Självständigt arbete på grundnivå

*Independent degree project - first cycle*

Datateknik
*Computer engineering*

Persiennbutiken
The construction of a web shop using Symfony

Oscar Reimer
MID SWEDEN UNIVERSITY
Department of Computer and System Science (DSV)

Examiner: Felix Dobslaw, felix.dobslaw@miun.se
Supervisors: Johan Timrén, johan.timren@miun.se,
Daniel Wisenhoff, Persiennbutiken, daniel.wisenhoff93@gmail.com
Author: Oscar Reimer, oscar.reimer@reimerbrothers.com, osre1300@student.miun.se
Degree programme: Software Engineering, 180 credits
Main field of study: Construction of a web shop using Symfony
Semester, year: 1, 2016
Abstract

The aim of this work has been to suggest e-commerce implementations to a partially complete web shop using the script language PHP: Hypertext Preprocessor and the related framework Symfony. The motivation behind the work has been the rapidly changing requirements which web sites have seen during the past few years as a result of the introduction of smart phones and tablets. This in turn led to the company behind the web shop Persiennbutiken needing a new web site.

In addition to taking the latest web design trends into consideration, the work is also studying behavioural economics in order to provide a solution which not only work, but also helps increasing profit with the help of clever technology. The importance of relevant statistics when developing new features is also highlighted and how to gather that statistics in a non-intrusive way.

Each suggested implementation is measured against quite a few performance measures emphasising functionality, performance and security. All very important aspects when building a web shop as is seen in this report.

The results indicate that the suggested implementations are able to fulfil the set requirements to a varying degree with different estimated development time. This leads to a recommended final solution which tries to balance development time versus features, relying on existing libraries where possible.

Finally, the report also suggests a new way of dealing with database cache invalidation when using Symfony in conjunction with the object-relational mapping tool Doctrine.

**Keywords:** Symfony, E-commerce, web, PHP, behavioural economics.
Acknowledgements

I would like to say a big thank you to my tutor Johan Timrén for his excellent feedback on the report throughout my work. I would also like to thank the people behind Persiennbutiken for letting me work with them and give me the insight I needed to be able to make a meaningful project.
# Table of Contents

Abstract .................................................................................................................. iii  
Acknowledgements ................................................................................................. iv  
Terminology ............................................................................................................ vii

1 Introduction .......................................................................................................... 1  
   1.1 Background and problem motivation ......................................................... 1  
   1.1.1 About Persiennbutiken ................................................................. 1  
   1.1.2 Previous work towards a new web shop ...................................... 1  
   1.2 Overall aim ......................................................................................... 1  
   1.3 Scope ............................................................................................ 2  
   1.4 Concrete and verifiable goals ............................................................. 2  
   1.5 Outline ....................................................................................... 2  
   1.6 Contributions ............................................................................ 3  

2 Theory .................................................................................................................... 4  
   2.1 Statistics ....................................................................................... 4  
   2.2 Swedes opinion on payment methods ............................................. 5 
   2.3 Psychological pricing ....................................................................... 9 
   2.4 Symfony and related bundles ............................................................. 10 
   2.4.1 Doctrine .................................................................................. 11 
   2.4.2 Twig ...................................................................................... 12 

3 Methodology ......................................................................................................... 13  
   3.1 Development methodology ................................................................. 13 
   3.2 Statistics ....................................................................................... 13 
   3.3 Frontend requirements ...................................................................... 14 
   3.4 Backend requirements ...................................................................... 15 
   3.5 Overall requirements ....................................................................... 16 
   3.6 Payment requirements ...................................................................... 17 
   3.7 Currency requirements .................................................................. 18 

4 Design and suggested implementations ............................................................... 20 
   4.1 Currency ..................................................................................... 20 
   4.2 Payment ...................................................................................... 22 
   4.2.1 Extensibility/flexibility ............................................................. 22 
   4.2.2 Enabling speedy payment and ease of use .................................. 23 
   4.2.3 Integrating PayPal .................................................................. 26 

5 Results .................................................................................................................... 28 
   5.1 Suggested currency implementations .................................................. 28
5.2 Database caching implementations ........................................... 29
  5.2.1 Automatic use of caching................................................... 29
  5.2.2 Invalidation of cache....................................................... 30
5.3 PayPal integrations .................................................................. 30
5.4 Compliance with overall requirements ....................................... 31
6 Conclusions.................................................................................. 32
  6.1 Analysis of result.................................................................... 32
    6.1.1 Currency......................................................................... 32
    6.1.2 Payment.......................................................................... 33
  6.2 Ethical aspects of the research................................................ 34
  6.3 Future work............................................................................ 35
  6.4 Final words............................................................................. 36
References......................................................................................... 37
Appendix A: Code............................................................................. 45
Terminology

Abbreviations and acronyms

API Application programming interface [1].
CSRF Cross-site request forgery [3].
CSS Cascading Style Sheet [4].
DBMS Database management system [5].
DKK Danske Kroner. Danish currency.
DDR3 SDRAM Double data rate type three synchronous dynamic random-access memory [6].
DQL Doctrine Query Language [7].
EU European Union [8].
HHVM HipHop Virtual Machine. A PHP-interpreter from Facebook [9].
HTML Hypertext Markup Language [10].
MVC Model-view-controller. A common design pattern for structuring an application [12].
NTFS New Technology File System [13].
ORM Object-relational mapping [14].
PHP PHP: Hypertext Preprocessor [15].
PSR PHP Standards Recommendations [16].
RAID Redundant array of independent disks [17].
RAM Random-access memory [18].
REST Representational state transfer [19].
SDK Software development kit [20].
SEO Search Engine Optimisation [21].
SEK Svensk Krona. Swedish currency.
SQL Structured Query Language [22].
SSD Solid State Drive [23].
WWW World Wide Web [24].
XML Extensible Markup Language [25].
XSS Cross-site scripting [26].
YAML YAML Ain’t Markup Language [27].
1 Introduction

1.1 Background and problem motivation

E-commerce has been a rapidly growing form of retailing since Internet’s big breakthrough in the 1990s [29]. It has evolved from being a niche market for B2B (business-to-business)[1] and the technology minded to being widely used by the general population in developed countries such as Sweden in recent years [30]. The growth has been fuelled by faster Internet connectivity and new mobile, yet powerful, platforms such as mobile phones and tablets [30]. A trend which is just set to continue. These changes have led to a constant change of expectations by customers. Today, four out of ten [48] Swedish customers using mobile phones cancel their purchase because of the site not being adjusted to work well with their device. To adjust to this ever changing environment, Persiennbutiken, the store which this work has been done for, has decided to build a new website taking all the latest trends into consideration.

1.1.1 About Persiennbutiken

The company behind Persiennbutiken, ATW Handelspartner Handelsbolag, was founded in 1993 with the aim of offering sun screen protection. It has since 2001 had a presence on the WWW (World Wide Web)[24] when its first web shop was launched. Over the years the web shop has gone through several iterations to keep up with time, with the latest being introduced back in 2010. Now the time has come to make a new iteration, built from the ground up.

1.1.2 Previous work towards a new web shop

In a preceding university course to the current independent degree project course (DT133G), called DT002G, the students were tasked with a smaller project of little or no scientific value. In that course, I first came into contact with Persiennbutiken and helped them lay the foundation on to the new web shop. In that course we made quite a few decisions and implementations such as what language to use, selected a suitable framework to that language and implemented a configurator which lets the user easily build their own roller blind, just like car configurators from car companies. It is upon this work which the current project is based on.

1.2 Overall aim

The project’s aim is to analyse and suggest appropriate implementations which, when implemented, enables a fully functional web shop, built upon the work that was discussed in chapter 1.1.2. To accomplish that, e-commerce features will need to researched and implementations outlined. Namely multi-currency support and handling of payments. These features are to be well thought out so that
the utility of the suggested solutions are maximised according to a few well defined performance measures outlined in chapter 3.

1.3 Scope
The work has its focus on what is needed to get the web shop to a working state. This means that not all features usually found at a web shop will be studied, such as an order view and product view. Instead the two features needed, multi-currency support and handling of payments, will be the ones researched in this paper. They will be researched with Persiennbutiken’s needs in mind, but the main parts of the study’s conclusion should apply to any web shop. We will try to avoid discussing the server environment running the code in this report because it is such a vast field it deserves a report of its own, instead we will focus on the code needed to achieve our goals.

1.4 Concrete and verifiable goals
The goals are:

- Suggest a solution which is free of well-known security risks such as SQL injection [28] and XSS (Cross-site scripting)[26].

- Suggest a solution which tailors to Persiennbutiken’s demographics’ expectations when it comes to website performance and device customisation.

- Suggest a solution which utilises the chosen framework Symfony.

- Suggest a payment implementation which is likely to be trusted by Persiennbutiken’s demographic using PayPal as payment processor.

- Use the field of behavioural economics to enhance the functionality of the suggested multi-currency implementation.

1.5 Outline
- Chapter 2 describes the facts which we gathered in order to make informed decisions about the project’s methodology, design and implementation.

- Chapter 3 contains the methods which we used when working with the project. The chapter also contains information about requirements which we set up to be fulfilled by the final solution.

- Chapter 4 describes the design and implementation of the web shop. It contains details about the implementation including small code snippets.
Chapter 5 covers the result of the various design and implementation decisions in an objective way according to well defined performance measures outlined in chapter 3.

Chapter 6 contains the analyse and discussion of the results presented in chapter 5. It also contains reflections related to the work, such as its importance and news value. The chapter also include potential ethical issues encountered while doing the research.

1.6 Contributions
The whole project has been conducted by me, Oscar Reimer. However, the people at Persiennbutiken contributed with some very valuable background information which was particularly useful when doing research for the multi-currency support.
The second chapter contains facts which we gathered in order to make informed decisions about the project’s methodology, design and implementation. It also contains a subchapter with information about Symfony and related bundles which we use in the project.

2.1 Statistics

One of the most important aspects while building a new web shop, or any site for that matter, is its target audience. If you do not connect well with your audience, you will not sell well either. This led us to early on in this work take a look at the visitor statistics for the current website. Having a current website’s statistic is gold, but that statistic is not always available, such as when starting a website from scratch. In those cases, you would have to rely on good old techniques for start-ups, such as publicly available statistic and your own questionnaires.

Collecting statistics from users can be done in many ways, you could for example, even with a well-established company, run a survey, collect statistics on your website and hold conversations with your customers. As with everything the various ways of collecting statistics have different advantages and disadvantages, some ways being more controversial than others.

Starting with website statistics gathering, this field is wide and can be done in many ways all based on using your website as a source. A simple way is to just gather as much information as possible from your users requests without storing any data on their computers. It is possible to create some valuable statistics this way, such as how many visitors you have, which of your pages they visit the most and their location based on both IP-addresses and browser language settings. This statistic can be created by tools like AWStats[32] which gather data from your web server logs and then present that in a web user interface. Because no personal information is stored about the user there is less legal and ethical challenges to this approach.

Another way is to use a third party tool such Google Analytics to get a deeper knowledge of your users. Google Analytics can be configured in multiple ways with various levels of privacy intrusion such as enabling collection of users’ age and gender via an option called “Enable Demographics and Interest Reports”[33]. Per default it is not very intrusive at all, but also not very functional, offering functionality similar to what can be achieved by looking at user requests. While its functionality is limited it still requires a cookie, which in all EU (European Union)[8] countries means that the website needs to clearly state it is saving information on its users’ computers[36],[37]. This is often done by a popup attached to the bottom or the top of a website where the user has to explicitly accept the use of cookies. These popups have attained some criticism for being irritating leading to web browser add-ons such as “I don’t care about cookies”[31] trying to hide these popups and statistics[38] have shown very few users act on them. This is clearly a drawback with Google Analytics, but one also has to remember
that one is likely to use cookies on its website for other purposes. In our case we have planned to store user preferences such as a “remember me” functionality (user remains logged in on multiple visits from the same device) on the new website. This does indeed require cookies and so Google Analytics’ cookie requirement does not matter, at least not in that regard.

To get all features of Google Analytics the website owner has to enable options such as “advertising reporting features”, which makes it possible to track for example users’ age, gender and interests. While it is hard to connect this data to a particular person, the data is collected anonymously[35], this vast information collection may still be controversial. On one side we have the webmasters and marketers arguing this kind of information collection is crucial to their business’ success, on the other side we have privacy advocates. In Sweden, the main market forPersiennbutiken as seen in Table 1, several studies [34],[39] have shown that the general population is not particularly worried about their personal integrity being violated.

Table 1 Percentage of visitors’ countries. Data source, current website’s Google Analytics data.

<table>
<thead>
<tr>
<th>Country</th>
<th>Sessions. % of total.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sweden</td>
<td>95.96 %</td>
</tr>
<tr>
<td>Norway</td>
<td>0.94 %</td>
</tr>
<tr>
<td>Denmark</td>
<td>0.4 %</td>
</tr>
</tbody>
</table>

2.2 Swedes opinion on payment methods

Handling payments clearly is a vital part of any web shop. But to implement them well we need to understand what our mainly Swedish customers expect when it comes to payment methods. We did this research with the knowledge that we would be using PayPal as payment processor, at least initially. The reason to why we are using PayPal as a payment processor is easy, it was the client’s wish because of many factors such as ease of administration and fees.

As you may or may not know, PayPal supports both payment via PayPal accounts and also straight card payments, which do not require a PayPal account, via a service called “Website Payments Standard” (only available in selected countries, including Sweden). It also supports digital invoices, but they still need to be paid via a PayPal account or card payment. Lacking is the ability to pay straight from your bank or via a more traditional invoice.[47]

According to the payment processor DIBS’ research into Swedish e-commerce [48], Swedes strongly prefer to pay with cards. 41 % prefer card payment and another 7 % prefer PayPal and similar services, with a rise in popularity of both alternatives in recent years, see Figure 1. When looking at payments made from
mobile platforms, an overwhelming 74 % have paid with cards the past 6 months and an additional 12 % with solutions such as PayPal, this is illustrated in Figure 2. Combining these facts with the fact that 34 % have cancelled purchases because of lacking payment method, the most common reason to cancel, see Figure 3, it is clear that it is important to offer a wide range of payment option if one does not want to lose out on customers.

Figure 1 Payment method preference over the past three years. Data source, DIBS’ research [48].

Figure 2 "Which payment method(s) have you used over the past 6 months when using smartphone and tablet?". Multiple choices possible, hence the total can exceed 100 %. Data source, DIBS’ research [48].
Another interesting fact presented in DIBS’ research is the reason why people selected the payment method they did. As Figure 4 shows, when it comes to cards the major reason (71%) is simplicity with the other three options, safety, bonus points and cheapest, being quite equal at about 20%. The situation is quite different when it comes to paying with PayPal, the main reason here is safety (61%), followed by simplicity (52%). The other two methods of paying being compared, paying via bank and invoice, also gives some interesting statistics. With bank payments, simplicity is yet again on top with 59%, followed by a rather large 49% for safety. When it comes to invoice 65% prefer this option because of safety. This is probably both due to technical safety, as in not a lot of sensitive details being saved, and also financial safety, not having to pay unless the product actually arrives. A further 28% prefer invoice because they think it is the easiest. It is indicative that different prioritises should be made when developing a payment experience depending on which payment method is integrated.
Figure 4 "Why do you prefer the payment method?". Multiple choices possible, hence the total can exceed 100 %. Data source, DIBS’ research [48].

Finally, another point made in DIBS’ research is how sought after the feature to save credit/debit card details is. As seen in Figure 5, a majority of mobile users want to save card details, as opposed to 47% on computers. Even more interesting, from a web shop’s perspective, is that almost a third of the mobile users would buy more if they did not have to enter the card details time after time. The same figure for desktop users is just less than a quarter. It shows that the expectation on the payment process differs depending on the platform which the user uses.
2.3 Psychological pricing

The pricing of products is one of the most basic factors which affects any shop’s income. Hence all web shops support pricing of products in one or several ways. The most basic implementation of letting the shop owner set a price for each product is not very advanced and for sure not directly groundbreaking. To advance beyond that, we decided to take a look at psychological pricing, part of the wide behavioural economics field [43].

Psychological pricing [45] is a marketing strategy where you try to make product’s prices appear more appealing to the customer than what a straight margin calculated price would give. For example, if we wanted to have a roller blind with 50% margin which has the purchase price of 120 SEK (Svensk Krona), we would give it a price of 180 SEK. Now this has been shown to not be an optimal pricing point, at least not if we are trying to maximise our sales. By changing the price to 179 SEK we could expect our sales to increase in the range of 10-35%, easily outweighing the 0,56% lower pricing point. We could further increase sales by combining the “magic” 9 with a sale tag. In our example this could mean that we say the regular price is 210 SEK, but that it is currently reduced to 179 SEK. [44]

One more method belonging to psychological pricing is “Useless” price points. A study by Dan Ariely [46] have shown that clever price points can increase sales of more expensive options, changing people’s mind set from finding the cheapest alternative to finding the alternative giving the most value. For instance, if we offer three different roller blind fabrics quality, normal, premium, exclusive, we might have positioned them according to the magic 9 and a certain profit margin:
1. Normal quality – 399 SEK
2. Premium quality – 459 SEK
3. Exclusive quality – 499 SEK

Which quality would you have chosen and why? Perhaps you would immediately start to reason whether you really need anything more than the normal quality. Your mind set is tuned to find the lowest price available. What happens then if we change the prices to:

1. Normal quality – 399 SEK
2. Premium quality – 499 SEK
3. Exclusive quality – 499 SEK, 549 SEK

If the exclusive quality now seems like a more viable option to you, then the useless price points technique worked. The term “useless” comes from the middle option being completely pointless, no one in their right of mind would choose the lesser quality “premium” for the same price as the “exclusive” quality.

### 2.4 Symfony and related bundles

As mentioned in chapter 1.1.2, we did previously choose a framework for the new web shop. We ended up selecting the Symfony framework [49] because of its extensibility via “Bundles”[50], its promotion of good practises such as enforcing PSR (PHP Standards Recommendations)[16] and support for various security features such as form validation and CSRF (Cross-site request forgery)[3] protection. To be able to understand the rest of this report, it is important to have some knowledge about this framework and some related bundles.

Symfony itself is made up of several standard “components” [51] which can be used just as any other bundle to add new features. They include an Asset component, which does what it says on the tin, deal with various assets, such as CSS (Cascading Style Sheet)[4], and JS (JavaScript)[11] files and makes it easy to apply certain actions to these files including minifying and combining them. Common for all the standard bundles are that they deal with fairly easy and mundane tasks which most PHP (PHP: Hypertext Preprocessor)[15]-developers have repeated in various projects. By using these components it is possible to save time by not having to reinvent the wheel all the time and as it happens, thousands of minds tend to create something better than one mind alone.

If one wants to get the most out of Symfony, one should use the Symfony Standard Edition/full stack [53], which is based on the MVC (Model-View-Controller)[12] pattern. It includes all standard components as listed in [51], Doctrine and Twig support (more about them in the next couple of sections), and provides some neat developer interfaces with its Symfony profiler as seen in Figure 6. Additionally, its default project structure enforces the current Symfony’s best practises [52], avoiding starting on the wrong foot. Apart from governing a project’s structure, the best practises also include recommendations
about storage format of configuration files, security, dealing with web assets and much more.

Figure 6 Part of the Symfony Profiler. A powerful development interface which makes it possible to trace performance of a request, request parameters, cookies, executed database queries and their performance and much more.

2.4.1 Doctrine

What is Doctrine you might wonder. Doctrine [54] is a ORM (Object-relational mapping)[14] tool, which abstract away specific DBMSs’ (database management system)[5] PHP driver implementations. It makes it possible to use a wide variety of relational databases, such as MySQL, Oracle, Microsoft SQL Server, PostgreSQL and more, with a common code base. To change from one system to another, in Symfony’s Doctrine integration, you simply change a configuration file. Its ORM revolves around so called entities which are classes which usually correspond to a table in the database. In these classes each variable is mapped to a column or relation via any supported configuration format, annotations, YAML (YAML Ain’t Markup Language)[27], XML (Extensible Markup Language)[25] or PHP. Based on this configuration data Doctrine then knows how the classes relate to the database and vice versa. In addition to the wide support of database systems and the common ORM-attributes, it has extensive support for caching. Making it possible to drastically improve the performance of a database heavy
2.4.2 Twig

The other mentioned supported feature/bundle, Twig [55], is a templating engine [56]. It offers a cleaner form of writing code which is turned into HTML (Hypertext Markup Language)[10]/XML output than raw PHP-code. The Twig code itself is turned into executable PHP-code by the Twig engine, a conversion which can be cached for improved performance. As many other templating engines like Java EE’s Facelets, it supports simpler statements such as for-loops to iterate over passed variables and has its own syntax. Twig is also very extensible supporting writing custom extensions which can then be used in the code. It is for example possible to write a custom filter which formats the number “5500” to “5.500,00”. Finally, arguably one of the most important features of Twig is its security. Twig, when using Symfony’s Standard Edition, automatically applies auto escaping to any variable. It also supports a sandbox mode via an extension, which limits what the template can access in terms of tags, filters and variables. Exactly what is limited is configurable by the developer.
3 Methodology

This chapter describes the methods which we used when working with the project. The chapter also contains information about requirements which we set up to be fulfilled by the final solution.

3.1 Development methodology

Based on experience from previous work on the website outlined in chapter 1.1.2 we have decided to continue with the incremental build model [40]. See Figure 7 for how development is split into iterations/increments with five stages each. The model let us incrementally extend the web shop with new functionality with the latest ideas and concerns from the shop owner taken into consideration on each iteration. It further let us deploy the parts of the web shop which is considered ready before the website as whole is done, which is the client’s wish. However, some care must be taken so that we do not exceed the designated development time of a certain feature and we must also carefully plan the architecture so that we can minimise the risk of having to rethink major design decisions later on. Both are well-known risks with this development method.

Figure 7 Incremental build models development phases [57].

3.2 Statistics

With the discussion in chapter 2.1 in mind, we have opted to enable the advanced Google Analytics’ features on the current website in the hope that the advantages outweigh the disadvantages. This will enable us to better understand our demographic and spot weaknesses in Persiennbutiken’s current website which can be ratified in the new website while not being intrusive.
Something which immediately became very apparent when we looked at the web browser statistic is that the current website has a very low conversion rate, in other words amount of visits which resulted in sales, on mobile platforms, being about 50-70 % less than on computers. This is especially true for smartphones, with slightly better statistics for tablets. It indicates that the current website does not work well with the smaller screen sizes associated with smart phones and potentially also touch input. By taking mobile platforms into consideration when developing the new website, we have a very real potential of sales growth.

3.3 Frontend requirements

Based on web browser statistics from the current website, see Figure 8, we have decided to actively try to support the browsers shown in Table 2. To make the list a browser needs to be used by more than 5 % of Persiennbutiken’s customers. The greater total share the lower minimal version will be required of a particular browser. We have one exception and that is Edge which still makes the list despite only having about 3 % share. We made this exception because it is used by the shop owner and it is Microsoft’s promoted web browser on Windows 10. Actively supported means that we will be testing the new website against these browsers and that they should work with only minor glitches being acceptable. Minor glitches are defects which do not change the functionality of the web page and do not fundamentally change the appearance such as a missing margin. Other browsers than the actively supported might work, but are not taken into consideration when developing the website.

![Figure 8 The browser share and the version share of every browser on Persiennbutiken's current website.](image-url)
Table 2 Actively supported web browsers

<table>
<thead>
<tr>
<th>Web browser</th>
<th>Minimum version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safari (desktop)</td>
<td>7.0 (OS X v10.9 [61])</td>
</tr>
<tr>
<td>Safari (mobile)</td>
<td>600.1.4 (Apple iOS 8.0 [59])</td>
</tr>
<tr>
<td>Chrome (desktop and mobile)</td>
<td>34.0.1847.76</td>
</tr>
<tr>
<td>Internet Explorer</td>
<td>10.0</td>
</tr>
<tr>
<td>Firefox (desktop and mobile)</td>
<td>45.0</td>
</tr>
<tr>
<td>Edge</td>
<td>12.10240 (Windows 10 RTM build [58])</td>
</tr>
</tbody>
</table>

Since we have used HTML 5, CSS 3, JS, jQuery 1.11 (a JS library) and Bootstrap (a HTML, CSS and JS framework) in the previous work discussed in chapter 1.1.2, they are the techniques which are to be used when creating new features in this project. Additional dependencies can be added if we deem them as really necessary, but should be avoided because of the impact they have on page size and therefore loading time.

### 3.4 Backend requirements

Previous development towards the web shop has used the PHP language with the PHP-framework Symfony 3.0 Standard Edition. All backend code should be written with this language and framework. Symfony bundles can be added freely, but with some care because every bundle introduces a third party dependency – a third party may stop developing the bundle creating a risk of it becoming obsolete with new versions of Symfony. To make the code as maintainable as possible Symfony’s Best Practises will be used, see the official documentation [52].

To execute code written in PHP we need a so called interpreter. We agreed to set the minimal level to PHP 7.0, the latest PHP revision as of writing, with tweaks to also support Facebook’s HHVM (HipHop Virtual Machine)[9] interpreter if necessary. Tweaks will be necessary if we want to use any of the features/behaviours listed at HHVM’s GitHub page as incompatible [60]. We wanted to support HHVM because of its large performance advantage in certain circumstances [68],[69] over the standard PHP-interpreter.

In order to have a good flexibility when it comes to hosting the website it needs to support both Apache and Nginx, the two most common web servers in the
world when it comes to active sites according to Netcraft’s April 2016 web server survey [62]. That means that the same routes should work on both web servers via rewrite and that the same level of security should apply, as far as feasible. Finally, the website must fully support HTTPS 2.0 TLS and no unencrypted content (which when included on an encrypted site is blocked and a warning raised in most modern browsers [63]) is allowed. This is due to safety reasons and that the warnings raised by browsers may confuse customers or make them feel insecure, a common reason to cancel purchases as outlined in chapter 2.2.

As main storage a SQL (Structured Query Language)[22]-database will be used. With the use of Doctrine, see chapter 2.4.1, it is possible to support the most popular relational databases with the same code base. Hence, all database code should be utilising Doctrine.

### 3.5 Overall requirements

No matter what feature the particular implementation brings it must satisfy the requirements laid out in this section. The requirements are supposed to ensure good safety throughout the website and promote minimal load times. Promoting low load times is good for two reasons, it is good for SEO (Search Engine Optimisation)[21][64] and for making customers buy more [65],[66]. See Table 3 for all required and optional security and performance measures. We have decided to lower the priority of form CSRF-protection and make it optional because not nearly all forms deal with sensitive data and for these forms, form validation provides enough security combined with routing restrictions. Also, Symfony’s in-built CSRF-protection does not work well with pages that uses AJAX requests to send form data and simultaneously have two or more forms, based on experience from the work which we did previously, see chapter 1.1.2. All the other security measures are equally important, and needed, to ensure a truly safe website. Based on studies on web performance [66],[67] we decided that a one score penalty for every 100 ms should be rewarded due to satisfaction is gradually reduced with a granularity in hundreds of milliseconds. In addition to the performance and security measures, Symfony’s best practises should be followed as outlined in the official documentation [52]. Among many benefits, this should ease long time maintenance, boost productivity and make it easier to introduce new developers which are familiar with Symfony development. Because it is not critical to the web site’s functionality, this is an optional requirement with a lower score.
Table 3  Performance measure common for all implementations.

<table>
<thead>
<tr>
<th>Performance description</th>
<th>Form validation (server and client side)</th>
<th>Form CSRF-protection</th>
<th>SQL-injection protection</th>
<th>Output escaping</th>
<th>Penalty for load time</th>
<th>Follow Symfony’s best practices</th>
<th>Total score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td>10</td>
<td>-1 per 100\text{th} millisecond</td>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td>Solution/required</td>
<td>Yes</td>
<td>No (Yes, when dealing with sensitive data such as account details and admin settings)</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

3.6 Payment requirements

All payments should be done with PayPal as payment processor, the reason why was briefly explained in section 2.2. The implementation needs to support both PayPal accounts and debit/credit cards based on Swedes’ opinion on payment methods mentioned in chapter 2.2, simply supporting PayPal is not good enough. This should be done via PayPal’s Website Payments Standard [47] service. As described in section 2.2 it would be hugely beneficial if the new web shop in the future support more payment methods, such as Klarna invoice. Suggested implementation does not need to support more payments methods from the start because they require separate payment processors which leads to more administrative work for Persiennbutiken and potentially longer development time. With that in mind we have added easily extendable to the performance measure. If the implementation offers an abstract interface which only needs to be implemented for every payment method/processor, then the implementation is regarded as easily extendable. Another point made in section 2.2 is that most Swedes would like to save their debit/credit card details. It would be great if the implementation supported this, however, it is not allowed to save these details locally for security reasons outlined in chapter 4.2.3. See Table 4 for how the implementation will be measured. Finally, for every week estimated development time, one point is de-
ducted from the total score of the solution. As a small company, a week development time is a lot of money, hence an appropriate addition to the performance measurement.

Table 4  Performance measure for payment implementations.

<table>
<thead>
<tr>
<th>Performance description</th>
<th>Supports PayPal accounts and debit/credit card</th>
<th>Integrates PayPal’s Website Payments Standard</th>
<th>Easily extendable with other payment methodprocessors</th>
<th>Offers saving of debit/credit card details using PayPal/other trustworthy third party</th>
<th>Approximate development time (weeks)</th>
<th>Total score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>-1 / week</td>
<td>30</td>
</tr>
<tr>
<td>Solution/required</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

3.7 Currency requirements

The new web shop initially only needs to support Swedish krona, but should be extensible enough to also be able to support additional currencies, such as the Euro. Any conversion rates must be fetched from PayPal as those are the rates which Persiennbutiken will have to use. Additionally, the implementation should take Psychological pricing in chapter 2.3 into consideration if/when doing currency conversion because of its impact on sales. See Table 5 for how the implementation will be measured. Multi-currency and psychological pricing are not required as they are not necessary for accepting payments, hence they score lower. Lastly, just as with payment implementation’s measurement mentioned in chapter 3.6, for every week estimated development time, one point is deducted from the total score of the solution.
Table 5 Performance measure for currency implementations.

<table>
<thead>
<tr>
<th>Performance description</th>
<th>Supports Swedish krona.</th>
<th>Fetches conversion rates including any fees from PayPal.</th>
<th>Supports multi-currency (1..n currencies).</th>
<th>Takes psychological pricing into account</th>
<th>Approximate development time (weeks)</th>
<th>Total score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>-1 / week</td>
<td>16</td>
</tr>
<tr>
<td>Solution/required</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>
4 Design and suggested implementations

The fourth chapter describes the design and implementation of the web shop. It contains details about the implementation including small code snippets.

4.1 Currency

Because Persiennbutiken would like to offer an even better service to customers outside of Sweden it makes sense to support multiple currencies. This is something which the current website does not support and there is a good reason for it - handling one currency is easy, handling two currencies is much harder.

The big challenge is to offer multiple currencies while still maintaining good price accuracy, with minimal economic risks and administrative work. Even a difference between the given converted value to a user and the actual converted value of a few percentage points could result in hundreds or even thousands of Swedish kronors in lost income on big orders. With that in mind we consider a few different ways of implementing multicurrency support with different strengths and weaknesses.

One way is to have multiple prices, one for every currency supported, for each product. The big benefit with this is that you have full control of what you’re charging customers in every currency. This makes it easy to utilise psychological pricing, which we talked about in chapter 2.3, without giving unfair conversion rate fees – it is possible to round down as well as up to give an accurate price, which still utilises psychological pricing. Another benefit is that this way of handling multi-currency should require less development time compared to, for example, solutions relying on a third party service for getting current exchange rates. A big disadvantage with the technique is that it requires a lot of administration, suddenly each product requires multiple prices which all needs to be updated frequently in order to be kept relevant and in turn minimise economic risks and maintaining fairness. This might not be a problem when you have small shop with very few products or you are a big company with relatively few products like Tesla Motors. It could, however, be a problem if you have shop like Persiennbutiken with hundreds of products and little manpower. The additional administration could be overwhelming, especially at times when the currency market fluctuates a lot. As an example of fluctuations over a year between two currencies see Figure 9.
Another way is to use a base currency, logically SEK, and calculate other currencies’ prices based on this currency. The biggest advantage with this method is that all prices but the base price are automatically calculated, saving a lot of time. However, as one would expect, the shop owner has then very little control over individual prices in different currencies, making it hard to benefit from psychological pricing. This could at least partly be countered by clever algorithms determining the prices which doesn’t blindly look at the exchange rate. One part of such algorithm could be to always round up the prices to nearest magic 9 and enforce “useless” price points, while the latter is harder to implement, it’s certainly possible. A potential pitfall, and thereby drawback, is to display obsolete prices to your customers. One way of making sure this never happen is to constantly get the current exchange rate from whatever payment processor we choose to use. While this minimises the risk of showing prices too low (or too high), it could lead to poor performance because remote calls to third party services tend to be really slow. Additionally, it could lead to a lot of fluctuating prices if our exchange rate frequently changes, potentially making customers very confused when the price from one screen to another changes by a few percentages and then on the next screen changes back. A middle ground would probably be preferably, where we take minimal economic risk, perform well and avoid confusing customers.

Such a solution could be to have a database table where we save the exchange rate. We then have a service regularly pulling the exchange rate from our chosen payment processor, which, if the rate has changed from our saved rate, updates the database table and invalidate cached query results from this table. In this way most page loads, unless the exchange rate changes extremely frequently, will
come from a cached query result leading to very good performance. This method also poses minimal economic risk. However, it does not solve the problem with potentially confusing customers. This problem seems to be hard to solve when doing automatic updates of the exchange rates, leading us to a third way.

The third way is to let the administrator manually set the exchange rates. This works well if the exchange rate rarely changes, such as when pre purchasing a foreign currency at a fixed rate. It vastly reduces the risk of the user seeing different values from one page load to another. The drawback is that it has to be administrated and because a human has to update the values there is always a risk of human error such as failing to update the exchange rates when appropriate.

To further enhance the second and third way adjustable constants could be added which could target both a certain currency and a certain country. For example, Switzerland generally have much higher prices than Sweden [41]. To take advantage of this fact a constant in the form of increased price by a certain percentage points to customers from Switzerland could be used to align our prices with the rest of the market in that particular country.

For maximum flexibility optimally all three ways could be implemented to be able to use their advantages when appropriate. The only drawback really being increased development time caused by increased complexity.

4.2 Payment

Handling payments clearly is a vital part of any web shop. Making a payment should ideally be fast [64],[66],[67], safe, reliable and convenient. Convenience is especially true for those who choose to pay with cards, as the statistics show in chapter 2.2. At the same time security is important to those who prefer paying with PayPal. Being secure is something which we had planned to be anyway, but it shows that we must also give the impression of being safe in our suggested implementation. We discussed some technical requirements on how to appear safe in chapter 3.4 which we will ensure to fulfil. We took those points and others mentioned in the discussion about Swedes opinion on payment methods from chapter 2.2 into account when designing the shopping cart and payment experience.

4.2.1 Extensibility/flexibility

Judging by the numbers mentioned in chapter 2.2 regarding people’s preferred payment methods and reason to cancel, it is clear that we at least in the future will need to support more payment methods if we do not want to lose lots of potential customers. This means that the implementation we make now should be extensible enough to accommodate more payment methods in the future, such as Klarna invoice.
4.2.2 Enabling speedy payment and ease of use

To be able to provide a speedy payment we need to make the website really fast. This is something which is easily achievable with the current website, it is just a lot of static HTML pages built by an offline page builder. To compete with this speed for our dynamic website we need to optimise both the server code and the environment within the server code is running, in other words the webserver, network connection and PHP-interpreter.

Optimising code can be done in many ways, everything from the basics such as always breaking a foreach which has fulfilled its purpose and use INNER JOINs as much as possible when accessing database data, to more advanced topics such as page and database caching. When working with database access we have managed to minimise the amount of database accesses to an average of about 10 queries per page, including queries fetching data for display, using what sometimes is really long queries, see Code 3 in Appendix A: Code as an example. While all data from the database is meant to be used for dynamic content, such as payment options being changeable for the web shop owner, a lot of that data is not changed very frequently. One option then is to obviously not store that data at all in the database, but instead in a file saved on disk. This is something which we do for a few basic preferences such as database and mailer settings, which very rarely changes. This is about twice as fast as accessing the database, see Table 6, but has its limitations. A file system generally does not scale as well as a database, requiring special solutions such as RAID (Redundant array of independent disks)[17], we also move over a lot of the processing logic from the storage solution to our programme, increasing development time. The third option is to cache as much data as possible in RAM (Random-access memory)[18]. This can be done both with data stored in a database and data stored on the filesystem. However, the latter would require third party solutions or advanced operating system configurations, and would still require processing of the file. Most database systems are more suited for caching than file systems are. Database systems such as Maria DB, which we are using, can be configured to cache parts or all the data residing in the database for fast access. While this certainly relieve the performance problem of accessing database data, it is still slower than a filesystem as can be seen in Table 6. We also have additional latencies if the database, or filesystem for that matter, resides on a remote computer.

Code 1 Typical query execution in our programme using both query cache and result cache.

```php
$qB->getQuery()->useQueryCache(true)->useResultCache(true, self::$cacheTi
meOut, __METHOD__) ->getSingleResult(\Doctrine\ORM\Query::HYDRATE_ARRAY);
```

To overcome this problem, we can use client cache, where the client in this case is the PHP-interpreter. This is something which the mentioned ORM tool in chapter 2.4.1, Doctrine, offers in three ways. Firstly, it offers a query cache, this is caching the translation from DQL (Doctrine Query Language)[7], to SQL. Secondly, it offers a metadata cache, this is caching of the metadata gathered “from
a few different sources like YAML, XML, annotations, etc.”[70] containing information such as table and column names. Unless you are developer, the queries and metadata will never change so it is wise to enable this on queries throughout the application. Unfortunately, caching queries must be done in the code on every single query, how this is done can be seen in Code 1. The third way is a result cache, which makes it possible to cache the whole result of a particular query with two optional settings, cache id and cache timeout/lifetime. This is very performant, as can be seen in Table 6, and avoids any network delay (provided the caching solution is on the local machine, which it often is). Result cache also needs to be applied per query.

Table 6 Approximate latency of accessing stored data via three different methods. The data is stored on the same system running the application. Test conducted using Symfony Profiler.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Database (SSD (Solid State Drive)[23] and InnoDB cache &gt; database size)</th>
<th>NTFS (New Technology File System)[13] file system (SSD)</th>
<th>Cached database result, metadata and query (1866 MHz DDR3 SDRAM (Double data rate type three synchronous dynamic random-access memory)[6])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most simple query with one column (~1 KB of data)</td>
<td>~0,14 ms</td>
<td>~0,070 ms including processing of file</td>
<td>N/A, virtually instant (not possible to measure accurately due to its speed)</td>
</tr>
<tr>
<td>More advanced query requiring one join (~4 KB of data)</td>
<td>~0,20 ms</td>
<td>~0,075 ms including processing of file</td>
<td>N/A, virtually instant</td>
</tr>
</tbody>
</table>

After a while we realised it would be great if it were possible to automatically use at least query cache and perhaps also to opt in of result cache per default, with the option to opt out when needed. One way of solving this problem would be to create a simple method which takes the query/query builder, applies the query and result cache and then returns the query. This method could have the optional parameters, whether to use result cache or not, the cache time out and the cache id. The drawback with this method is that it would involve calling this custom method everywhere, requiring rewrite of all existing code which uses query and result cache. An option would be to override the methods which creates
the queries. They all reside in Doctrine’s EntityManager, a class used to communicate with the database. By extending this class and then configure Symfony/Doctrine to use it, it would be possible to apply the caching on the queries when they are created. In turn making it possible to continue to use the old syntax for the old code, without affecting its behaviour, and at the same time use a much shorter syntax when creating new queries. The drawback with this method is that it is slightly more advanced, with some minor penalty when it comes to development time, which may or may not be countered by the fact that no time is spent on changing the creation of all existing queries.

Using cache is great, but sometimes we need to invalidate it because a change has happened in the underlying database. Invalidation is a quite tricky concept to get right. We usually always want to have the latest data in cache, but we also do not want to invalidate cache which is still valid since that would lead to unnecessary performance loses. To handle cache invalidation, we are looking at two alternatives.

The first alternative involves creating a very simple interface with a single method clearCache() which should clear cache for all queries of the implementing class. This makes it possible for other classes to clear cache which they know will be outdated. While this alternative is fairly simple to implement, it has a couple of noteworthy drawbacks. One of them is the fact that other classes need to know that their update of a certain data affected other classes - we increase our code coupling. The other drawback is that when we create new classes that uses cached queries, we must make sure that they implement this interface - we get increased reliance on programmers doing the right thing.

The second alternative would be to have a service [71] which let entity classes, see chapter 2.4.1, register and subscribe to cache groups. In this alternative every entity class would subscribe to this service, preferably automatically. Then other classes with cached queries which uses these entities could subscribe to the relevant group via, for example, annotations. When an entity would be updated, then all subscribers to the cache group would be notified. Making it possible to clear the cache in all relevant classes. In this way the entity which was updated does not need to know which classes depend on it, reducing code coupling. However, this solution still heavily relies on the developer, he/she needs to remember to subscribe to the right groups. Another disadvantage is that this solution is estimated to require substantially more development time, at least initially, because it would require addition of custom annotations, a service which can handle events and a change of all existing entity classes.

Code 2 Doctrine caching settings in config_prod.yml affecting only production environment.

```yaml
doctrine:
  orm:
    metadata_cache_driver: apc
    result_cache_driver: apc
    query_cache_driver: apc
```
To not interfere with development, we have disabled Doctrine’s caching completely when running in Symfony’s “dev” environment, avoiding problems with cache invalidation all together. This is done by only specifying cache drivers in the configuration file used by the production environment, see Code 2.

### 4.2.3 Integrating PayPal

With PayPal being one of the world largest payment processors based on its Alexa rank [72] it is not surprising that there are many ways of integrating PayPal, even with fairly strict requirements, such as that they need to work with Symfony and support both PayPal and debit/credit card payments, as mentioned in chapter 3.6. One way is to utilise PayPal’s PHP SDK (Software development kit)[20] [73] which can be used to communicate with PayPal’s RESTful APIs (Application programming interfaces)[1]. With this SDK one has access to all of PayPal’s REST (Representational state transfer)[19] services, which is incredibly powerful. As we all know, with great power comes great responsibility. It is for example possible to store card details locally on your own server, but that also means you have to be PCI compliant [74]. To ease the burden of the developer and the web shop it is possible to instead use PayPal’s Vault service [75], which moves the responsibility of storing card details from the web shop to PayPal. This is just one of many steps involved when integrating PayPal to fulfil the requirements which we laid out earlier in chapter 3.6. We realised it would be better if we could find a solution which was of higher code level, thereby reducing development time and risks.

Such a solution is PayumBundle[76], which integrates the Payum payment processing library [77] into Symfony. It supports a wide variety of payment processors from the start, including Paypal, but also Klarna Invoice. With the ability to write your own gateway implementations for unsupported payment processors. The included PayPal REST implementation abstracts away a lot of the code which would otherwise have to be written when using PayPal’s PHP SDK, with most of the code needed to handle payments being reusable for other payment processors. Unfortunately, it does not support PayPal’s Vault service. Hence, to support saving card details, without achieving PCI compliance, we would have to extend the Payum PayPal implementation with Vault support ourselves. Still, to integrate PayumBundle should be fairly simple thanks to its integration with Symfony and a lot of readymade code with classes for data such as customer address and credit card where security has been taken care of [78]. Its extensibility is also a huge benefit, because as we discussed in chapter 2.2 and 3.6 it would be greatly beneficial if we could support more payment processors in the future.

Because of the many benefits and seemingly few disadvantages we decided to make a light implementation of the PayumBundle as a proof of concept. This turned out to be harder task than we initially expected. Instead of taking one day to implement, only using the official documentation, it took four days. The reason to the much longer development time were many. To start with the current releases of the PayumBundle (2.0.1) and Payum PayPal Rest extension (1.3.2) are not strictly compatible with each other. For example, the extensions require a trait
that was introduced in Payum 1.3, a version which the bundle does not include. This is however fairly simply solved by using the latest development version of PayumBundle, which includes the latest Payum core. With the incompatibility solved we could continue the integration, which involved reading the documentation of the PayumBundle and PayPal extension and then try to merge the examples given. This was a fairly simple task; the real time consumer was the storage solution.

According to PayumBundle’s documentation [79] the recommended way of storing payment data is to use Doctrine or any other non-filesystem storage. There is one problem with that recommendation, the PayPal extension does not seem to support any other storage than the filesystem storage. This is due to how PayPal’s PHP SDK stores its variables in a hash map based system. Because data is stored in a hash map, instead of regular variables, it is not possible to use Doctrine’s annotations. To circumvent this problem, we had to extend the SDK’s domain classes used for payment, create corresponding variables as actual variables, annotate these variables and override their get and set methods to set both the hash map variables and the actual variables. Take a look at the Doctrine enabled class in Code 6 found in Appendix A: Code and compare it to its base class [79], to see what changes were required to make the domain classes Doctrine capable. After this rather lengthy process we were able to make our first test payment, but apart from being time consuming, we now also had a much larger code base than anticipated to maintain.
5 Results

This chapter covers the result of the various design and implementation decisions in an objective way according to well defined performance measures outlined in chapter 3.

5.1 Suggested currency implementations

In chapter 4.1 we discussed various ways of implementing currency support. That discussion resulted in three different ways with a fourth combining them all. How they score according to the table first introduced chapter 3.7 can be seen in Table 7. We estimated the first solution to only take two weeks of development time because it requires very little logic with most work done by hand. The second solution is estimated to take a substantial longer time to develop because of its algorithms and service fetching the exchange rate, we estimated this time to be six weeks. The third solution is estimated to take slightly less time, four weeks, because it does not require automatic exchange rate fetching. Finally, the fourth solution which combine them all is estimated to take 10 weeks to implement. It benefits from only having to implement certain things once, such as the pricing algorithms, but still requires a lot of time to implement the non-common features.
Table 7 Score table for suggested currency implementations.

<table>
<thead>
<tr>
<th>Performance description</th>
<th>Support Swedish krona.</th>
<th>Fetches conversion rates including any fees from PayPal.</th>
<th>Supports multi-currency (1..n currencies).</th>
<th>Takes psychological pricing into account</th>
<th>Approximate development time (weeks)</th>
<th>Total score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>-1 / week</td>
<td>16</td>
</tr>
<tr>
<td>Solution/required</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>First way – manual handling</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>N/A (dependent on the administrator)</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>Second way – base currency with clever algorithms and with service handling automatic exchange rate fetching</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>Third way – Same as second, but manual control of exchange rate</td>
<td>Yes</td>
<td>N/A (dependent on the administrator)</td>
<td>Yes</td>
<td>Yes</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>Fourth way – All previous solutions combined</td>
<td>Yes</td>
<td>Yes, when wanted</td>
<td>Yes</td>
<td>Yes, when wanted</td>
<td>10</td>
<td>6</td>
</tr>
</tbody>
</table>

5.2 Database caching implementations

5.2.1 Automatic use of caching

Because of its many advantages we decided to go ahead and create our own custom EntityManager as discussed in chapter 4.2.2, the result of that can be seen in Code 4 which you find in Appendix A: Code. This is complemented by a custom
query builder seen in Code 5, also found in Appendix A: Code. To use them, one just need to tell Doctrine to use the custom EntityManager instead of the default one. It requires no other code changes.

### 5.2.2 Invalidation of cache

Due to time constraints we decided to go ahead with the first alternative mentioned in chapter 4.2.2, which enables us to invalidate cache when needed.

### 5.3 PayPal integrations

In chapter 4.2.3 we discussed two different solutions for integrating PayPal. How they score can be seen in Table 8, which is the same table found in chapter 3.6. We see that both solutions meet the required performance descriptions, but PayumBundle is additionally regarded as easily extendable with its inbuilt support for multiple payment processors. We have estimated the development time to be slightly longer with the pure PayPal SDK integration, because it essentially requires the developer to create the same functionality already found in the Payum solution.

**Table 8 Score table for the suggested PayPal integrations.**

<table>
<thead>
<tr>
<th>Performance description</th>
<th>Supports PayPal accounts and debit/credit card</th>
<th>Integrates PayPal’s Website Payments Standard</th>
<th>Easily extendable with other payment method/processors</th>
<th>Offers saving of debit/credit card details using PayPal/other trustworthy third party</th>
<th>Approximate development time (weeks)</th>
<th>Total score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>-1 / week</td>
<td>30</td>
</tr>
<tr>
<td>Solution/required</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>PayPal PHP SDK</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>PayumBundle</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>3</td>
<td>27</td>
</tr>
</tbody>
</table>
5.4 Compliance with overall requirements

In chapter 3.5 we outlined a few overall requirements which all solutions need, or should, meet. All of the mentioned implementations either meet those requirements or have a potential of meeting them. All currency implementations have a potential to conform with the requirements, but the developer writing the actual implementation is responsible for making it happen. The situation is the same with the database implementations. Both of the suggested PayPal integrations also have the possibility to conform with the requirements. Worth noting though is that the PayumBundle seems to be following Symfony’s best practices only to some extent. This can be seen by the fact that it, for example, keeps the structure of the bundle as recommended and also uses YAML as the suggested configuration format, but at the same time it uses XML configuration files internally which is a breach of the best practises.
6 Conclusions

6.1 Analysis of result

While the result of the suggested implementations was given in chapter 5 it does not give the complete picture of an implementation’s usefulness in certain situations. In this section containing the analyse of the result we will try to remedy that fact.

6.1.1 Currency

We came up with a total of four different solutions to implement currency support. The first of them, leading to manual control of all prices, were never really an option to Persiennbutiken as it, combined with the vast amount of product the shop offers, would create far too much administrative work. This subpar option also resulted in a low score of six, showing the relevance of the performance measures introduced in chapter 3.7. However, for a shop which only has a few items, this solution may very well be worth considering, especially since it has the lowest estimated development time of them all.

The second option, which included clever algorithms taking psychological pricing into consideration and automatic exchange rate fetching from PayPal, was the one to score the highest with ten points. It indicates that the solution is able to match the requirements set by Persiennbutiken, without adding too much development time. The option has a good balance between functionality versus development cost, albeit with limited control of individual pricing across multiple currencies.

If a shop owner wants more control it would be worth to take a look at solution number three instead. This solution has the extra benefit of having reduced development time thanks to lower complexity. However, because it does not fetch conversion rates automatically from PayPal, it does not fulfil Persiennbutiken’s requirements, making it a non-viable option. The lack of this feature also makes the implementation to have modest score of seven. This solution may still be a good alternative for a shop which has an exchange rate which very rarely changes, or simply do not mind the additional administrative work.

For maximum flexibility the choice is clear, the fourth implementation is the way to go which implements all features from the previous solutions and let the administrator decide what to do on a product basis. Because it is heavy on features, it is also heavy on development time. This, in combination with that the implementation does not offer any new features, make it score an equally low six as the first solution. However, I would say this is not completely fair. The added flexibility may very well be business critical to a shop, hence one must not blindly look at the score tables when making a decision of what implementation to choose.

Performance wise all the solutions have the potential of being very fast. While
all but the first option offers truly dynamic pricing, they should still perform well because the price is calculated only on the first view after changing exchange rate or the base price, and not on every view. To further enhance the performance of the implementations it would be a good idea to introduce warming (caching data before it is accessed for the first time) of the cache. A good trigger for this warming of cache would be whenever the exchange rate or base price changes.

Security wise all the solutions have the potential of fulfilling the security requirements outlined in chapter 3.5 as discussed in chapter 5.4. The same can be said about device tailoring, none of the solutions favour any platform before another.

While none of the solutions explicitly require Symfony, the three dynamic ones which relies quite a lot on database access and caching would strongly benefit from a framework such as Symfony in conjunction with Doctrine. Thanks to them it would be possible to automatically apply caching, as discussed in chapter 4.2.2. A benefit of Symfony not being strictly required for the suggested solutions is that they can be of use to more e-commerce sites using different frameworks.

Based on the results given in chapter 5.1 and the analyse in this chapter I would recommend Persiennbutiken to go with the second option to start with. If Persiennbutiken in the future feel like it need more flexibility when pricing products, I would recommend going with the fourth option. The extended functionality which this fourth option brings should only take roughly four weeks to develop, thanks to most of the work has already been done while implementing solution number two.

### 6.1.2 Payment

Initially we set out to provide at least one payment implementation which were to achieve Persiennbutiken’s requirements. While researching what people expected from payment solutions we found that speed was of great concern not only to payments, but the whole shopping experience as we mentioned in chapters 3.5 and 4.2. This lead us to four interesting caching solutions, two for automatic caching and two for invalidation of cache based on Doctrine.

As we mentioned in chapter 5.2.1 we decided to go ahead and implement the EntityManager based automatic caching solution. Implementing this solution was as easy as expected, only taking about a day, and provides many benefits over the other solution with a custom method as mentioned in 4.2.2. Still, this other solution could be valuable if only a few of an application’s queries were suitable for caching, if the application is just very small or if the potential extra maintenance of a custom EntityManager sounds frightening.

Also when it comes to invalidation of cache we made a decision to implement one of the solutions as described in chapter 5.2.2, namely the solution including an interface because of time constraints. Despite its tight coupling it serves its purpose well for the time being, but I would strongly recommend Persiennbutiken to consider the second, more advanced, invalidation solution.
will discuss this second solution’s importance in greater detail in chapter 6.3.

When it comes to the actual payment implementation we had two suggested implementations. The first being using PayPal’s own PHP SDK. As the result show in chapter 5.3 this suggestion scores comparatively low with 16 points. Mainly suffering from not being easily extendable, in fact not extendable at all, to be used by other payment processors. This is to be compared by the 27 points which the PayumBundle is able to muster thanks to its freely available payment processor/gateway extensions. From the score it is clear who is the real winner here, but PayPal PHP SDK may still be a viable option if one considers PayPal’s SDK to be easier to integrate than PayumBundle, do not want to mess with Doctrine storage and if no other payment processors are likely to be integrated in the future.

Just as with the currency implementations, all solutions have the potential of being secure. How we feel about PayPal’s security is mentioned in chapter 6.2. Performance wise we are confident that the performance measures taken will be enough to offer a fast experience even for the most impatient customers. With regards to device customisation the story is slightly different, the choice of payment implementation could affect its appeal on different devices, as seen in the statistics in chapter 2.2. Gladly, both of the solutions support the aspect of saving card details, a particular important aspect for mobile users. PayPal’s interface also boosts a responsive design which is sized appropriate on all kind of devices.

Compared to the currency implementations, all but the direct PayPal SDK solution rely on the combination of Symfony together with Doctrine. That makes the payment implementations slightly less useful for web sites using different frameworks, but could still be used as a source of inspiration.

With seemingly few drawbacks and despite longer development time than initial expected, I would recommend the PayumBundle solution to Persiennbutiken. Its biggest benefit being that it let Persiennbutiken relatively easily extend with more payment processors in the future. Something which I also recommend Persiennbutiken to do based on the statistics in chapter 2.2.

### 6.2 Ethical aspects of the research

We briefly touched upon privacy concerns with statistics gathering in chapters 2.1 and 3.2. Despite this we decided to go for the most aggressive form of Google Analytics available. We argued that thanks to no personal identifiable information being collected, Swede’s would be fine as the research we mentioned earlier indicates. The data is also, we believe, securely stored on Google’s servers, minimising the risk of data theft.

Making people buy things which they perhaps do not really need is always controversial, something which we are trying to achieve with psychological pricing. If psychological pricing results into a customer buying more things, it could potentially be a waste of resources and an environment concern. The latter
being a constant growing concern as we keep receiving depressing reports about the state of the environment from institutions such as United Nations Environment Programme. However, with our psychological pricing we strive to make the customer choose higher quality materials, rather than buy more and as with any other for profit business we will always try to maximise the profit as long as the customer does not suffer.

When dealing with sensitive, personal information, which Persiennbutiken does when for example collecting address information, it would be morally irresponsible to have known security risks floating around on the live website. To avoid this issue we have enforced stringent security requirements outlined in chapter 3.5, which all solutions, at least, have the potential to fulfil. When it comes to the payment implementation this has also involved moving the responsibility of storing card data from our website to a third party, in this case PayPal, as discussed in chapter 3.6. One could argue that Persiennbutiken still has some responsibility because it made a choice of storing card data with PayPal. It is a valid point, but we have made the decision in good faith that initiatives like PayPal’s Bug Bounty Program [81] will encourage security researchers and hackers to let PayPal fix any security flaws they find rather than selling them to criminals or releasing them to the public, leading to very high security.

Going forward, it is also important to make sure that people with access to sensitive data, such as the shop owner, uses secure passwords in order to minimise the risk of unauthorised access. This could be enforced by having privileged users having stricter password requirements then what applies to customer accounts. However, this is an issue for when it is time to implement the user system.

6.3 Future work

From Persiennbutiken’s perspective all what remains is to embrace the results given and strongly consider implementing the suggested solutions.

For the general development community, it is worth paying particular close attention to the suggested database caching solution with cache groups. If implemented and made open source as a bundle, it could be a huge benefit not only to Persiennbutiken, but to any website which needs a simple, reliable and performant way of dealing with cache invalidation. Today there does not seem to be any alternative offering this kind of functionality, with the closest probably being the DoctrineCacheInvalidatorBundle [82], which unfortunately does not seem to be in active development.

Another suggested implementation worth paying extra attention to is the extended PayPal REST extension to Payum offering Doctrine support. If further refined from the example given in chapter 4.2.3 it could very well be included in the official Payum or PayumBundle repositories. The main issue that is stopping me from sharing my complete solution to this problem is that it is not very generic and tightly coupled to the underlying PayPal SDK. I would like to further research the problem and see if it is possible to make the solution more generic.
Perhaps there is a way to only have to redefine the internal fields with appropriate Doctrine annotations, instead of having to define custom variables and override the getter and setter methods for every domain class.

6.4 Final words
The project has successfully suggested solutions to the problems presented in 1.2 and I would of that reason regard the whole project as successful. That the project was successful was easy to verify thanks to the detailed performance measures presented in chapter 3. Apart from ease the verification of the result, they also helped to steer the development in the right direction by defining clear goals which was then taken into consideration when developing the solutions. Despite the project being successful, it would have been beneficial to actually implement the suggested implementations in order to get hard numbers on development time, instead of estimates, and it could also have revealed unforeseen advantages and disadvantages.

The chosen development methodology’s importance is clear; it is thanks to the methodology that the research was at all possible. If we would have chosen a stricter methodology such as the Waterfall method, we would have had missed the communication, planning and modelling stages, which this research is, a long time ago. While the report’s conclusions are mainly specific to Persiennbutiken, the research has also presented some interesting ideas, see chapter 6.3, which should be useful for any e-commerce site, or web site for that matter, particularly if using Symfony.
References

Retrieved 2016-05-17.


Retrieved 2016-05-12.


dia.org/wiki/Object-relational_mapping

ual/en/preface.php

http://www.php-fig.org/psr/
Retrieved 2016-05-12.


cess_memory

dia.org/wiki/Representational_state_transfer
Retrieved 2016-06-03.

dia.org/wiki/Software_development_kit

dia.org/wiki/Search_engine_optimization


state_drive

dia.org/wiki/World_Wide_Web

Persiennbutiken – The construction of a web shop using Symfony
Oscar Reimer 2016-06-06


Retrieved 2016-03-25.

Retrieved 2016-03-25.

[33] Google, “Enable Remarketing and Advertising Reporting Features in Analytics”,
https://support.google.com/analytics/answer/2444872?hl=en-GB&utm_id=ad&authuser=2
Retrieved 2016-03-25.

[34] Internetstiftelsen i Sverige (iis), “Personlig integritet, övervakning och politik”,
http://www.soi2014.se/personlig-integritet-overvakning-och-politik/
Retrieved 2016-03-25.

Retrieved 2016-03-25.

Retrieved 2016-03-25.

[37] Post- och telestyrelse, “Frågor och svar om kakor för webbplatsinnehavare”, https://www.pts.se/sv/Bransch/Regler/Lagar/Lag-om-elektronisk-
Persiennbutiken – The construction of a web shop using Symfony
Oscar Reimer 2016-06-06

Retrieved 2016-03-25.


Retrieved 2016-04-02.

Retrieved 2016-04-01.

Retrieved 2016-04-02.


Retrieved 2016-04-03.

Retrieved 2016-04-03.

Persiennbutiken – The construction of a web shop using Symfony
Oscar Reimer 2016-06-06

[60] GitHub, HHVM, “Issues”, https://github.com/facebook/hhvm/issues?q=is%3Aopen+is%3Aissue+label%3A%22php7+incompatibility%22
Retrieved 2016-05-05.


Retrieved 2016-05-06.

Retrieved 2016-05-06.

Retrieved 2016-05-06.

Retrieved 2016-05-06.


Persiennbutiken – The construction of a web shop using Symfony
Oscar Reimer

Retrieved 2016-05-17.

Retrieved 2016-05-17.

Retrieved 2016-05-17.

Retrieved 2016-05-17.

Retrieved 2016-05-17.


[76] GitHub, “Payum/PayumBundle”, [https://github.com/Payum/PayumBundle](https://github.com/Payum/PayumBundle)

[77] GitHub, “Payum/Payum”, [https://github.com/Payum/Payum](https://github.com/Payum/Payum)


[80] GitHub, “PayPal-PHP-SDK/Payment.php”, [https://github.com/paypal/PayPal-PHP-SDK/blob/a37b880e96e270303c7e131fcccde6425c1e7972/lib/PayPal/Api/Payment.php](https://github.com/paypal/PayPal-PHP-SDK/blob/a37b880e96e270303c7e131fcccde6425c1e7972/lib/PayPal/Api/Payment.php)

Appendix A: Code

This section contains code examples referenced throughout the report.

Code 3 A typical long, but reasonably fast, SQL query taking about 3.17 ms. The mentioned query has been optimised and uglified/minimised by our ORM tool, Doctrine.

```sql
SELECT w0_.word_source AS word_source_0, w1_.word_source AS word_source_1, r2_.id AS id_2, r2_.position AS position_3, r3_.position AS position_4, r4_.id AS id_5, w5_.id AS id_6, w5_.word_source AS word_source_7, w5_.translateable AS translateable_8, w6_.id AS id_9, w6_.word_source AS word_source_10, w6_.translateable AS translateable_11, w7_.id AS id_12, w7_.word_source AS word_source_13, w7_.translateable AS translateable_14, r8_.id AS id_15, r8_.position AS position_16, w9_.id AS id_17, w9_.word_source AS word_source_18, w9_.translateable AS translateable_19, w10_.id AS id_20, w10_.word_source AS word_source_21, w10_.translateable AS translateable_22, r11_.position AS position_23, p12_.id AS id_24, p12_.image_name AS image_name_25, p13_.id AS id_26, p13_.path AS path_27, p14_.id AS id_28, p14_.extension AS extension_29, r15_.position AS position_30, p16_.id AS id_31, p16_.image_name AS image_name_32, p17_.id AS id_33, p17_.path AS path_34, p18_.id AS id_35, p18_.extension AS extension_36, w19_.id AS id_37, w19_.word_source AS word_source_38, w19_.translateable AS translateable_39, r3_.regulator_category AS regulator_category_40, r3_.regulator_categories_main_characteristic AS regulator_categories_main_characteristic_41, r11_.main_id AS main_id_42, r15_.regulator_category AS regulator_category_43 FROM regulator_categories r2_ INNER JOIN word_ids w0_ ON r2_.name_word_id = w0_.id INNER JOIN word_ids w1_ ON r2_.description_word_id = w1_.id INNER JOIN regulator_categories_characteristics r3_ ON r2_.id = r3_.regulator_category LEFT JOIN regulator_categories_main_characteristics r4_ ON r3_.regulator_categories_main_characteristic = r4_.id LEFT JOIN word_ids w5_ ON r4_.name_word_id = w5_.id LEFT JOIN word_ids w6_ ON r4_.description_word_id = w6_.id LEFT JOIN word_ids w7_ ON r4_.button_word_id = w7_.id LEFT JOIN regulator_categories_main_characteristics_options r8_ ON r4_.id = r8_.main_characteristic LEFT JOIN word_ids w9_ ON r8_.name_word_id = w9_.id LEFT JOIN word_ids w10_ ON r8_.description_word_id = w10_.id LEFT JOIN regulator_categories_pictures r11_ ON r4_.id = r11_.main_id LEFT JOIN picture p12_ ON r11_.picture = p12_.id LEFT JOIN picture_paths p13_ ON p12_.image_path_id = p13_.id LEFT JOIN picture allowed_extensions p14_ ON p12_.image_extension_id = p14_.id LEFT JOIN regulator_category_pictures r15_ ON r2_.id = r15_.regulator_category LEFT JOIN picture p16_ ON r15_.picture = p16_.id LEFT JOIN picture_paths p17_
```
Code 4 Custom EntityManager which caches all queries using both query and result cache.

```php
namespace AppBundle\ORM;

/**
 * This class offers a cached entity manager, where all queries per default
 * uses both query and result cache. Apart from that, it behaves just like
 * the default Doctrine EntityManager.
 *
 * @author <a href="mailto:oscar.reimer@reimerbrothers.com">Oscar Reimer</a>
 */

class CachedEntityManager extends \Doctrine\ORM\Decorator\EntityManagerDecorator
{

    /**
     * Factory method to create EntityManager instances.
     *
     * @param mixed $conn An array with the connection parameters or an existing Connection instance.
     * @param Configuration $config The Configuration instance to use.
     * @param EventManager $eventManager The EventManager instance to use.
     *
     * @return EntityManager The created EntityManager.
     *
     * @throws \InvalidArgumentException
     * @throws ORMException
     */
    public function create($conn, Configuration $config, EventManager $eventManager = null)
    {
        return new self(\Doctrine\ORM\EntityManager::create($conn, $config, $eventManager));
    }

    /**
     * Creates a cached query
     *
     * @param string $dql query string
     */
```
Persiennbutiken – The construction of a web shop using Symfony
Oscar Reimer 2016-06-06

```php
* @param string $cacheId for caching, default none (needs to be cleared by
  * the entity manager's clearAllCache() method)
  * @param int $cacheLifeTime lifetime of cached object, default 3600
  * seconds (1 hour)
  * @param bool $useResultCache defaults to true, set to false to disable
  * result cache
  *
  * $return

  \Doctrine\ORM\Query

  */
  public function createQuery($dql = '', $cacheId = null,
    $cacheLifeTime = 3600, $useResultCache = true)
  {
    $query = parent::createQuery($dql);
    $query->useQueryCache(true);
    $this->applyResultCaching($query, $cacheId,
      $cacheLifeTime, $useResultCache);

    return $query;
  }

  /**
   * Creates a cached named query
   *
   * @param type $name
   * @param type $cacheId for caching, default none (needs to be cleared by
   * the entity manager's clearAllCache() method)
   * @param type $cacheLifeTime lifetime of cached object, default 3600
   * seconds (1 hour)
   * @param type $useResultCache defaults to true, set to false to disable
   * result cache
   *
   * $return

   \Doctrine\ORM\Query
   */
  public function createNamedQuery($name, $cacheId = null,
    $cacheLifeTime = 3600, $useResultCache = true)
  {
    $query = parent::createNamedQuery($name);
    $query->useQueryCache(true);
    $this->applyResultCaching($query, $cacheId,
      $cacheLifeTime, $useResultCache);

    return $query;
  }
```
/**
 * Creates a cached native SQL query.
 * @param type $sql
 * @param Doctrine\ORM\Query\ResultSetMapping $rsm
 * @param type $cacheId for caching, default none (needs to be cleared by
 * the entity manager's clearAllCache() method)
 * @param type $cacheLifeTime lifetime of cached object, default 3600
 * seconds (1 hour)
 * @param type $useResultCache defaults to true, set to false to disable
 * result cache
 * @return Doctrine\ORM\NativeQuery
 */

public function createNativeQuery($sql, Doctrine\ORM\Query\ResultSetMapping $rsm, $cacheId = null, $cacheLifeTime = 3600, $useResultCache = true)
{
    $query = parent::createNativeQuery($sql, $rsm);
    $this->applyResultCaching($query, $cacheId, $cacheLifeTime, $useResultCache);
    return $query;
}

/**
 * Creates a cached named native SQL query.
 * @param type $name
 * @param type $cacheId for caching, default none (needs to be cleared by
 * the entity manager's clearAllCache() method)
 * @param type $cacheLifeTime lifetime of cached object, default 3600
 * seconds (1 hour)
 * @param type $useResultCache defaults to true, set to false to disable
 * result cache
 * @return Doctrine\ORM\NativeQuery
 */

public function createNamedNativeQuery($name, $cacheId = null, $cacheLifeTime = 3600, $useResultCache = true)
{
    $query = parent::createNamedNativeQuery($name, $cacheId = null, $cacheLifeTime = 3600, $useResultCache = true);
    $this->applyResultCaching($query, $cacheId, $cacheLifeTime, $useResultCache);
```php
return $query;

/**
 * Applies result caching with given parameters
 * @param type $query the query to apply caching to
 * @param type $cacheId for caching, default none (needs to be cleared by
 * the entity manager’s clearAllCache() method)
 * @param type $cacheLifeTime lifetime of cached object, default 3600
 * seconds (1 hour)
 * @param type $useResultCache defaults to true, set to false to disable
 * result cache
 */
private function applyResultCaching(&$query, $cacheId, $cacheLifeTime, $useResultCache)
{
    if ($useResultCache)
    {
        $query->useResultCache(true, $cacheLifeTime, $cacheId);
    }
}

/**
 * Creates a CachedQueryBuilder instance.
 * @return \AppBundle\ORM\CachedQueryBuilder $qB
 */
public function createQueryBuilder()
{
    return new \AppBundle\ORM\CachedQueryBuilder($this);
}

/**
 * Clears all query and result cache (it does NOT clear meta cache)
 */
public function clearAllCache()
{
    $queryCacheDriver = $this->getConfiguration()->getQueryCacheImpl();
    $queryCacheDriver->deleteAll();
    $resultCacheDriver = $this->getConfiguration()->getResultCacheImpl();
    $resultCacheDriver->deleteAll();
}
```
A custom QueryBuilder which lets the user specify cache options when getting query. Used by the CachedEntityManager.

```php
/**
 * A QueryBuilder which lets you specify whether or not to use result cache
 * and its settings. Per default it uses result cache with a sane life time.
 * Otherwise identical to Doctrine's standard QueryBuilder.
 * @author <a href="mailto:oscar.reimer@reimerbrothers.com">Oscar Reimer</a>
 */
class CachedQueryBuilder extends \Doctrine\ORM\QueryBuilder
{
    /**
     * Constructs a Query instance from the current specifications of the builder.
     *
     * @param type $cacheId for caching, default none (needs to be cleared by
     * the entity manager's clearAllCache() method)
     * @param type $cacheLifeTime lifetime of cached object, default 3600
     * seconds (1 hour)
     * @param type $useResultCache defaults to true, set to false to disable
     * result cache
     */
    public function getQuery($cacheId = null, $cacheLifeTime = 3600, $useResultCache = true)
    {
        $query = parent::getQuery();
        $query->useResultCache($useResultCache, $cacheLifeTime, $cacheId);
        return $query;
    }
}
Use PayPal SDK domain class with variables for Doctrine storage.

```php
use Doctrine\ORM\Mapping as ORM;
use PayPal\Api\Payment as BasePayment;

/**
 * Entity used to contain PayPal payment information using Doctrine as storage
 *
 * @ORM\Table(name="payum_payment_paypal", options={"charset":"utf8mb4","collate":"utf8mb4_general_ci"})
 * @ORM\Entity(repositoryClass="AppBundle\Repository\PayumPaymentPayPalRepository")
 * @ORM\HasLifecycleCallbacks
 *
 * @author <a href="mailto:oscar.reimer@reimerbrothers.com">Oscar Reimer</a>
 */
class PayumPaymentPayPal extends BasePayment
{
    /**
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     *
     * @var integer $idStorage
     */
    private $idStorage;

    /**
     * @ORM\Column(name="paypal_id", type="string", length=255, nullable=true)
     *
     * @var string $id
     */
    private $id;

    /**
     * @ORM\Column(name="intent", type="string", length=255, nullable=true)
     *
     * @var string $intent
     */
    private $intent;

    /**
     * @ORM\OneToOne(targetEntity="PayPalPayer", cascade={"persist"}, fetch="EAGER")
     */
```
private $dPayer;
/**
 * @ORM\OneToOne(targetEntity="PayPalPotentialPayerInfo",
cascade={"persist"}, fetch="EAGER")
 * @ORM\JoinColumn(name="potential_payer_info", referencedColumnName="id")
 */
private $dPotentialPayerInfo;
/**
 * @ORM\OneToOne(targetEntity="PayPalPayee", cascade={"persist"}, fetch="EAGER")
 * @ORM\JoinColumn(name="payee", referencedColumnName="id")
 */
private $dPayee;
/**
 * @ORM\OneToMany(targetEntity="PayPalTransaction", cascade={"persist"}, mappedBy="dPayment", fetch="EAGER")
 */
private $dTransactions;
/**
 * @ORM\Column(name="billing_agreement_tokens", type="simple_array", nullable=true)
 */
private $dBillingAgreementTokens;
/**
 * @ORM\OneToOne(targetEntity="PayPalPaymentInstruction",
cascade={"persist"}, fetch="EAGER")
 * @ORM\JoinColumn(name="payment_instruction", referencedColumnName="id")
 */

```php
* @var \AppBundle\Entity\PayPalPaymentInstruction $dPaymentInstruction
*/
private $dPaymentInstruction;
/**
 * @ORM\OneToOne(targetEntity="PayPalRedirectUrls", cascade="{"persist"}, fetch="EAGER"
 * @ORM\JoinColumn(name="redirect_urls", referencedColumnName="id")
 * @var \AppBundle\Entity\PayPalRedirectUrls $dRedirectUrls
*/
private $dRedirectUrls;
/**
 * @ORM\OneToMany(targetEntity="PayPalLink", cascade="{"persist"}, mappedBy="dPayment", fetch="EAGER")
 * @var \Doctrine\Common\Collections\ArrayCollection $dLinks
*/
private $dLinks;
/**
 * @ORM\Column(name="state", type="string", length=255, nullable=true)
 * @var string $dState
*/
private $dState;
/**
 * @ORM\Column(name="experience_profile_id", type="string", length=255, nullable=true)
 * @var string $dExperienceProfileId
*/
private $dExperienceProfileId;
/**
 * @ORM\Column(name="note_to_payer", type="string", length=255, nullable=true)
 * @var string $dNoteToPayer
*/
private $dNoteToPayer;
/**
 * @ORM\Column(name="failure_reason", type="string", length=255, nullable=true)
 */
```
Persiennbutiken – The construction of a web shop using Symfony
Oscar Reimer 2016-06-06

```php
/*
 * @var string $dFailureReason
 */
private $dFailureReason;

/**
 * @ORM\Column(name="create_time", type="string",
length=255, nullable=true)
 * @var string $dCreateTime
 */
private $dCreateTime;

/**
 * @ORM\Column(name="update_time", type="string",
length=255, nullable=true)
 * @var string $dUpdateTime
 */
private $dUpdateTime;

/**
 * Loads the internal hash map fields after we have loaded the
 * @ORM\PostLoad
 */
public function loadInternalFields()
{
    $this->id = $this->dId;
    $this->intent = $this->dIntent;
    $this->payer = $this->dPayer;
    $this->potential_payer_info = $this->dPotentialPayerInfo;
    $this->payee = $this->dPayee;
    $this->transactions = $this->dTransactions->toArray();
    $this->billing_agreement_tokens = $this->dBillingAgreementTokens;
    $this->payment_instruction = $this->dPaymentInstruction;
    $this->state = $this->dState;
    $this->experience_profile_id = $this->dExperienceProfileId;
    $this->note_to_payer = $this->dNoteToPayer;
    $this->redirect_urls = $this->dRedirectUrls;
    $this->failure_reason = $this->dFailureReason;
    $this->create_time = $this->dCreateTime;
    $this->update_time = $this->dUpdateTime;
    $this->links = $this->dLinks->toArray();
}
```
public function __construct($data = null)
{
    parent::__construct($data);
    $this->dTransactions = new \Doctrine\Common\Collections\ ArrayCollection();
    $this->dLinks = new \Doctrine\Common\Collections\ ArrayCollection();
}

/**
 * @return int
 */
public function getIdStorage()
{
    return $this->idStorage;
}

/**
 * Magic get method which returns the relevant data based on the key. It will first try to return Doctrine data, and if not found, data from the hash map.
 *
 * @param type $key
 * @return mixed appropriate value
 */
public function __get($key)
{
    switch ($key)
    {
    case 'id':
        return $this->dId;
    case 'intent':
        return $this->dIntent;
    case 'payer':
        return $this->dPayer;
    case 'potential_payer_info':
        return $this->dPotentialPayerInfo;
    case 'payee':
        return $this->dPayee;
    case 'transactions':
        return $this->dTransactions;
    case 'billing_agreement_tokens':
        return $this->dBillingAgreementTokens;
    case 'state':
        return $this->dState;
    case 'experience_profile_id':
        return $this->dExperienceProfileId;
    case 'note_to_payer':
        return $this->dNoteToPayer;
    case 'redirect_urls':
    }
return $this->dRedirectUrls;
    case 'failure_reason':
        return $this->dFailureReason;
    case 'create_time':
        return $this->dCreateTime;
    case 'update_time':
        return $this->dUpdateTime;
    case 'links':
        return $this->dLinks;
    }
    return parent::__get($key);
}
/**
 * Magic set method which attempts to set both the hash map and Doctrine data.
 * @param type $key
 * @param mixed $value data to set
 */
public function __set($key, $value)
{
    if (!$value instanceof Doctrine\Common\Collections\ArrayCollection)
    {
        parent::__set($key, $value);
    }
    switch ($key)
    {
    case 'id':
        $this->dId = $value;
        break;
    case 'intent':
        $this->dIntent = $value;
        break;
    case 'payer':
        if ($value instanceof AppBundle\Entity\PayPalPayer)
        {
            $this->dPayer = $value;
        }
        break;
    case 'potential_payer_info':
        if ($value instanceof AppBundle\Entity\PayPalPotentialPayerInfo)
        {
            $this->dPotentialPayerInfo = $value;
        }
        break;
case 'payee':
    if ($value instanceof AppBundle\Entity\PaypalPayee)
    {
        $this->dPayee = $value;
    }
    break;
case 'transactions':
    /* @var $transaction AppBundle\Entity\PaypalTransaction */
    foreach ($value as $transaction)
    {
        if (!$transaction instanceof AppBundle\Entity\PaypalTransaction)
        {
            break;
        }
        $transaction->setDPayment($this);
        $this->dTransactions->add($transaction);
    }
    break;
case 'billing_agreement_tokens':
    $this->dBillingAgreementTokens = $value;
    break;
case 'state':
    $this->dState = $value;
    break;
case 'experience_profile_id':
    $this->dExperienceProfileId = $value;
    break;
case 'note_to_payer':
    $this->dNoteToPayer = $value;
    break;
case 'redirect_urls':
    $this->dRedirectUrls = $value;
    break;
case 'failure_reason':
    $this->dFailureReason = $value;
    break;
case 'create_time':
    $this->dCreateTime = $value;
    break;
case 'update_time':
    $this->dUpdateTime = $value;
    break;
case 'links':
    /* @var $link \AppBundle\Entity\PayPalLink */
    foreach ($value as $link)
    {
        if ($link instanceof PayPal\Api\Links && !$link instanceof AppBundle\Entity\PaypalLink)
        {
            break;
        }
    }
```php
$newLink = new \AppBundle\Entity\PayPalLink();
$newLink->setDataFromPayPalApi($link);
$link = $newLink;
}
else if (!$link instanceof \AppBundle\Entity\PayPalLink)
{
    break;
}
$link->setDPayment($this);
$this->dLinks->add($link);
}
break;
```