Project CS - Product Report
Klarna E-commerce Framework

Christian Rennebruk, Shahzad Gul, Nicklas Nordenmark
Manzoor Ahmad Mubashir, Mikael Nordström, Kim Honkaniemi
Tanvir Ahmad, Yujuan Zou, Daniel Widgren

January 15, 2010
Abstract

There are not a shortage of e-commerce frameworks out there. Most of these lack the ability of any good distribution and scalability. What this project aims to do is building a ready-to-use, highly customizable and stable e-commerce framework all written in Erlang that gives the abilities other frameworks are missing. Not to compete with other frameworks but rather experiment with new cool technologies at hand.
## Contents

1 Introduction .................................................. 4

2 Preliminaries ................................................. 5
   2.1 Concepts .................................................. 5
      2.1.1 E-commerce framework ............................. 5
      2.1.2 N'Sync .............................................. 5
   2.2 Glossary .................................................. 5
   2.3 Development tools / software / languages ................. 6
   2.4 Target platforms ......................................... 7

3 Cookie Cart .................................................. 8
   3.1 Why Cookie Cart? .......................................... 8
   3.2 General Features ......................................... 8
   3.3 Administration features ................................. 9

4 System Description .......................................... 12
   4.1 Riak - Key-value store .................................. 12
      4.1.1 db.erl .............................................. 12
      4.1.2 db_appsettings.erl .................................. 13
      4.1.3 db_cart.erl ........................................ 14
      4.1.4 db_category.erl ..................................... 15
      4.1.5 db_content.erl ...................................... 16
      4.1.6 db_currency.erl ..................................... 16
      4.1.7 db_customer.erl ..................................... 17
      4.1.8 db_globals.erl ...................................... 19
      4.1.9 db_languages.erl .................................... 19
      4.1.10 db_media.erl ...................................... 20
      4.1.11 db_order.erl ...................................... 21
      4.1.12 db_pages.erl ...................................... 21
      4.1.13 db_product.erl ..................................... 21
      4.1.14 db_search.erl ...................................... 22
      4.1.15 db_tax.erl ......................................... 22
      4.1.16 db_user.erl ....................................... 23
   4.2 N'Sync .................................................... 25
      4.2.1 n_sync.erl ......................................... 25
   4.3 Plug-ins .................................................. 25
   4.4 Util ....................................................... 26
      4.4.1 Time stamps ......................................... 26
      4.4.2 Prices .............................................. 26
      4.4.3 BLOBs .............................................. 27
   4.5 Template system .......................................... 27
      4.5.1 Nitrogen ............................................. 27
      4.5.2 ErlyDTL ............................................ 27
      4.5.3 Globals ............................................. 27
C.12 Theme management ........................................ 41
C.13 Shipping module ........................................... 41
C.14 A default index page ....................................... 41
C.15 Usability issues ............................................ 41

D References and Websites ..................................... 42
1 Introduction

The goal of this project was to build a ready-to-use, highly customizable and stable e-commerce framework that combines the functionality of a normal web framework with the features a distributed key-value store provides and the scalability and stability abilities that Erlang offers.

There are many e-commerce systems out there today, and many of them are open-source. For us, our goal was not in any way to compete with them. Rather we wanted to try out new cool technologies and see if we could build a working e-commerce systems using these.

For this project we would use the Nitrogen [1] web framework. This is an new event-driven framework for Erlang [2], quick to learn and with lots of web 2.0 capabilities. As most traditional web databases are relational and not to distributable, we would use the newly developed Riak [3] as our underlying data store. Riak is a decentralized key-value store, which could scale well, and in many ways be a more natural fit for rapid web development than traditional RDBMS.

This project consists of nine master students in computer science from Uppsala University. The project was a cooperation between Uppsala University and Klarna, and was part of the course Project CS during the autumn of 2009. Klarna [4] provided us with the project outlines, concept, and lots of help and feedback during this time.
2 Preliminaries

2.1 Concepts

2.1.1 E-commerce framework
An e-commerce framework is basically an web-shop, an general all-in-one solution. This includes

- Well design web shop for shoppers
- Administration web interface for shop employees
- Built-in databases
- Content management
- Payment systems
- Customizable
- Extendible with plug-ins, payment solutions, themes and databases
- Ready-to-use

2.1.2 N'Sync
N’Sync is module for syncing web page content (pages, images, style, etc.) across all web server nodes. If content is added to the wwwroot directory without being properly synced through NSync, the content would not be available on all nodes. Consequently the page could crash, when connecting to a node without the uploaded content. Our goal was to create an system where everything could be managed through the administration interface. Since we can't upload or create new themes from the administration interface, we have not succeed with this entirely.

2.2 Glossary

**Bucket**  Buckets are containers for documents (see document-oriented database), basically a document namespace with some parameters and a loose schema.

**Cookie Monster**  Fictional Muppet character on the television show Sesame Street. Great inspiration for all of us.

**HTTP**  Hypertext Transfer Protocol is an application-level protocol for distributed, collaborative, hypermedia information systems. It is used for retrieving inter-linked resources, called hypertext documents.
**Document-oriented database**  Document-based databases store each record as a document that has certain characteristics. Any number of fields of any length can be added to a document. Fields can also contain multiple pieces of data.

**Key-value store**  Key value stores allow the application developer to store schema-less data. This data is usually consisting of a string which represents the key and the actual data which is considered to be the value in the key-value relationship. Key-value store provide a high performance alternative to relational database systems when it comes to storing and accessing data.

**Plug-in**  A computer program that interacts with a host application usually with specific a function that is called on-demand.

**Template**  A pre-developed page layout to make pages with a similar design, pattern, or style.

### 2.3 Development tools / software / languages

**Apache**  Web server software.

**Computer**  Magic thing, that no one truly understands.

**CSS**  Cascading Style Sheets is a style sheet language used to describe the presentation semantics of a document written in a mark-up language. Commonly used to style web pages written in HTML and XHTML.

**Django**  High-level Python web framework. [5]

**Emacs**  Rich text editor with useful Erlang mode.

**Erlang**  General-purpose functional programming language and runtime system, especially good for concurrency. It was designed by Ericsson to support distributed, fault-tolerant, soft-real-time, non-stop applications.

**ErlyDtl**  Template engine based on Django templates. [6]

**EUnit**  Lightweight unit testing framework for Erlang.

**gedit**  General-purpose text editor including tools for editing source code.

**gettext**  Internationalization (i18n) and localization (l10n) library.
**HTML**  Hyper Text Markup Language, is the predominant mark-up language for web pages.

**Hudson**  Continuous integration tool. [7]

**mochiweb**  Lightweight HTTP-server for running Erlang code. [8]

**Nitrogen**  Small and simple web framework for Erlang. Provides a robust, event-driven platform for creating dynamic Ajax web applications.

**Redmine**  Web-based project management and bug-tracking tool. [9]

**Riak**  Decentralized document-oriented key-value store with a flexible map/reduce engine. Truly fault-tolerant system that scales out as well as scales down easily.

**Subversion**  Version-control system. [10]

**Trackmania**  Arcade racing game to make breaks a little more fun.

**Whiteboard**  Glossy surface for social writing.

### 2.4 Target platforms

Cookie Cart is designed to work on any machine that can run the Erlang environment. Currently the key-value store used for the application can only run on Unix/Linux systems. Since the key-value store necessarily does not need to be on the same machine as the web server, theoretically, the web server can run an Windows OS while the key-value store resides on an Linux machine.
3  Cookie Cart

3.1  Why Cookie Cart?

Some e-commerce solutions seem to be complicated programming exercises and nearly impossible to install and use without an IT degree. Cookie Cart can be installed and set-up by anyone with the most basic website building and computer skills.

Cookie Cart offers a lot of features, some of the standard features are listed here. Please note that you can extend the functionality of Cookie Cart using plug-ins to make it do what you need!

3.2  General Features

- **Ready-to-use**
  You could have a professional-looking site up and running in minutes.

- **Web server scalability**
  The site can be extended to run on several web servers with an automatic syncing of content between them.

- **Paging, sorting, searching**
  Paging is applied on products. This means that products in a category is divided into several pages. The customer can choose how many products that should be viewed on every page. Products can be sorted according to its price, name and if its in stock or not. Products can also be search by name or tags applied to the product.
- **User accounts**
  An returning shopper can register an account and manage it at any time. Passwords can be reset if an shopper loses its login information.

- **Save carts**
  A customer can save multiple carts between visits to the shop. These carts can be modified and managed by the shopper.

- **Order history**
  A shopper can view all their previous order details.

- **Multiple languages**
  The e-commerce framework can handle multiple languages that can be easily switched by the customer on any time of a visit. **Price display**
  Currency and tax (with our without tax) display can be switched easily.

- **Guest checkout**
  Checkout can be done without creating an account (a so called guest account). This makes the checkout process quick and a bit more painless.

- **Product availability**
  The customer are showed the expected delivery time.

### 3.3 Administration features

- **Product and category management**
  An unlimited amount of products and categories can be managed. An product can be linked to several categories making it easy to create special offer categories for example. Products are added through a simple wizard.

- **Product attributes**
  Custom attributes (like size or color) can be added to the products.

- **Shop statistics/overview**
  An basic overview is given what items needs to be restocked, orders pending, sold items, monthly/yearly revenue etc. Simple statistics are also showed with Google charts.

- **Order status management**
  The systems gives basic functionality to handle orders. The system keeps track of all orders and an order status can be easily changed. An order can also be split into several ones.

- **Currencies and taxes**
  The webshop can handle several currencies and taxes. These can be easily managed through the simple administration system.

- **Flexible price display**
  Number and currency formatting, if tax is included or excluded can be changed.
• **Extensive language support**
  New languages can be easily added and extended through the administration interface. Default languages can be changed both on admin and customer side.

• **Basic shop preferences**
  The system gives basic control over shop preferences (e.g. shop name, contact information).

• **Administration security**
  Cookie Cart comes complete with an administration permission system, where roles and users can be defined. Users can be assigned multiple roles and roles can be given specific permissions to every page in the administration interface.

• **Content management**
  Uploading and management of pictures and other media can be done through an easy content management module. All uploaded content will be synced between all server nodes.

• **Plug-in management**
  Installation and uploading of different plug-ins can be done in the administration taking use of Erlang’s hot-code swapping feature.

• **Theme and template management**
  Complete control is given over HTML and CSS. Flexible layout of dynamic content are provided via the ErlyDtl template language. All user pages can be modified and extended without ever leaving the admin interface. Entire themes can be switched with the click of a button. You are also able to change CSS with an in-built editor.
• **Help pages**
  A simple help panel is provided for template variables.

• **Node management**
  Web server nodes can be easily added or removed through the admin interface. Listening of nodes up and running are displayed and all content are synced automatically between nodes.
4 System Description

This section contains information about the system architecture. It is split up into three different subsections where each part of the architecture is explained in further detail.

4.1 Riak - Key-value store

On the bottom layer sits Riak, our key-value store. A basic API was developed on top of the API provided by Riak to enable us to easily make use of a different DBMS (e.g., MySQL). This core API exports all the necessary functionality and thus makes it easy to write the more specific APIs.

Below you can find more detailed information regarding the different specific APIs and its corresponding structure in the key-value store. The image below explains the different entities in the structural diagrams.

4.1.1 db.erl

Core abstraction API that receives calls from the more specific APIs listed below. Contains the basic functionality such as INSERT, RETRIEVE and DELETE. As the open source version of Riak lacks a few vital features, these are
also implemented here. These features are: automatically create lists of added objects, enable the user to create indexes when inserting objects. This API also takes care of merging conflicting values as well as initialization of default values.

**LIST** The list bucket was implemented as Riak lacked reliable listing functionality. In order for us to be able to list specific information, keys are automatically inserted/deleted according to bucket name when an INSERT/DELETE is called. For example when a product P is added, its id is added to the list of ids at the key product.

**INDEX BUCKET FIELD** Another functionality that Riak lacked was the possibility to index on different fields than the key. We figured that this could be a very useful feature to have. Thus, we implemented an optional call that will create an index on the data belonging to the inserted object. This could be useful for finding all carts owned by a certain customer for example. The picture below illustrates the indices created for product names.

4.1.2 db_appsettings.erl

Takes care of general display settings such as default language, contact information, store name and terms of use but also more technical settings such as base currency and default images for products.
APPSETTINGS Instead of fetching everything or each attribute at one time, the application settings data is divided into groups, as the picture shows. The groups so far are INFORMATION, BASECURRENCY and SETTINGS.

INFORMATION is for holding contact info, addresses and such, for the web shop and its owner(s).

BASECURRENCY holds data for which currency is currently set as default, price formatting rules and conversion rate.

SETTINGS holds the settings about our customer-side pages, for example what theme to use, default front page, what language the client have selected, the company logo, the default product image and so on.

4.1.3 db_cart.erl

Takes care of functionality related to the carts stored in the key-value store. Contains functionality to create carts, list carts by customer id etc.

CART This bucket is for storing user-saved carts for returning(logged in) customers, each entry holds a product list, the customer ID of the customer owning this bucket, a customer chosen bucket name and a time stamp saying when the bucket was created. The product list is a list of one or more tuples, each tuple contains a product ID of a corresponding product and quantity representing how many of that product the cart contains.
4.1.4  db_category.erl

Database API for handling categories, with functionality for linking/unlinking of products to categories, fetching categories, adding categories, removing categories, editing categories, listing by all/visible/hidden, name to id converting (with the help of the above-mentioned index), and more.

**CATEGORY**  The purpose of the CATEGORY-bucket is to hold the data for all categories, each entry/key contains the category name, a flag that determines the visibility and a list of the products it contains.
4.1.5  db_content.erl

Takes care of the content management, i.e. all content that is stored in the database. This is for example images, style-sheets and template files. This also includes an inbuilt subversion system for files to be shared between different web servers. See the topic n sync for more details.

**CONTENT**  The content storage structure is very simple. Content files are stored with their respective paths as keys and their binary representation as value.

4.1.6  db_currency.erl

Takes care of the storing/manipulating of different currencies.

**CURRENCY**  The currency bucket is basically an extension to the previously mentioned baseCurrency section of the APPSETTINGS bucket. All available currencies in the system will be stored here.
4.1.7  db_customer.erl

Deals with customers added to the system. Handles both customers with accounts related to them and customers without accounts (people who have ordered goods without being logged in). Uses two different buckets to separate user information from account information (non-logged in customers will only have a record in the CUSTOMER bucket, not in the CUSTOMERUSER bucket).

CUSTOMER  The CUSTOMER bucket contains regular customer information, such as firstname, lastname, social security number (ssn) and contact information.
**CUSTOMERUSER**  This bucket contains additional information about customers that have an account in the web shop as well. CID (Customer ID) will serve as a link between the account information and the customer information. Also contains login information and a reset password token.
4.1.8  dbGlobals.erl

Deals with all global variables that are exported to the ErlyDTL template system.

**GLOBALS**  Stores all global variables that is accessible by the system (for more information about global variables, see the Nitrogen/Template section). It has a basic tuple structure of two elements where the first one represents the global and the second element the module, function and arguments used to evaluate the value of the variable.

4.1.9  dbLanguages.erl

Deals with languages added to the system. Handles extra information such as what physical translation file to use for a specific language.

**DB LANGUAGES**  LID represents the abbreviated country code (sv for Swedish e.g.), language the full language name and profile holds the value of the physical translation file where the translations can be found.
4.1.10  db_media.erl

Handles media files (images, audio files, flash files etc).

MEDIA This bucket has the same basic structure as the CONTENT bucket; a media path as key and a binary representation of the media file as value.
4.1.11 db_order.erl

Handles orders sent by customers. Also deals with the administration of orders.

ORDER  ORDER contains a customer id to see who submitted the order, a list of products and their ordered quantity as well as general order information such as creation date, shipped date, state of the order and total price. Also has a payment module field which says what payment module was used for this specific order.

4.1.12 db_pages.erl

Takes care of dynamic pages in the system. Specifies templates, plug-ins and globals on a page basis.

PAGES  Every page is stored with its name along with its selected ErlyDtl template (see Nitrogen/ErlyDtl for more details) as well as a list of apply-tuples (the format of the globals).

4.1.13 db_product.erl

Takes care of all information related to products.

PRODUCT  The information stored about products is pretty straightforward. It contains default fields for example name, description and sell price but our system also allows the shop owner to add additional fields such as colour, weight or size. When adding a new / modifying an existing product you can set
the tax rate field to an available tax rate (see TAXES below).

4.1.14 db_search.erl

This module is for making it possible to search for products or users or basically every kind of search-tags to anything, that you would want to be able to search for. So this module contains those basic functions for adding, removing and modifying entries in the search Key-Value store. When adding you add some sort of data, mostly in our cases these are product IDs, and some search tags you want to associate with this product ID and when someone searches for any of the words added for this product, the product ID will be in that list of search hits. When a search is made, the system gathers all hits for each search-word entered in the search, then merges the data together and sorts accordingly to most total hits first. i.e. searching for a b c d could result in 2 entries both having 2 hits, entry 1 hits a and b, as entry 2 hits c and d. This way both entry 1 and entry 2 will be as highly ranked in the search.

Known problems: When misspelling a word or entering a part of the word no hits will be found. More information about this will be found in Appendix C: Future Work.

4.1.15 db_tax.erl

Deals with different tax rates added to the system.

TAX TAX has a basic structure with name and respective tax rate to contain available tax rates in the system.
4.1.16  db_user.erl

This API takes care of employee accounts. It also contains the functionality of the role system (i.e. who has access where). Finally, it contains the functionality for the non-editable root account.

USER  USER contains employee accounts with user names and passwords. Every user also has a list of RIDs (role ids) which refers to the objects in the ROLE bucket as well as a status (active/inactive).
**ROLE** The ROLE bucket contains different roles to support an access level system. Every role has a list of administration pages that it should be allowed access to.

**ROOT** As some other E-commerce frameworks we looked at gave us the possibility to lock ourselves out of the system, we added the very simple ROOT bucket to hold a root account separated from all other accounts. Its user name
and password are set when installing the system and can not be removed later on.

4.2 N'Sync

4.2.1 n_sync.erl

This file contains all the functionality for making the content distribution to work, such as adding/removing nodes from the cluster, perform an update-check including fetching binary file data from Riak and writing to disc and functions for broadcasting new revisions.

N'SYNC NSync is the part of the system that acts as a glue in between the Nitrogen/Mochiweb nodes. Since Nitrogen reads its content from disc and we have multiple nodes supposed to serve the same homepage, this will mean that all nodes will have to serve the same version of the file. So whenever a node uploads a new file to Node A, it must be available on Node B,C,D and E also. To achieve this we have made NSync, which consists of 2 modules namely db content and n sync. Where db content is just the API for fetching, adding and deleting content in the key-value store. NSync is basically a small automated subversion control system, where all the nodes in the cluster have their local revision number that keeps track of what version of the files they currently have on disc. NSync keeps its global currently highest revision number in the Key-Value store. So whenever a node uploads a new file, such as a template or a picture or anything that needs to be distributed, NSync sends out a notification to every other node with a notification message containing a new content-tag and a revision number. When a node receives the new content-message it will update its content if the revision number is greater than the local revision number. Also whenever a node starts up it will automatically check if its content is up-to-date.

4.3 Plug-ins

Thanks to the hot-code swapping nature of Erlang, we soon realized that custom user created plug-ins could easily be developed and we wanted a somewhat painless system for adding new plug-ins. As we sat down and discussed new
features we realized that many of them could in fact be implemented as plug-ins. Plug-ins don’t have a general key-value storage structure like products or categories. Plug-ins are in fact compiled Erlang modules (.beam) files that are uploaded and added to the code path. Information regarding the plug-in is then provided by the plug-in itself through certain standard functions.

We have three different types of plug-ins, namely tags, transformers and page plug-ins. The tags are of a very simple nature as they’re basically global variables that prints some sort of information. The transformer plug-ins act on certain global variables that already exist and transform them in some way. This could be useful for a profanity filter for example. The framework comes pre-packaged with a few already installed transformer plug-ins, for example the Klarna payment system. The third type of plug-ins are page plug-ins. This kind of plug-in actually serves an entire page on the administration side of whatever content the plug-in chooses to display.

The plug-ins have certain plug-in API functions provided by the plugin.erl module but can also use any database API if need be. This enabled the plug-ins to do anything in the system basically, even removing default things. Thus, the plug-ins could be seen as a security flaw and this is something that could be improved with further development.

4.4 Utils

As we developed the different database APIs and designed the general structure of data stored in the key-value store we realized the need for certain utility functions. Thus, utils.erl was created to contain common functions that are used by many different modules.

The utility module could be divided into four major sections; timestamp, price, validator and BLOB functions.

4.4.1 Timestamps

Time values are represented as UNIX integer timestamps to enable easy sorting as well as custom format for printouts (different date formats for different countries belong to future development though).

4.4.2 Prices

A large part of the development was focused on price arithmetic; multiplication of tax rates and conversion rates. We changed our minds a couple of times when it came to representation of prices but eventually ended up using integers in the key-value store.

The relevant price functionality consists of applying tax, multiplication of conversion rates as well as price formatting for display on the web pages.

So when we want to display the internally represented price of 25000, to a customer, we would need some function to convert it to a more readable representation, for example 250,00 kr. These functions that help us deal with this
conversion and handling of internal versus external prices, are great examples of utility functions.

4.4.3 BLOBs

This somewhat misleading term grew as a concept when we needed a name for our general structure in the key-value store. A BLOB in our system is a list of two-tuples containing the field key and the field value, for example [name, House, price, 45000]. The utility functions get option and set option were created for easy access of specific fields in those BLOBs.

4.5 Template system

4.5.1 Nitrogen

Nitrogen is a web framework that enables quick and easy development of event driven web pages. Nitrogen uses simple Erlang records which are translated into Javascript/AJAX/HTML code. This neat feature allows us to make extensive use of Javascript updates and sending/catching events without reloading entire pages.

4.5.2 ErlyDTL

ErlyDtl is an Erlang implementation of the Django Template language. The idea is to let Erlang modules pass variables to the ErlyDtl compiler and in turn allow us to use these variables together with simple HTML in the template files. We modified this slightly to suit our needs. Instead of passing those variables to the compiler we actually have our own global variables that enables us to use the different variables regardless of template file. [11]

4.5.3 Globals

The global variables are used in the templates and can contain just about anything from Nitrogen buttons to complex tuple lists related to products. These global variables do share a common format:

{ VARIABLE, [ MODULE, FUNCTION, ARGS , ] { } }

VARIABLE is the global variable in question and MODULE, FUNCTION and ARGS tells the system what function (in what module) should be called with what arguments in order to evaluate the value of VARIABLE. To add a tag plug-in to the system you basically only have to add a new global variable like the above mentioned. A transformer plug-in would basically add a MODULE, FUNCTION, ARGS tuple to a global and use this information to transform the given data.
4.6 Internationalization

Internationalization is one of the main features of any web-shop. Considering this we looked for the best solution. We used gettext for it. The name gettext is from the GNU package.

4.6.1 Gettext

The gettext application makes it possible to internationalize your application in an easier way. All translations are based on PO files. The PO files would contain all the possible strings of your application that should be possible to translate. It makes a DB for every language. In order to change language you need to read that particular database.

The PO files not only contain the translations but also some meta data. This describes the unicode transformation format.

Example:

------------
...removed some lines here...
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=utf-8\n"
"Content-Transfer-Encoding: 8bit\n"

4.6.2 How gettext works

Whenever you want to translate a particular piece of text, you wrap that text in a macro. These macros are defined in gettext.hrl. In those macros we are mentioning the required language.

Example:

...-
include(".../gettext/include/gettext.hrl").
changeLanguage(LangCode) ->
?TXT2("Hi", sve).
...

The ?TXT macro will be substituted with a call to:

ggettext: key2str("Hi", sve)

This call looks for Hi in the database sve. If there is no word found in the database it would return nothing but the same text. The data is stored in a dets file in the GETTEXT_DIR. In order to locate the directory we use an environment variable. The environmental variable is used to locate the GETTEXT_DIR. For example export GETTEXT_DIR=/klarna/lib/klarna-languages

The lang directory contains all the required information. The directory structure would look like this:
$(GETTEXT_DIR)/lang/gettext_db.dets
$(GETTEXT_DIR)/lang/default/$(GETTEXT_DEF_LANG)/gettext.po
$(GETTEXT_DIR)/lang/custom/$(LANG)/gettext.po
5 Evaluation and Testing

One of the points in doing this project was to have a fault-tolerant system that could perform and scale well. It would be impossible to accomplish this without some kind of testing. This collection of aspects and their respective testing environment or testing method we included in this project are presented below. To a large extent, our testing became de-prioritized due to not having sufficient time.

5.1 Testing of database API’s

5.1.1 EUnit

To ensure that our database APIs would meet the quality goals on the code level and to make full use of the scrum development method (which encourage re-factoring) we had to employ some sort of unit testing. During the work process people using these APIs needed to be ensured getting the correct output when doing calls so that as little time as possible would be spent on bug testing due to faulty APIs. Testing was done using EUnit mainly due to the following reasons.

1. It was recommended to us in the beginning of the course.
2. It was including in main distribution of Erlang.
3. It was easy to use.

Since our API’s changed a lot during the project our EUnit test are not entirely up to date. Due to time limits testing was not prioritized in the later part of the project.

5.2 Testing of web pages

Since most functionality done in web pages consists of side effects, EUnit testing was made impossible here. No other framework for web testing was used. Instead testing of these pages was done in two aspects.

1. See that the page renders correctly.
2. Since Nitrogen is an event driven system, every event had to be tested, simply by activating it and see that the expected result was given.

Without any web testing framework to our help, this was the simplest and most natural way to do it. This was in some way naturally included in the work process of doing a web page.
5.3 Performance testing

5.3.1 Scaling

One of the main goals of this project was to make an e-commerce framework that could scale well. You should easily be able to add and remove web server nodes or key-value stores. Due to problems in meeting project deadlines, testing on how large the system could scale was de-prioritized by the product owner. In the later part of the project, every workstation became a web server node. A few tries with success was also made adding several key-value stores. So even if a large scale test was not performed we could say that the system can scale, we just don’t know to which extent.

5.3.2 Load

Since we experienced some performance problems when loading a page on the customer site, we did some testing on how fast the page would load to determine which parts needed improvement. We did this by simply adding a huge number of products to the database to see if it was the rendering of the pages or the database calls that took time to perform.

No real testing was made how much traffic load the web page could handle. This was de-prioritized by the product owner due to time limits.
6 Related work

6.1 Zotonic

Zotonic [12] is a completely new content management system for dynamic web pages. Since this CMS is built with many of those development frameworks we use, it made the application interesting for us to take a look at. It is built with Erlang, Nitrogen and ErlyDtl but in contrary to us use PostgreSQL as a data store. Unfortunatly the source code for this application was released quite late in our project and therefore we could not take advantage of many of the great ideas Zotonic offers when we were building our core system. Would we have had this to look at in the beginning of the project, our system would probably looked a bit different.

6.2 PHP e-commerce frameworks

There is not a shortage of e-commerce frameworks out there. Most of these are done in scripting languages like PHP, Ruby, Python with an standard relational transactional SQL database. We looked on several different frameworks that we took some ideas from when making our system. We mainly looked at those below.

- osCommerce http://www.oscommerce.com/
- Zen Cart http://www.zen-cart.com/
- Magneto http://www.magentocommerce.com/
- Prestashop http://www.prestashop.com/

At first glance, these frameworks does not differ much from ours, except a bit more functionality of course (since ours just a prototype). The core difference between our framework and the examples above are our possibility of scalability and distribution (at least in theory).
7 Conclusions and future work

We developed a e-commerce framework using Nitrogen web framework and de-centralized database Riak. This new framework is highly customizable, flexible, scalable and fault-tolerant. It provides all basic and more and less advance functionalities that one e-commerce framework can provides, for example it has two major interfaces one for Administrator and one for clients.

From administrator site it has different functionalities like adding different users and assigned them different roles, new products and categorize it according to different types of categories, new languages, new currencies, new tax system, managing customer orders. It has more flexible feature Plug-in system, with the help of Plug-in system any body can integrate new functionality in this system and modify it according to his desires.

From client side it has main functionality that customer can buy a products. There are we introduced two methods for shopping one for logged in customer and one for without logged in customer. There is a built in payment gateway system is available it name is Klarna payment system.

Its cookie cart work in different ways. Cart can save products for seven days but if client want to save cart in database. He needs logged in after logged in he can save cart in his login. He can save more than one carts in his login and whenever he wants he can purchase products directly from saved cart or he can also edit and purchase products.

At the end of this project we completed more and less all requirement but due to shortage of time and we could not implement some functionalities that we feel that it should be inside this project in future. For example Integration with other databases, customer support module, auto detection of threshold level of stock, scan products with barcode scanner, recommendation of products to clients according to his search, strict administration login to specified IP addresses.
A Installation instructions

This guide is how you install all the tools and programs that you need for using our web-shop, this guide is for Ubuntu and haven't been tried on other operating systems. This would probably work on other Linux distributions as well.

A.1 Erlang

1. We need SSL headers in order for the crypto application to work, install by typing: sudo apt-get install libssl-dev

2. Go to http://erlang.org/download.html and download the latest source version of erlang.

3. Open a terminal and cd to where you downloaded the .tar.gz file to.

4. Unpack the file using tar xzvf <filename>

5. Change directory into the one you just unpacked into.

6. We need to install ncurses before continuing, so do that by typing: sudo apt-get install libncurses5-dev

7. When that is complete, run ./configure

8. make

9. sudo make install

10. Restart your terminal and start the erlang prompt by using erl

11. Verify that crypto is working by writing: crypto:start(). It should return ok.

A.2 Cookie Cart

If you got the program on cd copy the files into a directory that you choose.

A.2.1 Start Riak

To get everything starting you must first start Riak then go into the folder and use the start script. This is if you are using it on your own machine, if you have a server it must be running there before you start nitrogen. This is if you don't have Riak it will be problems when you are trying to connect. Go into your Riak folder and write the following in the terminal:

./start-fresh.sh config/riak-dets.erlenv

This will start up riak with the correct configfile.
A.2.2 Start Nitrogen/webserver

Go into your folder where Cookie Cart resides and write:

`.start.sh`

A.3 Documentation

When you compile the files you will have a documentation for the functions in the doc directory. Open `index.html` with your web browser to view the documentation.
B Maintenance instructions

B.1 Upgrading

In theory, the system could easily be upgraded with a checkout from a version control (e.g. Subversion) repository. Changed files could then be recompiled in the terminal with the command `sync:go()`.

B.2 Themes

B.2.1 Uploading a new theme

Uploading of entire themes are not yet supported doing in the administration interface. A new theme has to be uploaded in the theme folder. It will then appear in the administration interface for you to change. To make it sync between all web servers it has to be added to the data store with an `nsync` command.

B.2.2 Creating a new theme

Creating a new theme from an existing one is not yet a supported feature in the administration interface. Currently, you have to copy an existing theme into a new folder. You then have to add it to the data store with an `nsync` command.

B.3 Plugin development guide

Plug-ins have the potential to do just about anything, so be careful when developing them.

You should probably have a look at what is said concerning global variables before venturing into developing plug-ins, although the development of plug-ins in our framework isn't too much of a hassle. You're also required to know the basic structure of an Erlang module.

All plug-ins should contain a couple of standard functions, namely:

- `init/0` - adds the plug-in to the system
- `description/0` - returns a string which describes the plug-in and what it does
- `delete/0` - deletes the plug-in from the system
- `page/0` - OPTIONAL if you want your plug-in to also serve a web page on the admin interface, create this function

In order for a plug-in to be added to the system, the list of global variables needs to be modified. To do this, a couple of functions were created and put in the plugin API plugin.erl.
Global = atom()
FunctionList = [{module, function, [argument1, argument2, ...]}]

- get global(Global) - Returns the functionlist for Global
- add global(Global, FunctionList) - Adds a new global according to Global and FunctionList
- delete global(Global) - Deletes the global Global
- set global(Global, FunctionList) - Sets the functionlist of Global to FunctionList

There are two type of plug-ins, tag plug-ins (which is a new global variable basically) and transformer plug-ins (which takes the value of a current global variable and transforms it in some way).

B.3.1 Example tag plugin - exampletag.erl

The tag plug-ins are very trivial since they only need to add a global variable in the init() function and delete the corresponding one in the delete() function.

-module(exampletag).
-export([init/0, description/0, delete/0, do/0]).

init() ->
Global = exampletag,
% [{MODULE, FUNCTION, [ARGS]}]
FunctionList = [{exampletag, do, []}],
plugin:add_global(Global, FunctionList).
description() ->
"This is an example tag plugin that comes prepackaged with the system".
delete() ->
Global = exampletag,
plugin:delete_global(Global).
do() ->
"This is what the example plugin global returns".
% Does not have a page() function, thus will not try and render its own page

B.3.2 Example transformer plugin - exampletransformer.erl

The transformer are a bit more tricky since they need some more modification to the global variables table. In the init function we add our corresponding function list to the existing function list of whatever global variable we want to transform. When deleting, we delete the corresponding function list. Note that the plug-in can chose in what position of the function list it should act.

-module(exampletransformer).
-export([init/0, description/0, delete/0, do/1]).
init() ->  
% We want to transform the global variable header for example 
Global = header, 
% Get the old functionlist of the global 
OldFunctionList = plugin:get_global(Global), 
% [{MODULE, FUNCTION, [ARGS]}] 
NewFunctionList = [{exampletransformer, do, []}], 
% Set the new functionlist to the example variable 
plugin:set_global(Global, OldFunctionList ++ NewFunctionList). 

description() -> 
"This is an example transformer plugin that comes prepackaged with the system". 
delete() -> 
Global = header, 
% Get the old functionlist 
OldFunctionList = plugin:get_global(Global), 
% The new functionlist should be the old functionlist but 
% with exampletransformer removed 
NewFunctionList = 
lists:keydelete(exampletransformer, 1, OldFunctionList), 
% Set the new functionlist 
plugin:set_global(Global, NewFunctionList). 
do(Data) -> 
% Just return the value of the global we want to 
% transform and some extra string 
Data ++ " transformed". 
% Does not have a page() function, thus will not try and render its own page 

B.3.3 Installation of plugins 
Once you've finished your dazzling brand new plug-in you want to install it to the system, right? 

- The name of your plugin is the module name 
- Plugins are uploaded as .tar archives (containing the compiled .beam file as well as any image files or other files) 
- If your plugin module is called example.erl, the compiled file will have the name example.beam and the tar archive must have the name example.tar 
- Once you have your .tar file, head over to http://YOURDOMAIN.COM/web/admin/pluginlist to upload it 

Plug-in .beam files are stored in ebin/plugins/PLUGINNAME/ Any other files uploaded in the same .tar archive are stored in wwwroot/plugindata/PLUGINNAME/ The example plugin files can be found in the priv/ directory
C  Suggested future work

Though we have nearly all main functionalities what a good web-shop should have but there are couple of stuff that we wanted to have in our product but due to shortage of time or technical reasons we could not implement in our product. Some of main remaining features are given below.

C.1  Integration of other RDBMS

The idea is to communicate to existing DBMS with Riak. Due to shortage of time we could not implement it. This would mean that somehow we could use existing table of any DBMS for instance customer table in our current system without transferring all data into our database i.e. Riak.

C.2  Customer support

A good web shop should backed by 24/7 live technical and customer support. In order to provide satisfaction guarantee we could have customer care section for users. This can be done by providing live chat system, messenger, email and by telephone.

C.3  Auto management Control

Let the web shop manage your stock levels by itself. Re-order system could be used to mange stock. Set the re-order levels and web shop would take care of your stock by keeping stock levels at the right level. It would prompt If any item falls below your re-order level or goes out of stock. There could be system for automatically generate orders to give to your suppliers. By applying triggers over different events we can generate different analytical reports for administration purpose.

C.4  Bar-code scanner integration

In future, bar-code scanner could be connect with system to reduce the adminis-trative work. Checking in stock would be done with great efficiency. Moreover, This would allows you to perform stock checks much more precisely.

C.5  Knowledge based product recommendation system

System should recommend products to customers. This can be done by different ways. For instance there could be two type of recommendations, Global and Personalized. In Global recommendation system could recommend products price, quality, most sold. Where as in personalized product recommendations It uses previous shopping history.
C.6 Security

It ought to be possible to require that the admin interface can only accessed from a specific port or ip range. It should still be possible to do remote administration over HTTPS, for shops that want to enable that (perhaps a single shop owner who absolutely needs to access it even when on vacation abroad). Due to prioritization of other tasks we didn't apply hash functions on passwords stored in the database. They are now stored in clear-text but should in future be stored as one-way hashes (e.g. SHA-1 or MD5).

C.7 Scalability of the system

Current system supports the scalability of system. In order to do that, we need some configurations. There could be easy way to add new Riak node or web server.

C.8 OpenID user login

OpenID an decentralized authentication system for user login. This would allows customers to login to our system by using digital identity. It would replace the common login process that usually based on username and password.

C.9 Product specification

Product can be described with more rich ideas. Multiple images can be good feature which can be shown as sideshow. Product specification (size, weight, color...) should be added in product description. Zoom in and out of product images to get more detail information about the product overview.

C.10 Efficient Search system

In our current search implementation, the user is limited to searching of whole words, meaning that if a user spells it wrong or just enters a part of the search tag the user wont get the desired result. E.g. searching for red house wont give you the result for redhouse. This is a flaw that we knew of at the time we started to make the system, but we had to prioritize other parts of the system due to lack of time. Later on it came to our attention that there is a data-structure called Suffix Tree that can solve this problem for us.

Another improvement would be to suggest the correct spelling of misspelled words.

C.11 Extending the ErlyDtl filters

In the future we would like to extend the ErlyDtl filters with more tags and filters that could be used when writing templates. An example of this would be custom validation filters so a user could apply validation on text boxes.
C.12 Theme management

A better theme management is a wanted feature in our system. You should be able to upload themes and create themes from existing ones.

C.13 Shipping module

The system needs new module for handling shipping options.

C.14 A default index page

Right now the index page doesn’t contain anything useful. It would be suggested to have a default look of the index page of for example new products, discount prices and so forth.

C.15 Usability issues

Lack of sufficient feedback. The feedback system should provide enough messages to both customers and admin users after successfully completing a task or occasionally making a mistake.

After successful registration, other than putting all the account options on the top menu, a separate section called the My account can be added, where all the account management are putting together, and can be easily manipulated by customers.

Tax rate can be set by categories, when generating a new category, the tax rate can be added to this category, then user do not need to always specify the tax rate for each individual product.

Lack of Back and Forward button, users will be easily lost when they go deeply down in the hierarchy. The website should provide sufficient Back and Forward buttons for user navigation.

A complete search function. The site can add a search by category refinement into the site search for easier access to site sections. Customers can also refine the product results by color, shape, size etc, based on the product features, and allow them to clear the refinements any time.

Show per page sort option. When the number of items on the page increase, it is better to allow users to show a designated number of items per page any time with a show per page sort option. User can choose from several options, such as 20, 50 or 100 items per page. This needs to be implemented on all pages that lists products, customers and so forth.
D References and Websites


