Bachelor Thesis, Institute of Computer Science, Freien Universität Berlin

Development of a mobile payment service with focus on evaluating API recording for automated mocking

Johannes Würbach (4278779)
johannes.wuerbach@googlemail.com

Berlin, January 23rd, 2013

Supervisor:	Prof. Dr. Lutz Prechelt
Second supervisor:	Prof. Dr. rer. nat. Adrian Paschke
Adviser (external):	Dipl.-Ing. (BA) Robert Glaser

Contents

1 Introduction **4**
- 1.1 The service . . . 4
- 1.2 The company . . . 4
- 1.3 Goal . . . 4
- 1.4 Approach . . . 4

2 The existing solution **5**
- 2.0.1 Overview . . . 5
- 2.0.2 Mobile payment . . . 5

3 Current problems and required improvements **5**
- 3.1 Technical problems . . . 5
  - 3.1.1 Difficult to maintain and extend . . . 5
  - 3.1.2 Payment provider implementation . . . 6
  - 3.1.3 Alarming statistics . . . 6
- 3.2 Bad usability . . . 6

4 The new solution **7**

5 Implementation of the service **10**
- 5.1 The environment . . . 10
  - 5.1.1 Rails instead of the Zend Framework . . . 10
  - 5.1.2 Continues integration as a service . . . 11
  - 5.1.3 Code management . . . 12
  - 5.1.4 Development live cycle of a feature . . . 12
  - 5.1.5 Hosting . . . 12
  - 5.1.6 Task management . . . 13
  - 5.1.7 Downtimes . . . 13
- 5.2 The structure . . . 13
  - 5.2.1 Javascript-API . . . 15
  - 5.2.2 Server-to-server request . . . 15
- 5.3 Provider implementation . . . 16
  - 5.3.1 Types of integrations . . . 16
  - 5.3.2 Example Google Wallet . . . 17
  - 5.3.3 Example MobiTown . . . 17
- 5.4 Do not invent the wheel again . . . 18
- 5.5 Testing . . . 18
  - 5.5.1 Testing the Javascript-API . . . 19
  - 5.5.2 Testing the ruby code . . . 19
  - 5.5.3 API-Recording . . . 19
- 5.6 Broken holidays . . . 21

6 Conclusion **23**

7 Forecast **23**

8 References **24**

1 Introduction

This thesis covers the creation of a payment service for HTML 5 products, which could be used as a drop-in replacement for in-app purchases outside of App Stores. As testing is a really important aspect of creating a payment service, this thesis will give you an inside look how to test different communication ways like AES encrypted communication, signature based communication, SOAP via HTTP or a RESTful communication.

1.1 The service

As a developer of a mobile HTML5 free-to-play game it is quite difficult to integrate a proper payment solution, which works in various countries. On the one side there are providers with their own customer base like PayPal or Google Wallet on the other side there are SMS and WAP payment providers with different regulations and price points in every country. The goal of the payment service created in this thesis is to allow 3rd-party developers to bill customers everywhere, without knowing any country specific regulations or prices.

1.2 The company

The Softgames GmbH is a company build around mobile gaming. They started with creating and distributing mobile java games six years ago and have extended their business to Android and HTML5 games.
Their latest product is called 'SG Connect', 'which solves developers biggest challenges: Distribution, Monetization and Discovery of their HTML5 Games' (1).

1.3 Goal

The main goal of this thesis was to extract the payment integrated in the existing application (SG Connect) and to reimplement it as a service. The resulting service should be usable without using the 'SG Connect' platform.

1.4 Approach

The following steps have been chosen to achieve the goal:

1. Identify and list the problems of the existing solution from the technical side
2. Collect wishes and issues other departments have with the current system from the management side
3. Decide, whether refactoring or reimplementation is the better approach to extract the payment as a service and present that solution to the company
4. Develop the service

These steps are also the structure of this thesis. The advantages and disadvantages of the existing implementation are explained in section 2. Required improvements and current problems are listed and explained in section 3. The final specifications for the service and explained flows are mentioned in section 4. Decisions and problems during the implementation of the service are written down in section 5.

2 The existing solution

2.0.1 Overview

The existing solution was build using the PHP based Zend Framework 1 and some Python scripts. PostgreSQL was used as database in the background. Developers were using a SVN repository hosted on an own server to manage their source code. A feature or bug fix was developed in a feature branch, this branch was merged by the lead developer of the team into trunk and the trunk was deployed with some Python scripts to some dedicated servers. These servers were managed by the Softgames IT and one additional external server administrator in Poland. Only the lead developer had access to the live system and the knowledge, how to deploy the system.
The existing solution had approximately 63.000 lines of code [1]. Maintaining the existing code and developing new feature was hindered by nearly no documentation and the complete absence of any automated tests.

2.0.2 Mobile payment

In the world of mobile payment you have a lot of different providers for different regions and markets. Well-known examples are PayPal and Google Wallet, but for games, users prefer SMS- or WAP-payment, which is provided by many small providers. These providers are offering some specific price packages in a list of countries. So a main goal for the payment service should be, to make the integration of mobile provider as simple as possible to have access to a large number of country specific packages and to replace providers and case of rate changes. In the existing solution the implementation of Google Wallet[2] payment was done using four files and approximately 400 lines of source code.

3 Current problems and required improvements

3.1 Technical problems

3.1.1 Difficult to maintain and extend

The biggest problem of the existing solutions was the full lack of code tests. This made any code changes in the existing nearly undocumented codebase really complicated

[1] counted using cloc version 1.56 on a trunk checkout from the 27.12.2012
[2] `http://www.google.com/wallet/`, Accessed: 18.01.2013

and developers were always trying to make the easiest solution as possible, even if this solution included copying a lot of existing code. On top of that nobody fully understood internal flow in the application between an incoming http request and the generated response. Also the reasons behind some uncommon architectural decisions were not clear anymore. The reason for that was, that nearly nothing was documented and that most the developers of the existing solution had already left the company.

3.1.2 Payment provider implementation

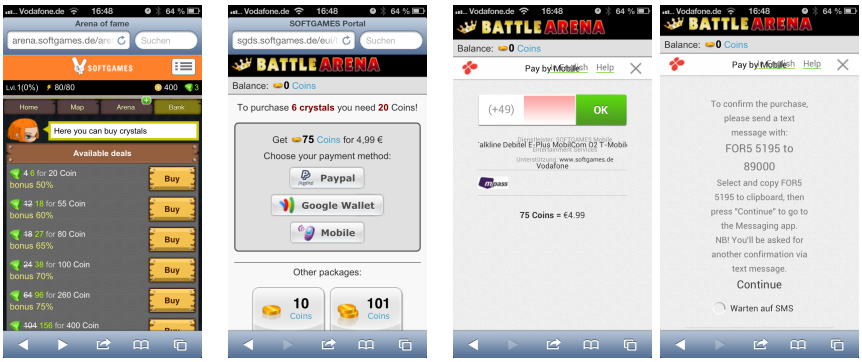
Normally the implementation of a new provider was done by copying a previous implementation and perform all necessary changes in that copy. During this process around 50% of the copied code were not changed and additional 20% of the code required only the renaming of variables and classnames. This process produced a lot of duplicated code.

3.1.3 Alarming statistics

To understand the distribution of transactions between countries and providers some analysis were made. This resulted in some alarming indicators. On the one side around 97% off all transactions were canceled and on the other side some providers implementations were not working at all. One example for a broken implementation was Google Wallet. The analysis of the company's data resulted in zero successful transactions and no revenue, but creating the same statistics on the data provided by Google itself was reporting 70% successful transactions and several thousand Euro of revenue.

3.2 Bad usability

Purchasing additional in-game currency was quite difficult to understand from a user perspective. In the first step a user had to select the amount of coins (1(a)), he wanted to buy for his game (in-game currency). In the next step (1(b)) a new page opened and the user had to exchange SGDS coins for the in-game currency. These SGDS coins were shared between all games in the portfolio of the company. If the user had not enough SGDS coins, he could buy a package of SGDS coins with real money (1(c)). To do that, he had to select a provider (mobile in the example) and had to perform the steps the selected payment provider required. For the most used SMS payment flow, these steps are: Enter your phone number. Receive a PIN from the provider in a text message and enter this PIN back into the website, if the network operators in a country are allowing 'MT-Billing'. If the operators are using 'MO-Billing', the user has to sent a text message including a PIN to the payment provider to finalize the transaction. An provider using the 'MO-Billing' flow is shown in the figure 1.



(a) Select package (b) Buy SGDS coins (c) Enter number (d) Send message

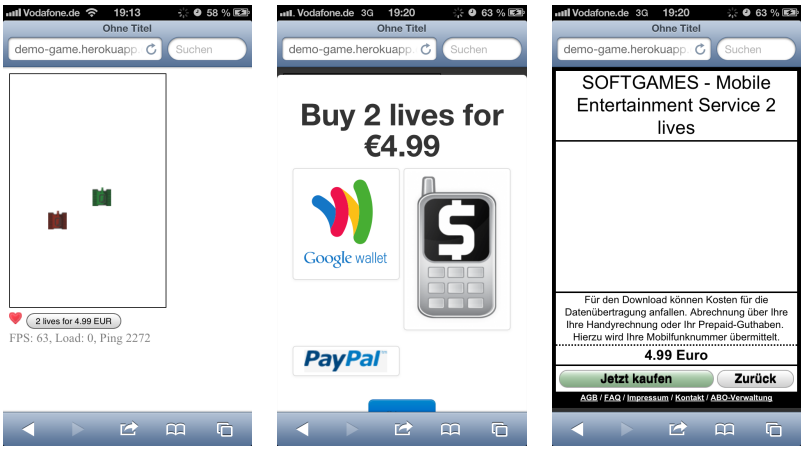
Figure 1: Existing payment flow

4 The new solution

For creating the service and simplifying future discussions about it, an overview of all possible paths an user can follow was created and is shown in figure 4. The general consensus in the management was, that the payment has to be simplified and unnecessary steps should be removed. To start the payment the user selects a package like before, but instead listing SGDS coins, real prices have to be displayed below the packages. After the user selected a package, he can decide between the available payment providers available in his country. This list of providers is presented in an overlay to the user and he is not 'visually removing' from the game like before. After the user selected a provider, the interface of the provider is displayed. Those provider interfaces are not customizable by the services and because of that marked as black box in the overview. After the user finished all required steps, he will see a success or an error page from the service.
The figure includes already possible future improvements like automatically select a payment provider and skip the selection list. Displaying other payment methods again after the user canceled the purchase in the black box could be also tried to increase the overall percentage of finished transactions. Between those things and the normal way A/B testing should be possible to optimize the service customers in specific countries.

To simplify the integration of mobile payment a tier based approach has been chosen. As already mentioned, a world wide mobile payment solution consists of a large number of single price points.
For example the SMS payment provider Fortumo[3] can bill 0.49, 0.99, 1.99, 2.99, 3.99 and 4.99 EUR in Germany, but only 1.45, 3.63, 5.20 and 7.26 EUR in Spain. IPX[4], which is providing SMS payment only in the United Kingdom and Germany, is offering

[3] `http://fortumo.com`, Accessed: 15.01.2013
[4] Not online accessible



(a) Select the item (b) Select the provider (c) Pay

Figure 2: New payment flow

0,19, 0.49, 0.99, 1.29, 1.49, 1.99, 2.99, 3.99, 4.99, 9.49 and 9.99 EUR to be billed. As game developers should not care about different currencies and country specific buying power of a playing user, the services provides price tiers.
A tier represents one fixed package per country and has nearly the same amount everywhere. An excerpt from the used table of tiers is shown in figure 3.
Developers can now sell their product for a tier instead of a concrete price and the service provides an API to convert the selected tier on-the-fly into the related amount of money and currency for the current user. For example a Samurai sword should be sold for approximately one Euro in Germany. According to the tiers table provided by Softgames this price is represented by tier '1'. If a user from Spain is now visiting the shopping page the developer requests the concrete for the price for this user, the payment service will return '0.86 €' as this is the best matching package available in Spain to be billed.

List of tiers

Tier	BE	CH	CZ	DE	DK	ES	FR	PL
1	1.00 EUR	2.00 CHF	20.00 CZK	0.99 EUR	10.00 DKK	0.86 EUR	1.00 EUR	2.46 PLN
2	2.00 EUR	3.00 CHF	50.00 CZK	1.99 EUR	15.00 DKK	1.45 EUR	2.00 EUR	7.38 PLN

Figure 3: Excerpt from the tiers table

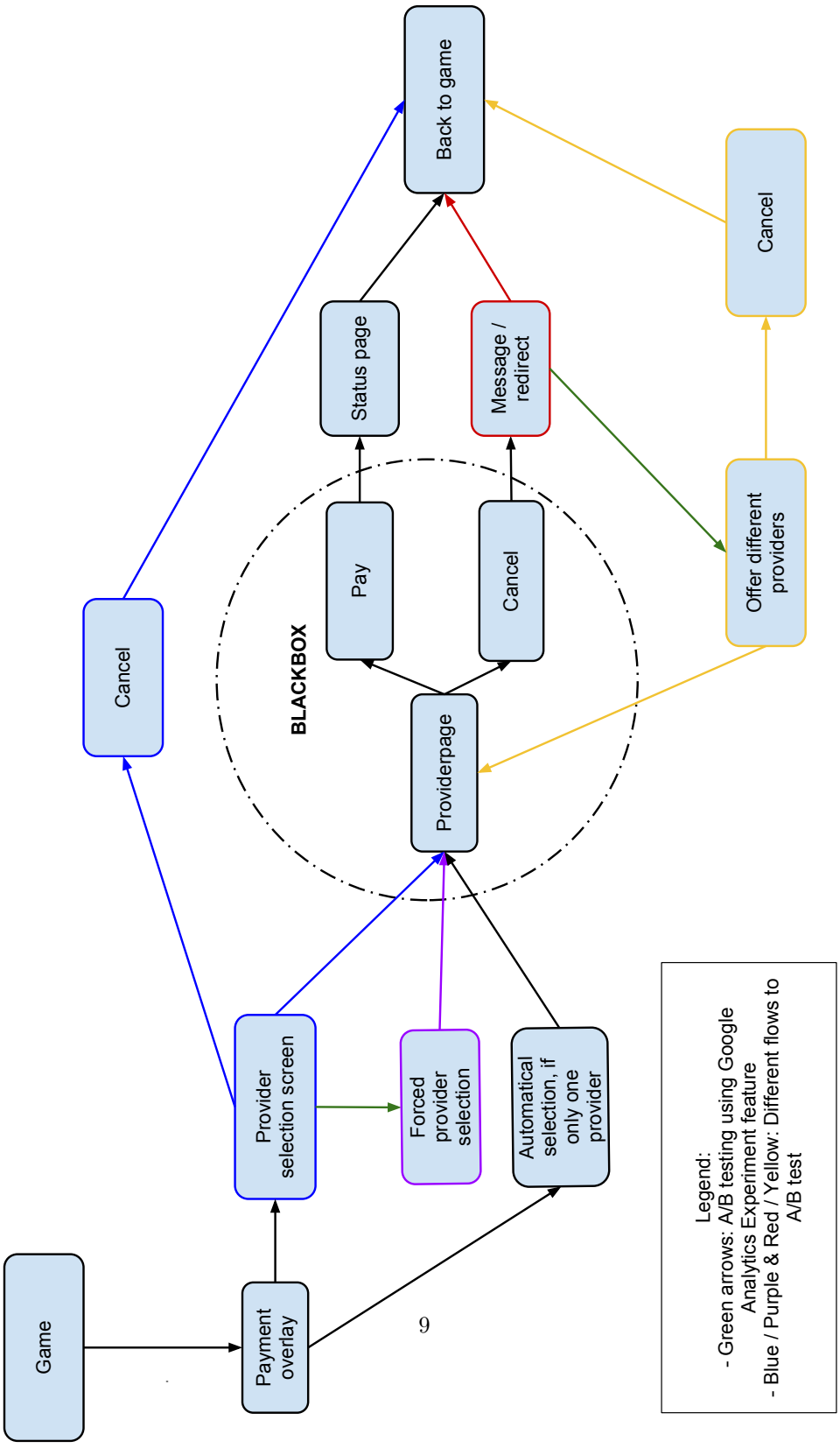


Figure 4: Steps a user can take

5 Implementation of the service

5.1 The environment

After analyzing the existing solution, it was decided to recreate the new payment service from scratch and ensure test coverage from the beginning. To do that, the service should be implemented using test-driven development.

5.1.1 Rails instead of the Zend Framework

As mention already the previous service build using the PHP based Zend Framework 1, Apache, Memcache, Postgres and some custom database migration and deployment scripts written in Python.
The installation and maintaing of a PHP based application is not very developer friendly. Indeed PHP 5.4 includes a web server and it is not required to configure and run apache on your local machine anymore, but there are a lot of libraries incompatible with that version. As development of a web services includes using several libraries, you should also be able to update them with a few steps and other developers should be able to install those libraries in the same version quite simple. This is not possible at the moment in a PHP application without requiring a lot of manual work. There is a dependency manager for PHP called 'Composer'[5], but this manager supports only a few packages at the moment.

Beside a simple installation and automated dependency management, a solution for database migrations and assets compilation and minification should be also included in the new framework.

As the most developers in the company had already more experience in Ruby than in Python, Ruby on Rails (RoR)[6] was decided to be the framework for developing the payment service. Node.js[7] was also included in the evaluation as it delivers an impressive performance, but it was discarded because there is no solution like Rails for Node.js, except the early alpha Tower.js[8].
RoR has the advantage that every developer can setup the application in a few minutes without being forced to use specific operation system.
After that decision was done, it was evaluated, whether PostgreSQL is still the right database. As executing payments is always tied to generate statistics about what was sold in which time frame a pure key-value storage like Redis[9] was discarded. As the document based database MongoDB[10] has a really advanced integration into Rails, the database came also into consideration. MongoDB was discarded, because a payment system is always storing transactions and those transactions are unlikely to be

[5] `http://getcomposer.org/`, Accessed: 23.01.2013
[6] `http://rubyonrails.org/`, Accessed: 23.01.2013
[7] `http://nodejs.org/`, Accessed: 23.01.2013
[8] `http://towerjs.org/`, Accessed: 23.01.2013
[9] `http://redis.io`, Accessed: 15.01.2013
[10] `http://www.mongodb.org/`, Accessed: 23.01.2013

extended with additional data, so a schema-less approach was not required. As the majority of the developers was also more experienced in writing SQL queries then working with MongoDB, it was decided to use a SQL based database again. Since the old system was already using PostgreSQL and is already preconfigured on the selected hosting partner, PostgreSQL was selected to be the storage backend of the payment service, too.
The installation of the old application required to install a web server, a specific PHP version, Postgres[11] and Memcache[12] in the beginning. After that you had to find some special PHP extensions compiled for your operation system or compile them. Next you had to checkout the code from the SVN repository. Creating a local test database was normally done by dumping the live system and loading this dump into your local database, which required some hours as the dump was usually some giga bytes large. Running the database migration scripts used to update the live database on your local environment required to install and setup a specific Python setup and modify the scripts to use the local environment, which nobody had done before.

Moving from the self-coded PHP to RoR simplified to this process to install RVM[13] or Pik[14], if you are using Windows. Install Postgres on your machine, create the default database user and clone the git repository. Run 'bundle install' to install all required extensions. After that the database could be created using 'rake db:setup'.

As running and managing the servers used by Softgames to serve the application and hosting the tools, which were used to support the development, was constantly stealing a lot developer time, the payment service should be build upon external services to save money and create clear responsibilities.

5.1.2 Continues integration as a service

To support the test-driven development a continues integration solution was required to ensure that all commits were tested. The most common solution for testing and building an application, is the continuous integration server 'Jenkins'[15]. The problem with Jenkins is, that it requires a server and somebody, who is maintaining this server. Configuring a Jenkins server to test every commit and configure which steps are executed before and after the testing requires also a training period. Instead of using Jenkins the service is using CircleCI[16]. This service is strongly created around Github[17], tests automatically every commit and is supporting the Github's commit status API[18]. The service automatically detected that a Rails application should be

[11] `http://www.postgresql.org/`, Accessed: 22.01.2013
[12] `http://memcached.org/`, Accessed: 22.01.2013
[13] `https://rvm.io/`, Accessed: 15.01.2013
[14] `https://github.com/vertiginous/pik`, Accessed: 15.01.2013
[15] `http://jenkins-ci.org/`, Accessed: 21.01.2013
[16] `https://circleci.com`, Accessed: 18.01.2013
[17] `https://github.com`, Accessed: 18.01.2013
[18] `https://github.com/blog/1227-commit-status-api`, Accessed: 18.01.2013

tested, created a test database and executed the example tests. As every build of the project is executed in a new virtual machine, it is not possible that former runs falsify the current build.

5.1.3 Code management

As SVN caused some problems already in the existing team. The payment service is now using git as source code management system and Github as provider for git. Using Github allowed also to enhance the code review process, as it is possible to comment directly on changed lines of code in a commit. Github also provides a simple wiki solution, which is used for documentation instead of the old self-hosted wiki.

5.1.4 Development live cycle of a feature

Using all this services together made the workflow of development much cleaner. To develop a feature or a fix a new branch is created like before. After the development is finished a pull request is created. This notifies the person responsible for merging the code changes and gives also an overview of the changes made during the development. CircleCI is testing the pull request automatically, whether all tests are still passing and provides the result directly on the Github page as shown in figure 5.

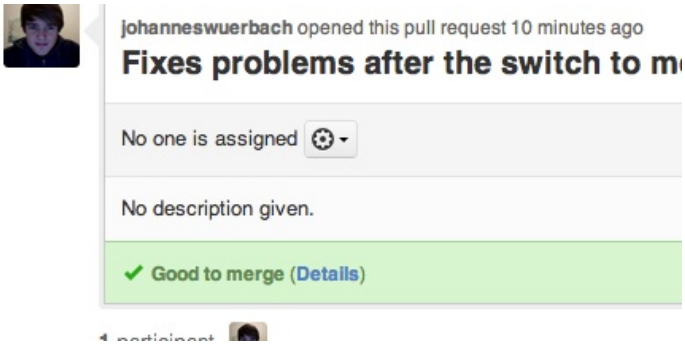


Figure 5: Github pull request status

5.1.5 Hosting

To remove the dependency from the external and sometimes unresponsive server administrator and to speed up the process for new developers, the payment service was hosted on Heroku[19]. Heroku provides hosting for Rails applications and does not require any administration or configuration. Deployment is done by simply pushing the code into a remote git repository. A nice advantage of Heroku is, that they are providing a development environment for free. This includes one running instance and

[19] `https://heroku.com`, Accessed: 15.01.2013

small postgres database. If you need other software beside ruby and postgres, Heroku provides a list of 'add-ons'[20] to use additional services provided by other companies. For the service, the 'MyRedis' add-on was used. This add-on provides a hosted Rails database, which is used for caching in the service.

5.1.6 Task management

The bug tracking software Mantis[21] and a list of notes on the iPad of the lead developer of the company was mainly used to coordinate the development of new features and bug fixes for the old service. During the creation of the service, the company was already in the process of moving everything into PivotalTracker[22]. As GitHub provides already an task management called 'GitHub Issues' and PivotalTracker provides a lot features not needed for a single developer, 'GitHub Issues' was used in the beginning to coordinate and plan the creation of the payment service. As a normal payment provider integration is done by receiving several integration documents in different formats the missing file upload in the issue system provided by GitHub was becoming a problem and the project was moved the PivotalTracker as well.

5.1.7 Downtimes

As most of the services used now are operated by 3rd parties, there is always the risk of unexpected downtimes. During the three months used for developing the service CircleCI and GitHub were down once during normal office hours. As the process was designed, to allow continuing the work during a downtime the option to manually push to Heroku was always available.
The only influence, caused by those downtimes, was less comfort and an additional mail to the support of the service somebody had to write.
In the old setup the whole development team was waiting until the responsible person had fixed the problem and the service was operational again. This caused two big problems. On the one side nobody wanted to have additional responses for a service, because those response where always causing a stressful time, if something was not working. On the other side running and configuring a service required also some inside knowledge of linux, which most developers in the company did not had, so most of the knowledge how to operate most of the services was combined into one person.

5.2 The structure

After collecting the requirements of the Softgames payment service the structure of the system was created in figure 6.

[20] `https://addons.heroku.com/`, Accessed: 15.01.2013
[21] `http://www.mantisbt.org/`, Accessed: 22.01.2013
[22] `https://pivotaltracker.com`, Accessed: 15.01.2013

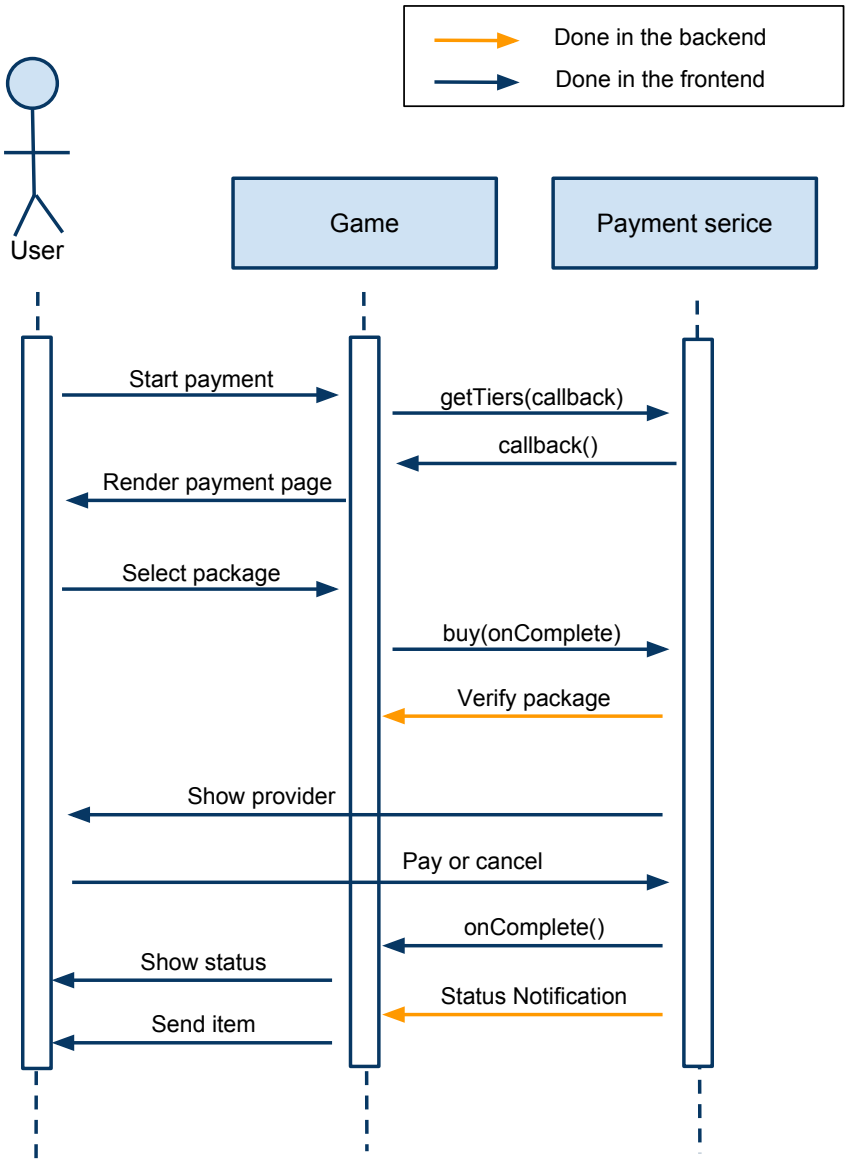


Figure 6: Request flow

5.2.1 Javascript-API

To allow developers to render concrete pricing information and to open the payment overlay, a javascript API was required. As this API would be included into the page of the developer some additional requirements have been arisen.
As it is not possible to build a versioning with a hard included files and some web browsers are gladly ignoring cache settings, the API had to be as small as possible and the service should never require a specific version of that file.
A solution to fix that, would have been a small loader, which loads the API with the current timestamp for cache busting attached. This solution was discarded, as the amount of data transferred in mobile networks is limited for most of the customers and this solution would not allow any caching on the device of an user.
In order to simplify the development of the API, it was written using CoffeeScript[23], which provides a the availability of classes and inheritance in the frontend and is compiled into javascript before being delivered to the client by Rails. CoffeeScript is already bundled with Rails by default and it is really common in the Rails community. Using Rails for delivering the API has also the advantage to split up the frontend code into multiple files, as it is possible to define dependency on top of each file. The files defined as dependency of a file are automatically loaded before loading the file itself.

5.2.2 Server-to-server request

After the user selected a package, the payment service verifies the incoming transaction with the backend of the game. This is required, because the relation between the SKU of an item and a specific price tier is transmitted from the game to the payment service using GET parameters, which are easily changeable by a user. After the provider backend verifies the relation between userid, SKU and tier, the user is able to continue the purchase.

This approach was used instead of pre-signing all parameters send to the payment service to save some resources for the game developers. As a game normally offers several packages, which had to be pre-signed, and just a small percentage of users is visiting the shop during each session, it does not make sense to pre-sign all packages in every session. Lazy pre-signing the packages using XHR would also impact the shop rendering time and decrease the usability of the payment flow.

The next server-to-server request will be sent, after the payment provider has confirmed the transaction by sending an update notification. This request usually arrives a few seconds later, but the transaction can also be pending for some hours depending on the external provider. As long pending requests are not helpful to satisfy the user, the service tries to only use payment providers, which are delivering quick updates. After the payment was confirmed or rejected by the payment provider and the service is updated via a notification, the developer has to be informed about this. Those notifications are created, signed and pushed into a queue during the update request.

[23] `http://coffeescript.org/`, Accessed: 20.01.2013

This is done to encapsulate the response time of the service from the response time of the game backend and to be able to repeat notifications. A worker daemon running in the background is pulling these notifications from that queue and tries to delivers them. If the delivery of a notification fails, because the response of the developer was not correct or the target server was not available, the notification is requeued with a growing timeout before delivery.

5.3 Provider implementation

The difficulty of implementing a new payment provider is really provider specific. On the one hand there are providers providing fully featured sandboxes and easy to understand documentations and on the other hand there are providers with broken implementations, wrong documentations and without any possibility to test a payment from Germany.

5.3.1 Types of integrations

In general there are two different types of integrations. Some providers are using a redirected based integration, which means the user is redirected to their side and others are using an integration based on javascript, which opens an overlay on the current page.

Redirect based

To implement a redirect provider you have to create an URL, which includes the amount of money and the currency you want to bill, a return URL and some provider specific security parameters like a signature. After that URL was created, the user is redirected to that. Now the user can pay on that specific site and will be redirected to the return-URL with some parameters attached.
Some providers are already sending a payment status in those return URL parameters, others are sending an additional server to server request for confirming the payment.

Javascript based

The implementation based on javascript is normally done by creating a JSON object with the country specific amount of money you want to bill, adding some security measures and inserting that together with the script of the provider into the message body of the response. The browser of the user is now executing a javascript function after the page has finished loading. This function receives the JSON object as a parameter and is opening an overlay. After the user finished or canceled his purchase in the overlay of the provider a javascript callback is executed.
As sending unsigned or unencrypted data via javascript is insecure, every javascript provider is confirming or rejecting a transaction using a server to server request. Those requests are signed or encrypted in all cases.

5.3.2 Example Google Wallet

As already implemented in the existing system (2.0.2), the reimplementation of Google Wallet into the new application was a great case study, whether the implementation of a provider was really simplified. The implement of this javascript based provider was done this time in 68 lines of code instead of over 400 lines of code in the old service and the number of files was reduced from four to two.

The reduce of complexity was mainly caused by identifying that there are only two types of integrations. This allowed to build two abstract flows and let every concrete provider just fill the gaps.
To do that all providers are extending a general provider class. This general provider class acts as database model and implements common methods require by more than one provider. A concrete provider has to implement maximal four methods. The first method is returning the type of the integration, which is used by the provider. If the provider is using a redirect based implementation, the implementation has to respond to at least two methods, the first one is returning the redirect-URL and the second one is called after the user came back from the provider. If the provider uses notifications an additional method has to implemented, this method is also required for javascript based providers. A javascript based provider is implementing instead of a redirect and return method, a method, which is returning a JSON object. For a javascript provider an additional view partial is required, which is rendered to start the payment, filled with the JSON object and loads the external javascript.

5.3.3 Example MobiTown

MobiTown[24] is a provider, which offers WAP payment in Thailand and is using a redirect based implementation. The documentation of their payment was clear until it comes to the security part. After the explanation which parameter in the query string has which function, they described to encrypt the resulting query string like this: (2, Page 8):

```
BYTE2HEX [ AES [ parameters ] ]
```

without any additional explanations what they mean exactly with AES or BYTE2HEX. The mail, which was included this document, contained also two strings labeled as key and salt. As two keys where provided it was reasoned that they are using a CBC mode instead of EBC. As both keys were 16 chars long and are only including ASCII chars it was decided to use AES-128-CBC. After some invalid request and additional mails they confirmed, that they are using AES-128-CBC, but are manually padding the string with whitespace before encrypting.
As the provider still could not decode our strings, the code used by them for encoding was requested. After some hours of debugging, the problem was finally found in the provided code. As MobiTown is using Java in their system they had to convert the key and salt string into bytes. Instead of using the getBytes() function provided by Java,

[24] `http://www.mobitown.asia/`, Accessed: 16.01.2013

they implemented their own function. The function is converting the string into a hex string in the first step and converting this hex into an array of bytes in the second step. The first step was implemented in a wrong way and the error resulted in a wrong hex representation. As the ruby AES implementation requires the key and the salt to be a string, the service performs now the following to generate the correct encrypted string:

```
hex_to_string(mobi_string_to_hex(provided_key))
```

5.4 Do not invent the wheel again

In contrast to the old solution, Ruby and especially Rails allows to include and manage 3rd-party-extension really easily. There are a lot different extensions used during the development of the service and here is an extract.
First of all the services is using ActiveAdmin[25] to provide an easy to use administration panel. ActiveAdmin has embedded authentication and allows to generate pages automatically to create, read, update or delete your database entries. If you are not conferrable with having all those options available or do not want to display specific fields of a model you can hide them. ActiveAdmin does most of the things without the requirement to write a single line of, until you want to have something custom or you are using complex relations in your database model.
To implement the queue used for sending notifications 'delayed_job'[26] was used. This gem allows to queue method calls in Rails and provides already a worker implementation, which you just have to start.
As the service deals a lot with money and currency conversations and there are a lot exceptions like currencies without fractions for example the Japanese yen (JPY) or currencies which are usually displayed with three decimal positions like the Kuwaiti dinar (KWD) the 'Money Rails'[27] gem was used, as it includes all those exceptions and provides methods for printing the currency with the correct symbol and exchange between currencies.

5.5 Testing

As testing should be taken serious this time, the whole service was created using test-driven development. Test-driven development means to write a failing test first. After you wrote that test, you can write the application code which make the test pass. It is only allowed to write only the necessary code to make test pass. To speedup the ruby gem Guard[28] was used to rerun tests automatically again, if a file was changed. As the test suite of a product is continuously growing Guard tries runs only the test, which are written for the changed file by following the naming conventions in Rails.

[25] `http://activeadmin.info/`, Accessed: 18.01.2013
[26] `https://github.com/collectiveidea/delayed_job`, Accessed: 18.01.2013
[27] `http://rubymoney.github.com/money/`, Accessed: 20.01.2013
[28] `https://github.com/guard/guard`, Accessed: 20.01.2013

5.5.1 Testing the Javascript-API

The tests for the Javascript-API, where written in the beginning using Jasmine[29]. As the integration of Jasmine into the CI-process was not really stable, the tests were rewritten using Mocha[30]. As the API is bundled with the rest of the application and will be merged and compiled by using the Rails assets pipeline, Konacha[31] provides a simple integration into Rails and allows to run the mocha tests in the browser or in a CI-environment by using Capybara[32]. Konacha also allows the tests to be written using CoffeeScript like the rest of the frontend code.

5.5.2 Testing the ruby code

For testing the backend code written in ruby, the service uses Rspec[33], FactoryGirl[34], Capybara with PhantomJS[35], Webmock[36] and VCR[37].

Rspec provides an intuitive language for writing your tests and is the glue, which holds together the other helpers. FactoryGirl is used for easily creating models and complex relations during the tests. Capybara allows to write integration tests as it allows to control a real browser from your tests. Instead of using the selenium[38], which uses by default the Firefox installed on the computer, the service is using PhantomJS, because it really speeded up the tests and stops the annoying popping up of the browser during a test run. To use PhantomJS together with Capybara, the service uses Poltergeist[39], which provides and easy PhantomJS integration into Capybara. Mocking HTTP requests is done by Webmock, which allows to intercept HTTP before they are send by ruby and to evaluate, whether those requests are done with the correct parameters and method or not.

5.5.3 API-Recording

Mocking a complete API with Webmock is possible, but not ideal as you have to manually update your mocks every time an API changed and you can not be sure that your mock implemented every edge case correctly. To solve that problem the service is using VCR. VCR allows the application to make real http connections in the first request and records them into a 'cassette'. Every additional http request is just answered by VCR using the recorded version. VCR is using Webmock to intercept requests directly in Ruby, so you do not have to change the application code to use VCR.

---

[29] `http://pivotal.github.com/jasmine/`, Accessed: 20.01.2013
[30] `http://visionmedia.github.com/mocha/`, Accessed: 20.01.2013
[31] `https://github.com/jfirebaugh/konacha`, Accessed: 20.01.2013
[32] `http://jnicklas.github.com/capybara/`, Accessed: 20.01.2013
[33] `http://rspec.info/`, Accessed: 21.01.2013
[34] `https://github.com/thoughtbot/factory_girl`, Accessed: 21.01.2013
[35] `http://phantomjs.org/`, Accessed: 21.01.2013
[36] `https://github.com/bblimke/webmock`, Accessed: 21.01.2013
[37] `https://github.com/vcr/vcr`, Accessed: 21.01.2013
[38] `http://seleniumhq.org/`, Accessed: 21.01.2013
[39] `https://github.com/jonleighton/poltergeist`, Accessed: 21.01.2013

This allows to simply test against a new version of an API by deleting the cassettes related to that API. VCR provides some helpers for using it together with RSpec for creating cassettes automatically named like the test they are recorded from and it is automatically using separate cassettes for every test, if you want to.
This approach does not fully work with testing an implementation of a payment provider, because a frontend interaction with the page of the provider is always required between different API requests. As those interim actions are done on a website served from another server, they are not recordable by using the VCR running in the payment service.
To solve that problem, different approaches have been tried.

Proxy approach

An option to work around this limitation is to visit the side of the provider and do all required frontend steps by using Capybara. As this is already done for testing the complete payment flow with some fake providers, it does not require additional software or knowledge.
The first problem, which was occurring here already, is that only payment providers providing a sandbox environment are testable like this. Otherwise every test run has to be made using real money, which is to expensive.
To prepare that approach a test for Paypal based on Capybara was created using the following flow. First login to the developer page of Paypal, to receive a session cookie. After collecting this cookie, the browser is visiting the payment service to start a Paypal payment, logs in with sandbox credentials received by Paypal and submits the payment. This tests requires approximately 50 seconds to complete and only covers a successful payment.
To keep tests fast and independent from the availability of the payment provider, those frontend actions done by Capybara should also be recorded. A graphical overview of the required steps is shown in figure 7.
PhantomJS, which is used by Capybara, as a headless browser, allows to specify an HTTP proxy. After implementing a simple proxy using ruby and VCR based on an idea posted on StackOverflow[40] the next problem occurred.
Most of the payment providers are using HTTPS to secure the payment process, but the proxy provided by Ruby's WebRick library does not support decrypting of HTTPS requests. After doing some research, how this HTTP proxy is processing HTTPS and whether there are other existing solutions for that problem, two options where remaining. The first is to modify the CONNECT method used by the WebRick proxy to allow decrypting of HTTPS traffic and the other option is to develop a man-in-the-middle HTTPS proxy from scratch.
PhantomJS allows to disable the validation of HTTPS certificates, so faking a certificate authority for signing certificates is not a problem, but there a still some steps to perform. The first step is to generate a valid certificate for the host name, PhantomJS is connecting to. After that the socket has to be updated into an SSL/TLS server socket and you have to intercept and parse the incoming requests and forward

[40] `http://stackoverflow.com/q/13039251`, Accessed: 21.01.2013

them afterwards using the library already used by the payment server to do HTTPS requests. As those requests are now done directly on the proxy, VCR is able to record them.
A proper solution using this proxy would require also to filter binary data to not bloat the code repository with recorded graphics and to find a solution for synchronizing between rerecording of VCR cassettes stored in the payment service and VCR cassettes stored in the proxy.
In addition to that work all payment providers which are providing a sandbox are still not fully testable, because of the lack of notifications, which are normally sent to inform the service about payment status changes. The only provider providing a sandbox and working without update notifications is Paypal. As Paypal is only one of seven providers implemented into the service, this solution was discarded and an hybrid approach was implemented.

Hybrid approach

Instead of trying to test every API request using VCR, only API requests are recorded now, which are done before an user interaction is required. This allows still to check, whether a partner has broken his implementation by deleting the related cassettes and whether the initializing of the payment process works.
Every request, which is executed after the user interaction, is tested against a manually created mock using Webmock. This approach also allows to test providers which do not provide a sandbox and providers which are requiring that the user is visiting their page by using specific mobile internet connection.

Summary

In general VCR provides all methods to test a pure REST-API and the hybrid approach could probably make sense, if the target pages are browsable via HTTP. In the case of this mobile payment service, where every page should be accessible only via HTTPS and most of the providers do not provide sandboxes and or are requiring server-to-server notification it did not made sense to implement.

5.6 Broken holidays

On 28th of December the CI service reported suddenly timeouts during the build-process. After reverting all changes the build was still failing. The problem was also, that the test suite was always crashing the complete ruby binary and not showing any special source of the problem.
After some hours of debugging the source of problem was found in an updated version of ruby. On 26th of december Ruby 1.9.3-p362 was released and the CI environment was updated automatically, as the service was just setting ruby version 1.9.3 as a requirement and not a specific patchlevel. This patchlevel release had an issue with with rails [41] and the problem was fixed by defining a concrete patch level for ruby.

[41] `http://bugs.ruby-lang.org/issues/7629`, Accessed: 08.01.2013

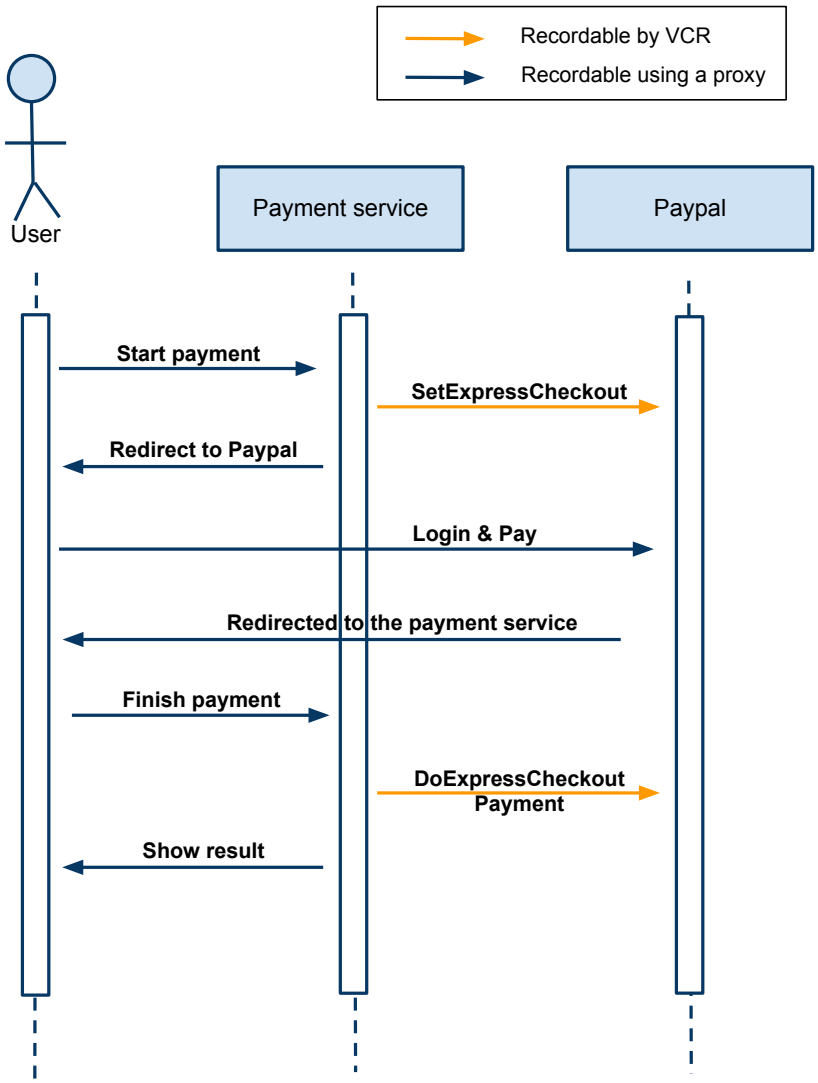


Figure 7: Proxy approach

Ironical the inventor of ruby, Yukihiro Matsumoto, closed his release notes with: 'Have good holidays, and happy hacking!' (3), which it really was not for me, because of that update.

6 Conclusion

The payment service can now be seen as a true service and the integration of new payment providers was simplified. 100% test coverage of the written code has made additional feature development and code refactoring less dangerous. By providing a tier based API 3rd-party-developers do not have to worry about different country specific price points or currencies. This simplifies the integration of real money payment in web applications a lot.

With using external services instead of self-hosted solutions, management costs were reduced a lot and developers do not have to care about running a build infrastructure or server management anymore. This resulted already in faster feature development in the company's core business.

7 Forecast

Currently the SG connect API is being redeveloped using the same software stack and principles as this payment service. After that reimplementation has been finished in the mid of february[42] the payment service will handle all real money payments in Softgames.
During march the ability to handle subscriptions, using various payment providers will be added to the service to provide all income types the Google Play Store or the Apple AppStore providing today.
Currently it is in discussion to open the service to developers not using our game publishing service, but this will also require to build a web interface for managing their settings, analytics about payment transactions, a sandbox to test their implementation and an automated billing solution.
In general the service is not tied to games as developers can sell everything they want for example music, software or movies.

[42] Current ETA, may change

8 References

[1] Softgames company page. http://softgames.de/distribute-monetize-mobile-html5-games-softgames-connect/. [Online; accessed December 25, 2012].

[2] Thawalit Junpoung. *MobiTown API Specification*, 2011. Document only available after signing a contract with Index Corp. (Thailand) Limited.

[3] Ruby 1.9.3 patchlevel 362 release announcement. http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/402471. [Online; accessed December 30, 2012].

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben. Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, den 23.01.2013

Johannes Würbach