FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Jan Škvařil

Web System for Crowdfunding Based on Selling Items with Custom Imprint

Department of Distributed and Dependable Systems

Supervisor of the master thesis: RNDr. Pavel Parízek, Ph.D.

Study programme: Computer Science

Study branch: Software Systems

Prague 2016

I am much obliged to my thesis supervisor, Dr. Pavel Parízek, for his advices and hints that helped me to successfully implement and write this diploma thesis.

Název práce: Webový systém na prodej předmětů s potiskem pomocí crowdfundingových kampaní

Autor: Bc. Jan Škvařil

Katedra / Ústav: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: RNDr. Pavel Parízek, Ph.D.

Abstrakt: Cílem této práce bylo navrhnout a implementovat webový systém na prodej předmětů s potiskem, který zároveň umožní sbírat peníze pomocí kampaní fungujících na bázi crowdfundingu. Výsledná aplikace automatizuje a usnadňuje celý proces od návrhu potisku v integrovaném editoru přes vyhodnocení kampaně až po odeslání hotových výrobků zákazníkům. Během celého procesu tedy existuje minimum bodů, kdy je nutný administrátorský zásah zvenčí. Částí implementace bylo také napojení aplikace na externí služby, jako jsou on-line platby kartou, tiskárna či sociální sítě.

Klíčová slova: crowdfunding, webový systém, potisk

Title: Web System for Crowdfunding Based on Selling Items with Custom Imprint

Author: Bc. Jan Škvařil

Department / Institute: Department of Distributed and Dependable Systems

Supervisor of the master thesis: RNDr. Pavel Parízek, Ph.D.

Abstract: The aim of this thesis was to design and implement a web system for organization of crowdfunding campaigns that are based on the sale of items with custom imprint. The implemented system automates and facilitates the whole campaign lifecycle from the design of the imprint in an integrated editor through campaign evaluation up to the distribution of final products to customers. Therefore, there is a minimum number of points where the administrator's action is required. Part of the implementation was also seamless integration with external services like on-line payments, printing house, and social networks.

Keywords: crowdfunding, imprint, web system

Contents

1 Introduction

These days, e-commerce has become a common way for selling goods and services. Great examples of this expansion are websites like E-bay or Amazon which has managed to get a significant market share. They both use a standard model of selling items – when a customer buys an item, it is delivered; there are just two main participants involved in the shopping process. But many other ways to sell and distribute goods exist and one of them is called crowdfunding.

Crowdfunding is a practice of funding a project or a venture by raising monetary contributions from a large number of people, typically via the Internet [1]. This model of funding custom projects has experienced a great expansion in the past few years around the world. One of the reasons for that is that there is no financial risk for all the participants involved in the model.

The most well-known model of crowdfunding is what KickStarter.com [2] or GoFundMe.com use. A project initiator creates a new campaign, sets its properties and defines rewards for people who wish to financially support his project. Those rewards are typically graded according to the amount of donation. Then the campaign is launched and shared with potential supporters. When the campaign ends, the crowdfunding platform sends the money to the project initiator and his responsibility is to deal with all the rewards for his supporters.

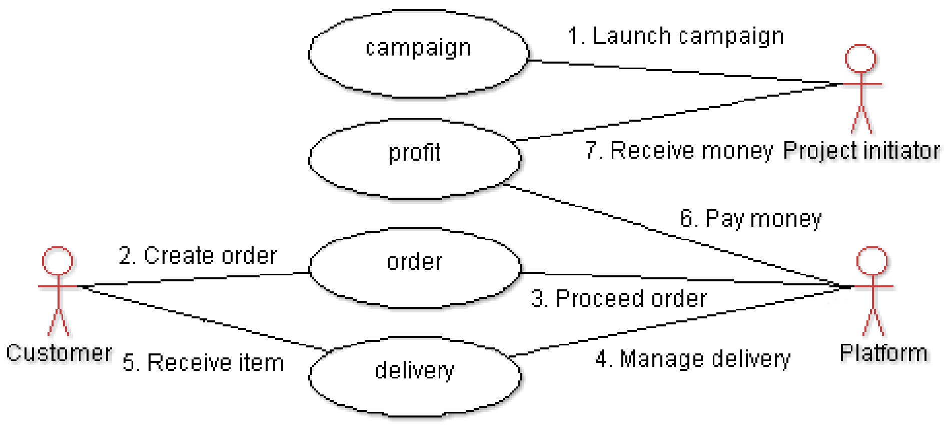


Figure 1.1: Interaction of project initiator, customer and platform in the ideal case.

Our goal is to create a new crowdfunding platform with a less common business model which is suitable for groups of people who wish to sell items with a custom imprint, e.g. T-shirts, or fund their project by selling those items. A simplified interaction of involved participants in this model is illustrated in Figure 1.1. As we can see, there are three major participants involved: the platform, the project initiator and the customer.

The first step is similar to the standard crowdfunding model – a project initiator launches a new campaign and shares it among its potential supporters. But as a part of the campaign he has to define the design of the imprint. In contrast with the previous model there are no rewards for the project supporters, i.e. customers, but technically speaking we could consider the delivered items as rewards.

When the campaign ends successfully, the platform's responsibility is to create the ordered items, distribute them to the customers and transfer the profit to the project originator. This step is also different from the standard crowdfunding model where it is the project initiator‘s responsibility to reward his customers. This way we avoid one disadvantage of the standard model which is related to the trustworthiness of the platform. It happens from time to time that the project initiator just collects the money and does not reward his customers at all.

1.1 Motivation

Our original motivation is to create a successful crowdfunding platform which is able to succeed in the real world. Our vision is to end up with a practically useful system tested on real users, with real campaigns, cooperating with real external services and handling with real money, not just something what ends up in a drawer. A great inspiration for our platform is TeeSpring [3] which we discovered after the idea of implementing the crowdfunding system and which we would like to accommodate for our local conditions.

Selling T-shirts is often a way to collect money for some community or public project; project supporters buy the T-shirts and some percentage of that money is used to fund the project. There are also many groups of people such as bands, sport teams, classmates or even firemen that would like to create their own community T-shirts but they typically have to solve several problems related with that idea. First of all, they have to create a T-shirt design in a format suitable for a printing house and sign a contract with that printing house.

The problem is that they usually have no idea how many T-shirts they should create and in which colours and sizes. When they estimate such properties, they risk creating too many or too little T-shirts of each kind. Then they have to collect money from people in their community, send an order to the printing house, pick-up the T-shirts from the printing house and deliver them to those people.

This is a quite complicated process that could be simplified from the point of view of the project initiator. For those people it would be a great simplification to have a platform where they could create a custom design of their community T-shirt, share it across the community via social media and then just wait until their T-shirts are printed, delivered and then simply obtain their profit. This is our motivation for creating such a platform which shields the end users from the technical details of the T-shirt creation, the money collection and the delivery of final products.

1.2 Goals

The main goal is to design and implement a web crowdfunding system that allows the project initiator to design an imprint, set the campaign properties like its deadline and goal and launch the campaign. After the launching, the system should generate a special page for the campaign which can be easily shared via social media. On this page the supporters should be able to buy the item with a designed imprint in a selected size and colour. In general, we do not want to limit the system only for a sale of T-shirts. It should be possible to configure it also for a sale of other types of items that can have a custom imprint. But we use T-shirts as a demonstration of the system functionality.

When the campaign ends and it is successful, an order to the printing house should be generated in a suitable format. When the ordered items are created, they have to be manually packed and issued to the delivery service which delivers them to the customers. When the campaign ends unsuccessfully, then we should ensure that no money move at all; therefore, no fees need to be paid; how to solve this problem is described in Section 2.4.3.

One of the important properties of the system is the great level of automation of the system processes. The system should automate processes not only from the point of view of the campaign initiator and the customers but also from the point of view of system administrators. This automation has of course some natural limitations which we will discover and describe in the following chapters. In some cases, we also outline a possible theoretical solution.

Creating a complex web application is a difficult process which involves many technologies, procedures and design patterns. Because of that, first we have to analyse the requirements for such a system and design a suitable solution. Afterwards, we will also focus on an effective testing of the final product from the point of view of functional and also non-functional tests and we discuss the limitations of those approaches, especially when testing the integration with third party services.

1.3 Basic Terminology

In this section we explicitly define important terminology and concepts used throughout the rest of this thesis.

Project: In this thesis we understand a project as “something that is able to be supported”, e.g. an activity, an invention, a community, a single person with some idea etc.

Campaign: An entity in the system which represents the real project. Basic properties of the campaign are its name, deadline, goal, designed imprint and URL.

Customer: The campaign supporter – pays the money for some item, e.g. a T-shirt.

Campaign goal: A minimum number of T-shirts that have to be sold to customers in order to send the T-shirt design to the printing house to produce the real T-shirts. If the goal is not reached, no T-shirts are produced and no money is charged from the customers.

Campaign progress: A fulfilment of the goal of the campaign – a number of ordered T-shirts given the campaign goal; this value is specified in percent.

Running campaign: A campaign which ends in the future.

Successful campaign: A campaign with enough sold T-shirts, i.e. more or equal than the campaign goal.

Unsuccessful campaign: A campaign that is not successful; no T-shirts are printed when the campaign ends as unsuccessful.

Printable area: A rectangle on the T-shirt where the printing house can print to.

1.4 Thesis Structure

As a result of this thesis, we successfully implemented and deployed a web-based system which we call DesignTeeLine.

In the next chapter we present existing solutions and analyse requirements on the final system. We also find out the limitations of the system‘s full automation. In Chapter 3 we design a solution based on the analysis. This also includes designing software architecture and a database structure which fulfils all the requirements.

Chapter 4 describes the final implementation with necessary technical details and then in Chapter 5 we talk about the testing of the DesignTeeLine system.

The last chapter then summarises the whole thesis, evaluates it and presents a list of possible improvements. A user manual, which also includes a description of system installation, can be found as an attachment of this thesis.

2 Analysis

In this chapter we analyse the requirements on the final system, we discuss problems that we discover and design a theoretical solution to those problems. We also search for the most suitable technologies and we point out their advantages and limitations. After reading this chapter, it should be clear what kind of system we want to implement and how to deal with the key challenges.

2.1 User Roles

Here we describe different user roles in the whole campaign lifecycle, their actions, and what they should be capable of in the system. This analysis is based in Figure 1.1, shown in the previous chapter.

- **Unregistered user**: Those users can browse launched campaigns and they can buy a T-shirt by clicking on a buy button in the campaign detail which redirects them to an ordering form which must be filled. Then a payment has to be done in order to consider the T-shirt as bought.

  After buying the T-shirt, those users just wait for the result of the campaign about which they get informed via e-mail. When the campaign is successful, they wait for the T-shirt they bought to be delivered.

- **Registered user**: Those users can do the same actions as the unregistered users, but in addition they can login to their profile and manage it. They also have an ability to launch a new campaign and thus become a campaign initiator, return to some drafts of their campaigns or see the history of their orders.

- **Campaign initiator**: Every registered user can become a campaign initiator when he launches a new campaign. When the campaign ends, he gets notified about its result via e-mail. If the campaign ends successfully, he can request a pay-out in his profile and afterwards the money is transferred to his bank account.

- **System administrator**: A system administrator is responsible for managing all the launched campaigns. When the campaign is successful, he has to send all its orders to the printing house and then pass the created T-shirts to the delivery service. When the project initiator applies for the pay-out, he transfers money to his bank account.

2.2 System Requirements

In this section we define the system requirements based on the informal description of the campaign lifecycle described in Chapter 1 and the different user roles that must be supported. The main parts of the system are an editor, a registration and a login, a campaign detail, an order, a user profile and administration. We discuss the requirements on each of these parts in the following subsections.

2.2.1 Editor

The first step in simplification of a T-shirt creation is the possibility to design a custom imprint for the T-shirt. After analysing existing solutions, we found out that most of them solve this problem with an integrated editor. In our case, this editor should also guide the user through the whole process of launching a new campaign. The required features of the editor include:

- The editor must be able to work with graphics in bitmap and vector formats. We would like to support JPG and PNG from the bitmap formats and SVG and EPS from the vector formats.
- There should be an option for uploading a custom image or to use some artwork from an integrated gallery. Artworks must be easily manageable by the system administrator. They might be in the bitmap or vector format.
- The editor offers an option for writing a custom text on the T-shirt. The user can pick up a font from an integrated font database which is managed by the system administrator.
- The editor supports basic operations with graphical or textural elements such as move, scale, rotate, duplicate, delete and mirror.
- There is a possibility to preview the campaign or share it with other designers via e-mail.

We would also like to support more bases than just T-shirts, for example bags or sweatshirts. Therefore, when creating a new campaign, the user should select a default base and colours in which he wants to sell the item. Adding a new base is not a common operation in the system but it should not be difficult for the administrator to do so. And least but not last, the editor should provide an interface to gather all the parameters of the campaign such as its name, description, deadline, goal, author margin, URL etc.

Now, we need to pick up a suitable format for manipulation with the data in the editor. It should be definitely some vector-based format because we need to preserve the best possible quality of the editor output. After a brief research, we decided to use the SVG format due to its good integration with HTML and JavaScript and its wide support in all modern web browsers. Moreover, libraries for XML manipulation can be exploited for a colour counting on the server side and libraries for manipulation with SVG exist on both, client and server side. Also as we mention in Section 2.6, it can be easily converted to bitmap and a conversion from EPS to SVG can be also performed.

2.2.2 Registration and Login

Registration and login are standard parts of most of modern websites but implementing it right is not as trivial as it might seem. We would like to support three ways of registration and login: via e-mail, via Facebook account and via Google account. The last two of them, which are described in Section 2.4.4, are very convenient for users because this way they can register at one click. Moreover, there is also an advantage for the website because this simplicity encourages more users to register.

Hand in hand with the registration goes a need to implement verification of e-mail addresses, a possibility to reset a password when the user loses it and providing a high level of security.

To fulfil that security requirement we will use standard techniques like storing the user‘s password, encrypted in the database, for the case it is corrupted or limiting the maximum number of login attempts per a given period of time.

2.2.3 **Campaign Detail and List**

Every campaign should have its detail on the URL specified in the editor. This is also the URL that should be shared via social media by the campaign initiator. The main part of this page should contain a generated image of the T-shirt design. On the second part of the page there should be a form where the customer can select a type of textile, its colour and size and click on the buy button which redirects him to the order.

When the campaign is ended, it should still be possible to view its detail, but it should not be possible to buy it anymore. The maximum length of each campaign must be limited because the items are created in one batch after the campaign ends and we do not want to let the customers wait for their T-shirts indefinitely. We limit the maximum length of the campaign to 21 days; another reason for that number is described in Section 2.4.3.

It should be also easy to find some campaign when we know its name or author; therefore, we need to support searching in running campaigns and then clearly list the search results.

2.2.4 **Order**

On the page with the order should be a standard ordering form with a selection of shipping and payment methods. We want to support three payment methods: credit card payments via GP webpay, PayPal payments and direct money transfers.

When one of the first two is selected, the customer is redirected to an appropriate payment gate, fills in all the necessary data and then the payment gateway redirects him back to the pre-specified URL where the result of the payment is visualised. In the third case, we just show a generated variable symbol and instructions how to pay. When the payment is confirmed from the payment service, we must not forget to actualise the progress of the campaign.

When creating a new campaign, the campaign initiator can decide that he wants to allow his customers to pick up their orders at his place. When a customer wishes to pick up his order there, the shipping is for free.

2.2.5 User Profile

In the user profile there should be a list of launched campaigns, a list of drafts created in the editor, a list of orders that the user has made, a possibility to change his address and profile and a section where a withdrawal of collected money is possible. In order to withdraw money the campaign initiator has to make a request and then it is a responsibility of the system administrator to transfer the money to his bank account.

We would like to support simple user actions like re-launching ended campaigns, ending launched campaigns and sending an e-mail message to all the campaign customers.

2.2.6 Administration

We decided to have administration as a part of the system. The system administrator will have all the necessary views at one place and can make a quick conclusion about what is happening in the system at the moment. Another advantage of having the administration is that we can put there functionality for data exports we need as we discuss in Section 2.4.1.

This decision also implies that we need to implement different user roles – at least the unlogged user, the normal user and the system administrator which we have already described in Section 2.1.

2.3 Existing Solutions

In this section we describe similar systems that already exist and their similarities and differences to our system. A table which compares all the similar systems with the DesignTeeLine system can be found in the end of this section.

2.3.1 KickStarter

KickStarter [2] is an American crowdfunding system based on the crowdfunding model described in Chapter 1. Campaigns are time limited and the campaign initiator must define rewards for his supporters when launching a new campaign. The main difference between KickStarter and our system is that in the case of KickStarter there is no need for an editor.

The reason for that is simple: at KickStarter the campaign initiator does not sell any custom goods but just collects money from his supporters. In return, he rewards them with rewards he defines when launching the campaign. The reward might be for example sending a post card, telling a story, dedicating a book etc. It is his responsibility to keep his word and reward all his supporters as he promised. If the reward includes production of some goods, he is responsible for producing them.

In contrast with KickStarter, campaigns in our system are based on selling of items with custom imprint which is designed online in the editor. A campaign initiator does not define rewards as a part of each campaign, but he offers items with designed imprint as a reward, i.e. by buying an item in the campaign, a customer supports this campaign with pre-defined amount of money.

This is also one of the differences between those two systems – in our case a campaign initiator receives constant amount of money from each sold item. In case of KickStarter a customer can choose how much money he is going to give to the campaign. Rewards in case of KickStarter are more of the symbolic nature and these rewards have nothing to do with the platform itself, i.e. KickStarter has no responsibility in delivering rewards to the customers of campaigns. If a campaign initiator does not deliver rewards, it impacts mainly his reputation.

In case of our system, the reason to support a campaign is more pragmatic – a customer wants to buy a nice T-shirt, sweatshirt or other item with an imprint. Supporting a campaign initiator might be perceived as a side effect which might not be noted by the customer at all if he does not learn how the system works. This is because the amount of money which goes to the campaign initiator is already included in the final price of each sold item. Anyway, both of the systems, KickStarter and DesignTeeLine, are crowdfunding platforms; there are just some differences in the philosophy of collecting money which result in different systems with different business processes.

2.3.2 **T-shock**

T-shock [4] is Czech website where a user can design own T-shirt and then buy it. The main purpose of this site is to provide a platform where the user can create an original T-shirt just for himself. For designing of T-shirts they use an integrated editor. Unfortunately, this editor does not support vector images in SVG or EPS.

In addition to that basic functionality, it is also possible to sell the T-shirts to other users, but first a contract between the campaign initiator and the website needs to be signed, which we find as a quite complicated process. Campaigns are not time limited and when some customer buys a T-shirt, they create it immediately. This is an advantage for the customer, but on the other hand it raises the final price of the T-shirt. The campaign initiator also cannot set own price margin; it is fixed to 10 % of the final price.

It is not a crowdfunding platform in its original sense. It is meant as a platform where authors of nice T-shirt designs, who want to earn some money, can sell these designs and make some profit. Because of the absence of time limited campaigns that have some goal, T-shock is pushed to create each ordered item individually and they cannot benefit from discounts for creation of more items of the same kind in one batch. Thus the base price of each T-shirt created on this site is bigger than for T-shirts created with DesignTeeLine.

We found out that from this platform we could take the possibility to pay with direct bank transfers which we discuss in Section 2.4.3. This feature is suitable for our system, because in the Czech environment, customers still appreciate this option.

2.3.3 **TeeSpring**

TeeSpring [3] is an American crowdfunding website where a user can design his campaign in the editor which is created in the very user-friendly way. This platform allows the campaign initiator to launch own campaign in a similar way as in our system. We discovered TeeSpring after our original idea for the crowdfunding system based on the selling of items with a custom design. Anyway, when discovered, it became a great inspiration for us, especially in the area of frontend layout, supported features and the campaign lifecycle.

Nevertheless, we need to design all the processes from the administrator's point of view in our own way, create the administration and implement everything from scratch. We do not know how this platform works inside, what technologies are used to run it, how they managed to integrate their system with external services like payment gateways and what level of automatization they achieved. These are challenges we have to solve on our own.

One of our goals is to adjust TeeSpring's solution, which is in many aspects very mature, to the Czech environment, i.e. use regional payment and delivery services and support features like direct bank transfers which are appreciated by local customers. We did not want to implement all the features offered by TeeSpring or be a competition for this platform which has a really big budget for development.

In time we started to implement the DesignTeeLine system, TeeSpring did not support multiple languages and currencies. Unfortunately they implemented this feature while we were working on our system, thus we lost the advantage of having this feature while they do not support it.

We were also thinking about supporting possibility to design an original T-shirt which is then bought just by its author in the similar way as T-shock or TeeSpring do. Unfortunately, after a discussion about pros and cons, we had to reject this feature, because T-shirts made this way would be too expensive and it is much more work for the system administrator to deal with each order individually then to process all the orders of some campaign at once.

The second feature we were discussing is the list of all the running campaigns. We have implemented this feature and it could be turned on with a little modification of the source code. We do not have it in default, because when there are not many running campaigns in the system, it does not look trustworthy, so it is better to separate campaigns from each other in the beginning.

2.3.4 **Comparison**

In Table 2.1 we show the comparison of systems discussed in previous three sections. This table summarizes the text above and shows differences and similarities between those systems on the set of selected features.

	KickStarter	T-shock	TeeSpring	DesignTeeLine
Crowdfunding platform	✓	✓	✓	✓
Editor for custom items	✗	✓	✓	✓
Design of items just for author	✗	✓	✓	✗

Multiple languages	✓	✗	✓	✓
Multiple currencies	✗	✗	✓	✓
Rewards for supporters	✓	✗	✗	✗
List of all running campaigns	✓	✓	✓	✗
Payment with bank transfer	✗	✓	✗	✓

Table 2.1: Comparison of properties of similar systems.

2.4 **Communication with External Services**

Because the DesignTeeLine system is bound to several external services such as printing house, delivery service, payment services and social networks, we have to analyse the interface between those services and our system. The right integration must also respect certain rules and limitations. This analysis will result in more technical and detailed requirements on the DesignTeeLine system.

2.4.1 **Printing House**

When the campaign ends successfully, we need to pass all its relevant data to the printing house in order to continue in the campaign lifecycle process (the lifecycle is described in detail in Section 3.1.2). After consulting with a real printing house [5] we found out that they require those data sets:

- A PDF file with a table with all the orders – they should easily recognise from this table which T-shirt of which colour and size they should print on.
- Pure graphics of the imprint in the vector format (SVG or EPS); this file should contain only the printable area of the T-shirt. We decided to export the T-shirt imprints in the SVG format because we work with this format in the editor; therefore, there will be no need for the data conversion.

Theoretically, we could fully automate those exports. For example, we could automatically send a generated e-mail with the exported data to the printing house. But we need to have a control over this process because the quality of exported graphics must be manually checked. This brings us to the idea of manual exports, but there is still a question where those exports should be realized.

For example, we could export data automatically, save them in some pre-defined directory and notify the system administrator about a new export via e-mail. Then the contents of the export should be manually validated and sent to the printing house.

As we have said, we cannot fully avoid this manual intervention by the system administrator, but we chose a different approach – we decided to have the administration as a part of our system. The advantage of this approach is its generality – in the administration we can have all the necessary exports and views we need at one place.

There is also another important requirement on our system related to the printing house: a base price of the T-shirt. After the analysis of the price table and consultation with the technical support, we found out that the base price of the final T-shirt depends on a number of used colours from the front and from the back and total quantity of created products. There is also a little surcharge when both sides of the T-shirt have some graphics.

Therefore, we will need those input data in order to count the base price of the T-shirt:

- the number of used colours at the front of the T-shirt,
- the number of used colours at its back and
- the goal of the campaign.

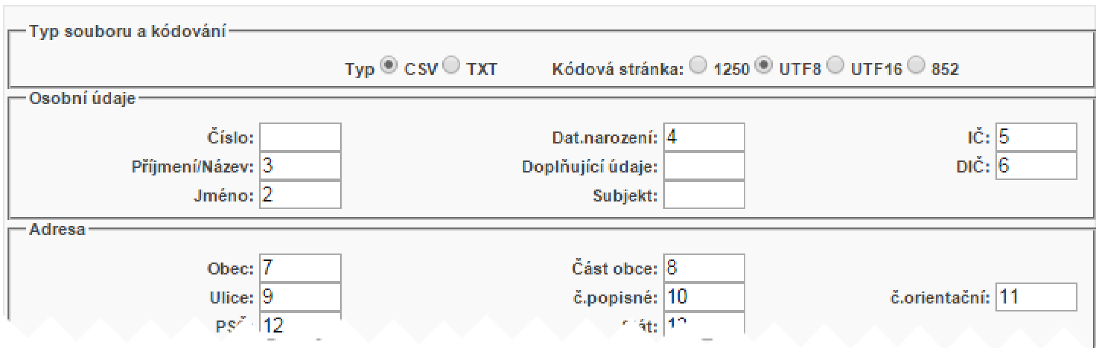
The printing house also provides two types of printing methods: a screen printing and a digital printing. We found out that when too many colours are used or when the campaign goal is too little, we should use the digital printing which is in general more expensive, but in those particular cases it is cheaper. Moreover, for too complex T-shirt imprints it is not technically possible to use the screen printing. The price of the digital printing is constant, i.e. it does not depend on the input parameters.

Creating T-shirts in greater batches is much cheaper than creating just one because a special form for the imprint must be made. This is also the main reason why we need to have the goal at each campaign – too small batches sent to the printing house are just too expensive.

2.4.2 **Delivery Service**

The last step in processing a successfully ended campaign is sending the final products to the customers. We decided to exploit the Czech Post as a delivery service because it fulfils our needs in the cheapest way. We achieved a low final price per T-shirt delivered because T-shirts can be packed into envelopes which are much cheaper than packages.

The Czech post has its own application [6] for printing postal labels for the envelopes. We would like to connect to that application, but unfortunately, it has no useful API and the only thing we can do is to take advantage of their imports in the CSV format. Implementing the export feature in the system administration will be pretty straight forward task, but first, according to the user manual, we have to configure a structure of imported CSV documents. It is not difficult, but little bit confusing; we need to set the numbers of columns in the CSV which correspond to the respecting features as shown in Figure 2.1.


Typ souboru a kódování

Typ [✓ CSV] TXT   Kódová stránka: 1250 [✓ UTF8] UTF16 852

Osobní údaje

Číslo:		Dat.narození:	4	IČ:	5
Příjmení/Název:	3	Doplňující údaje:		DIČ:	6
Jméno:	2	Subjekt:			

Adresa

Obec:	7	Část obce:	8		
Ulice:	9	č.popisné:	10	č.orientační:	11
PS	12	át:	1		

Figure 2.1: Settings of the CSV format in the Czech post application.

In our administration we will also need to generate PDF files with invoices which will be attached to the envelopes passed to the post. Those invoices will be also useful for checking whether the right T-shirt goes to the right address.

It is possible to extend our system for another delivery service but because the Czech post is very specific in the export format, it is very likely that communication with that delivery service would have to be implemented from scratch.

2.4.3 **Payment Services**

The greatest challenge associated with payment services is to invent a workflow that would satisfy all those requirements:

- Full amount of money must be refunded to a customer when a campaign ends unsuccessfully.
- When a campaign ends successfully, money for all the associated orders must be transferred to the platform’s account.
- Avoids fees when transferring money back to the customer’s account.

GP webpay

According to the manual of GP webpay [7] (page 10, Introduction to the system) we can see that each order to the GP webpay has a parameter called DEPOSITFLAG. When this flag is set to “1”, it means that money from customer’s credit card should be transferred immediately. When set to “0”, money should be just blocked for some time and it can be transferred later by calling the “deposit” method of their Web Services.

There is also another interesting method in the GP webpay Web Services called “credit”. By calling this method we can refund the transferred money back to the customer.

As we can see, we have two possible scenarios of interaction with the GP webpay to fulfil the requirements listed above:

1. Create orders with DEPOSITFLAG=0 and when the campaign becomes successful, we can call the “deposit” method for all the associated orders and therefore, transfer money to the platform’s account.

   The advantage of this approach is that we do not have to do anything when the campaign ends unsuccessfully because in this case no “deposit” is called and the money stays at customer’s account. This also means that no fees are charged because there are no real money transfers; it is like nothing happened at all.

   But there is a little catch in that – we do not know in advance for how long the money stays blocked in the customer’s account, so when we call “deposit” after a long time interval it might happen that it fails and no money transfers at all. The failure can happen for example due to a lack of financial resources on a credit card in the moment of the deposit call – the bank guarantees to block the money only for a certain period of time.

2. Create orders with DEPOSITFLAG=1, money transfers immediately and we avoid problem described in the previous scenario. But when a campaign ends unsuccessfully, we would have to call the "credit" method for all the associated orders and refund all the payments.

   In this case money really moves between accounts, so we would not avoid the fees.

As we can see, none of the approaches fulfils all the requirements but we have to pick up one of them. There is a constant fee 2 CZK per transaction at the GP webpay. It is not much, but considering the worst case when campaign has a goal n and n-1 T-shirts are ordered, we find out that the total amount of fees for greater n (e.g. 200) is economically unacceptable.

Regarding the problem mentioned in the first approach, we have been informed directly from the bank that they can guarantee to block the amount on the customer's account for seven days, but technically it is not a problem to deposit money even after three weeks. This is another reason for limiting the maximal length of the campaign to 21 days. We also immediately tested to deposit the money after 21 days and it worked with no problem; therefore, we chose the first approach.

Nevertheless, some risk of failure when depositing money remains, but it holds that sooner the deposition is made, the lower the probability of failure is. Thus we can optimize the first approach by depositing the money right after the campaign becomes successful, not at its end. Moreover, we can create orders at the GP webpay with DEPOSITFLAG=1 when a customer orders a T-shirt of already successful campaign.

Communication with the service is maintained via URLs, i.e. all the exchanged data are passed at the URL. For security reasons the data have to be signed with the self-signed certificate [8] that we obtained from the GP webpay. When our system gets a response from the service, it has to validate its authenticity via the certificate in order to avoid spurious data.

PayPal

According to the technical documentation [9] it is possible to create orders at PayPal with two intents: “sale” and “authorize”. Those work in the same way as DEPOSITFLAG at the GP webpay, thus we can implement for PayPal a similar workflow to the one used for the GP webpay.

For communication with PayPal we will use their REST API for PHP library [10] which encapsulates a lot of functionality and eases the communication with the payment gateway. The most important methods from that library for us will be those:

- PayPal\Api\Payment::create – Creates a new payment at PayPal, it can be created with a “sale” or “authorize” intent.
- PayPal\Api\Payment::execute – This method has to be called after a customer fills in all the credentials at PayPal. It returns an authorisation token which we have to store in the database in order to be able to deposit the money when the campaign becomes successful.
- PayPal\Api\Authorization::capture – Captures the money from the customer’s PayPal account to the DesignTeeLine’s PayPal account. This method requires an authorisation token obtained from the *PayPal\Api\Payment::execute* method.

The same problem as we have discussed in the case of the GP webpay might appear, i.e. transferring money after a long time could fail. But we have not found any time limitation for calling the “capture” method at PayPal documentation and we also tested to deposit the money from several PayPal accounts after 21 days and it worked with no problem.

Bank Transfers

Last of the supported payment methods are direct bank transfers which come with further technical difficulties. First of all, it is not possible to ensure that no fees are paid in case of unsuccessful campaigns. Because this payment method is based on transferring money between accounts, we have to live with that fact.

When a customer decides to pay by a direct transfer, we need to generate a variable symbol for the payment. Later we use that symbol to pair the transaction with the payment in our database. There is a little difference to the previous payment methods: we cannot include the customer's order into the campaign progress until we are sure that the payment has been successfully transferred.

Unlike the other payment methods, the bank transfers take a significant amount of time to complete. Therefore, there is a question what to do with delayed payments, i.e. those that arrive after the campaign end. At least, we should notify the system administrator about that fact, for example by e-mail.

We also need to upgrade our workflow when a campaign ends successfully – when there are some unpaid orders, we need to wait for some time whether they get paid or not. We decided to set this deadline to two days after the campaign ends. After that deadline the system administrator can send the orders to the printing house. Orders assigned to payments arriving after that deadline should be cancelled and those payments should be refunded back to the customers.

When a campaign ends unsuccessfully, we need to manually refund all the payments paid by the direct transfer to the relevant customers. Because the bank house we cooperate with has no API for automated communication, this is another point where the intervention of the system administrator is required. Therefore, we need some view in the system administration where it is possible to list all those payments and then mark them as refunded.

When a new payment is accepted at a bank, we would like to be notified, so we could try to pair that transfer with some order in our system based on the total amount and variable symbol. But missing API complicates that otherwise simple process. Luckily, it is possible to configure an e-mail address where a notification e-mail is sent every time a new payment is received at the bank.

We decided to solve this problem by creating a special mailbox for those e-mails, periodically check its contents for new messages, parse all the necessary information like the amount and the variable symbol from the e-mail body and then try to pair those messages with the payments in our database.

2.4.4 **Social Networks**

From social plugins we need "Facebook like box", "Facebook comments" and "Facebook like button" which we will use in the campaign detail. We also plan to show a Twitter box in the footer of each page. Implementing those social networks plugins is a trivial action in those days; it is just copying of pre-generated HTML snippets that can be obtained from the Facebook developer's page [11] and from the Twitter account.

We would also like to implement login with Facebook and Google account. After the analysis of both APIs we found out that communication protocols with those services are very similar and they conform to the protocol illustrated in Figure 2.2.

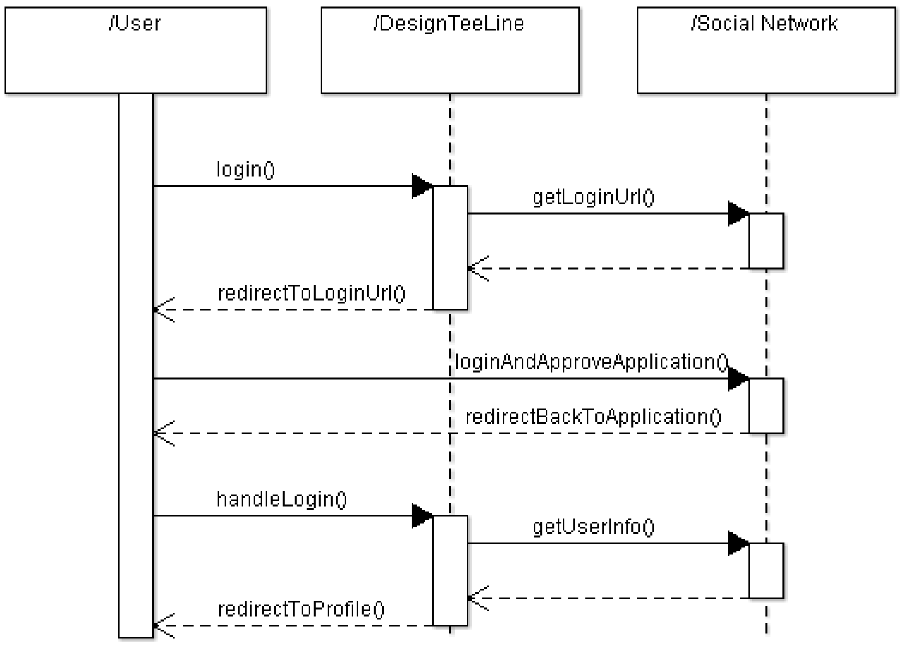


Figure 2.2: Communication with social networks when user logs in.

1. Obtain an URL for login from the social network API and pass it URL where DesignTeeLine handles the results of the login process.
2. Redirect user to the URL obtained from API.
3. User logs into social network and eventually approves our Facebook or Google application.
4. Social network redirects him back to the URL we passed it in the first step.
5. DesignTeeLine uses social network API to get an e-mail of logged user at social network; we have to handle possible errors in this step.
6. When we do not find the e-mail in our database, we automatically register the user.

7. Login the user with given e-mail and redirect to the original URL.

For communication with Facebook, we can use their PHP SDK and for communication with Google we can communicate by curl extension of PHP.

2.5 Manual Actions

In this section we present a summary list of points in the processes where the action of the system administrator is required. As we have seen, those actions cannot be fully automated for various reasons:

- Exporting and sending data to the printing house.
- Exporting orders for the Czech post.
- Refunding of directly transferred payments of unsuccessfully ended campaigns.
- Refunding directly transferred payments delayed for more than two days after the campaign end.
- Sending the profit to the campaign initiator when the campaign is successfully ended and the campaign initiator requested for pay-out.

2.6 Used Technologies

This section is dedicated to the technologies we decided to use and we also reason why we made such decisions. We discuss technologies on both server and client side and we also point out some libraries used for image generation.

2.6.1 Server Side

On the server side we use PHP, but this language has many weaknesses; therefore, we would like to pick up some framework that would shield us from them. Another good reason for the framework is that it eases the programmer's life and it implements many features suitable for almost every application like the security against attacks or debugging tools.

There are many PHP frameworks, but only few of them have earned an excellent reputation over the time. From those we can name Zend framework, Symphony or

Nette framework [12]. All of those frameworks are highly reliable, well-documented, well coded and they all lead the programmer to well-designed applications.

The main reasons why we picked-up the Nette framework are that it is still quite light-weight, it is really easy to learn to work with it and it has a great performance. The last reason is not as important as the first two of them because performance of web applications depends also on other aspects such as how many resources are loaded, the size of the required resources, the usage of cache on the client side etc. But still it is nice to have a framework that does not consume too many resources on the server.

2.6.2 **Client Side**

On the client side we will have only a few simple scripts – those which make the UI more user-friendly. Therefore, we need some light-weight framework which would shield us from the differences in JavaScript interpretation between different web browsers and which would provide us a suitable API for common tasks like manipulation with a DOM tree.

We decided to use jQuery [13] as the JavaScript framework because it is perfectly suitable for our needs, it is well-documented, reliable and these days it has become almost the standard. We can also exploit jQuery functionality to help us with the implementation of features in the client part of the editor which is the only more complex script at the client side.

Because we need to work with the SVG format at the client part of the editor, we also need to pick up a library for this task. We were choosing between Raphaël [14] and svgjs [15] libraries. They both seemed like mature solutions, but Raphaël had a little bit better documentation; therefore, we finally picked up this library. Today we would probably choose Snap.svg [16] which is a new generation of Raphaël library but this library did not exist at the time of this decision.

2.6.3 **Image Generation**

When a new campaign is launched, all of the images that preview the sold items with the designed imprint have to be generated. This includes all the images in predefined resolutions in all the colours selected in the editor. We need those images for the campaign lists and for the campaign detail where we want to provide a preview of the item when a customer selects its colour.

To make the image generation work we have to find useful technologies for conversion of the imprint design in SVG format to the PNG image of the item with the imprint.

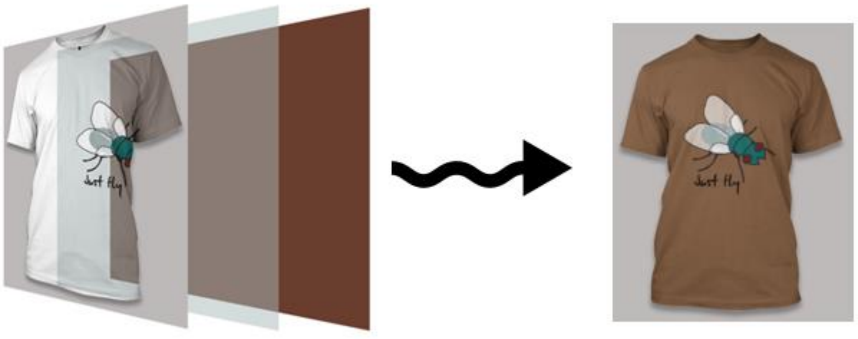


Figure 2.3: Layers of the generated image.

When generating an image in the specified colour, first we have to be able to create a temporal SVG image with three layers which are illustrated in Figure 2.3.

- In the bottom layer is a rectangle with the item colour.
- The middle layer contains the design of the imprint created in the editor. In Figure 2.3 it is the layer with the fly and text “Just fly”.
- The top layer covers everything up; it is a special PNG image of the item where the shape of the item is transparent in order to make the design in the middle layer visible. There are also shades on the image which make the impression of a plastic look of the final image with the layers merged. The surroundings of the item are made in grey to match the design of the web.

First we tested the SVGlib [17] for PHP but it did not suit our needs because we were not able to make it work with custom fonts which we want to use in the editor. The second library we tried was the Imagick [18] extension for PHP with support for the SVG format. But to make the Imagick work with the custom fonts, they have to be installed in the operating system. See the installation instructions in the user guide attached to this thesis.

For conversion of EPS images to SVG format we picked up the ps2pdf [19] program in combination with Inkscape [20]. Those can be easily launched from PHP script.

3 Design

In this chapter we explicitly define the processes which we need to implement. Then we create the database structure of our system which results from the previous analysis and from those processes. And as an important part of this chapter we describe the design of the system architecture.

3.1 System Processes

In this section we use UML activity diagrams to present processes that we later implement. Those diagrams are based on the previous analysis of the system; they summarize all the processes and make them easier to understand. Together with each diagram we provide a detailed textural specification of each process. Definition of those processes is very important because later we need to implement them. They are crucial for the intuitive administration of the system and for all the system users to track their campaigns, orders, payments and pay-outs.

3.1.1 Campaign

As we can see in Figure 3.1, every campaign starts as a draft and it might stay in this state forever. When the campaign is successfully launched by its initiator, it switches to the "launched" state. The campaign remains in this state until it reaches its end date, then it switches to the "inactive" state. This transition can be also taken when the user manually ends the campaign before its end date as specified in Section 2.2.5.

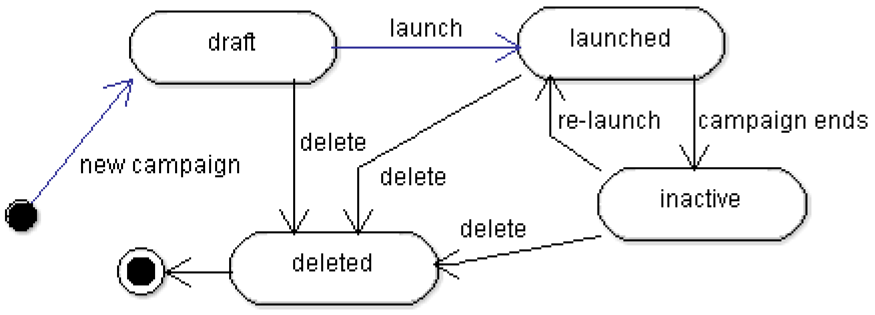


Figure 3.1: Lifecycle of a campaign.

The campaign can be re-launched and then it passes back to the "launched" state. This loop can be taken as many times as the user considers appropriate.

The user that owns the campaign should be allowed to delete it only when it is in the “draft” state, otherwise it might get messy for example when the campaign has some orders associated. On the other hand, the system administrator can delete the campaign at any state because he is aware of all the consequences. Campaigns should not be physically deleted from the database, but just marked as deleted, thus they could be restored in case of false deletion. It is also clear that no campaign in the “deleted” state should appear in the administration or on the frontend.

3.1.2 **Launch**

Because every campaign can be re-launched several times, we need to analyse the lifecycle of campaign launch separately. A new launch is created every time when we enter the “launched” state in the diagram in Figure 3.1. As shown in Figure 3.2, each launch is created in the “running” state. When the launch becomes successful, it is switched to the “successful” state and all customers that made their orders are informed by an e-mail message.

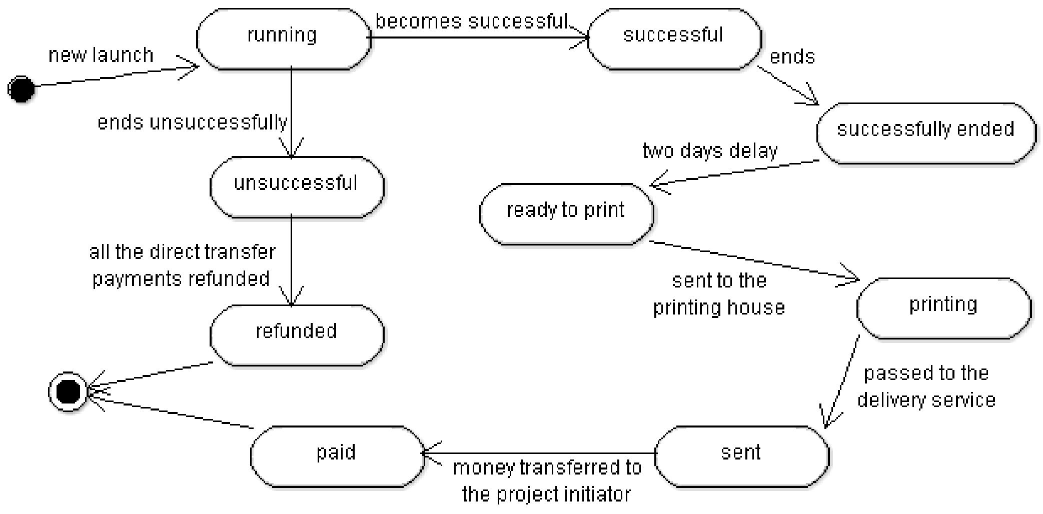


Figure 3.2: The lifecycle of a campaign launch.

Afterwards when it ends, it takes the transition to the “successfully ended” state. When the goal of the campaign is not reached till the launch end, the transition to the “unsuccessful” state is taken. The campaign initiator and all the customers are informed about those two transitions by e-mail.

When the transition to the “unsuccessful” state is taken and there are some orders successfully paid by direct transfers, the system administrator is informed by e-mail about that situation. In the “unsuccessful” state the system administrator is responsible for a manual refund of all the direct transfer payments. Then the launch can be switched to the “refunded” state.

In the “successfully ended” state the system administrator has to wait for two days for delayed bank transfers, then the launch is switched to the “ready to print” state and it can be exported and sent to the printing house. In our diagram the “printing” state represents this part of the scenario.

When the items are received from the printing house, the administrator can export information about the orders for the Czech post. Then all the T-shirts are packed and passed to the Czech post, the launch is now in the “sent” state. When earned money is transferred to the campaign initiator, the launch lifecycle terminates in the “paid” state.

3.1.3 **Payment**

When a customer fills in the ordering form, a new order and an associated payment are created. We will split the design of payment lifecycle to two parts: payments paid by the GP webpay or PayPal and payments paid by direct bank transfers.

The diagram in Figure 3.3 presents the lifecycle of payments paid by the GP webpay or PayPal. In that case, each payment is created in the “unauthorized” state. If there is no response from the payment gateway, for example when the customer closes the web browser, it can last in that state forever.

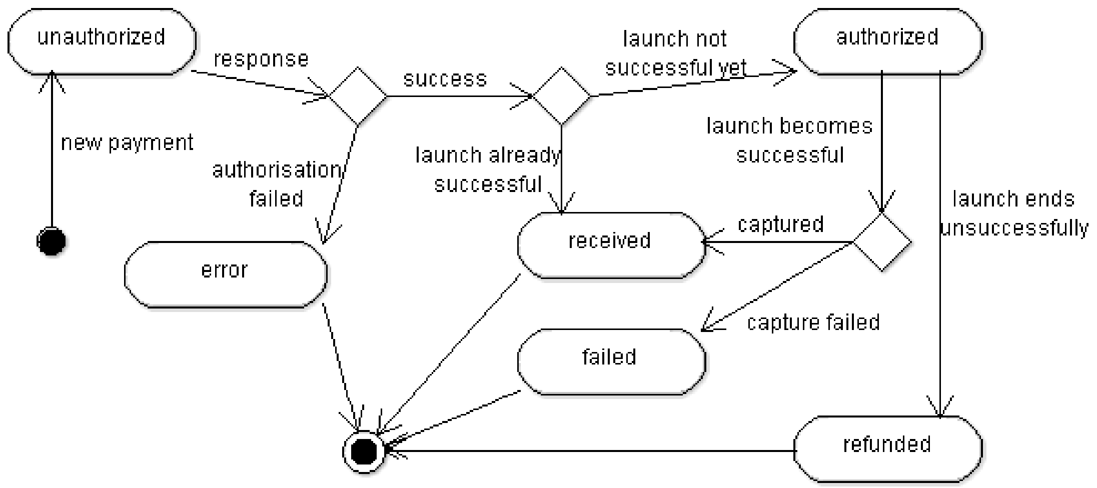


Figure 3.3: The lifecycle of payment paid by GP webpay or by PayPal.

When our system receives a response from the payment gateway, it checks whether the payment was successfully authorised. Payments for orders of already successful launches are immediately switched to the “received” state; payments for orders of launches still in the “running” state are switched to the “authorized” state.

The transition between “authorized” and “received” states is taken when the launch associated with the payment becomes successful. When it ends unsuccessfully, we switch the payment to the “refunded” state, but we do not have to take any action as we analysed in Section 2.4.3.

Also we have to count with a possibility of a payment failure. In our process, this can happen at two places:

- When the payment is authorised after its creation.
- When the payment is automatically captured after the campaign goal is reached.

The first type of error can happen due to a lack of financial resources on a credit card, an invalid credit card, timeout at the payment gateway and many other reasons. When the error occurs, our system is informed in the response from the payment gateway about its nature. The system should log the error and switch the payment to the “error” state, then the process terminates. We should also inform the customer by showing an appropriate message.

The second type of error is much more serious because we have already included the order to the launch progress and suddenly we did not receive the money. The system administrator has to be immediately informed about that situation by e-mail and he has to handle it. But based on the analysis, we assume that this type of error should happen really rarely.

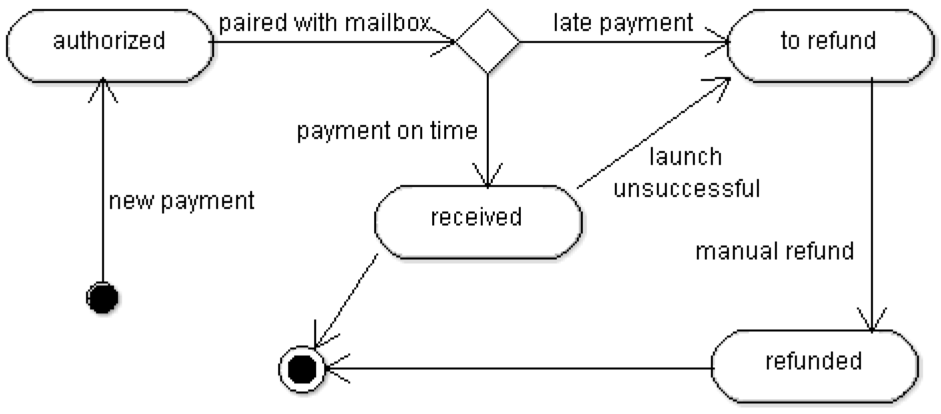


Figure 3.4: The lifecycle of payment paid by direct transfer.

The figure 3.4 shows the lifecycle of payments realised by direct bank transfers. In this case each payment starts its lifecycle immediately in the “authorized” state. There it waits to be received, i.e. it waits until we are able to automatically pair the payment with some e-mail message in the dedicated mail box as mentioned in Section 2.4.3. It can also happen that the customer decides not to pay, and then the payment stays in the “authorized” state forever.

If the launch ends unsuccessfully and the payment has already been received, it is switched to the “to refund” state. If the payment is delayed more than two days from the launch end, it is switched to the “to refund” state immediately. The system administrator is responsible for refunding the payment and switching it to the “refunded” state manually.

The customer should receive an e-mail message when his payment is successfully paired with some message in the dedicated mailbox. In this e-mail he should be informed about the current state of his order and payment, i.e. whether the payment was received on time and what is the current state of the associated launch. The customer should be also informed via e-mail when his payment is manually refunded.

3.1.4 **Order**

We already have most of the information about each order in the state of associated payment and launch. We can obtain the lifecycle of an order as a combination of the lifecycle for a launch and the lifecycle for a payment. But not every combination of states is meaningful, e.g. sent order which has payment in the error state. In the implementation we have to guarantee that only meaningful combinations of states can be reached.

We decided to implement only three explicit states for orders and the rest of them we infer from the state of associated launch and payment. Each customer is informed precisely about the state of his order in the order detail view in his profile. A diagram of order states is shown in Figure 3.5.

Each order is created in the “reserved” state where it remains until it is marked as unsuccessful which can happen from many different reasons – the payment failed, the launch is unsuccessful, the payment is too late etc. When the launch is marked as passed to the delivery service, all paid orders are transferred to the “sent” state. Theoretically, we could also infer this state from the state of associated launch, but we wanted to be able to mark each order individually as sent in the administration.

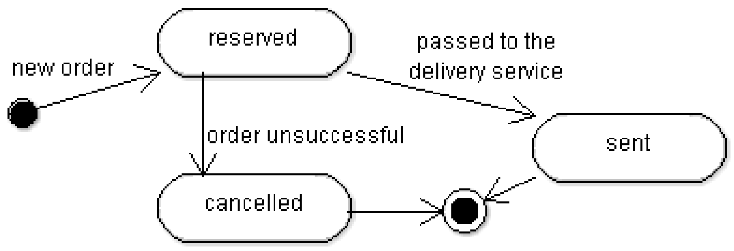


Figure 3.5: Lifecycle of an order.

We can also easily extend the diagram with states that could not be inferred from the associated launch or payment. For example, if we decide to add a new “delivered” state after the “sent” state, it will be easy to extend this diagram.

3.1.5 **Pay-out**

The lifecycle of pay-out demonstrated in Figure 3.6 is very simple. Each pay-out is created when the transition from the “successfully ended” state to the “ready to print” state is taken in Figure 3.2. Then the pay-out waits until the campaign initiator requests the money. The system administrator can see all the pay-outs in the “requested” state in the administration and it is up to him to transfer the money to the campaign initiator’s bank account. Afterwards, he has to manually switch the pay-out to the terminal “paid” state.

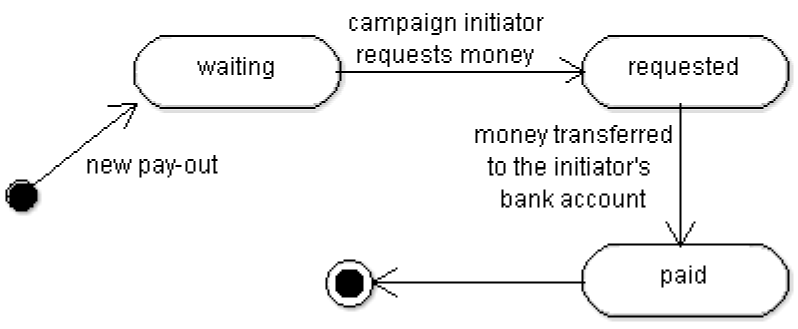

campaign initiator requests money
waiting
requested
new pay-out
money transferred to the initiator's bank account
paid

Figure 3.6: Lifecycle of a pay-out.

3.2 **Database Structure**

Now we design a structure of a database suitable for our system. First of all, we need to identify the entities present in the system, their relationships and properties. This will result in a structure of the database tables – for each entity we create a separate table. The resulting database structure is based on our previous analysis of the system requirements and the processes described in the previous section.

3.2.1 **Motivation**

When designing a database structure, there are two opposing trends between which we need to find a balance. On one hand, we would like to have a database with minimal redundancy of information so we could pick up some of the normal forms [21] and split our tables to correspond with that form. But that approach also increases the total number of tables, and therefore it increases the overall number of joins needed in the queries and decreases performance of those queries. On the other hand, there is the requirement for high performance and we need to find equilibrium between those two.

One of the ways to provide a database schema with low redundancy preserving also a low number of joins is usage of views. We tried to create views that would save us the necessity of using joins, but when we tested those views with a few thousands of entities in the campaign and launch tables, the performance of the queries was behind a tolerable level – some simple selects took almost a second.

The performance of the final application is essential for us; therefore, we decided not to strive for some normal form and accept the database with a higher level of redundancy. The consistency of the database is provided on the application level as discussed in Section 3.2.6.

3.2.2 **Campaign**

The central entity in our system is the campaign. As we have analysed in Section 3.1.2, each campaign can have zero or more launches; therefore, another entity that should be reflected in the database structure is a campaign launch.

The most important properties of each campaign identified during the system analysis are:

- Campaign name and description.
- Campaign alias which has to be unique upon all the campaigns because the URL of page with campaign detail is based on that alias.

- Campaign goal and its length in days.
- Data for the imprint design.
- Number of colours in the imprint design from front and from its back.

We could easily infer the knowledge about the current state of each campaign illustrated in Figure 3.1 from the related campaign launches – when the campaign has no launches, it is a draft, when its newest launch ends is in the future, the campaign is launched and when its newest launch ends is in the past, it is inactive. For the deleted state we could have a column with a flag indicating that state.

We identified those properties that characterize each launch: a launch start date, launch end data and its state. Similarly, the current state of each launch can be easily inferred from its other properties. E.g. a launch is successful when it is not ended and more items were sold than the goal of the associated campaign. Only the states "printing" and "sent" would have to be stored explicitly.

In the first version of our implementation we inferred the state from other properties where it was possible because this approach decreases the redundancy of the information in the database, thus it also decreases the possibility of its inconsistence. But we still had to store some information about the state explicitly and as a result it was an inconsistent mixture between the explicit and implicit state storage which was hard to maintain. With this approach, it is also much more complicated to extend the system processes when necessary and the code of the application is more error prone because we cannot work with the state directly.

Therefore, we decided to redesign the database structure and rewrite the code of the application to support the states of individual entities explicitly. We use the same set of states and transitions as described in Section 3.1.

We can see the entities related to the campaign in Figure 3.7. The pay-outs table serves to both campaign initiators and system administrators and its purpose is the same for both of them – to provide a brief history of money paid and received. The category table stores the information about the supported categories which can have a hierarchical tree structure. Each campaign has to be placed in some category.

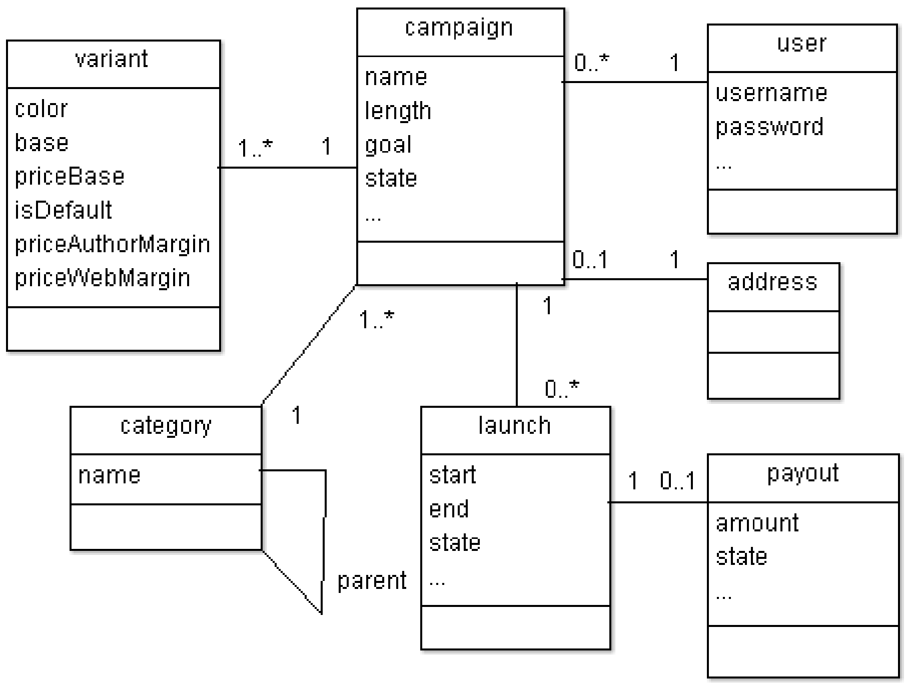


Figure 3.7: Database schema of campaign-related entities.

Because in the editor the campaign initiator can check that he wants to allow his customers to pick up their orders at his place, we need to store the address of the place somewhere. This is a reason for the relation to the “address” table. Tables for variants and users are discussed in the following subsections.

3.2.3 Sale Variant

As we have already said, we store designs of imprints in the table for campaigns. But the campaign initiator can decide that he wants to sell the item in more than one colour; therefore, we need to store that information somewhere too. We could create a special table for colours in which the design is sold and it would solve our problem. But we decided to solve this task in a more general way.

In the analysis we have decided that we want to have our system extensible in a way that we could easily add new items for sale, i.e. new bases. With that in mind, we decided to create a table for sale variants of designs which are available to the customers. Every variant is a concrete item in a concrete colour, e.g. a red T-shirt or a green sweatshirt.

One great advantage of this separate table for campaign variants is its flexibility. In the future, we can easily add new items – those will be just new rows at that table. We can also explicitly set which variants we want to offer to the customers – from a minimum where we offer just one variant to a maximum where we offer all the possible variants (but this would require too much space for each campaign, in fact all the combinations of possible items and their colours) and anything in between.

We decided for a compromise: the campaign initiator can pick up a default base (a default item which is offered to his customers in the campaign detail) in the editor and up to four colours in which the campaign will be available. After the launch the customers can buy any of the supported items with the designed imprint in the colours that were selected in the editor. E.g. when the system supports selling of T-shirts, bags and sweatshirts and the campaign initiator wants to have as default base a T-shirt and he wants to sell the items in red and in yellow, then those variants are stored in the variants table: a red T-shirt, a yellow T-shirt, a red bag, a yellow bag, a red sweatshirt and a yellow sweatshirt.

This design of the table with sold variants also improves the modifiability of the system. For example, it is possible to refactor the mechanism for generating the variants in the way that the campaign initiator selects exactly which variants he wants to sell. This change would impact only on the code of the editor frontend and backend but it would not affect the database schema.

Now we can summarize all the columns that we need in our "variant" table for sale variants:

- Variant colour and used base.
- The column indicating whether it is a default variant used in the campaign detail.
- Base price which represents the costs to create an item.
- Author's margin on the item which represents the amount of money he earns when the item is sold.
- System margin which represents the profit of the platform when the item is sold.

The final sale price of each variant is a sum of the base price, the author's margin and the web margin.

3.2.4 **Order**

Another important entity in the database is an order. Each order is always related to some campaign launch. The customer can order more items at once, which leads us to the table with the ordered variants. As we can see in Figure 3.8, this table is related to both, the table with the orders and the table with the campaign variants, thus providing information which variants of which launch the customer ordered.

With each order there has to be an associated payment where we need to explicitly store its state in order to successfully implement the process described in Section 3.1.3. Because there is a 1-1 relation between orders and payments, it might make sense to merge those two tables into one and save some necessary joins. We decided to prefer a distinction between the payments and the orders because this way the database structure is more extensible. For example, if we decide to store also payments that are not related to any order, for example, payments with a wrongly set variable symbol, we could easily do so.

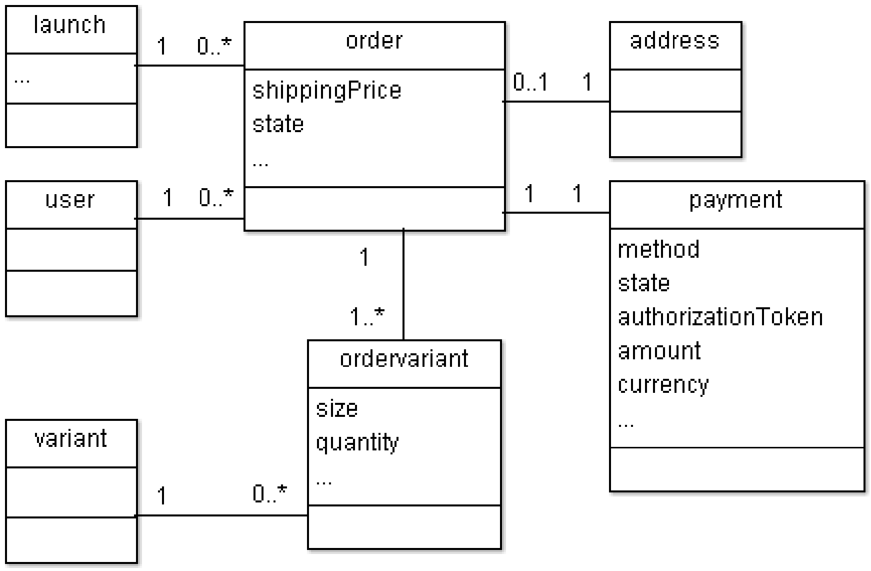


Figure 3.8: Database schema of order-related entities.

In the sake of simplicity, each order is also related to exactly one user. But on the other hand, we need to support orders of non-registered users too. We solved that issue by introducing a row with unregistered user in the table with users and all the orders of non-registered are related to that user. We should also mention that we recycle the “address” table to store the address where the order should be delivered.

3.2.5 **User**

We have already mentioned the table with users. This table also plays a central role in the system and its purpose is obvious – to store information about registered users. We need to distinguish different user roles as a consequence of having the system administration. There is a natural many-to-many relationship between those roles and users although we do not get a benefit out of that fact at the moment. Therefore, we have two additional tables in the database structure – first of them for the user roles which we call the user groups, and second of them for the relations between those groups and users. All of the mentioned tables and their relations are shown in Figure 3.9.

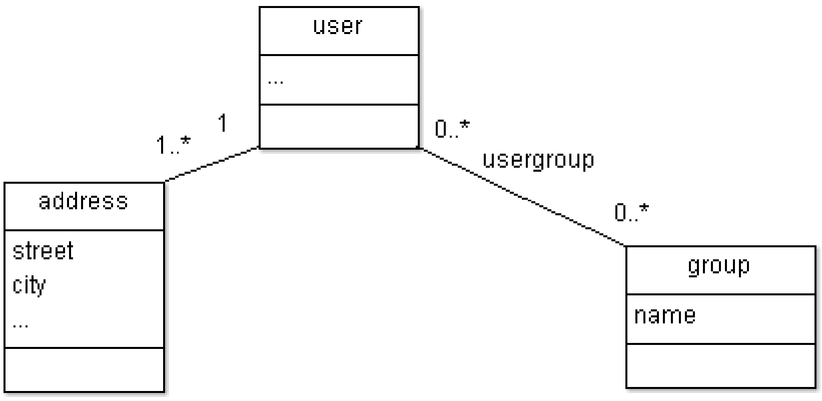


Figure 3.9: Database schema of users and their relation to the groups.

We decided to support the possibility of multiple addresses associated with one user and we recycle the table with addresses for that purpose. In the implementation we support only one address per user, but the database structure is ready for multiple addresses and in the future the application may be simply extended in that way. We just shortly recapitulate the types of addresses stored in the "address" table:

- Addresses of users where they wish to deliver their orders.
- Address of each order where it should be delivered.
- Address related to a campaign. When the campaign initiator decides to allow his customers to pick up their orders at his place, he needs to fill in this address.

3.2.6 Discussion

There are more tables used in the database structure, but we do not analyse them because they are less important for the overall application functionality. As an example of such a table, we can mention a table for queue of e-mail messages that should be sent or a table for tokens that are used during the registration or when a user forgets a password.

Later on during the performance optimizations we wanted to reduce a total number of queries to the database, especially at the campaign detail and at the homepage. For example, we wanted to write a progress of the campaign at its detail. In order to get that information we need to count the orders associated with the current launch of the campaign, i.e. it is necessary to perform a special query for that.

We managed to get rid of that query by altering the schema of the launch table where we added a new column for caching a total number of orders associated with the launch. This increases the redundancy of information in the database, but it is not difficult to manage a database consistency of the database when we have a single point in the application where update that column, i.e. we have one place where we add a new order and update the cache column. We call that place a repository and it is described in Section 3.3.3.

There are more cache columns in the database structure that reduces the overall number of database queries or their complexity. For example, we can mention a column “preferedColor” in the “campaign” table which stores a value of the default colour from the associated table with campaign variants (i.e. a colour from the table “variants” where “isDefault” is set to 1 and it is a variant associated with the given campaign).

The consistency of the database is provided on the application level, i.e. in the repositories. We also wrote a few test queries that check whether the database is consistent or not. Not consistent database would mean an error in the application implementation.

3.3 System Architecture

When designing system architecture, first we need to define the functional requirements. We have already defined most of them in Sections 2.2, 2.4 and 3.1.

The next step is to decompose the system to smaller units of decomposition in order to fulfil those requirements and in the end of this section we shortly focus on the quality attributes of the DesignTeeLine system.

3.3.1 **Overall View**

We are going to implement a web system with a typical overall structure. The client is a user with a web browser who performs HTTP requests to the server. The responsibility of the server is to handle those requests, prepare a response and send it back to the client over HTTP.

There are two separated parts of the system:

- Client side which consists of scripts executed in the client web browser.
- Server side which consists of the database and scripts executed on the server.

We will focus on the server side because it is much more interesting from the point of view of the system architecture. The overall structure of the system is shown in Figure 3.10 and as we can see, there is a database layer which stores the persistent data. The Nette framework handles all direct communication with the database layer and provides a great abstraction of communication for the rest of the system. We will iterate over all the parts of the system illustrated in Figure 3.10 in the following subsections.

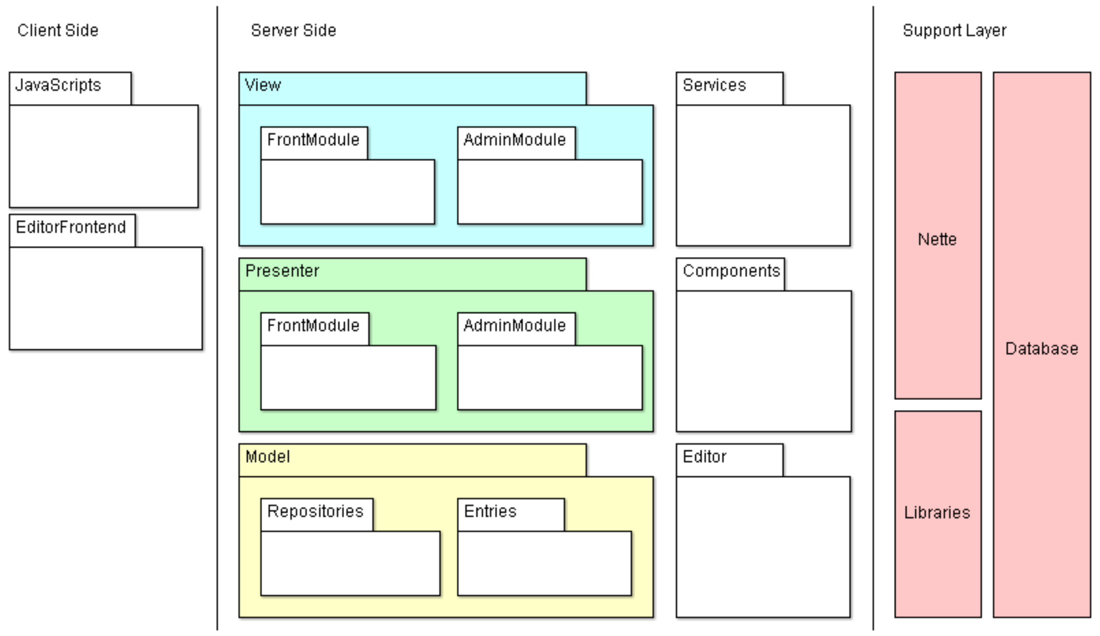


Figure 3.10: Basic decomposition of the system.

Our architecture is highly influenced by the usage of the Nette framework. One of the core functionalities of the framework is the support of a three layer design pattern called Model-View-Presenter. We briefly describe what each of those layers do:

> *“Model is data and especially functional base of whole application. It contains application logic. Model is managing its internal state and provides solid interface outside. By calling methods of this interface, we can read or update its state. Model does not know anything about view or controller.*
>
> *View is an application layer that is taking care of displaying result of request. For that purpose it uses the Latte templating engine.*
>
> *Presenter handles requests from the user, calls relevant application logic from the model and then asks view to display data.” [22]*

3.3.2 **Presenters**

Each presenter is a PHP class which extends the base class *Nette\Application\UI\Presenter*. The action of a presenter is a piece of code that is called when a certain user request appears. Each presenter defines a set of those actions thus providing a code to handle related user requests. The Nette framework also abstracts from the mapping of HTTP requests to the actions of presenters. It is enough to provide a list of routes that describe this mapping between URLs and relevant presenter actions.

Actions of the presenters are always represented by a pair of methods called *action<ActionName>* and *render<ActionName>* where *<ActionName>* represents the name of the action. In the implementation, one of those methods might be omitted. We will describe the presenters and their actions that we need to implement later in this section.

The usual workflow, when a new HTTP request arrives to the server, is as follows. Firstly Nette asks the router to find the appropriate presenter and action to handle the request and then the action is called. The action asks the model layer for the relevant data, passes those data to the view, renders the view and returns a response to the original request.

We start with decomposing the system into the presenters and their actions which we need to handle. We divide those presenters into two modules: a module for frontend users and module for the administration, but here we describe only a decomposition of the frontend module and the most important presenters and their actions.

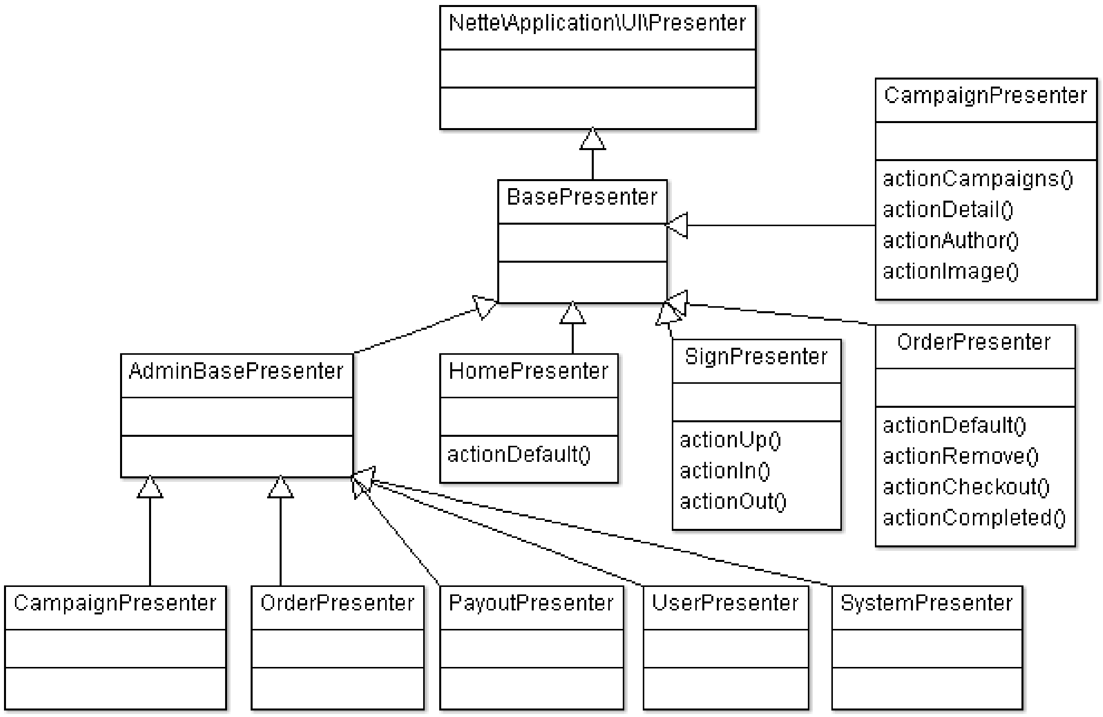


Figure 3.11: Hierarchy tree of presenters.

We should also mention that all the presenters in both modules extend the class called *BasePresenter* where the functionality common to all the presenters is located. A hierarchy tree of the most important presenters is illustrated in Figure 3.11.

The communication between separate presenters is never done directly, i.e. by calling the methods of one presenter by another presenter in the code. Presenters communicate with each other on the level of HTTP requests via links, via forms and via sharing a state. This state is always captured on the level of the model layer. Some of the communication workflows are described further in the text together with the presenter actions.

Homepage Presenter

Homepage presenter serves as a presenter for the application homepage. It has only one default action which asks the appropriate view to render the homepage content.

Campaign Presenter

The responsibility of this presenter is to handle all the actions related to the listing of the campaigns. We need to implement those actions:

- **Campaigns**: An action for listing the campaigns as a result of the search. When the user writes a string into the search form, the search component redirects him into this action.
- **Detail**: An action for handling the request for showing a particular detail of the campaign. The campaign is identified by its alias which is defined when the campaign is launched. This alias is passed as a parameter in the URL. This action has to handle the situation when no such campaign exists.
- **Author**: An action for listing all active campaigns from a particular author. Each author can set his own nickname in his profile. When this nickname is set, then it can be passed as a parameter to that action and the campaigns created by that author are listed. This action has to handle the situation when no such author exists.
- **Image**: An action for generation of an image of a particular campaign. Which image (i.e. its resolution, colour of the item, which item etc.) should be generated is defined in the parameters of this action.

Order Presenter

The responsibility of that presenter is to show the order form, redirect the user to the payment service, accept the response of the payment service and show the result of the communication with payment gateway to the customer. We need to implement those actions in order to fulfil all payment scenarios described in Section 2.4.3:

- **Default**: An action which renders the ordering form.
- **Remove**: An action for removing an item from the shopping cart, this action should redirect the customer back to the default action.
- **Checkout**: An action where the user is redirected after a successful creation of the order and the payment, i.e. when the ordering form is successfully sent. It redirects the customer to the payment gateway or if he selected to pay by bank transfer, it just redirects him to the action called “completed”.
- **Completed**: An action which receives a response from the payment service, validates it and shows the result of the payment to the customer. If everything is ok, it also asks the model to update the state of the payment.

Sign Presenter

Sign presenter serves for implementation of the processes related to the registration and login of the users. But most of the application logic is located in the handling of the forms, not in the actions. There are also two additional presenters that handle the login via Facebook and Google; those are called *FacebookPresenter* and *GooglePresenter*.

- **Up**: Shows the registration form.
- **Confirm**: When a user performs a registration, an activation e-mail message is sent to his e-mail address. The link in that message points to this “confirm” action. An activation token is passed in the URL as a parameter to that action. We have to check the validity of the token (whether it is not expired, it exists in the model, etc.) and activate the user associated with the token or inform the user about an error.
- **In**: Shows the login form.
- **Out**: Performs the logout of currently logged in user and redirects him back to the previous action.
- **Forgot**: Shows the form for handling the situation when a user loses his password.
- **Renew**: A link in the e-mail sent by forgot password form points to that action. A confirmation token is passed in the URL as a parameter to that action. We have to check the validity of the token and if it is not valid, we redirect to the “forgot” action. If the token is valid, we show the form for setting a new password.

User Presenter

The aim of the user presenter is to handle all actions in the user private section where the user is allowed only when logged in. The actions of this presenter are:

- **Default**: Lists all the launched and inactive campaigns that the user has created.

- **End Campaign**: Ends the campaign that is currently in the launched state and belongs to the logged in user. A campaign id is passed as a parameter to that action.
- **Edit Campaign**: Shows the form which allows the logged in user to change the properties of the campaign (i.e. its name, description etc.). The campaign id is passed as a parameter to that action.
- **Contact Buyers**: Shows the form which allows a campaign initiator to contact all the buyers of his campaign. The campaign initiator can fill in the subject of an e-mail and its text and the system plans to send all the e-mail messages to the relevant customers.
- **Re-launch**: Re-launches the campaign which is currently in the inactive state and belongs to the logged in user. The campaign id is passed as a parameter to that action.
- **Drafts**: Lists all the campaigns that are in the “draft” state. User can go back to the editor and continue in creation of the campaign.
- **Delete Draft**: Deletes the draft of a campaign which belongs to the logged in user. The draft id is passed as a parameter to that action.
- **Orders**: Lists all the orders that the user has created. The user can see there the state of each order and the associated payment.
- **Order Detail**: Shows a detail of a particular order that belongs to the logged in user. User can check which products he has ordered and how much did it cost.
- **Profile Info**: Allows the user to change his profile, like the nick.
- **Address**: Shows the form for changing the address for delivering orders.
- **Change Password**: Allows the user to change his password.
- **Get Paid**: Shows a summary of all user pay-outs. It also shows the form that allows the user to request the money he has earned.

3.3.3 **Models**

We build a model layer above the database layer in order to abstract from the fact that our application is using a database. For each table in the database we will create a class called repository which will encapsulate all the queries to the table. Data returned from the repository are instances of model classes – each row in the database is returned as an instance of the particular model class.

From the logical point of view, repositories are used to perform tasks which change the state of the model layer or to retrieve the relevant model entries and pass them to the caller. Repositories are also used as single accesses point to the model entries. This uniformity of access eases the localization of bugs because we can easily localize the place where the bug might be.

But some data entries are not located in the database at all, they are just located in the memory because they are static data (i.e. their state does not change over the time) and there are not a lot of them. By defining those entries directly, we save some database queries and in some cases we do not have to connect to the database at all.

We have two different types of repositories: repositories with model entries that are reflected in the database and repositories with models that are not reflected in the database. In the case of non-database repositories, the model entries are stored directly in the repositories and in the case of database repositories, the repository creates an illusion of that. Therefore, repositories abstract from the fact that some models are located in the database and some are not and they provide a uniform access to both of them.

In Figure 3.12 we illustrate the hierarchy of the repositories and entries with the most important methods. There are also three generalizations that are not visible in the figure: all the classes in the namespace *Application\Model\Entry* extend the *Entry* class, all the classes in the orange part of the *Application\Model\Repository* extend the *Repository* class and all the classes in the blue part of the *Application\Model\Repository* extend the *DatabaseRepository* class. We did not include those generalizations into the figure in order to preserve its clarity.

Some of the entries in the repositories located in the orange part of the namespace *Application\Model\Repository* are so simple that they do not need to be represented by a separate class; it is enough to represent them as a *Nette\ArrayHash* instance. The artworks used in the editor are an example of that approach.

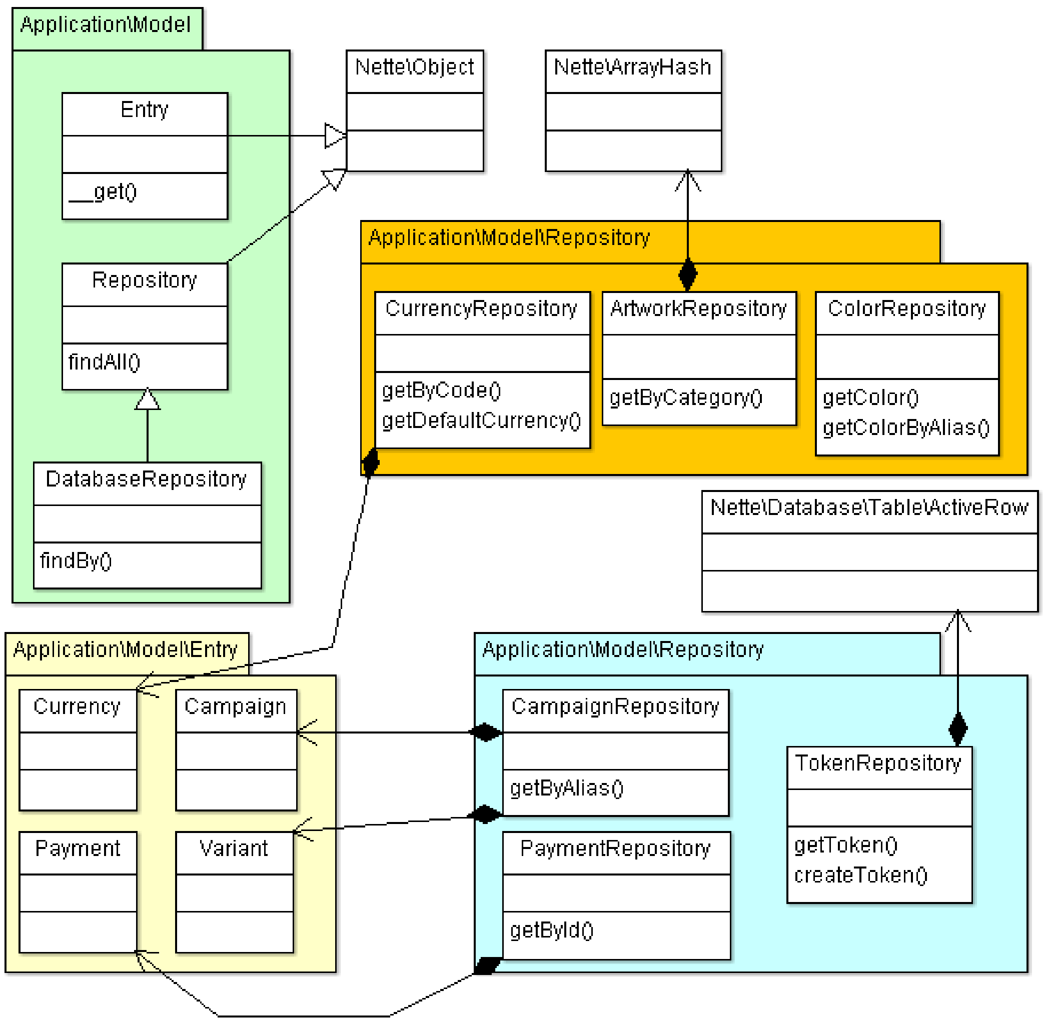


Figure 3.12: Part of the class hierarchy of the repositories.

Some entries are even simpler, for example colours. A colour is just a pair which has a name and an RGB code; therefore, we can represent them in the repository as an array where the key is a colour name and the value is its RGB code. But some entries are more complex and we need to represent them with a separate class even though they are not database entries. We need a separate class when we need to call some methods on the entries, for example methods for currency formatting.

We can represent simpler entities reflected in the database as instances of the class *Nette\Database\Table\ActiveRow*, for example tokens can be represented that way. But when we need to call some special methods on the entities, we need to represent them as a separate class. Those model classes which extend the class *Entry* behave like proxies – they wrap an original instance of *ActiveRow* and delegate the requests for unknown properties to the original object, but they also add a new functionality.

The most complex repositories are the *CampaignRepository* and *OrderRepository* which work with more than one database table. For example, the *CampaignRepository* works with the table for campaigns, their launches and sale variants.

3.3.4 **Components**

Another important concept of the Nette framework that we get benefit of are the components. Components in the context of Nette framework are PHP classes which represent renderable objects. Those might be forms, menus, inputs and so on. We do use some components as separate classes because we need make them reusable. For example, the sign in form and the sign up form, menus etc.

Components are always created by factory methods in the presenter whose name confirms to a pattern *createComponent<ComponentName>* where <ComponentName> is the name of the created component. Other components are created on a fly – we do not need to create a separate class for them. For example, when we need to create some form in the presenter we can use the class *Nette\Application\UI\Form* and then design the form on the fly.

3.3.5 **Services**

Services are PHP classes whose purpose is to provide functionality that is not directly related to none of the layers from the MVC design pattern. An example of this functionality might be a service for caching, a service for minification of JavaScripts or a service for communication with the payment gateway. Those classes are located in the *Application\Services* namespace and they usually extend the *Nette\Object* class.

3.3.6 **Editor**

Editor has a bit special role in the system architecture because it is the only part of the system which consists from two inseparable units: its frontend and its backend. The frontend of the editor is written in JavaScript and its backend is written in PHP.

Even though those two parts do not make sense without each other, we wanted to provide a loosely coupled interface between them. The communication between editor frontend and its backend is realised with AJAX [23] queries which transfer data in the JSON [24] format.

In the backend there is *EditorPresenter* with *actionAjax* which handles all the AJAX queries. This presenter has also an action for showing the preview of the campaign and an action for launching the campaign.

3.3.7 **Quality Attributes**

One of the quality attributes we would like to focus on is the system performance. We considered design choices for improving performance such as caching of generated HTML (see Section 4.3), minification and merging of JavaScript and CSS (see 4.4), reducing the overall number of database queries (see 3.3.3) and support for the CDN services such as CloudFlare [25].

4 Implementation

In this section we describe the implementation details of the DesignTeeLine system. Because of the scope of the implemented application, we are not going to present all the classes and their relationships; we do not want to provide an exhaustive and complex description. Rather, we describe the technically interesting or challenging parts of the application and the description of the rest can be found in the developer documentation generated from the source codes.

We also present some detailed technical aspects which we would like to point out as a demonstration of the application complexity. Those aspects are for example: support for multiple language mutations, multiple currencies or resource minification which increases the application performance.

4.1 Application Structure

First, we will describe the overall structure of the implemented application to introduce the reader where to find certain functionality. Only the most important parts of the application are described here.

- **app/AdminModule/** is a directory which contains all the presenters and templates used in the application administration.
- **app/components/** contains implementation of all the components used throughout the application. What is considered as a component is described in Section 3.3.4.
- **app/config/** contains configuration files for the application.
- **app/FrontModule/** is a directory containing all the presenters and templates used on the application frontend.
- **app/model/** contains implementation of all the repositories and model entries, see Sections 3.2 and 3.3.3.
- **app/presenters/** contains only the implementation of the *BasePresenter* class from which all the presenters used in the application inherit.

- **app/services/** contains all the services. What is a service is described in Section 3.3.5. Some of the services are described in the more detail further in this chapter.
- **app/tasks/** contains the implementation of cron tasks, see Section 4.2.
- **app/templates/** contains general purpose templates which are not related to the concrete action of some presenter. Those are for example templates of e-mail messages or templates of components.
- **files/js/** contains all the JavaScripts used in the application.
- **files/js-editor/** contains the implementation of the front-end part of the editor.
- **files/style/** contains CSS a LESS visual styles of the application.
- **install/** is an installation directory with a script used to install a new instance of the application.
- **libs/** contains all the external libraries and the implementation of the Editor class which is used for generation of the images and handling the file upload from the editor.
- **tests/** contains selenium and unit tests for the application.
- **www/** is the directory that should be reachable from the outside world. The root of the website should point to that directory and no other directory of the application should be reachable from the web browser.
- **www/images/campaign/** contains all the generated images for all the campaigns. The images in this directory are divided to the subdirectories where each of them can contain images for 2 000 campaigns at maximum. This feature is implemented to speed up the search for the image in the file system on the server.
- **www/static/** contains static files which can be directly loaded from the browser. All those static resources are located in the separate directory because then they can be viewed together at some subdomain and some CDN such as CloudFlare can be used to cache the static content and save the server resources.

- **www/static/css/** contains compiled version of CSS files located in the files/style/ directory. The *CssService* class is used for that compilation and the whole process is described in Section 4.4.
- **www/static/js/** contains the compiled version of JavaScript files.

4.2 Cron Tasks

In order to automate the system processes described in Section 3.1 as much as possible, we implemented a set of cron tasks that are automatically executed by the cron running on the server. Executing of all cron tasks on a server is easy; it is enough to periodically call the script cron/cron.php every five minutes.

The class responsible for storing cron tasks, their scheduling and launching is called *CronService*. Firstly, we implemented cron tasks as instances of the *Task* which accepted a method that should be performed as an argument in its constructor. Second important parameter of each task is the periodicity of its execution. The last execution time of each cron task is stored in the cache, thus the tasks can be scheduled according to their periods.

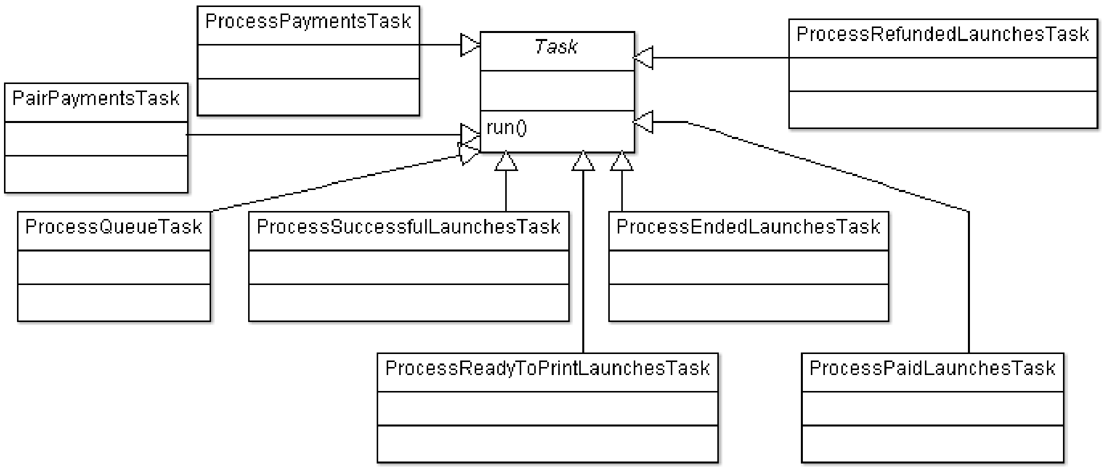


Figure 4.1: Class hierarchy of cron tasks.

But there was no clear logical place where to implement the methods that really represented the cron tasks. They were usually implemented at some repository or service but that was not logically correct. For example, some cron tasks send e-mail messages and it should not be a responsibility of any repository to send an e-mail message.

Therefore, we decided to re-implement the cron tasks in a way that each task is a separate class which inherits from the base *Task* class. Each such a class has to implement abstract method *run* which is called when the task is performed. The hierarchy of cron tasks is shown in Figure 4.1.

The instances of cron tasks are defined in the *BasePresenter::registerCronTasks* method. After analysing the processes that needed to be implemented in the application, we implemented those cron tasks:

- **ProcessQueue** – A task responsible for sending batch of e-mail messages stored in a message queue which is implemented as a database table called "message". There are more reasons why to send messages in batches and not all at once. First of them is that sending an e-mail consumes server resources, e.g. in a hypothetical situation when 10 000 customers support certain campaign, sending 10 000 e-mail messages at once is not a good idea. Another reason is that by sending too many messages at once the probability of marking those messages as spam at the receiver site increases.
- **ProcessSuccessfulLaunches** – When enough items of a campaign launch are ordered, the launch becomes successful. The purpose of this task is to automatically identify such launches, switch them to the "successful" state and inform all the customers about that situation.
- **ProcessEndedLaunches** – This task takes all ended campaign launches that were not processed yet and switches them to the "successfully ended" or "unsuccessful" state according to the Figure 3.2. Then an e-mail message is sent to the campaign initiator and e-mail messages for all the campaign buyers are added to the message queue. A message text is generated depending on the campaign launch successfulness. When the campaign launch is unsuccessful, all the related direct transfer payments that were captured are switched to the "to be refunded" state.
- **ProcessReadyToPrintLaunches** – When the campaign launch is ended for two working days, all the direct transfer payments should be captured. This task switches it to the "ready to print" state where it is up to the system administrator to send the order to the printing house. This task also creates

pay-outs for all the authors of successfully ended launches that do not have pay-out yet.

- **ProcessRefundedLaunches** – When the campaign launch is in the “unsuccessful” state and all its direct transfer payments are in the “refunded” state, this task switches the launch to the “redunded” state.
- **ProcessPaidLaunches** – Switches all the sent launches whose author has already been given the pay-out to the “paid” state.
- **ProcessPayments** – When the campaign launch becomes successful all the payments made by credit card or by PayPal are processed, i.e. the money from all relevant customers are captured via the payment authorisation tokens which were received when the customers made their payments. For each payment method a special service is responsible for communication with the payment gateway – the *GpWebPayService* for communication with GP webpay and the *PayPalService* for communication with PayPal. Occasional errors are logged for further analysis.
- **PairPayments** – This task is related to the direct transfer payments. It connects to a mailbox specified in the application configuration file, downloads the notification e-mails informing about incoming money and tries to pair those transactions with payments in the application database. The pairing is based on their amount and variable symbol. This task is an implementation of the process described in Section 3.1.3.

4.3 Caching

Caching is a good way to reduce the overall resources that application demands and increase its performance, i.e. speed the application up. We implemented two levels of caching mechanisms – caching on the level of templates and caching of generated HTML.

The first level of caching is provided by the Nette framework and we took advantage of it in the presenter actions that are supposed to be most visited. All that has to be done, in order to enable caching, is to wrap a block of code in the template with {cache}…{/cache} macro. With this mechanism, we managed to reduce the total number of SQL queries performed at the homepage and campaign detail actions to zero when the content is loaded from the cache. This is because SQL queries are performed inside of those cached blocks.

We also implemented the class called *HtmlCacheService* which serves as the second level of cache. The generated HTML is saved to the cache when the Nette event “onShutdown” is executed.

Loading of the content from the cache is performed just after the Nette framework is initialised in the *bootstrap.php* file. Therefore, when the content is loaded from the HTML cache, it is served in no time because no connection to the database is needed, no routing needs to happen and no presenter action is performed.

But because the whole generated HTML is stored in the cache, it needs to be used very carefully. For example, when some user is logged in, we cannot load the content from the cache because in the top right corner of each page the username of logged in user has to be shown.

4.4 Minification

Each request on the server from the web browser consumes server resources and it also takes extra time to connect to the server and download the resource. Therefore, it is a good practice to decrease an overall number of external resources needed for each page.

When generating a page, i.e. performing an action of a presenter, we define all the JavaScript and CSS files that the page requires in order to be fully functional. The *JsService* class is responsible for handling the references to the JavaScript files and the *CssService* class responsible for handling the references to the CSS files. Both of those services inherit from the base *AttachmentService* class which implements the common mechanism for storing the compiled files in the output directory.

All the resources passed to those services are first minified then merged into one file and finally saved to the *www/static* directory. This directory is directly accessible from the web browser; therefore, the resource can be later loaded as it is – no PHP scripts are involved.

As a result of that, only one request for JavaScript or CSS file is needed per page. Moreover a minimum amount of data is transferred via the internet because all the resources were first minified. The LessPHP [26] compiler is responsible for the minification of CSS/LESS files. Google Closure Compiler [27] was chosen for the minification of JavaScript files.

4.5 Image Generation

Generation of authentic campaign images is an important part of the application because a potential customer needs to see what he is going to buy. The *ImageRepository* class is responsible for locating the campaign images. When the image is not generated yet, an URL of the *CampaignPresenter::actionImage* action is returned as its URL.

This action is responsible for collecting the parameters for the image generation, delegating the generation to the *Editor* class and redirecting to the image real URL. The *Editor::generateCampaignImages* method is capable of image generation in different resolutions and formats. It merges all the image layers described in Section 2.6.3 into one SVG image and then it uses the *Imagick* extension of PHP to convert the image to the final PNG format. When the images for the printing house are generated, they are saved directly in the SVG format.

We generate those images:

- Image of the campaign for homepage on the default base and in the default colour from the front.
- Image of the campaign pure design for the homepage in the default colour.
- Mini image of the campaign used in the summaries in the user profile.
- Image of the campaign used in the search results on the default base and in the default colour from the front.
- Image of the campaign pure design used in the search results in the default colour.
- Image of the campaign design used for sharing of the campaign on Facebook.

- Images for the campaign detail in all the available colours and on all the available bases (i.e. types of sold items) from front and from the back.
- Images for the campaign pure design used in the campaign detail from the front and from back in all the available colours.

As we can see, it is quite a lot of images that need to be generated and this number grows with the number of supported bases and number of colour in which the campaign design is available. Therefore, quite a lot of server disk space is required for each campaign.

It is clear that majority of the disk space is occupied by the images generated for the campaign detail. Therefore, reducing the number of those images dramatically reduces the disk space requirements of each campaign. We solved that issue by taking advantage of the three layer nature of generated images.

For the campaign detail we generate only the image of the campaign design from the front and from the back, we use the same procedure for image generation, but the image is generated with a transparent background and with no base image. The final assembling of the images, which creates the effect of the T-shirt with its design, is finally solved in the campaign detail where the layers are rendered one over each other.

4.6 Language Support

Our system supports translation to multiple languages. At default only a Czech and an English version of the website are available but it is easy to translate the website to any other language. All that needs to be done is to introduce a new language in the *LanguageService* class and then make all the translations.

For translating the application we use the *GetTextTranslator* [26] extension of Nette which has also an easy to use frontend to make the translations. But it is also possible to use some other tool for editing .po and .mo files with translations.

Every language has a currency attached which is an instance of the *Currency* model class. This class is responsible for recounting prices from one currency to another based on the pre-defined conversion rates which are defined relative to the base EUR currency. All the currencies are defined in the *CurrencyRepository* class where it is easily possible to change their conversion rates.

It might be also possible to extend the application and implement a service which downloads the conversion rates every day from the official database of Czech National Bank and stores them in the cache. The *CurrencyRepository* class would then read the conversion rates from this cache.

4.7 Communication with Payment Services

From the analysis in Section 2.4.3 we can see that we can implement the same workflow at the GP webpay and at PayPal; therefore, we can abstract from technical details and introduce an interface as an abstraction of the common workflow for communication with external payment gateways. Communication with payment services is implemented as application services called *GpWebPayService*, *PayPalService* and *BankTransferService*; those classes implement the *IPaymentService* interface which is the mentioned abstraction.

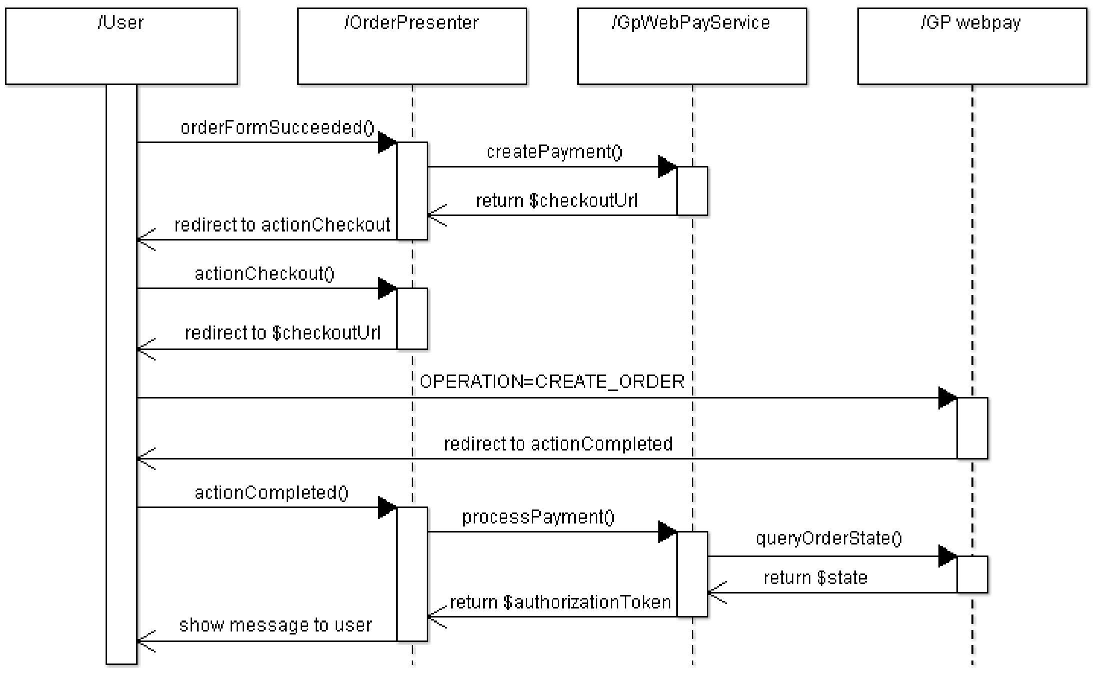


Figure 4.2: Communication with GP webpay gateway when creating a new payment.

The part of the common workflow for communication with payment gateways is illustrated in Figure 4.2. We picked up the GP webpay payments to have the illustration more concrete, but a similar figure applies also for a model of the PayPal payments. The entire workflow is as follows:

1. Create a payment – when the order form is successfully processed, the *OrderPresenter* is responsible for calling the appropriate payment service and calling its *createPayment* method as shown in Figure 4.2. The payment service initializes a new payment on the side of the payment gateway. Then the created payment is associated with the payment created in the database.

2. Redirect to the *actionCheckout* action of the *OrderPresenter* from where it is redirected to the payment gateway where the customer fills in the credentials to authorise the payment.

   We might redirect to the payment gateway in the previous action but we use this intermediate *actionCheckout* because we support the case when the payment gateway might be embedded directly to the web page, i.e. no redirect is needed. Then the embedded payment form would be shown at this action instead of redirection.

   In the case of GP webpay, payments are created implicitly by redirecting the customer to the URL where all the necessary information like amount and currency is passed as parameters. When communicating with PayPal API we have to explicitly create the payment by calling the "create" method of the class *PayPal\Api\Payment*.

3. When the authorisation is completed, the payment gateway redirects the customer back to the application to the *actionCompleted* of the *OrderPresenter*.

4. This action calls the *processPayment* method of the appropriate payment service implementation. This method is responsible for analysing of the response from the payment gateway and potentially it processes further communication with the payment gateway.

   For example, in the case of the GP webpay, it is recommended to verify whether the payment is really in the correct state on the side of the gateway as shown in Figure 4.2. Because the response is all passed in the URL parameters, it is necessary to verify its validity with passed signatures and secret certificates.

In the case of the GP webpay, an authorisation token is the id of an order at the GP webpay. We need to store the last id of the order in the database to maintain synchronization with the id counter at the GP webpay. The *DialRepository* class is responsible for that.

In the case of PayPal, it is necessary to call the *execute* method on the instance of the class *PayPal\Api\Payment* and obtain an authorisation token.

5. When a campaign becomes successful, deposit the payment. For the deposition, we use the authorisation token obtained when the payment was created. Cron task "ProcessPayments" is responsible for calling *authorize*, *refund* and *processBatch* methods of the appropriate payment services. In the implementation of the *GpWebPayService* and *PayPalService* the *processBatch* method captures all the payments that were previously authorised.
6. When a campaign ends unsuccessfully, do nothing. In our implementation, we ignore all the payments that should be refunded because it is enough just not to capture them. In that case, no money is transferred and no fees need to be paid. But the workflow is ready for the case when some action might be needed in the case of unsuccessfully ended campaigns and the *refund* method of the appropriate payment services is called.

In the case of direct transfer payments, we used the same interface but we had to do a few exceptions from the common workflow. In the step two we use an URL of *actionCompleted* as an URL of the payment gateway; hence the customer is immediately redirected to that action when he is informed about payment details. The payment in the database remains in the "authorized" state in this case.

In the *BankTransferService* class the methods *processPayment* and *processBatch* do nothing; they are just implemented in a way to fit the common workflow. The final pairing of the payments in the database with the real payments in the bank is accomplished by the cron task "PairPaymentsByVariableSymbol". This way we managed to fit direct bank transfers to the common workflow.

As we have said in Section 2.4.3, we have also tested to deposit the money after 21 days from the payment approval which is theoretically the longest possible time because 21 days is the maximal duration of any campaign. The deposit of the real money after that long time worked with no problem at both of the services, the GP webpay and PayPal.

4.8 Model Layer

When retrieving data from the database in some repository, we can rely on the *Nette\Database\Table\ActiveRow* class to provide us all the relevant data from the database. But for some database entries we need more than just read the values of the entry properties. We would also like to read some properties inherited from the data in the database, e.g. how many days are left till the launch end or the percentage of launch successfulness.

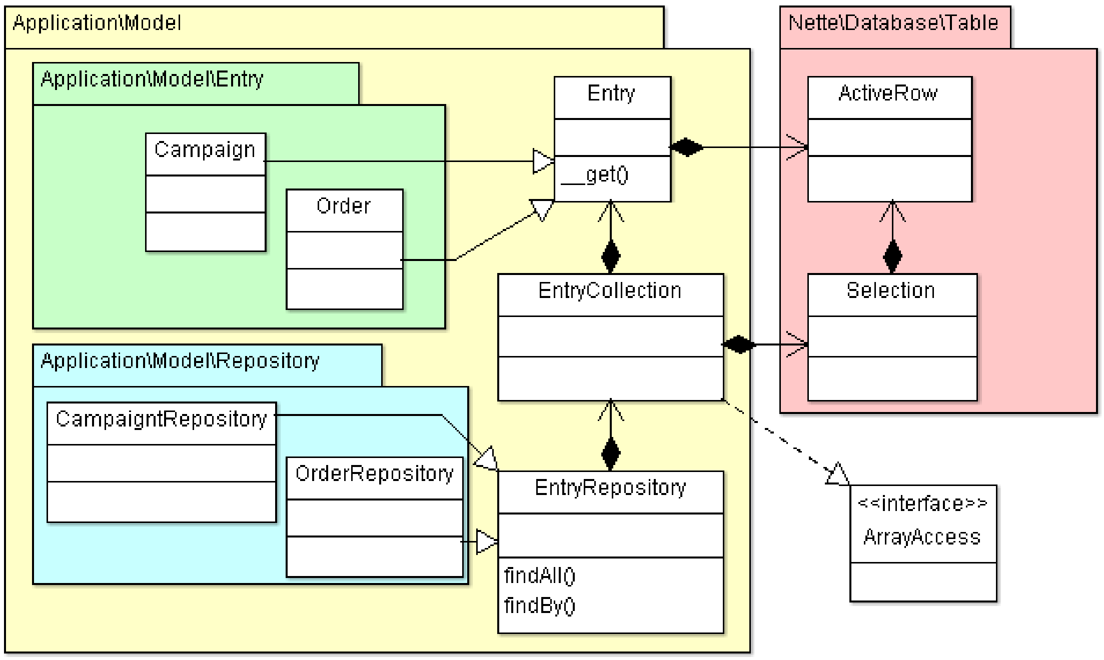


Figure 4.3: Relation of the EntryCollection class to the rest of the model layer.

It is against the encapsulation principle to put methods for computation of those values to some helper classes, more logical is to implement those in the model classes and this is exactly what we have done. Model classes are implemented as wrappers around the database or other entities and they enrich their behaviour.

One of the advantages of the Nette framework model layer is the on demand execution of database queries. Usually the query is just built in the repository and it is returned to the presenter as an instance of the *Nette\Database\Table\Selection* class. The query is executed afterwards when we start to read the result in the template. When this principle is combined with a caching mechanism in the template, we get an elegant way to save a query when the template content is retrieved from the cache because in that case the query is not executed at all.

But unfortunately in that moment we lost that advantage because when we return data from the repository, we need to convert all the model entries, i.e. instances of the *Nette\Database\Table\ActiveRow* class to the concrete model. Therefore, the query is immediately executed and it cannot be cached. We could solve that problem by moving the retrieval of the data to the template but it would not be a clean solution.

We solved that problem in a more elegant way. Probably the best solution we found was to implement a class that wrapped every instance of the *Nette\Database\Table\Selection* class created in the repository as shown in Figure 4.3. In the diagram we call that class *EntryCollection*. This class implements the *ArrayAccess* interface in order to enable looping over its instances in the templates. When retrieving an entry from the database, it gets wrapped with the concrete instance of the model.

In Figure 4.3 we can see a logical containment of instances of the *Nette\Database\Table\ActiveRow* class in the *Application\Model\Entry* class. The *EntryCollection* class logically contains some model entries – a name of a concrete model class could be passed as a parameter in the constructor. The *EntryCollection* class wraps the *Nette\Database\Table\Selection* class and instances of this class are returned from all the repositories that store their state in the database. In order to preserve the clarity, not all model entries and repositories are presented in the figure.

Another solution to this problem might be usage of some ORM framework [27], but we like the simplicity of the database layer in Nette and ORM framework seems as an unnecessary complication.

4.9 **Editor**

One of the key features of our application is the editor where a design of the imprint can be created. We tried to implement the editor as a separate entity that is more or less independent on the rest of the application. We achieved that by defining a communication interface between JavaScript part of the editor and its backend. This communication is based on the AJAX queries which exchange data in the JSON format. As a result of that, the JavaScript part of the editor is fully independent on the rest of the application and it might be re-used somewhere else without further complications.

On the server side the image generation and handling of the files uploaded from the editor is encapsulated in the *Editor* class. This class is also nearly independent on the rest of the application.

The core of the editor on the server side is the *EditorPresenter* class. This presenter is responsible for handling all the AJAX requests from the editor frontend and provision of reasonable responses in the pre-defined JSON format. The list of all AJAX requests that need to be handled can be found in the *EditorPresenter::actionAjax* method.

When a new campaign is launched in the editor, the draft is not switched to the "launched" state as shown in Figure 3.1, but a new campaign is created as a copy of the draft. This campaign is already created in the "launched" state. This is because we want to preserve the old draft for the user.

The editor is also easily extensible for new artworks, fonts and bases:

- When adding a new artwork, it is enough to copy it to the subdirectory in the *www/static/artworks* directory. The *ArtworkRepository* class is responsible for the artwork localization.
- When adding a new font, it should have its own subdirectory in the directory *www/static/fonts*. All the necessary files as in the case of other fonts should be present there. The *FontRepository* class is responsible for the font localization. In order to make the fonts work on the server side when the campaign images are generated by Imagick, the font has to be installed as a system font in the operating system.
- When adding a new base, its images should be copied to the *www/static/images/editor/base-dark* and *www/static/images/editor/base-light* directories. Also the base has to be registered in the *BaseRepository* class and its available colours have to be defined in the *ColorRepository* class.

4.10 **Colour Counting**

The editor automatically counts a base price of each variant of designed item based on the goal of the campaign, item type, item colour and total number of used colours at the item front and at its back. Therefore, it is crucial for the price estimation to count the number of used colours correctly.

The most difficult part of the colour counting is the estimation of colours used in the uploaded images. Counting the number of colours used in the rest of the design such as number of colours used in the texts or artworks is straight forward.

When a user uploads an image in the editor, it is saved on the server side and depending on the image mime-type, the colour counting algorithm is picked up.

- When a SVG image is uploaded, the algorithm first determines whether it contains embedded bitmap or not. When not, the algorithm for counting the colours in vector images is used. This algorithm is implemented in the *Editor::_countVectorColors* method. Otherwise the image is converted to the PNG format and algorithm implemented in the method *Editor::_countBitmapColors* is used.
- When an EPS image is uploaded, it is first converted to the SVG image and then the same procedure as described above is exploited.
- Bitmap images are first converted to PNG and then the algorithm from method *Editor::_countBitmapColors* is used.

Counting the colours in the vector SVG image is absolutely precise and quick but unfortunately it cannot be used at all occasions. Therefore, an algorithm for counting the number of colours in the bitmap images had to be developed. Because of the performance reasons the algorithm does not check every pixel in the uploaded image but it is designed to check at maximum 10 000 pixels.

We can look at each colour represented in the RGB as at the point in the three dimensional space with 256 points in each dimension, and therefore we can count an Euclidean distance between every two points. Our algorithm for colour counting is based on that idea – it divides the colours into clusters based on their similarity. The advantage of this approach is that it eliminates a great explosion of the colours when the image is not in 100% quality. We consider a distance between black and white colours as a maximal possible distance. Algorithm works as follows:

1. Get the relevant pixel from the image.
2. If the colour of the pixel is less than 5% different from some colour of the cluster, put the pixel into the same cluster.
3. Otherwise create a new cluster with the colour of that pixel as its representative.
4. If more colours remain to process, go to the first step.
5. Number of clusters is the final number of the colours returned to the caller.

5 Testing

In this chapter we discuss the functional and non-functional tests that we implemented above our application. First two sections are dedicated to the functional tests – first we talk about unit tests and in the second section we describe how we coped with system tests. In the third section we focus on the non-functional tests, namely on the performance tests. Throughout the chapter we also discuss the limitations of the web applications testing, especially in the area of communication with external services.

5.1 Unit Tests

Unit tests test the application from the developer's point of view. The goal of unit testing is to isolate each part of the program and show that the individual parts are correct [28]. This isolation is quite hard to preserve in context of web application because of the presence of database layer and communication with external services. We should provide such tests that do not interact with the real environment, but still test whether the tested methods behave in a specified way.

Most of the methods in our application communicate directly or indirectly with the database layer. There are two main approaches how to solve the isolation problem connected with the unit tests. The first of them is a semi-solution where we use a database designed especially just for the testing. Each test is then wrapped in a database transaction which is roll-backed in the end of the test. Thus the state of the database is not changed during the testing and the tests can run separately of each other thanks to the use of transactions.

The second approach is to mock the database layer and just check whether the tested methods call this layer in the expected way. We picked up this approach because it is much cleaner than the previous one. For mocking of the behaviour we chose the Mockery [29] framework. We chose the Tester [30] framework for writing the unit tests because it is easy to set up, it has simple, but powerful API and it supports parallelization when running the tests.

Our aim was not to have great code coverage with unit tests, but to show how to write unit tests in the context of real web application and how to test specific parts of the application written above the Nette framework. Writing tests to achieve a certain level of code coverage is time consuming and not very inventive work; therefore, we wrote only few tests to demonstrate the principles.

We wrote unit tests focused on testing of components, simple utility classes, tasks and services. All the tests are located in the *tests/unit* directory. There are a few services that are special and hard to test. Those are the services that facilitate communication with the GP webpay and PayPal.

Tests for utility classes are standard simple unit tests for classes without external dependencies. We also wrote tests for three services as a demonstration of principle how to test them. Each test extends the base class *Application\Test\TestCase* which implements methods for mocking various data types used throughout the application code. For example in tests for *Application\Services\CartService* class we had to mock a campaign variants added to the shopping cart, because there is a hidden communication with the database layer. Moreover when we construct a new variant instance it accepts an instance of the *Nette\Database\Table\ActiveRow* class in its constructor.

We had to solve the problem with injecting a model data to instances of model classes without breaking the encapsulation principle. For this purpose we added a new protected method *Application\Model\Entry::setEntry*. Thanks to its protected visibility level we did not break the encapsulation of model classes, but we have to call this method via reflection from unit tests.

Testing of components arises the question what to test in the relation with them. We picked up some simpler components and we decided to test whether they produce the right output or not. Because of the design of the Nette framework where each component has to be attached to some presenter when used, we had to implement the method *Application\Test\TestCase::attachComponentToPresenter* which mocks an application presenter and attaches the component to that presenter.

We have also decided to test one of the payment services – class *GpWebPayService*. Tests for this service are based on mocking up the external dependency on the GP webpay and mocking up the SOAP communication with the GP webpay. This approach works well for isolation of tests from the external environment, but it also comes with one disadvantage: when the API of the GP webpay or PayPal unexpectedly changes, the tests cannot detect that change because of the usage of mocks. This is one reason why we also wrote functional tests that are focused on the payment from the user's point of view.

In order to be able to mock dependencies of the tested class, we had to add new interfaces whose instances are accepted in the constructor of the tested class. These interfaces are called *ISignatureFactory* and *ISoapFactory*. For example the following code from the test for *processBatch* method shows how to mock the service which secures the communication with the GP webpay service.

```
$signatureFactoryMock = $this->mockInterface('ISignatureFactory')
    ->shouldReceive('create')
    ->andReturnUsing(function () {
        $signatureMock = $this->mockInterface('CSignature');
        $signatureMock->shouldReceive('sign')
            ->andReturn('valid');
        $signatureMock->shouldReceive('verify')
            ->andReturnUsing(function ($text, $signature) {
                unset($text);
                return $signature == 'valid';
            });

        return $signatureMock;
    })->getMock();
```

We can see that when we use the signature service in this test, we do not care whether the service encrypts data in the right way or not. This is the reason why we mock it – we just need to know that when it returns true for the call of verify method, the payment is processed in the right way. When it returns false, payment is not processed, because it might be a fake request.

One of the biggest challenges is how to test the processes described in Section 3.1. We decided to write unit tests for some of the cron tasks involved in the processes because it is interesting to invent tests for this part of the application. The assumption is when all the transitions illustrated in the process diagrams in Section 3.1 are taken at right circumstances, then the processes as a whole can be considered as it passed the test. Unfortunately, there are actions that require a manual intervention from the system administrator that make the process untestable as a whole.

Tests of cron tasks are interesting, because they do not execute any assertions. The only thing they test is whether tasks call the underlying services and repositories in the right way. This is done by configuring Mockery expectations – when these expectations are not met, i.e. when the task is broken, an exception is thrown by the Mockery framework in the *tearDown* method of the *Application\Test\TestCase* class and the test fails. The following code sample shows expected interaction with the *PaymentServiceContainer* service in the test of *ProcessPaymentsTask* class. We can see that method *getPaymentService* is expected to be called three times and method *getAllPaymentServices* is expected to be called exactly once with no arguments.

```
$paymentContaier = $this->mockService('PaymentServiceContainer');
$paymentContaier->shouldReceive('getPaymentService')
    ->times(3)
    ->with($paymentMethod)
    ->andReturn($paymentServiceMock);
$paymentContaier->shouldReceive('getAllPaymentServices')
    ->once()
    ->withNoArgs()
    ->andReturn([ $paymentServiceMock ]);
```

Tests we have implemented are meant as examples of testing of various types of classes used in our application. These tests demonstrate the principles that could be used for testing of the rest of services, components and tasks. We did not implement any tests of the model layer, but these tests could be implemented in the similar way as tests we have already created by exploiting the mocking framework. We also suggest using Testbench [31] library for testing of presenters and their forms.

5.2 System Tests

System tests are important because they test the application from a different perspective than the unit tests – from the user's point of view. In the test, we provide the inputs for the application that the user could provide and we test whether its response is correct due to our expectation. We could also say that those tests ensure that the system is functioning as users are expecting it to.

We decided to exploit the Selenium [32] tool for system testing because it is the most widely used tool for this purpose in the domain of web applications. It has gone through many years of development, has solid and wide base of users, it is stable, tested and well-documented.

Selenium is a tool which automates web browsers. Provided Selenium IDE is capable of recording the activity in the Mozilla Firefox web browser, defining asserts that should hold and saving such a scenario as a test case. Test cases can be grouped to tests suites and replayed later. When some assertion in some test case does not hold, it is considered as a violation of application functionality and Selenium provides us a report about all such cases.

We have created over 100 Selenium tests which are all located in the *tests/selenium* directory of the DesignTeeLine application. Those tests are focused on the main functionality of the application frontend. Each test covers a slice of the system functionality which can be simple, but it can be also implemented as a complex interaction of objects and their methods. The big advantage of Selenium is that it can also easily test functionality based on JavaScript and pages that use AJAX. Thus it was also possible to cover the functionality of the editor with test cases.

All the tests are grouped to the test suite located in *test/selenium/TestSuite.html* file. Before launching the test suite from Selenium IDE, one should edit the first test case called *Configure* and set there correct values for connecting to the database. Make sure that the used database user has a permission to create and drop a database. We also recommend setting the default timeout option in the Selenium IDE to 120 seconds. This is because of the installation test which might take more time to successfully finish.

Because selenium tests naturally change the state of application in the database layer, we designed our test suite in such a way that the second test case called *Initialize* installs a new instance of our application via its installation script. It also creates a backup of the original configuration file in *app/config*. Then all the actual tests for our application follow. The last test case called *Terminate* drops the test database and restores the original configuration file. Therefore, next time the application connects to the original database.

The test cases we implemented are dependent on each other which means that launching some of the test cases independently on the others or launching the test cases in the wrong order might result in a failure of some assertion. The right order of tests is in the *TestSuite.html* file, tests should be run in this order. Moreover, a failure of one test case might also result in a failure of several other test cases which are dependent on it. For example, many test cases assume that the administrator user is logged in. Thus when a test case for login fails, all those test cases fail as well.

We wrote a few tests that are focused on the right interaction with payment gateways. Namely, those are tests called *Order/BuyCard* and *Order/BuyPayPal*. Each newly installed instance of the DesignTeeLine application is configured in a way that it connects to the test endpoints of the payment gateways. Hence those tests do not cope with the real money, but they still test the correctness of the interaction because test endpoints guarantee the same behaviour as the real environment. But those tests are prone to fail because within those tests we also test UI of payment gateways. When this external UI is changed, the test might fail.

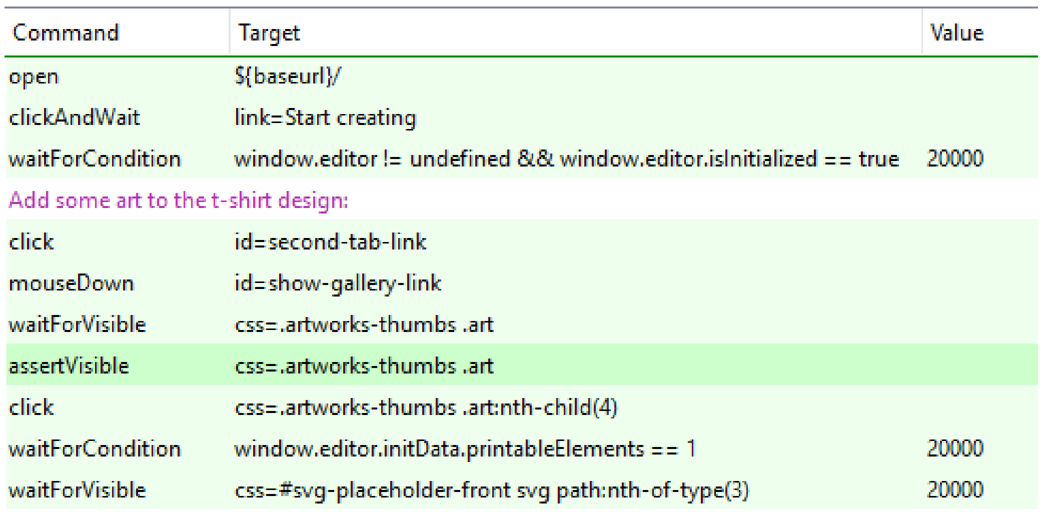
We managed to cover most of the functionality of the application frontend with tests. Still some functionality is hard or nearly impossible to test. For example when a new user registers to the system, he receives a confirmation e-mail where he has to click on the confirmation link. Then his account becomes active. Testing such functionality is very difficult because it involves clicking on the confirmation link in the e-mail.

If we would like to strictly follow the procedure, we would have to login to the tested mail box, wait for a confirmation e-mail, open it, find there a confirmation link and click on it. Testing this procedure in Selenium IDE is nearly impossible because we would have to locate the right e-mail in the mailbox. It would be possible to write such a test for Selenium WebDriver where we would connect to the mail box via IMAP, find the right e-mail, parse the link from the e-mail text and then navigate to the right URL.

But we decided not to employ Selenium WebDriver just because of a few test cases and we invented a different approach. We implemented a simple script located at *tests/selenium/support* which provides generated hashes for confirmation links. After a successful submit of the registration form, the test just asks this script for the confirmation hash of the user which has been registered. Then it navigates to the confirmation URL with this hash and the effect of this mechanism is the same as clicking on the confirmation link in the sent e-mail.

Unfortunately, we did not find a similar way to test the integration with Facebook and Google for the user registration and login. We found out that those processes would require having fake Facebook and Google accounts just because of those tests which is forbidden by terms and conditions of those services.

In the end of this section we are going to describe one of the test cases for the editor. We selected the test case called *Editor/ArtProperties* on which we can demonstrate some principles used during the testing. As shown in Figure 5.1 we start the test at the homepage and follow the link to the editor. Then we use “waitForConfition” command to wait until the editor is fully loaded and ready.

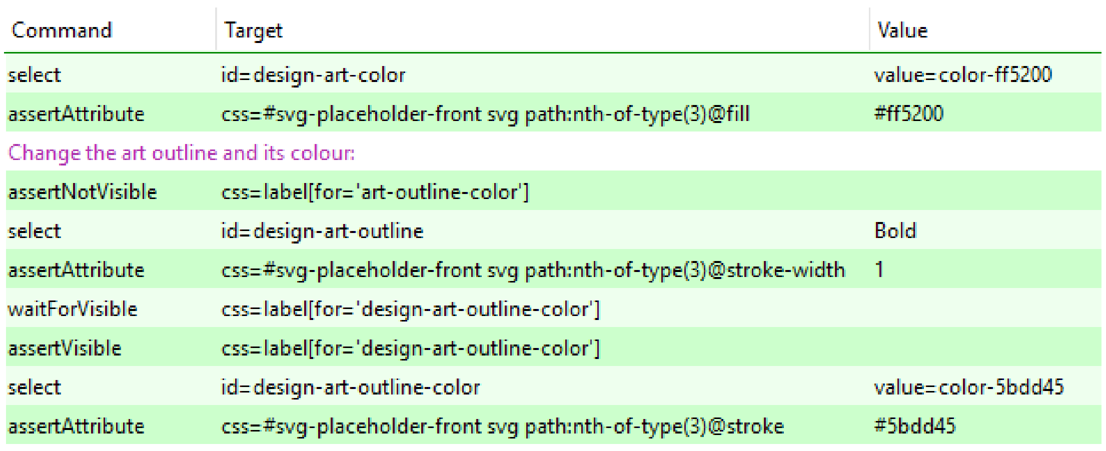


Command	Target	Value
open	${baseurl}/	
clickAndWait	link=Start creating	
waitForCondition	window.editor != undefined && window.editor.isInitialized == true	20000
Add some art to the t-shirt design:		
click	id=second-tab-link	
mouseDown	id=show-gallery-link	
waitForVisible	css=.artworks-thumbs .art	
assertVisible	css=.artworks-thumbs .art	
click	css=.artworks-thumbs .art:nth-child(4)	
waitForCondition	window.editor.initData.printableElements == 1	20000
waitForVisible	css=#svg-placeholder-front svg path:nth-of-type(3)	20000

Figure 5.1: First part of the selenium test case.

The fifth command opens the gallery with artworks and we use the “waitForVisible” command to wait until the gallery is loaded via AJAX. In the eighth command we select an art from the gallery we want to use in our design. Then we wait until the art appears in the printable area of the T-shirt. The “editor.initData.printableElements” variable contains a number of graphical elements used in the T-shirt design. The selector used in the tenth command is a bit complicated because we select the element inside edited SVG canvas. The “path:nth-of-type(3)” selects the third path on the canvas which represents the art we have just added.

The test case continues with the commands in Figure 5.2. First, we select a colour of the art and with the second command “assertAttribute” we check whether the “fill” attribute of our path was really set right. Then in the similar way we select an outline for the art and its colour. We also check whether the select box for outline colour becomes visible when we select some outline for the art. This way we verified whether it is possible to add a new art to the T-shirt design and change its properties.



Command	Target	Value
select	id=design-art-color	value=color-ff5200
assertAttribute	css=#svg-placeholder-front svg path:nth-of-type(3)@fill	#ff5200
Change the art outline and its colour:		
assertNotVisible	css=label[for='art-outline-color']	
select	id=design-art-outline	Bold
assertAttribute	css=#svg-placeholder-front svg path:nth-of-type(3)@stroke-width	1
waitForVisible	css=label[for='design-art-outline-color']	
assertVisible	css=label[for='design-art-outline-color']	
select	id=design-art-outline-color	value=color-5bdd45
assertAttribute	css=#svg-placeholder-front svg path:nth-of-type(3)@stroke	#5bdd45

Figure 5.2: Second part of the selenium test case.

5.3 Performance Tests

From the user's point of view performance is one of the most important extra-functional properties of the website. When a website is not loaded in a functional state after a few seconds from the user's request for load, the user loses attention and leaves the website or starts to feel frustrated. Moreover, website performance is easily measurable property and several tools for its measurement have been developed.

Those are the reasons why we focused on performance tests and optimizing the application in this dimension. However, it is interesting to see how much time it takes to load a page in the browser in order to determine the parts that should be optimized. We considered as more interesting to artificially simulate simultaneous users who use the application. This kind of testing is called load testing and its purpose is to see how the application performs under certain loads and thus find its limitations.

Those tests are of course hardware dependent. We launched the tests when application was running on the server with 8 GB RAM and Xeon E5504 2.0 GHz CPU. We picked up two online tools which are able to perform load tests.

First of them is LoadImpact [33] which provides a simple user interface to record an activity in the Chrome web browser. It records all the actions taken and then generates a load script which can be manually modified afterwards. As the last step we choose the duration of the test, the number of simulated simultaneous users which all perform the behaviour defined by generated script and then we can launch the test.

We wanted to take an advantage of the free version of the tool; therefore, we set up the values to the maximum, i.e. 250 simulated virtual users which are active during five minutes. All the simulated users first load the homepage of the application and then the detail of one campaign and they repeat those two actions in an infinite loop. We chose those two actions of presenters because we expect them as the most visited ones.

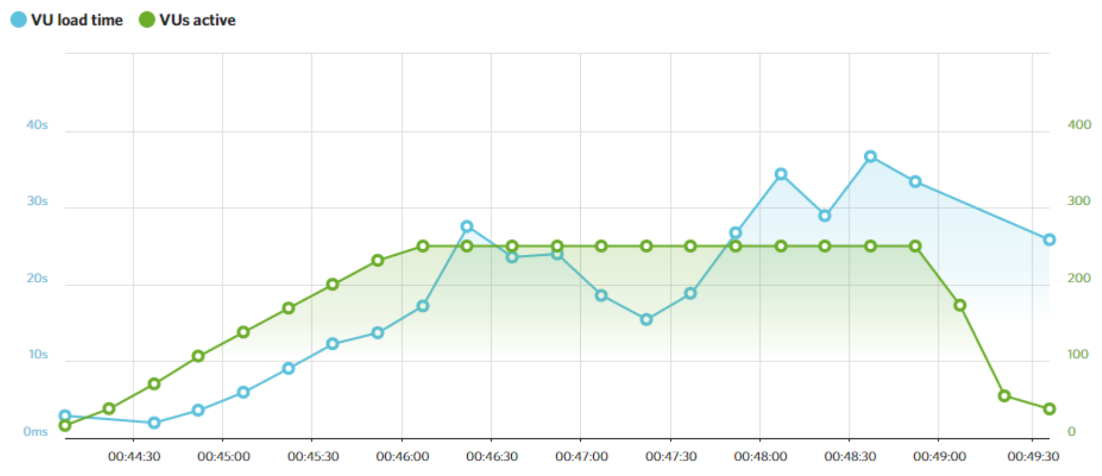


Figure 5.3: Results of the load test on LoadImpact with caching mechanism off.

As we can see, the results are not very impressive. At some points in time the average load time is above 30 seconds, which is way behind reasonable time. But we should also mention that 2,47 GB of data was transferred over the network during the test and there were 43 732 requests on the server.

We also wanted to check the impact of usage of the caching mechanism described in Section 4.3. Firstly, we performed a test without the caching mechanism; the results are shown in Figure 5.3. The green line shows a number of virtual users active in a certain point in time and the blue line shows an average load time of pages loaded by all the active virtual user in that time.

In Figure 5.4 we can see the results with the caching mechanism on and from the graph we can clearly see the impact of the caching mechanism on the application performance. Moreover, when the server is not blocked with many waiting connections, it is able to serve more data and handle more requests in the same time. In this case we were able to transfer 6,76 GB of data and handle 118 512 requests.

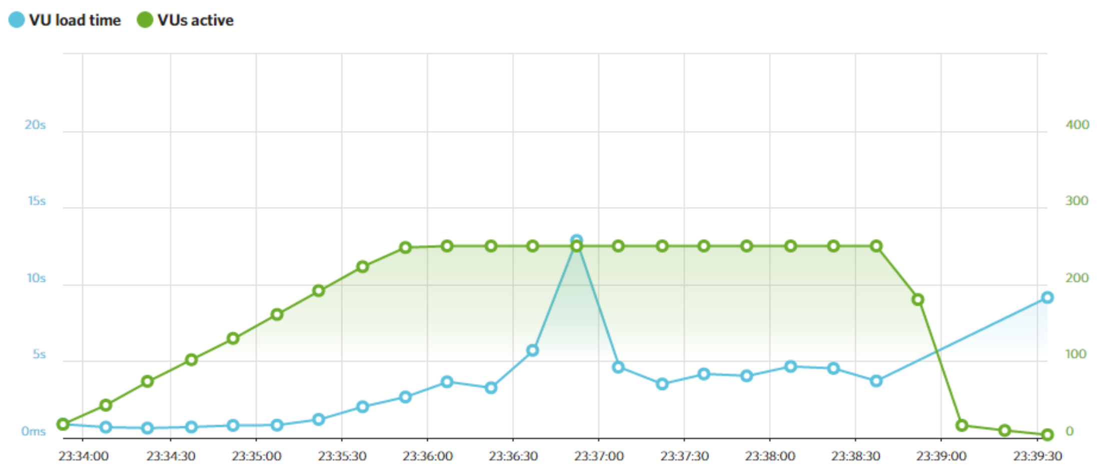


Figure 5.4: Results of the load test on LoadImpact with caching mechanism on.

The second tool we used for testing of the application load limitations is called loader.io [34]. This tool works in a different way than the previous one; therefore, it was interesting to test the application also in another way. In contrast with LoadImpact this tool does not record any script for virtual users who then load the page in the same way as normal users do in the web browser.

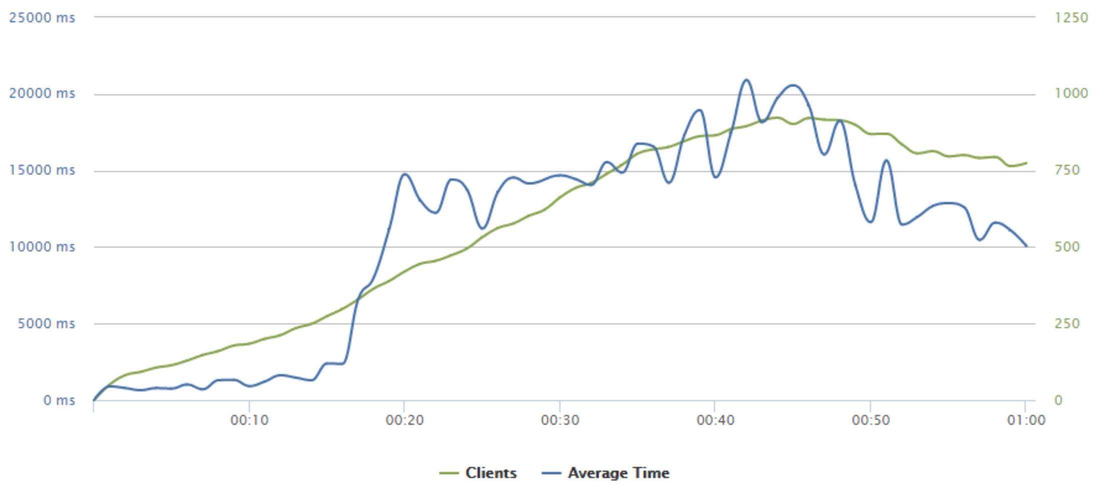


Figure 5.5: Results of the load test on loader.io with caching mechanism off.

Loader.io loads just the HTML content of the page but does not follow the links for attached files. This makes it a tool that tests how many requests can the server’s PHP scripts handle but it also makes this scenario less realistic. One advantage is that it can employ up to 10 000 workers per minute which connect to the server. We tested our application with 2 500 workers.

In Figure 5.5 we can see the results without the caching mechanism. There is a clearly visible congestion of the server when still 750 clients are active in the end of the test. In Figure 5.6 the results with the caching mechanism on are presented where we can again see a great impact of caching on the application performance. We also found a threshold where the response times of the server are too big even with the caching mechanism on. In our case this threshold is around 7 500 clients per minute.

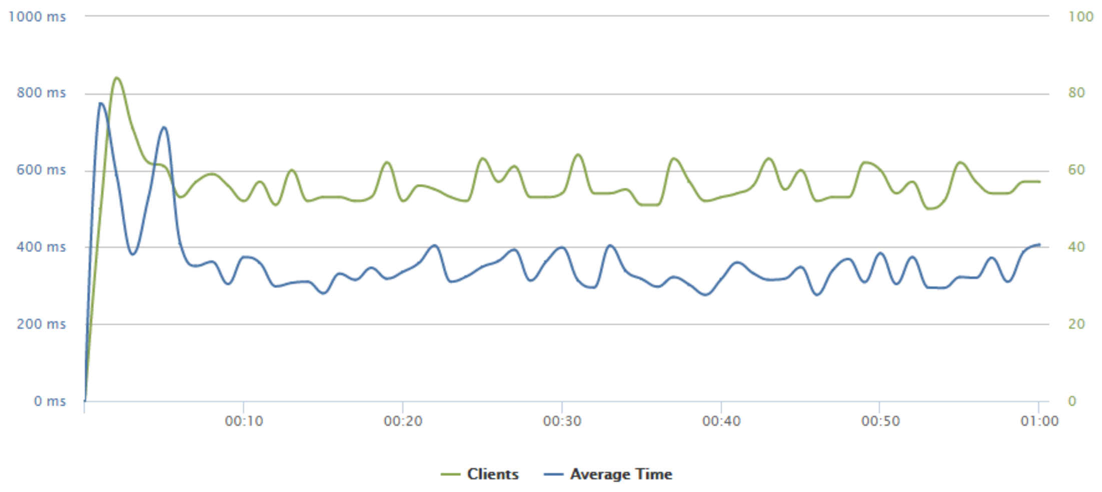


Figure 5.6: Results of the load test on loader.io with caching mechanism on.

6 Conclusion

In this thesis we describe the design and implementation of a web crowdfunding system based on the sale of items with a custom imprint. We demonstrate the system functionality on the sale of T-shirts, sweatshirts and bags. The whole process starts when a campaign initiator designs a T-shirt imprint in the editor that we implemented. From the architecture point of view this editor is highly independent on the rest of the system and could be reused somewhere else without further complications.

We managed to design processes for automating the campaign lifecycle as much as possible and we also discovered limitations of this automation. Part of the work was also seamless integration with external services like on-line payments, a printing house and social networks. In Chapter 5 we described how and why we test our application and we discussed the limitations of those approaches, especially when testing the integration with third party services.

All goals set in chapters 1 and 2 have been achieved. We were also able to cope with all the problems we discovered during all of the development phases. The thesis fulfilled its declared purpose.

A fully functional application was publicly accessible at the URL http://www.designteeline.com/ for a certain amount of time. We have registered few successful campaigns; therefore, we had an opportunity to test the whole system on real users with real data and everything worked as expected.

The high quality of the application, its source code, design principles and documentation are also demonstrated by a contract we managed to make with a real world company called RealGeek ltd. They successfully run our application at http://www.merchmaster.cz/. For this company we were obliged to implement a new credit card payment service called ThePay [35]. They also demanded a new method for pairing payments made with direct bank transfers, because they have a bank which has an easy-to-use API for that task. Thanks to our extendable design, it was not a difficult task to implement and integrate new classes for these payment methods.

6.1 Open Problems and Future Work

The development of the application is not terminated and we are ready to extend it in the future. Firstly, a big task to be done is to cover the application code with the unit tests. Regarding the size of the code this is going to be a really big task.

Second thing on our wish list is to rewrite the system administration, especially its frontend into a single page application, maybe with the use of AngularJS [36] framework. We have created a simple dashboard for the system administrator, so he does not have to think about all the actions that should be taken in the moment and search for the relevant entries in the system, i.e. he does not have to search for successfully ended campaigns that should be sent to the printing house, for orders that should be refunded etc.

Our aim is to create a cleaner dashboard where all the actions that should be taken by the administrator in the moment would be listed and arranged in independent widgets. This dashboard should be fully based on AJAX technology.

Third extension is related to the editor. We would like to implement new features for manipulation with elements on the canvas such as cut, copy, paste, undo and redo. These features should make the editor even more usable for campaign initiators.

Bibliography

[1] Crowdfunding. [online]. [cit. 2016-06-01]. URL: http://en.wikipedia.org/wiki/Crowdfunding

[2] KickStarter. [online]. [cit. 2016-06-01]. URL: http://www.kickstarter.com/

[3] TeeSpring. [online]. [cit. 2016-06-01]. URL: http://www.teespring.com/

[4] T-Shock. [online]. [cit. 2016-06-01]. URL: https://www.t-shock.eu/

[5] Danielson Printing House. [online]. [cit. 2016-06-01]. URL: http://www.danielson.cz/

[6] Czech Post On-line Application. [online]. [cit. 2016-06-01]. URL: https://klientskazona.cpost.cz/

[7] Manuals for GP webpay on-line payments. [online]. [cit. 2016-06-01]. URL: http://gpwebpay.cz/Download

[8] Self-signed Certificates. [online]. [cit. 2016-06-01]. URL: http://en.wikipedia.org/wiki/Self-signed_certificate

[9] PayPal API On-line Documentation. [online]. [cit. 2016-06-01]. URL: https://developer.paypal.com/docs/api/

[10] PayPal PHP SDK. [online]. [cit. 2016-06-01]. URL: https://github.com/paypal/PayPal-PHP-SDK/

[11] Facebook Social Plugins. [online]. [cit. 2016-06-01]. URL: https://developers.facebook.com/docs/plugins/

[12] Nette Framework. [online]. [cit. 2016-06-01]. URL: http://nette.org/

[13] JQuery. [online]. [cit. 2016-06-01]. URL: https://jquery.com/

[14] Raphaël. [online]. [cit. 2016-06-01]. URL: http://raphaeljs.com/

[15] SVG.js. [online]. [cit. 2016-06-01]. URL: http://svgjs.com/

[16] Snap.svg. [online]. [cit. 2016-06-01]. URL: http://snapsvg.io/

[17] PHP SVG. [online]. [cit. 2016-06-01]. URL: https://code.google.com/p/phpsvg/

[18] ImageMagick On-line Manual. [online]. [cit. 2016-06-01]. URL: http://php.net/manual/en/book.imagick.php

[19] PS to PDF Converter. [online]. [cit. 2016-06-01]. URL: http://www.ps2pdf.com/

[20] Inkscape. [online]. [cit. 2016-06-01]. URL: https://inkscape.org/

[21] Database Normalization. [online]. [cit. 2016-06-01]. URL: http://en.wikipedia.org/wiki/Database_normalization

[22] MVC Applications & Presenters in Nette. [online]. [cit. 2016-06-01]. URL: http://doc.nette.org/en/2.3/presenters/

[23] AJAX. [online]. [cit. 2016-06-01]. URL: http://en.wikipedia.org/wiki/Ajax_(programming)

[24] JSON. [online]. [cit. 2016-06-01]. URL: http://en.wikipedia.org/wiki/JSON

[23] CloudFlare - The web performance and security company. [online]. [cit. 2016-06-01]. URL: http://www.cloudflare.com/

[24] LESS Compiler in PHP. [online]. [cit. 2016-06-01]. URL: http://leafo.net/lessphp/

[25] Google Closure Compiler. [online]. [cit. 2016-06-01]. URL: https://developers.google.com/closure/compiler/

[26] Gettext Latte Add-on to Nette. [online]. [cit. 2016-06-01]. URL: http://addons.nette.org/h4kuna/gettext-latte/

[27] Doctrine – PHP ORM Framework. [online]. [cit. 2016-06-01]. URL: http://www.doctrine-project.org/

[28] Unit Testing. [online]. [cit. 2016-06-01]. URL: http://cs.wikipedia.org/wiki/Unit_testing

[29] Mockery – Simple PHP Mock Framework. [online]. [cit. 2016-06-01]. URL: https://github.com/padraic/mockery

[30] Nette Tester. [online]. [cit. 2016-06-01]. URL: http://tester.nette.org/

[31] Testbench for Nette framework. [online]. [cit. 2016-06-01]. URL: https://github.com/mrtnzlml/testbench

[32] Selenium Web Browser Automation. [online]. [cit. 2016-06-01]. URL: http://www.seleniumhq.org/

[33] LoadImpact. [online]. [cit. 2016-06-01]. URL: https://loadimpact.com/

[34] Loader.io. [online]. [cit. 2016-06-01]. URL: http://loader.io/

[35] ThePay [online] [cit. 2016-06-01]. URL: https://www.thepay.cz/

[36] AngularJS [online] [cit. 2016-06-01]. URL: https://angularjs.org/

Appendix A – Contents of the Enclosed CD

The enclosed CD contains the electronic version of this master thesis, scripts for generation of programmer documentation and the source code of the application including the test scenarios, external libraries, installation script and other tools.

The directory structure of enclosed CD follows:

/doc/

Scripts for generation of programmer’s documentation of the application.

/src/

Source code of the application.

/readme.txt

Description of CD contents.

/thesis.pdf

Electronic version of this master thesis.

/user-manual.pdf

User manual for the system.