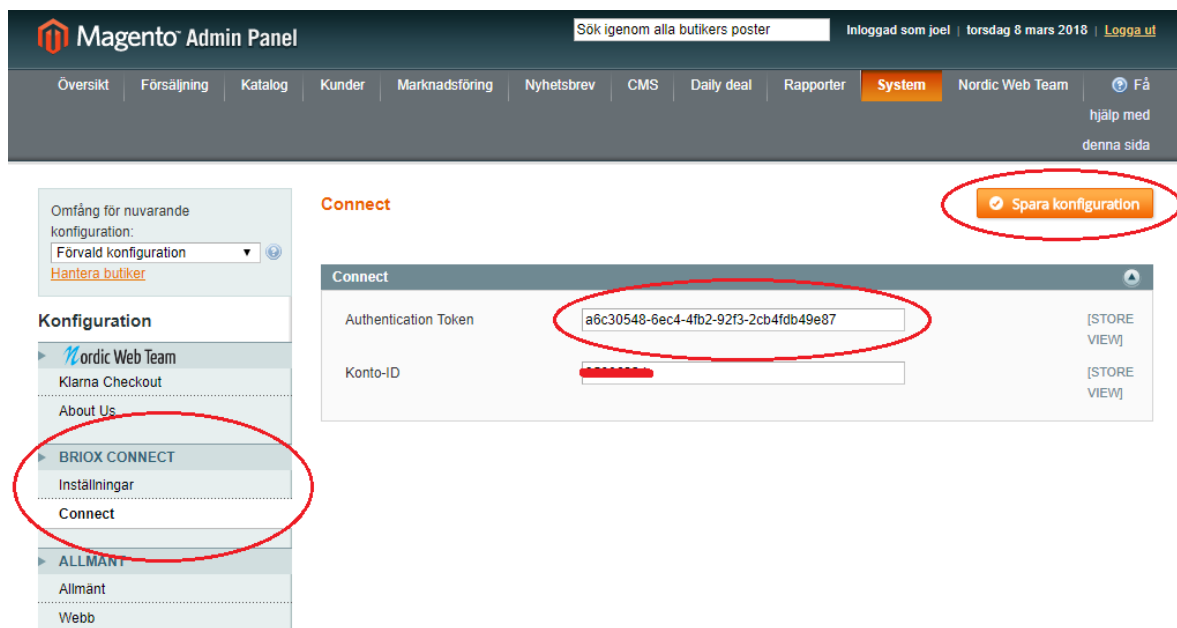


Figure 11: Authenticating in the Admin Backend

3.3.2 Retrieving the Access Token

By clicking the “Spara konfiguration” button, an observer is detecting a change in the Authentication form. The observer calls the initial method of the authentication *class*. The method retrieves the tokens from the form and sends them as parameters to a method called **getTokensBriox**.

The Observer function:

```
// the observer function
public function handle_adminSystemConfigChangedConnect(){

    // instantiating the authentication
    Mage::getModel('brxintegration/auth')->authenticate();

}
```

The authentication function:

```
public function authenticate(){

    // retrieve the Authentication Token
    $this->_authtoken =
Mage::getStoreConfig('brxintegration_connect/section_one/auth_token');
    // retrieve the Account ID
    $this->_accountid =
Mage::getStoreConfig('brxintegration_connect/section_one/account_id');

    // calling the prepare function
    $this->getTokensBriox($this->_authtoken, $this->_accountid);

}
```

The **getTokensBriox**-method prepares the CURL request by establishing the URL, the header and body. The authentication token and account ID is passed as the body object. After the preparation, the method instantiates the **curlExecute**-method in the CURL *class* to perform the communication and stores the retuning object, including the tokens and expire date, in an *array*. The array is parsed and every value is stored into unique *variables*, which gets be stored in the database table created by the installer *script*. Depending on the outcome of the insertion, a success or error message is presented to the user in the *backend*.

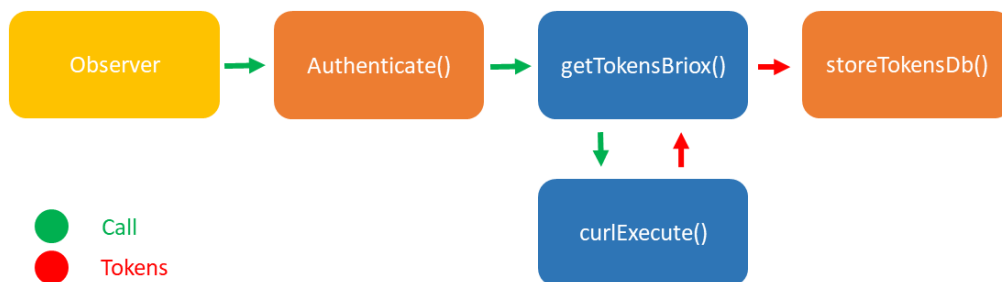


Figure 12: Authentication workflow

3.4 Initiation

The first and main function in the chain of events is called **exportOrders**. It organizes the workflow of the software in a chronological order. It starts of by validating the *tokens*, and refreshes them if necessary. After the validation, it fetches the orders from a method inside the same *class* and loops trough every order. For every loop, it calls a method to check if the customer exists, and calls another method to create a new customer in Briox if necessary. When the customer is reviewed, it calls a method to create a sales order and invoice to be posted. On termination, it saves a timestamp in the database that is used for the next execution

The next few sections will explain the process in more detail.

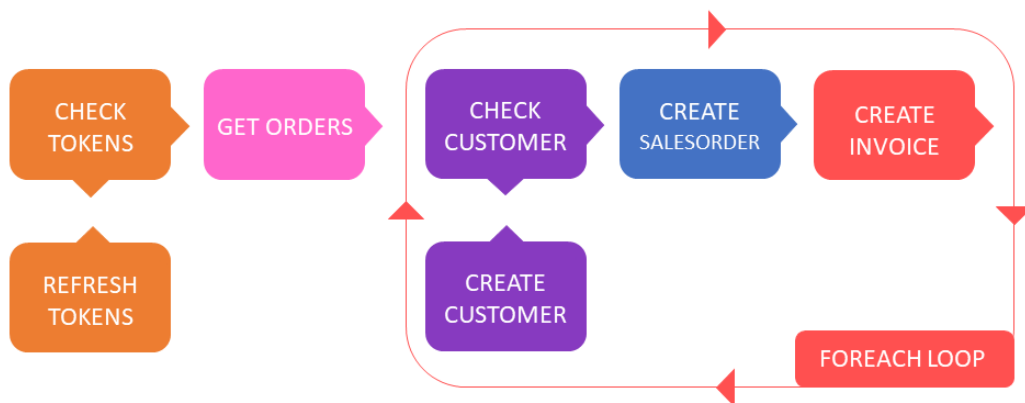


Figure 13: Main function workflow

3.4.1 Token refresh

The main function starts by calling a function to check expire dates of the *tokens* stored in the database. If the expire date is within 24 hours the function returns a false statement and the main function does a maneuver to refresh the tokens, by calling a method in the *CURL class* that sends a refresh token call to the *REST API*.

3.4.2 Collecting all orders

The next step for the export is fetching all the orders, placed since the last execution, from the web shop. The main function calls a method called **getOrders**, implemented to store data

of all orders into an *array*. The orders are collected from the sales/order model using the **getCollection**-method. The method supports attribute filter such as “created at” and “status”. The filters are applied to get the last unexported orders with a status of complete. The orders are stored into an object called “order”, and the orders products are retrieved with the **getItemsCollection**-method within the order class. The relevant data is then stored in a *multidimensional associative array* to minimize the size of the object to process.

The following snippet illustrates the *array* called “order_data”:

```
$order_data[0] = array(

    "name" => $order->getBillingAddress()->getName(),
    "customerid" => $this->getCustomerId($customer_email),
    "phone" => $order->getBillingAddress()->getTelephone(),
    "email" => $customer_email,
    "vatnbr" => $order->getCustomerTaxvat(),
    "currency" => $order->getOrderCurrencyCode(),
    "deliveryname" => $order->getShippingAddress()->getName(),
    "deliveryphone" => $order->getShippingAddress()->getTelephone(),
    "customerref" => $order->getCustomerName(),
    "invoicediscount" => number_format($order->getDiscountAmount(),2),
    "shippingcharge" => number_format($order->getShippingAmount(),2),
    "billingadress" => $order->getBillingAddress()->getStreetFull(),
    "billingZIP" => $order->getBillingAddress()->getPostcode(),
    "billingcity" => $order->getBillingAddress()->getCity(),
    "billingcountry" => $order->getBillingAddress()->getCountryId(),
    "shippingname" => $order->getShippingAddress()->getName(),
    "shippingadress" => $order->getShippingAddress()->getStreetFull(),
    "shippingZIP" => $order->getShippingAddress()->getPostcode(),
    "shippingcity" => $order->getShippingAddress()->getCity(),
    "shippingcountry" => $order->getShippingAddress()->getCountryId(),
    "orderdate" => $order->getCreatedAt(),
    "amount " => $this->getQtyOrdered($order),
    "price " => number_format($order->getGrandTotal(),2),
    "shippingmethod" => $order->getShippingMethod(),
    "billingAddressname " => $order->getBillingAddress()->getName(),
    "shippingAddressname " => $order->getShippingAddress()->getName(),

    "items" => array(
        0 => array(
```



```

        'product_name' => $item->getName(),
        'product_id' => $item->product_id,
        'briox_id' => $brioxId,
        'ean' => $ean,
        'unit' => $unit,
        'product_sku' => $productSku,
        'quantity_ordered' => round($item->qty_ordered),
        'discount_amount' => $item->discount_amount,
        'discount_percent' => $item->discount_percent,
        'base_price' => $item->base_price,
        'base_price_incl_tax' => $item->base_price_incl_tax,
        'tax_percent' => $item->tax_percent,
        'tax_amount' => $item->tax_amount,
        'free_shipping' => $item->free_shipping,
    ),
),
);

```

The “customerid”, “amount”, “briox_id”, “ean”, “unit” and “product_sku” are the only *variables* not accessible through the order or item instance. These variables are retrieved by custom functions, or from different *classes*. The “items” array is only declared inside the “order_data” array and filled from a separate foreach-function after the declaration. The function return the “order_data” array.

3.4.3 Check for customer in Briox

Before any data export can occur, the *module* needs to know if the customer already exists in Briox, or if it should be created. This action is needed to prevent duplicates of customers in the Briox database.

The main function loops through every individual order in the “order_data” *array* and calls a method inside the *CURL-class* to check if the customer exists. The method needs an email address as an in-parameter and sends a GET-request to Briox *API* with the email address as a parameter. If the email address is found in the customer register, Briox responds with the customer *JSON* object. Otherwise, it responds with an empty object including an error message.

The method returns the JSON object and the main function performs a check to decide if the customer exists based on the object data.

The following code snippet is an example of the condition:

```
<?php
$customerEmail = $order["email"];
$curlInstance = Mage::getModel('brxintegration/curl');
$custno = $instance->checkCustomerBriox($customeremail);

if($custno === false){
    //Customer does not exist and needs to be added
    $customerInstance = Mage::getModel('brxintegration/customer');
    $order['custno'] = $customerInstance->postCustomer($order);
}
else{
    //Customer exists, add custno to order
    $order['custno'] = $custno;
}
```

Scenario 1, the customer does exists:

If the customer already exists in the Briox database, the JSON object is not empty. The main function adds the Briox customer number (custno) to the active order array before advancing.

Scenario 2, the customer does NOT exists:

If the customer does not exists in the Briox database, the *JSON* object is empty. If the object is empty, the if-condition is true and the code inside the scope is executed. The customer is created by calling the **postCustomer**-method of the *Customer-class*. The method return the Briox customer number, which is added to the active order array before advancing.

3.4.4 Creating the Customer

If the active customer is not found in the Briox database, the returned customer number will be null, and the *Customer-class* will get instantiated. The class handle the process of creating the customer in Briox by using its own methods and the *CURL-class* for the *API* communication.

3.4.5 Preparing and posting the Customer

The Customer-class makes use of two major methods to operate, **postCustomer** and **setCustomer**. A third minor method is used to set the custom attribute called “Briox custno” to the number assigned to the new customer in Briox.

The **postCustomer**-method handles the chronological order of the process. It receives the order object as a parameter and passes it along to the **setCustomer**-method. The method creates and assigns values to a *JSON* object, which in turn will be posted to the Briox *API*. The object is filled with data from the order and data configured by the user in the admin configuration panel and retrieved from a class called Settings, such as sales account, references etc.

After the customer object is set, it is passed as a parameter to a method called **addCustomer** in the CURL-class.

3.4.6 Creating the Salesorder and Invoice

When the active order contains the Briox “custno”, a salesorder and an invoice can be created for the corresponding customer. The salesorder and invoice is set up by different *classes* and initialized by a call to the corresponding POST-methods, **postSalesorder** and **postInvoice**. The methods returns **true** if the object was successfully posted to Briox and **false** if an error occurred.

```
<?php
$salesorderInstance = Mage::getModel('brxintegration/salesorder');
if(!$salesorderInstance->postSalesorder($order)){

}

$invoiceInstance = Mage::getModel('brxintegration/invoice');
if(!$invoiceInstance->postInvoice($order)){

}
```

3.4.7 Preparing and posting the Salesorder

When posting a sales order, Briox is configured to receive a *JSON* object called “salesorder”. The sales order is prepared and posted from a *class* called Salesorder,

The Salesorder-class consists of two method, **postSalesorder** and **setSalesorder**. The **postSalesorder**-method is called upon initialization and handles the chronological workflow of the process. It receives the order and stores it in a variable. To set the format of the sales order, it calls the **setSalesorder**-method. It finishes off by calling the **addSalesorderInvoice**-method in the CURL-class, and passes the prepared sales order, along with a “salesorder” string.

The example of the **postSalesorder** method:

```
public function postSalesorder($order){
    $this->_order = $order;
    $this->setSalesorder();
    $curlInstance = Mage::getModel('brxintegration/curl');
    if($curlInstance->addSalesorderInvoice($this->_new_salesorder,
    "salesorder")){
        return true;
    }
    else{
        return false;
    }
}
```

The **setSalesorder**-method has one duty, to fill the JSON object with data. It is initialized by the **postSalesorder** method and starts by gathering the variables (“ourreference”, “salesaccount”, “shippingmethod, “paymentterms”) configured in the admin backend panel, from the **Settings** class.

Example of code retrieving payment terms configured in the backend:

```
$settingsInstance = Mage::getSingleton('brxintegration/settings');
$paymentterms = $settingsInstance->getPaymenttermsSettingsConfig();
```

After all the user-configured settings are retrieved, the method fills a class-member *array* with the variables and values from the order:

The sales order array:

```
$this->_new_salesorder = array(
    'salesorder' => array(
        'id' => '',
        'customerid' => $this->_order['custno'],
        'orderdate' => $this->_order['orderdate'],
        'deliverydate' => '2017-12-10',
        'admfee' => '0',
        'shipping' => $this->_order['shippingcharge'],
        'orderno' => '11',
        'paymentterm' => $paymentterms, //backend config
        'shippingmethod' => $shippingmethod,
        'shippingcondition' => 'DDU',
        'yourreference' => $this->_order['name'],
        'ourreference' => $ourreference,
        'costcenter' => '',
        'project' => '',
        'ordertext' => '',
        'customer_address' => array(
            'addressline1' => $this->_order['billingadress'],
            'addressline2' => '',
            'zip' => $this->_order['billingZIP'],
            'city' => $this->_order['billingcity'],
            'country' => $this->_order['billingcountry'],
            'county' => 'null',
        ),
        'customer_deladdress' => array(
            'name' => $this->_order['name'],
            'addressline1' => $this->_order['shippingadress'],
            'addressline2' => '',
            'zip' => $this->_order['shippingZIP'],
            'city' => $this->_order['shippingcity'],
            'country' => $this->_order['country'],
            'county' => 'null',
        ),
        'order_rows' => array(),
    ),
);
```

```

    ),
);

$orderItems = $this->_order['items'];
for ($i=0; $i < count($orderItems) ; $i++) {
    $this->_new_salesorder['salesorder']['order_rows'][] = array(
        'itemno' => $orderItems[$i]['product_id'],
        'description' => $orderItems[$i]['product_name'],
        'unit' => 'kpl',
        'amount' => $orderItems[$i]['quantity_ordered'],
        'price' => $orderItems[$i]['base_price_incl_tax'],
        'discount' => array(
            'value' => $orderItems[$i]['discount_amount'],
            'type' => '0',
        ),
        'account' => $salesaccount,
        'costcenter' => '2000',
        'project' => '',
    );
}

```

3.4.8 Preparing and posting the Invoice

The sales order and invoice object format expected by the Briox *API* are nearly identical. Therefore, the *classes* preparing and posting the data are similar.

Just as the Salesorder-class, the Invoice-class consists of two methods, **postInvoice** and **setInvoice**. The **postInvoice**-method is called upon initialization and handles the workflow of the action. It receives the order and stores it in a variable. To set the format of the invoice, it calls the **setInvoice**-method. It finishes off by calling the **addSalesorderInvoice**-method in the CURL-class, and passes the prepared invoice, along with a “customerinvoice” string.

The **postInvoice** method:

```
public function postInvoice($order){
    $this->_order = $order;
    $this->setInvoice();
    $curlInstance = Mage::getModel('brxintegration/curl');
    if($curlInstance->addSalesorderInvoice($this->_new_salesorder,
    "customerinvoice")){
        return true;
    }
    else{
        return false;
    }
}
```

Just as the **setSalesorder**-method, the **setInvoice**-method has one task, to fill the *JSON* object with data. It is initialized by the **postInvoice**-method and starts by gathering the variables (“ourreference”, “salesaccount”, “shippingmethod”, “paymentterms”) configured in the admin backend panel, from the Settings-class.

Example of code retrieving payment terms configured in the backend:

```
$settingsInstance = Mage::getSingleton('brxintegration/settings');
$paymentterms = $settingsInstance->getPaymenttermsSettingsConfig();
```

After all the user-configured settings are retrieved, the method fills a class-member *array* with the variables and values from the order:

The customer invoice array:

```
$this->_new_invoice = array(
    'customerinvoice' => array(
        'customerid' => $this->_order['custno'],
        'invoicedate' => $this->_order['orderdate'],
        'deliverydate' => '2017-12-10',
        'admfee' => '0',
        'shipping' => $this->_order['shippingcharge'],
        'orderno' => '1',
```

```

'paymentterm' => $paymentterms, //backend config
'shippingmethod' => $shippingmethod,
'shippingcondition' => 'DDU',
'yourreference' => $this->_order['name'],
'ourreference' => $ourreference,
'costcenter' => '',
'invoicetext' => '',
'customer_address' => array(
    'addressline1' => $this->_order['billingaddress'],
    'addressline2' => '',
    'zip' => $this->_order['billingZIP'],
    'city' => $this->_order['billingcity'],
    'country' => $this->_order['billingcountry'],
    'county' => 'null',
),
'customer_deladdress' => array(
    'name' => $this->_order['name'],
    'addressline1' => $this->_order['shippingaddress'],
    'addressline2' => '',
    'zip' => $this->_order['shippingZIP'],
    'city' => $this->_order['shippingcity'],
    'country' => $this->_order['country'],
    'county' => 'null',
),
'invoice_rows' => array(),
),
);

$orderItems = $this->_order['items'];

for ($i=0; $i < count($orderItems) ; $i++) {
    $this->_new_invoice ['customerinvoice']['invoice_rows'][] = array(
        'itemno' => $orderItems[$i]['product_id'],
        'description' => $orderItems[$i]['product_name'],
        'unit' => 'kpl',
        'amount' => $orderItems[$i]['quantity_ordered'],
        'price' => $orderItems[$i]['base_price_incl_tax'],
        'discount' => array(
            'value' => $orderItems[$i]['discount_amount'],
            'type' => '0',

```



```

    ),
    'account' => $salesaccount,
    'costcenter' => '2000',
  );
}

```

3.5 CURL Requests

Every *API* call is prepared and executed from the same model class called CURL. The class consists of five methods: **refreshTokens**, **checkCustomerBriox**, **addCustomer**, **addSalesorderInvoice** and **curlExcecute**. The first four methods mentioned is preparing the API request in its own unique manner, and the last one sends the requests. When the four methods have prepared the message header, message body and URL, they call the execution method and passes the prepared variables as parameters. The executing method return the API response.

The following figure demonstrates the functionality within the curl class:

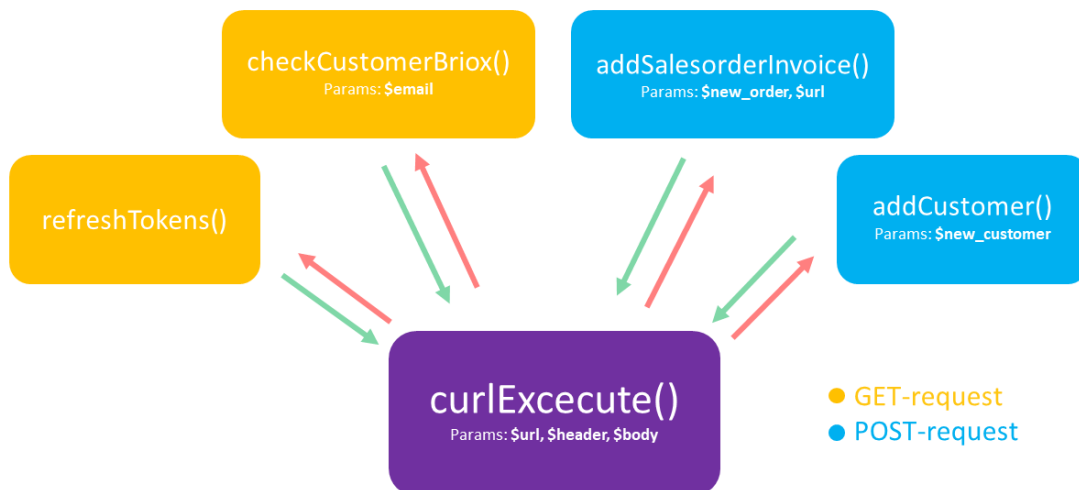


Figure 14: The CURL Class and its Method

3.5.1 Executing the HTTP request

CurlExecute gets called by every method within the scope of the CURL-class and is the only method which initiates contact with the Briox *API*. This allows for reuse of code in an effective way. The method uses three in-parameters to produce the call: Header, URL and body.

The header is needed for authentication purposes and consists of content-type, and the access *token*. The URL is used for directing the call to the right API method; this will change depending on the action. The body is the content of the request and is not an obligatory parameter. The method establishes the type of call by checking if a body-object was included as a parameter. If the body object is received, the method performs a POST-request to the server. If the body object is not received, the method performs a GET-request. By performing this check, there is no need for separate methods specified for each request. The method returns the response upon termination.

3.5.2 Refreshing the Tokens

RefreshTokens is fired by the main function when the checkExpireDate method returns false. The method retrieves the access *token* and refresh *token* from the database and stores them in a header-*array*. The header is passed as a parameter to the **curlExecute**-method, along with the URL. The return *JSON* object is parsed and the new tokens and expire date is stored in the database for later use. The method returns true or false, depending on the success of the refresh.

3.5.3 Checking for customer in Briox

When each order is being processed the module has to check if the customer is already registered in the Briox database. The check is performed by a method called **checkCustomerBriox**. The main function passes the active customers email address along as a parameter for the prepare method. The method adds the email address to the end of the URL string before calling the execution method. If the customer is found, the *API* returns the full customer data object, which is returned to the prepare method through the execution method. The prepare method parses the object and returns the Briox “custno” value.

3.5.4 Creating a new customer in Briox

The creation of a new customer in Briox is prepared by a method called **addCustomer**. The method receives a customer object as a parameter, which is stored in a body-variable. Before calling the **curlExecute**-method, it sets up the header-*array*, which contains the content-type of the message, and the access token. A url-variable is filled with the URL to the desired *API* method. The return value from **curlExecute** is stored in an array and the customer number, assigned by Briox, is parsed out and returned to the Customer-class.

3.5.5 Creating a new Salesorder/Invoice

A new sales order and an invoice is initiated and prepared by the same method, called **addSalesorderInvoice**. The method receives two parameters, one containing the JSON object to be posted, and a string defining what type of object (salesorder or customerinvoice). The type of object is set in the method instantiating the class. The preparation is similar to the **addCustomer**-method. The sales order or invoice is stored in a body variable and the header-*array* is filled with the content-type and access *token*. The url-variable is assigned with a URL-string and the url in-parameter is appended to the end of the string to complete the URL. When the preparation is complete, the method calls the **curlExecute**-method to communicate with Briox. The method returns true or false upon termination, depending on the success.

3.6 Testing

The module was developed on a Webcore.fi test store located on a development-server, accessible through the internal network. It was tested during all stages of development, by pushing to, and retrieving data from a Briox developer account. When the results were satisfying the test site was overwritten by copy of the real online store, including all the latest orders. The extension was installed successfully and the custom attributes and database tables were created without errors.

To test the extension, the orders that were registered during the last four days were exported with satisfying results. No data differed from the test site and the actual online store.

4 Results and Discussion

This section will analyze the results, suggest future work and evaluate the final product. The product is working but there can still be various functions implemented to make the process more dynamic.

4.1 Results

The project resulted in an integration between an Ecommerce store created with the *Magento Framework*, and an *ERP*-system called Briox. The integration is made possible with an extension for the online store, and uses a *RESTful API* service to communicate with the *ERP*-system.

The extension was requested by an organization called Webcore, and will primarily be used on their own online store, although it might be presented to customers in the near future. The extension was requested to automate the registration of customers, and creation of sales orders and invoices in the *ERP*-system. Previously the process was done manually, which resulted in time consumption and human typos.

Due to the Briox API being fairly new and still very much in development, a frequent email communication was maintained with the Briox support team. The teams implemented a few requested methods to the API, such as the possibility to GET a customer based on their email address. The team is still working on a few requested methods. Until these methods are implemented, the module has its restrictions.

The next step is to install the module on the real online shop and test it by exporting orders to the Briox developer account and analyze the data to find and eliminate bugs. Webcore will start using the extension as soon as all the precautionary tests are performed.

4.2 Future Work

Regarding future work, there are still many utilities that could be implemented to make the extension work even better. Just like every computer program receiving constant updates, the extension could always be updated to make it better, and it is rather difficult to conclude that it will ever be a 100 percent finished.

The extension is tested with Klarna, Bambora and invoiced payment methods, but it is rather difficult to know if it works with other methods activated, due to them probably being coded and structured differently. More tests are needed to determine the flexibility.

As for the admin configuration, various different settings could be implemented to make the extension fit every organization's needs. For example an organization selling product might have different demands than an organization selling services.

The supported shipment methods in the admin configuration panel could be coded dynamically to show all the active methods without revealing empty slots. This would require a custom html layout and a much broader competence about the Magento Framework.

4.3 Discussion

During the development I had rather free hands to develop the extension, and I chose a more stand-alone approach to make the extension more comprehensible to a person who is not that familiar with Magento development. To elaborate on my decision, the module could have made more use of Magento components, such as Helpers, Blocks etc. But this requires a much broader competence of the Magento Framework and to someone who is not that informed it might be confusing and look like magic.

The project has been educational but quite time consuming. Mostly because the Magento Framework was rather unknown to me before the project started, and it is an immense and complicated framework. The first few weeks were consumed by just learning the properties of the framework and custom module development.

During the process, I have also gotten familiar with the Briox ERP-system and all of its components and acquired a better understanding of how ERP-systems work and how it is implemented in the organization.

Before the project started my PHP knowledge was satisfying but not on the highest level. Although during the development, I often asked myself if the code I wrote was efficient? However, after a bit of research, testing and about 1000 lines of code I believe that I have reached a higher level of understanding PHP patterns, libraries and code efficiency.

5 Bibliography

- [1] PHP, "FAQ: PHP," 2018. [Online]. Available: <http://php.net/manual/en/faq.general.php>. [Accessed 26 February 2018].
- [2] PHP, "cURL," PHP, 2018. [Online]. Available: <http://php.net/manual/en/curl.examples.php>. [Accessed 15 March 2018].
- [3] Oracle, "Object-Oriented Programming Concepts: Oracle," 2017. [Online]. Available: <https://docs.oracle.com/javase/tutorial/java/concepts/>. [Accessed 1 March 2018].
- [4] Bob och JW01, "What is mvc really: Stackexchange," 2012. [Online]. Available: <https://softwareengineering.stackexchange.com/questions/127624/what-is-mvc-really>. [Accessed 20 February 2018].
- [5] Y. Trivedi, "Detailing the layout & flow of magento mvc architecture," 2014. [Online]. Available: <https://www.mconnectmedia.com/blog/understanding-magento-module-structure-and-code-execution/>. [Accessed 20 February 2018].
- [6] w3schools, "JSON Intro: w3schools," 2018. [Online]. Available: https://www.w3schools.com/js/js_json_intro.asp. [Accessed 24 February 2018].
- [7] N. Nurseitov, M. Paulson, R. Reynolds och C. Izurieta, "Montana State University," 2009. [Online]. Available: <https://www.cs.montana.edu/izurieta/pubs/caine2009.pdf>. [Accessed 25 February 2018].
- [8] Java T Point, "Json vs XML: Java T Point," 2018. [Online]. Available: <https://www.javatpoint.com/json-vs-xml>. [Accessed 25 February 2018].
- [9] S. Pearlman, "What are APIs and how do APIs work?: MuleSoft," 2016. [Online]. Available: <https://blogs.mulesoft.com/biz/tech-ramblings-biz/what-are-apis-how-do-apis-work/>. [Accessed 4 March 2018].
- [10] A. Rodriguez, "RESTful Web services: The basics: IBM," 2015. [Online]. Available: <https://www.ibm.com/developerworks/library/ws-restful/index.html>. [Accessed 4 March 2018].
- [11] Magento, "About," 2018. [Online]. Available: <https://magento.com/about>. [Accessed 20 February 2018].
- [12] Moose, "Why does Magento have 3 code pools?: Stckexchange," 2017. [Online]. Available: <https://magento.stackexchange.com/questions/11908/why-does-magento-have-3-code-pools>. [Accessed 6 March 2018].

- [13] Ian, "Magento Model vs Block vs Controller etc: Stackoverflow," 2013. [Online]. Available: <https://stackoverflow.com/questions/14283920/magento-model-vs-block-vs-controller-etc>. [Accessed 5 March 2018].
- [14] A. Strom, "Magento for Developers: Part 1—Introduction to Magento: Magento," 2018. [Online]. Available: <http://devdocs.magento.com/guides/m1x/magefordev/mage-for-dev-1.html>. [Accessed 26 February 2018].
- [15] F. Khattak, "How to Create a Magento 1.9 Custom Module: Magentician," 2017. [Online]. Available: <https://magenticians.com/create-magento-custom-module/>. [Accessed 3 March 2018].
- [16] A. Storm, "Magento for Developers Part 6 Magento Setup Resources: Magento," 2018. [Online]. Available: <http://devdocs.magento.com/guides/m1x/magefordev/mage-for-dev-6.html>. [Accessed 10 March 2018].
- [17] Briox, "Briox," 2018. [Online]. Available: <http://briox.fi/sv>. [Accessed 24 February 2018].
- [18] Briox, "API docs: BRiox," 2018. [Online]. Available: <https://apidoc-fi.briox.services/#/>. [Accessed 02 March 2018].
- [19] A. Storm, "Magento," 2018. [Online]. Available: <https://alanstorm.com/category/magento/>. [Accessed 22 February 2018].
- [20] A. Storm, "Magento Models an ORM Basics: Alan Storm," 2009. [Online]. Available: https://alanstorm.com/magento_models_orm/. [Accessed 13 March 2018].