Hashtux SAD Document

Aman Dirar Jerker Ersare Jonas Kahler
Dennis Karlberg Niklas le Comte Marco Trifance
Ivo Vryashkov

BSc. Software Engineering and Management,
Department of Applied IT,
Gothenburg University

December 2015

Contents

Chapter 1

Introduction

1.1 Product Vision (Updated)

> *HashTux is a platform for easy access to multiple social media feeds based on user search where the user has the option to filter information based on his or her preferences.*

HashTux is a web based service that displays the latest pictures, videos and other social media posts for a given hashtag. Media content is fetched from multiple social media sources and displayed in a web interface optimized for fullscreen view ideal for a conference or event context, having a screen illustrating different viewpoints of the chosen keyword subject.

As time passes, the different content objects come and go, new ones fade in replacing the old ones. Videos start playing automatically. The objects on the screen are ordered automatically so that all space on the screen is used properly.

There is already successful software that does something similar but with slight differences, like including only Twitter (Tweetdeck). Our selling point is that we include content from multiple sources (also including Instagram and YouTube, designed with a modular implementation that easily allows adding more) and also combine it with multiple media formats, including videos automatically playing. To fetch older information we want to include a time scroll functionality.

To ensure the system is available even in the case of a failure or maintenance in one or more servers, our goal is to have the system running on several servers. As a part of the constraints, a significant part of the software will be implemented in Erlang. We were also encouraged to use a NoSQL database and therefore use CouchDB.

1.2 Target Audience

Event and conference hosts
To easily show the latest multimedia from an event.

Social media users
For the ease of use, fun experience and the nice overview provided by our all-in-one solution.

General commercial use
Illustrate recent updates for a product or company name or other term.

1.3 Domain

- Information
- Social Media
- Trends

Giving internet users easy access to multimedia content for a particular search term, as well as information on current trends in search habits and on social media.

1.4 Business Cases

Main business case
Our service is free for personal use but requires buying a licence for commercial use. Right now we have the alternatives of buying either a 1-year license for €9.99 or a lifetime licence for €29.99.

Secondary business cases

- Aim for an acquisition by a third party.
- Collect general data about user habits, which could be valuable especially if we choose to integrate with a larger company that tracks user behaviour for marketing purposes or similar.
  Examples: search terms, related search terms, trends.

We also have a donate button that lets users to donate money if they want to.

1.5 Scope

The main purpose of our application is to provide the users with content gathered from several popular social media platforms and show it in an appealing way in the user's web browser.

For this project we decided to integrate Twitter, Instagram and YouTube as our data sources. After the end of this university project, we will possibly add more services (see Future).

The application comes with a history search functionality which supports Twitter and YouTube. Instagram is not supported at this time due to API limitations.

Our application is easily deployable and uses CouchDB as the data storage for user habit data as well as the application's cache.

From the beginning we wanted *HashTux* to have a very clean, uncluttered user interface to emphasize how easy it is to use. We also wanted to avoid a trap we fell into during our last terms project, of adding more and more features and not having enough time in the end to wrap up, debug and document everything properly. This is why we have focused quite a lot of our attention on things like availability tactics and making the whole application resilient, instead of adding a lot of flashy features.

We also consider improving this product after the scope of this university project, which is why we have also put some effort into providing a good infrastructure for that, such as using an issue tracker.

1.6 Definitions

To make it easier to understand this document, here are some terms we want to introduce:

1.6.1 Components

Client UI	Runs in the end user's web browser. We use JavaScript for dynamic elements.
PHP application	Runs on our Apache web servers. Written in PHP.
Backend server	Written in Erlang. Handles contacting external APIs and writing and reading to/from database.

1.6.2 External Components and APIs

CouchDB Used to cache social media post information as well as user habit data (related to how our system is used).

Twitter API

Instagram API

YouTube API

1.6.3 Other Terms

DB	Short for Database or CouchDB
Miner	Our word for the part of our backend server that connects to an external API to get data.
Miner component	The part of our backend server that relates to the above. We are considering making this its own component in a future release.
DB component	The part of our backend server that connects to CouchDB. We are considering making this its own component in a future release.

Chapter 2

Architectural Drivers

2.1 Functional Requirements

Must	
FR1	The user must be able to search for a term and get matching results in the form of multimedia content from social media services (initially we focus on Twitter, Instagram and YouTube). The way in which the content is displayed must conform to the terms and conditions for each social media API respectively.
FR2	The user must be able to select different options when searching for a term. Options are which services to search for (e.g. only twitter), content type (e.g. only images and videos), languages (e.g. only french).
FR3	The user must be able to select different options for how the content is displayed, such as tile size for displayed results (eg small) and refresh rate for updating the grid (e.g. fast).
FR4	The user can "freeze" (and then "unfreeze") a certain media item (tile in the grid), which means it will not be swapped out for a more recent media item in subsequent updates while it is "frozen".
FR5	The user must be able to search for content that is non-recent, for example content from two days ago.
FR6	The client UI must provide the user with the option of buying a license for commercial use. All users should also be able to donate money to our project through PayPal.
FR7	The client UI must be optimized for full-screen view and continue to show content if left unattended for an extended period of time.

Should	
FR8	The user should be able to see different user habit statistics in a separate part of the client UI. Dimensions: at least search terms, web browsers, platforms (operating systems) and their respective popularity among HashTux users.
FR9	The user should be able to view different periods of the user habit statistics dimensions.
FR10	The user should be able to search (free-text) in the statistics part of the client UI for keywords in each dimension.
Could	
FR11	On the front page or wherever appropriate, the user should be able to see current trends in social media and in the search habits of users to get some inspiration on what to search for.
Wont	
FR12	The user should be able to chat with other users of the application.

2.2 Quality Requirements and Scenario Analysis

Quality requirements for our project in order of significance (most important first). Quality scenarios are provided for each quality attribute as well as a brief description. For tactics addressing the QA mentioned, see Tactics.

2.2.1 Availability

Our goal for the product is to have an uptime of 99.999%. This value is expected when running the full server-side component stack on 3 servers or more. For a lower number of servers running the stack, we expect an uptime of the system of 99%.

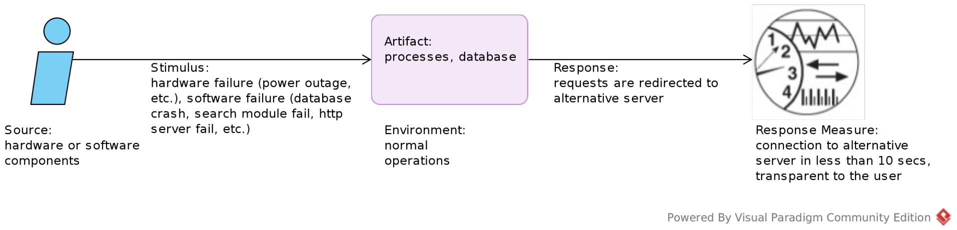
Source:
hardware or software components
Stimulus:
hardware failure (power outage, etc.), software failure (database crash, search module fail, http server fail, etc.)
Artifact:
processes, database
Environment:
normal operations
Response:
requests are redirected to alternative server
Response Measure:
connection to alternative server in less than 10 secs, transparent to the user
Powered By Visual Paradigm Community Edition

Figure 2.1: Availabilty Scenerio

2.2.2 Performance

The system needs to respond quickly enough for the user not to be discouraged from using our product. On average requests should be processed in less than 5sec. Database operations should be performed, on average, in less than 1 second.

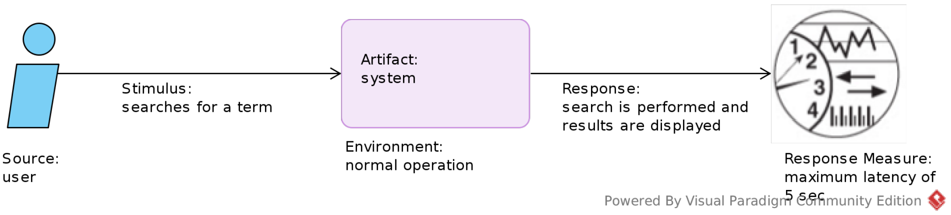

Figure 2.2: Performance Scenerio

2.2.3 Modifiability

The system should be designed to support easy modification. This requires aiming for having loosely coupled modules and a clear separation of concerns. For example, adding new modules to support different social media services should require low effort from the development team.

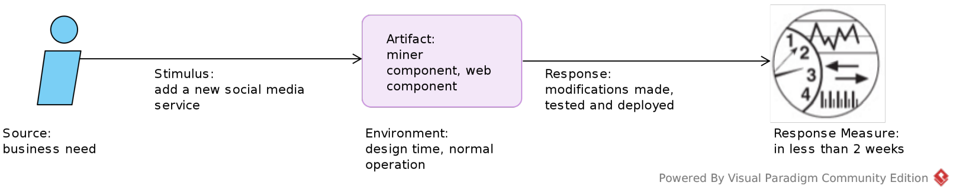

Figure 2.3: Modifiability Scenerio

2.2.4 Usability

The system should be easy to use in an intuitive way. This is hard to quantify, but we want our application to be as easy to use as the social media services we get content from, i.e. the typical user shouldn't have to consult a user manual to understand the provided features and options.

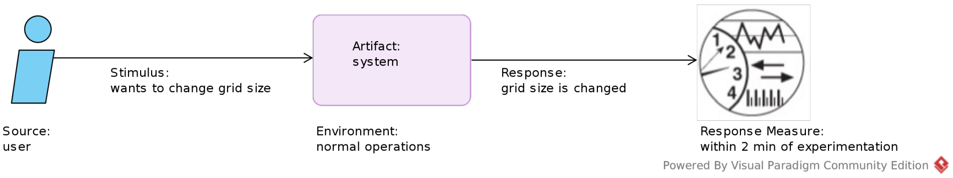

Figure 2.4: Usability Scenerio

2.2.5 Security

The system should have security mechanisms to prevent DDoS attacks and unauthorised access to our databases. Security is not our primary concern because we do not store or deal in any way with sensitive data like user login credentials or bank accounts. However, to guarantee availability, we must aim to minimize security vulnerabilities, both when designing, implementing and deploying our system.

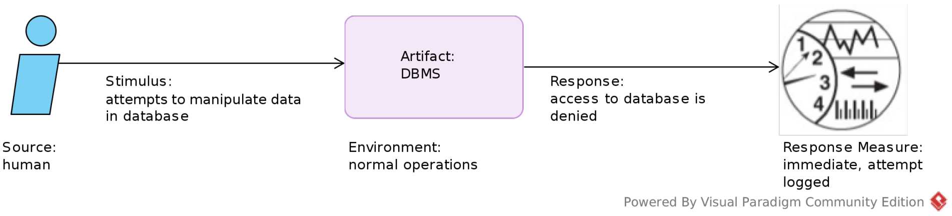

Figure 2.5: Usability Scenerio

2.3 Utility Tree (Tabular Form)

Quality Attribute	**Attribute Refinement**	**ASR**	**Impact**
Availability	H/W or S/W failure	Power outage: requests are redirected to an alternative server in less than 10 seconds	(H, M)
		Database failure: module responsible for database operations is reconfigured to use an alternative database in less than 3 seconds	(H, M)
Performance	Perform search	Search is performed and results are displayed in less than 5 seconds on average	(M, M)
	Check statistics	Statistics are fetched from database and displayed in less than 5 seconds	(M, M)
Modifiability	Add new social media service	Integrate a new social media service in less than 2 weeks (80 man-hours)	(M, M)
Usability	User wants to change tile size	The grid size is changed within 2 minutes of experimentation	(M, L)
Security	Database integrity	Unauthorised access to the database is denied	(L, L)

2.4 Tactics - Risks, Tradeoffs and Sensitivity Points

The tactics applied to achieve the above mentioned quality attributes are elaborated here and some risks, tradeoffs and sensitivity points are discussed.

2.4.1 Availability

Timeout - this tactic is used to detect if the backend server is available to serve requests. If the time to handle a request exceeds the configured value, the web application tries to connect to an alternative backend server. This helps to ensure that each instance of our PHP application can establish a connection with an available backend server.

A tradeoff for this tactic is a certain latency in responding to a search request when the currently preferred backend server goes down (see Reconfiguration for how we handle this).

Active Redundancy - for our application we have servers standing by (up and running live) and waiting to serve requests at any time. This also ensures the mentioned uptime of the system.

State Resynchronisation - used to provide a synchronised state of the collected user habit data across all available databases. This process takes place every minute.

Reconfiguration - we use this in two areas:

- We use this tactic to maintain availability when writing and reading to the database. If at any point the current database used for operations goes down, the module responsible for database procedures is reconfigured to use an alternative database.
- When a backend server fails to respond to a request in time, it will be marked in the web applications list of servers as down (with a timestamp of the incident). The web application will re-evaluate which backend server to connect to, and remember this decision. After a certain configurable time period, the incident will be forgotten so that the original server may be brought back to service (provided that it is now responding).

2.4.2 Performance

Limit Event Responses - this tactic is used when handling requests in the backend server to provide the required performance rates. This tactic is used together with the *Bound Queue Sizes* tactic. For our miner server, which is responsible for spawning workers to process requests, we have a queue size limit set to 1000. If the limit has been reached and the current request cannot be processed, the miner server sends a message back to the PHP application to notify it that there are currently too many requests being served and an alternative server should be tried.

Introduce Concurrency - a key tactic used in our system. It helps for parallel execution of operations and reduces the overall latency of serving requests. Concurrency is used in the backend server for managing clients, i.e. new threads are created for each request. Each request to the social media APIs are also handled in parallel. The same applies for database operations.

A risk of race condition when using concurrency is possible. This can occur especially when reducing the re-reduced results for caching the user statistics. Therefore, we created a custom pmap function to avoid these race conditions.

MapReduce - this tactic is used when caching summarizations of various user statistics for our statistics page. The user habits we collect generate a large amount of data and the use of MapReduce helps with precaching the statistics. The cached data is generated and stored in a separate database every hour and can then be quickly fetched when the client UI needs to display it.

2.4.3 Modifiability

A risk in general for modifiability is increased overhead when using intermediaries. This can affect the overall performance of the system. We do not identify this to be an issue for our product as it is. However, if the application is to be developed further and the size and scope increases, a tactic like *Reduce Overhead* should be considered to maintain the agreed performance levels.

Increase Semantic Coherence - this tactic addresses the separation of concerns. Each module has a clear responsibility and reduces the likelihood of side effects and errors when making changes.

Restrict Dependencies - this tactic addresses the concept of coupling between modules. Each module can interact with a certain number of other modules and low dependencies result in low cost and effort when implementing a change. It also keeps the complexity level low.

2.4.4 Usability

Aggregate - the different options for our application are presented in one simple view. This increases accessibility (all options are aggregated in one place) and reduces time in changing them (no need for the user to search for multiple menus containing different alternatives).

2.4.5 Security

Authenticate Actors - default security mechanism provided by the DBMS.

Change Default Settings - during deployment, we do not use the default settings that come with the different services in use. For example, CouchDB listens by default on port 5984, which we have changed in our configuration.

Firewall - from a deployment point of view, we consider it elementary to use firewalls on each server.

2.5 Constraints

According to the assignment definition, our application needs to...

- be distributed.
- consist of at least three tiers.

We were also encouraged to use a non-relational database management system.

All of the above affect the architecture significantly. So does the following details of our product vision:

- Our product is a web application, intended to be accessed via a web browser. The product should be optimized for full-screen display of content.

2.6 API Limitations

Requests to Instagram, Twitter and YouTube APIs represent a central part in most of the use cases described in this document. However, the APIs come with rules and limitations on how to use and render the retrieved data on third-party applications. You can find a list of references to the relevant documents in Appendix.

In this section we briefly discuss the limitations that have been considered relevant in the planning phase and the solutions we undertook during implementation.

2.6.1 Display Requirements

Instagram

No limitations imposed by Instagram regarding display of content.

Twitter

Limitation: Tweets rendered in third-party web applications must comply with the Twitter Display Requirements. In detail, Twitter identifies a minimum set of items (such as date, text and profile picture) and set some rules on how they are displayed.

Solution: The use of so-called embedded tweets would ensure compliance with the display requirements discussed above. However, as our application focuses primarily on the media content of retrieved feeds, we opted for a custom display format that that fit in well with the other social media content. Moreover, the use of embedded tweets is limited to 10 million tweets per day. Although the limit may seem fairly high for our application, the choice of rendering tweets in our own format makes *HashTux* not subject to this kind of limitation.

YouTube

Limitation: *"[...] any YouTube logo used within an application must link back to YouTube content or to a YouTube component of that application."*

Solution: We decided not to use any logo or trademark for rendered YouTube content.

2.6.2 Search Limitations

Instagram

Limitation: Unfortunately, Instagram does not provide a proper search service for non-recent posts (e.g. posts from 3, 4 or more days ago). They make use of pagination and each results page has a unique id that can be used later to retrieve results before or after a certain page id. The problem is that they have no connection between page id and the timeframe that this page represents. For example, if we want to fetch content posted 2 days ago, there is no practical way to know which page holds the results for that time period.

Solution: We decided to exclude Instagram from the history search function. For now we search only Twitter and YouTube for non-recent content.

2.6.3 Confidentiality Limitations

API Authorization Keys for all the involved APIs are subject to security rules. API Keys for *HashTux* are stored on our servers and are not accessible from outside.

2.6.4 Privacy

The content retrieved and displayed on our web page is flagged as public and fulfills the privacy requirements specified in the the APIs Terms of Use (see Appendix).

Chapter 3

Resulting Architecture Overview

We received clear recommendations from our supervisor and others to not use Erlang for serving web pages. Therefore we decided to use a common web server for hosting the web content, and PHP as our (web) server side language. Our solution consists of the Client UI, two server side components, and we also use the CouchDB DBMS.

- Client UI, rendered in a web browser, uses JavaScript
- PHP application, executed on an Apache web server
- Backend server, written in Erlang, connecting to CouchDB and social media APIs
- CouchDB Database (External component)

Our application can be said to use the layered architectural style. Each layer can use the services provided by the layer below. The modules in JavaScript and PHP can be said to make up the UI Layer, while the Erlang code is used both for the Application Layer and the Data Layer.

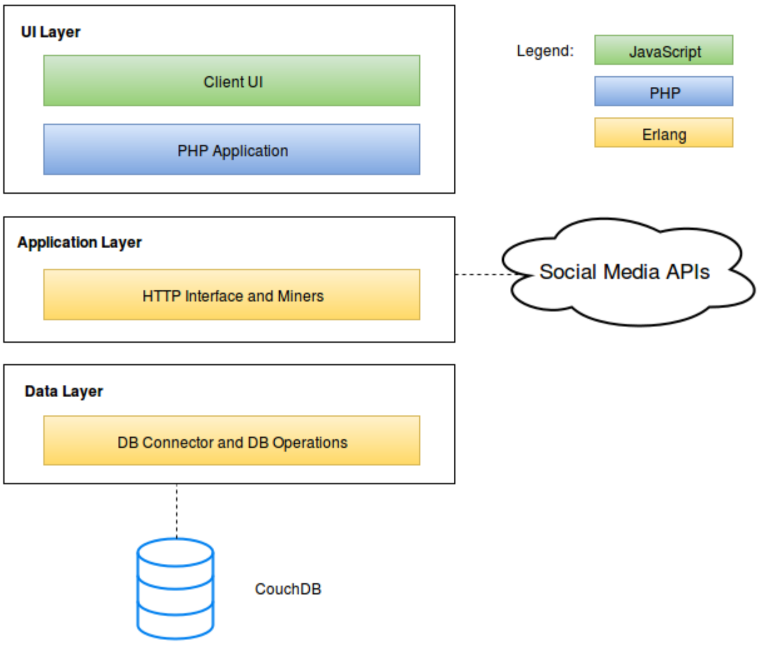

Figure 3.1: HashTux Layer Diagram

Chapter 4

System Characteristics and Challenges

4.1 Characteristics

How *HashTux* relates to the characteristics of distributed systems:

4.1.1 Resource Sharing

Since we dont have any particular scarce resource that we need to share, this is not very relevant to our system. Our components can run anywhere and wherever we choose to run our backend server, it can connect to the external APIs.

4.1.2 Concurrency

In the database component several processes run at the same time: for each request to the database a new worker is spawned and appended to the responsible supervisor. This is done by a dispenser gen_server which returns the reference to the requesting process.

For caching summarizations of the user habit data, we also use a concurrent approach. First, a map function is executed by CouchDB. Later, workers are spawned that run in parallel for reducing the data and saving the cached results it into our database.

4.1.3 Scalability

Regarding scalability, it is easy to add more server tiers if more requests need to be handled. It is a matter of installing any combination of the PHP application (and Apache web server), backend server and CouchDB on the machine and changing the config files.

Right now each of our server tiers hosts runs the whole stack of server side components, but it is also possible to run just one component on each server tier respectively (the PHP application, the backend server and CouchDB).

4.1.4 Fault Tolerance

Right now *HashTux* is deployed on four fully stacked server tiers. Our DNS servers are configured with all the IP addresses of our web servers. All modern browsers select one (which also provides load balancing) and check automatically if the host is available, otherwise they connect to another.

The PHP application tries to connect to the preferred backend server when a request is being processed. If no reply is received within a given time the PHP application connects to another node. Similarly, in the backend server, the database component automatically connects to another CouchDB instance if the primary database is down.

4.1.5 Transparency

The user will never notice if one of the stacks is not currently available because the request will be redirected to one that is running. If an instance of the backend server or CouchDB ever becomes unavailable, the next following search request may take slightly longer to process from a user perspective while the PHP application or backend server notices the timeout and reconfigures, but the user will see a loading bar during this time and we are confident that this can not be distinguished from the normal delays that sometimes occur on the internet.

4.2 Challenges

In this section, we discuss how we have handled or considered the common challenges of distributed system when developing *HashTux*.

4.2.1 Security

To make sure that nobody except a specific user is allowed to read and write from the database, we enabled the HTTP authentication feature in CouchDB, which uses the HTTP header for passing authentication credentials. This is a very basic security mechanism, but enough for our product since our database does not contain sensitive data like login credentials.

We also made sure that we are not using any default config values like the port the database is listening on. This is also important for availability.

4.2.2 Privacy

We do not handle sensitive customer information. However, we collect user habit data and use cookies to do so. We will have a text that clarifies this to the user, that will be accessible from a link in the footer of the start page on the client UI.

Also, see the API Limitations section for how our product is affected by the terms and conditions of the social media APIs we use.

4.2.3 Performance

While developing the module responsible for checking which of the CouchDB instances defined in the config file are available, we encountered a tradeoff between performance and availability.

To check if the databases are available, it would have been necessary to ping the databases very often or test the connection each time it is requested by the connector (a lot of requests are coming in within a short time, which should all be handled correctly). This would have decreased the performance of our system significantly this module would have been a bottleneck for our system.

Because we are using Erlang, this tradeoff could be solved in a nice way: when the supervisor structure of our database component starts up all the CouchDB instances are pinged once to check which ones are available. As soon as one of the databases becomes unavailable, any subsequent request will fail to be handled correctly and the database dispenser server will crash. This causes the whole supervisor to restart, again evaluating which CouchDB instances are available. With this solution, the performance is not decreasing while the system's availability and fault-tolerance is increasing.

4.2.4 Heterogeneity

For this project we used several programming languages (Erlang, JavaScript and PHP). This can provide a challenge for example in terms of communication. For sharing data between our different components, we consistently use JSON and this has been a very effective way to minimize the impact of this challenge.

4.2.5 Reliability

From the development perspective, we have tried to make each of our components as resilient and fault-tolerant as possible. This is also one of the strong sides of Erlang.

From the deployment perspective: right now, our server side components are deployed both at a professional virtual host hotel, and at three hobby server tiers at our homes. The latter ones will probably have a low reliability when everything such as the risk for power outages and consumer ISP reliability is considered.

However, it should be pointed out that we consider overall availability to be more important than reliability of specific components or tiers, which is why we generally have focused a lot on availability tactics in this project.

4.2.6 Availability

This is a very important area for us, and has been discussed extensively in other parts of this document, such as in the Tactics section.

Chapter 5

Use Case View

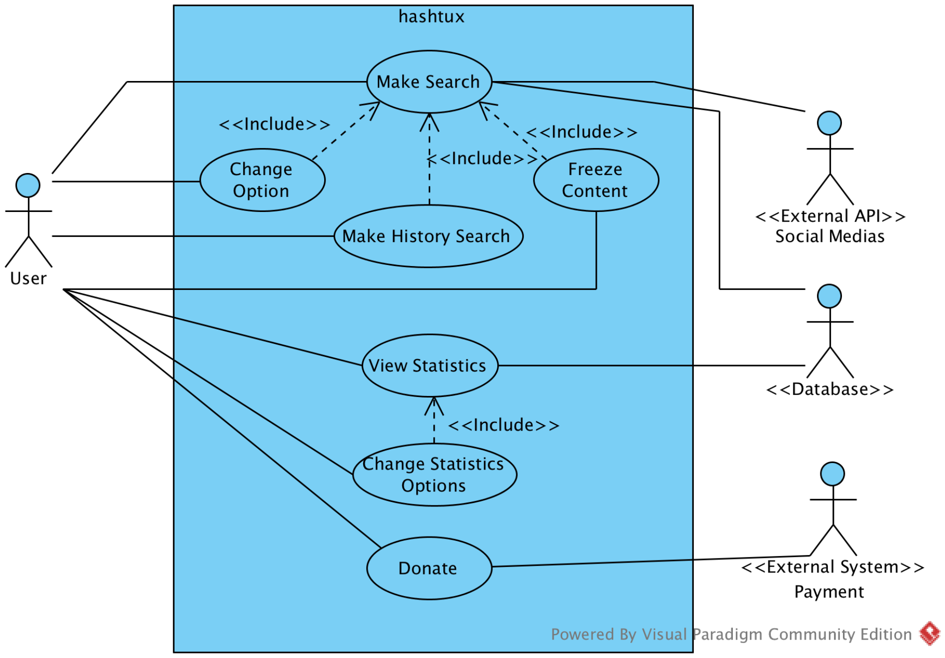

Figure 5.1: HashTux Use Case Diagram

5.1 Make Search

The user wants to search for a term to see content from the different social media services. See the corresponding sequence diagrams in Logical View.

1. Basic Flow
   (a) The user enters a search term.
   (b) The user can see the content.
2. Alternative Flow
   (a) Starts at 1.a.
   (b) There is no content for the given search term.
   (c) The user enters a new search term
   (d) Continues on 1.b.

5.2 Change Options

The user wants to change some options for the site.

1. Basic Flow
   (a) The user selects the options menu.
   (b) The user selects the preferred options.
   (c) The user selects save.
   (d) The user can see the new result.

Precodition:
Make search must have happened.

5.3 Make History Search

The user wants to look at posts from the past.

1. Basic Flow
   (a) The user selects the actions menu.
   (b) The user selects the preferred timeline.
   (c) The user can see the new results.

Precodition:
Make search must have happened.

5.4 Freeze Content

The user wants to freeze (pause) a post.

1. Basic Flow
   (a) The user selects the post's freeze feature.
   (b) The user can see that the post is frozen.
2. Alternative Flow
   (a) The user selects the actions menu.
   (b) The user selects the freeze feature.
   (c) The user can see that the post is frozen.

5.5 View Statistics

The user wants to see user habit statistics.

1. Basic Flow
   (a) The user enters the statistics page.
   (b) The user can see the statistics.
2. Alternative Flow
   (a) Starts after 1.a.
   (b) The user selects the prefered dimension.
   (c) Continues at 1.b.

5.6 Change Statistics Options

The user wants to change the options for how the statistics are shown.

1. Basic Flow
   (a) The user selects the preferred options.
   (b) The user can see the results.

Precondition:
View Statistics must have happened.

Chapter 6

Logical View

Here, we present sequence diagrams showing how different operations take place in the *HashTux* application.

The first 4 sequence diagrams describe the flow for a search request (the make search use case). For the sake of clarity we have made several diagrams, focusing on different parts of the system.

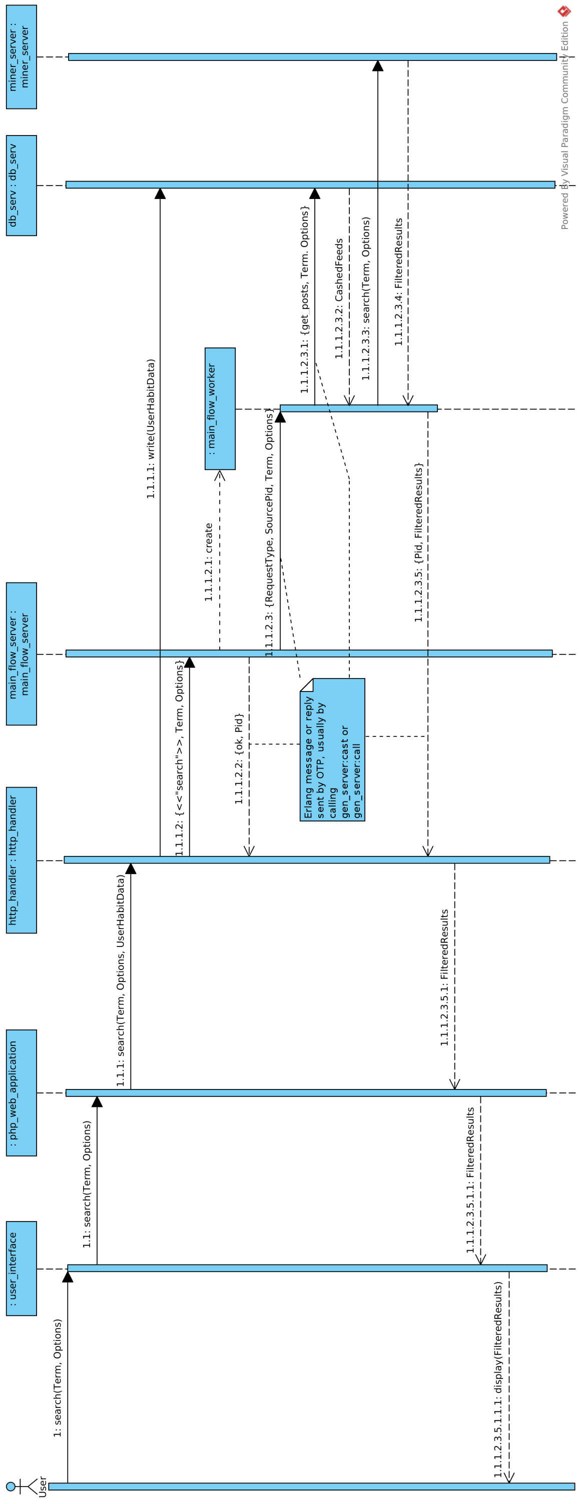
User
: user_interface
: php_web_application
http_handler : http_handler
main_flow_server : main_flow_server
: main_flow_worker
db_serv : db_serv
miner_server : miner_server
1: search(Term, Options)
1.1: search(Term, Options)
1.1.1: search(Term, Options, UserHabitData)
1.1.1.1: write(UserHabitData)
1.1.1.2: {<<"search">>, Term, Options}
1.1.1.2.1: create
1.1.1.2.2: {ok, Pid}
1.1.1.2.3: {RequestType, SourcePid, Term, Options}
1.1.1.2.3.1: {get_posts, Term, Options}
1.1.1.2.3.2: CashedFeeds
1.1.1.2.3.3: search(Term, Options)
1.1.1.2.3.4: FilteredResults
1.1.1.2.3.5: {Pid, FilteredResults}
1.1.1.2.3.5.1: FilteredResults
1.1.1.2.3.5.1.1: FilteredResults
1.1.1.2.3.5.1.1.1: display(FilteredResults)
Erlang message or reply sent by OTP, usually by calling gen_server:cast or gen_server:call
Powered By Visual Paradigm Community Edition

Figure 6.1: Sequence of operations when a user executes a search

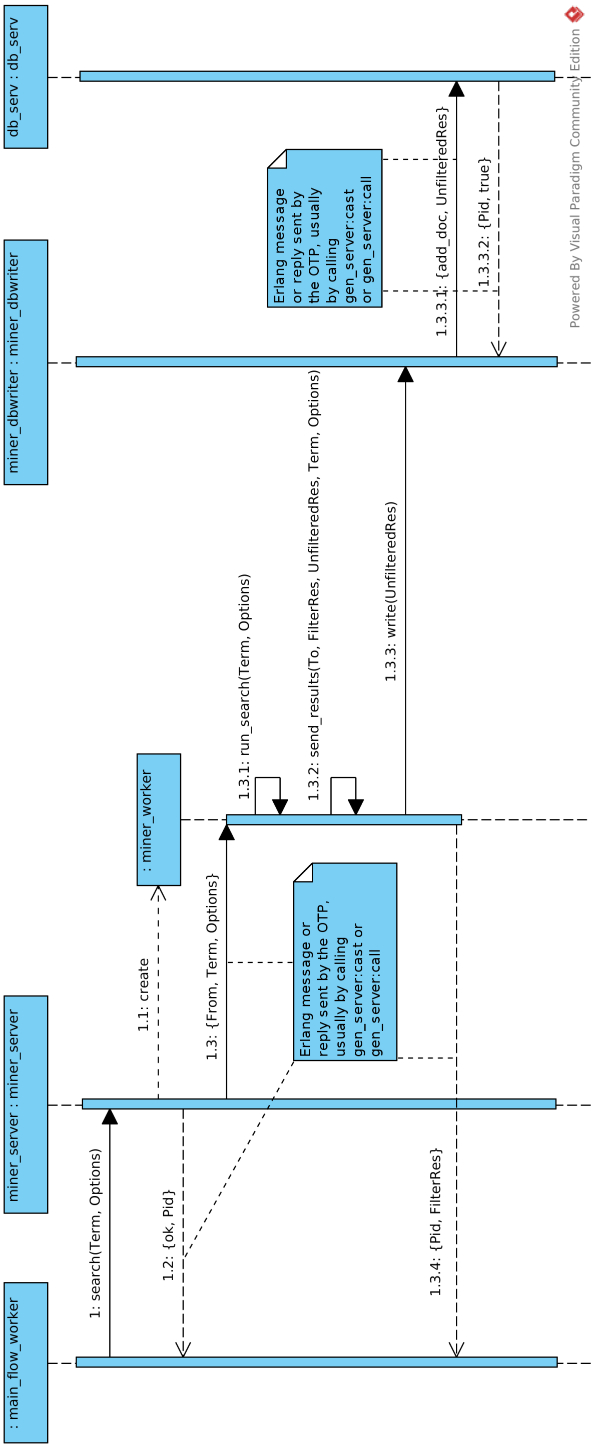
: main_flow_worker
miner_server : miner_server
: miner_worker
miner_dbwriter : miner_dbwriter
db_serv : db_serv
1: search(Term, Options)
1.1: create
1.2: {ok, Pid}
1.3: {From, Term, Options}
1.3.1: run_search(Term, Options)
1.3.2: send_results(To, FilterRes, UnfilteredRes, Term, Options)
1.3.3: write(UnfilteredRes)
1.3.3.1: {add_doc, UnfilteredRes}
1.3.3.2: {Pid, true}
1.3.4: {Pid, FilterRes}
Erlang message or reply sent by the OTP, usually by calling gen_server:cast or gen_server:call
Erlang message or reply sent by the OTP, usually by calling gen_server:cast or gen_server:call
Powered By Visual Paradigm Community Edition

Figure 6.2: Sequence of operations when a search request is received in the miner server

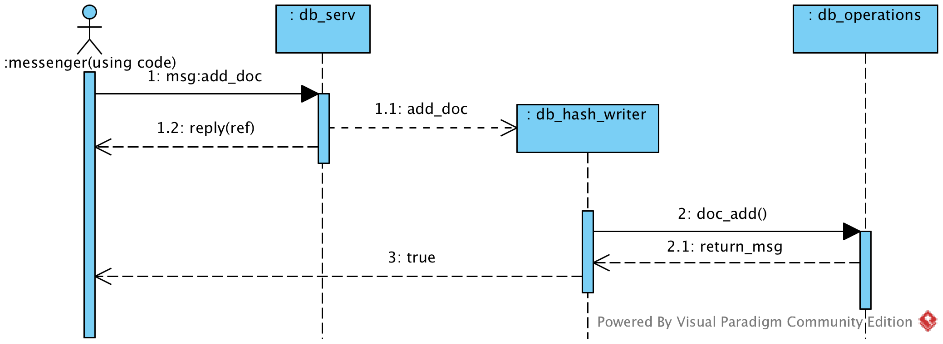

Figure 6.3: Flow of writing something to the HashTux database

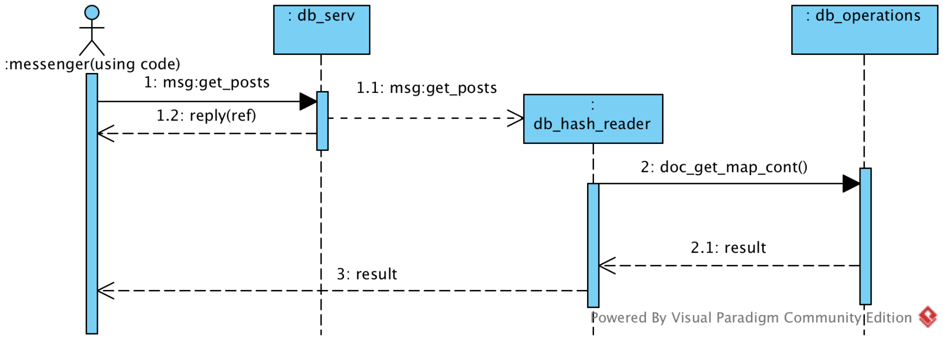

Figure 6.4: Flow of reading the HashTux database to get results based on a search term

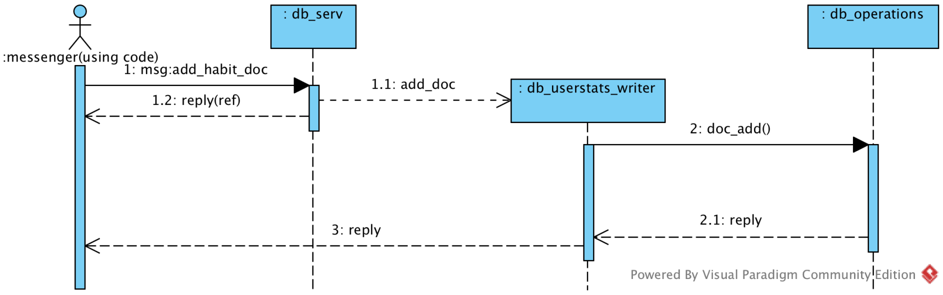

Figure 6.5: Flow of writing something to the hashtux_userstats database

Chapter 7

Deployment View

These are the components we need to deploy:

- Client UI, rendered in a web browser, uses JavaScript
- PHP application, executed on an Apache web server
- Backend server, written in Erlang, connecting to CouchDB and social media APIs
- CouchDB Database (External component)

The three server-side components could run on three different hosts, which would result in a four-tier deployment of our components since the client can also be seen as a tier.

However, we are running the full stack of server-side components on each physical server node (four in total), so the application could actually run even if up to three servers are down at any given moment.

In addition, we interact with external APIs:

- Twitter
- Instagram
- YouTube

So in total, we can say our solution needs at least 5 tiers: Client, Server (PHP application, backend server, CouchDB), and the three external API hosts, but is currently deployed with a redundancy of servers.

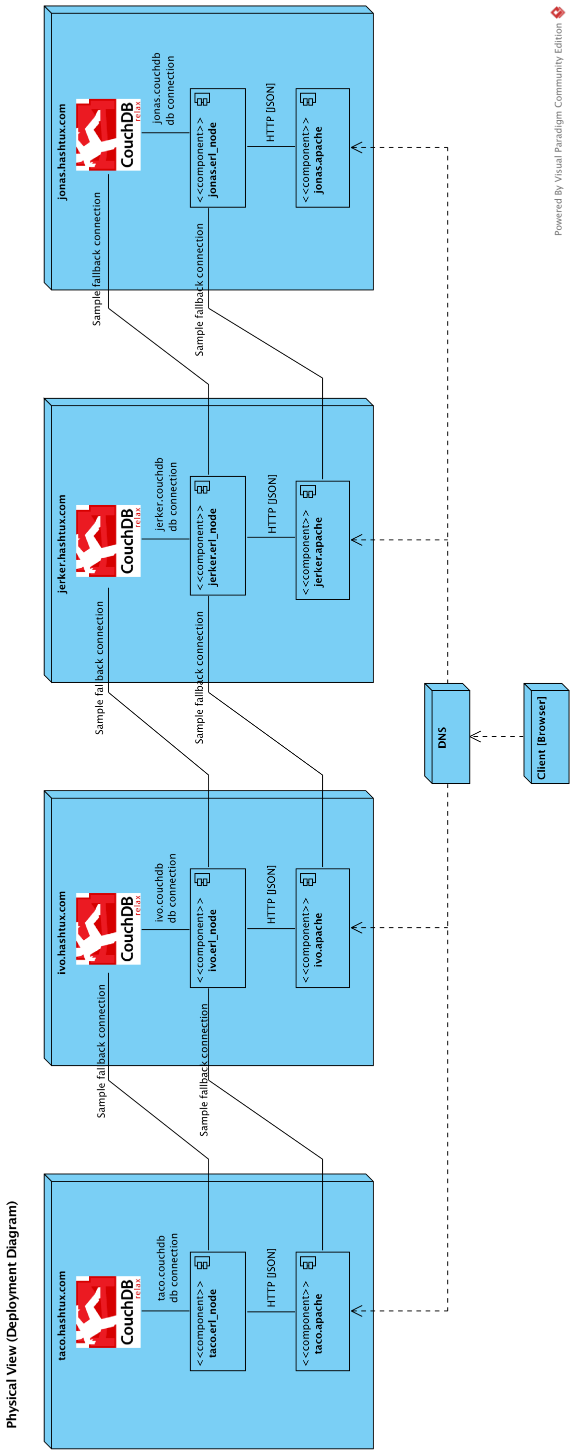
Physical View (Deployment Diagram)
taco.hashtux.com
CouchDB relax
taco.couchdb db connection
<<component>> taco.erl_node
HTTP [JSON]
<<component>> taco.apache
Sample fallback connection
Sample fallback connection
ivo.hashtux.com
CouchDB relax
ivo.couchdb db connection
<<component>> ivo.erl_node
HTTP [JSON]
<<component>> ivo.apache
Sample fallback connection
Sample fallback connection
jerker.hashtux.com
CouchDB relax
jerker.couchdb db connection
<<component>> jerker.erl_node
HTTP [JSON]
<<component>> jerker.apache
Sample fallback connection
Sample fallback connection
jonas.hashtux.com
CouchDB relax
jonas.couchdb db connection
<<component>> jonas.erl_node
HTTP [JSON]
<<component>> jonas.apache
DNS
Client [Browser]
Powered By Visual Paradigm Community Edition

Figure 7.1: HashTux Deployment Diagram

Chapter 8

Data View

This chapter gives a summary of what is stored in the databases of our application.

8.1 Social Media Post Document Format

Stored in the *"hashtux"* database.

Field	Purpose
search_term	The search term.
service	Social media service where it was posted i.e. Twitter, Instagram or YouTube.
insert_timestamp	The time the internal data object was created.
service_id	External item ID, as retrieved from social media service.
timestamp	The time the feed was posted on the social media service.
text	The text content of the post.
language	The language of the post (omitted by Instagram, not supported).
view_count	The view count of the post on the social media (not applicable for Instagram).
likes	Amount of likes retrieved from social media.
tags	A list of tags related to the post.
resource_link_high	URL to high-definition resource (Instagram provides two image resolutions. Where not applicable, put the available resource link in both fields).
resource_link_low	URL to low-definition resource (Instagram provides two image resolutions. Where not applicable, put the available resource link in both fields).
post_url	URL to feed webpage.
content_type	The content type of the post i.e. text, image or video.
free_text_name	Display name if it is separate to the username on that service (currently only applicable to Twitter).
username	The username of the poster.
user_id	User ID as retrieved from social media.

Field	Purpose
profile_link	URL to the user's profile.
profile_image_url	URL to the user's profile picture (only applicable to Twitter).
date_string	Formatted date of when the post was made (only applicable to Twitter).
location	Data is social media service specific and can vary but in general is a tuple of the form {latitude, longitude}. Can include additional information if provided by the social media service.

8.2 User Habit Data Document Format

Stored in the *"hashtux_userstats"* database.

Field	Purpose
term	The search term the habit data belongs to.
timestamp	The time it was collected.
session_id	The session id of the user.
ip_address	The IP address of the user.
browser	The browser that was used.
browser_version	The version of the used browser.
platform	The platform (OS) that was used.

In addition, all the options that were used for the request are also appended when storing user habit data. They may include the following:

Field	Purpose
request_type	The type of request the user made i.e. search, update, heartbeat or stats. Required.
services	Any combination of services i.e. twitter, instagram or youtube (omitted if no services were filtered out).
content_type	Any combination of content types i.e. image, video or text (omitted if no services were filtered out).
language	The language the request was filtered through (omitted if no language was chosen).
histroy_timestamp	The timestamp around which the history search results are centered

8.3 Cached User Habit Data Document Format

Stored in the *"hashtux_userstats_cached_data"* database.

CouchDB is not very efficient when executing MapReduce functions on large amount of data. To increase performance for the statistics page in the client UI, we use the *"hashtux_userstats_cached_data"* where we store documents consisting of map reduced summarizations of the data in our *"hashtux_userstats"* database. The documents contain {key, value} pairs where the key is a string such as a search term, and value is the amount of documents in the *"hashtux_userstats"* database that have that key.

The documents have predefined names that consist of two parts, first the dimension and then the interval e.g. "search_term_year". Each dimension name can be combined with any of the intervals.

Available dimensions: "search_term", "browser", "platform", "platform_browser" (a combination of the fields platform and browser), "browser_version".

Available intervals: "today" (last 24 hours), "week", "month", "year".

Chapter 9

Validation

Throughout the development, we have been testing the user interface and common features of the application manually. In the end of the development process, we were mostly interested in testing our redundancy mechanisms. We divided up the testing into three parts.

- We tried shutting down every web server except one, letting each one be the only available web server at a given time. This was to test how the DNS-based distribution works.

  We found that when we shut down other servers than our first DNS entry, the application was always available. When we shutdown the server at our first DNS entry, it was client-specific whether the client tried to connect to an alternative web server or not.

  We actually expected virtually all clients to handle this better, and were a bit surprised by this. However, we know our main web server is professionally hosted and has an uptime of at least 99%. Within the scope of the project, we will not try to solve this issue. An example of a possible way to handle it would be to run custom DNS servers with a lower "time to live value for all entries, that also regularly evaluates which web servers are available.

  The current DNS solution does however provide a degree of load balancing, as a significant share of clients will connect to other web servers than the first entry.

- We tried shutting down every backend server except one to see how the system behaves and if our availability and performance thresholds hold under unusual circumstances. Our tests proved successful and the system was able to maintain the proposed response times mentioned in the quality requirements. There was no obvious delay visible to the user in serving the search requests.

- We tried shutting down every CouchDB server except one to validate that the backend servers connected to that database. We let each instance be the only one available at a given time.

  We observed that the mechanism worked fine, but the time it took for the backend server to realize several instances were unavailable was unacceptably high. We changed a connection timeout value from 5 to 2.5 seconds as a result of this. It should be noted however that we were testing the extreme case where only one of four servers were available, which is very unlikely to occur.

The summarized result of the testing was positive and we got the chance to test the worst case situation. With some minor changes the system responded properly to what we wanted, except for the client-specific DNS behaviour explained above. In our eyes, the system as a whole can be classified as fault-tolerant based on the results of the testing and with the 4 different physical machines running the system it will have a very high availability.

For the performance aspect, we considered using some sort of automated test for seeing how the system behaves under heavy load, but lacking time near the end of the project, considered this out of scope. Instead, we manually applied as much load as possible by connecting and doing repeated searches simultaneously from all our devices. We noticed no particular delay.

Chapter 10

Tools and Dependencies

10.1 Tools, External Components and Services

Build Tool

We used rebar3 as our build tool. With its help it is easy to fetch the required Erlang dependencies and compile the whole project. It also provides a default folder structure. Another feature that was handy during development is the rebar3 shell which starts an Erlang shell running the project.

Version Control System

Our team has previous experience with Git, so we decided to use it for HashTux. Our Git repository is hosted at GitHub.

Scrum/Sprint Backlog

To have an online scrum/sprint backlog we created a project at pivotaltracker.com.

Database Management

For this project we decided to use a non-relational, document based database. After checking out Riak and CouchDB we came to the conclusion that the installation and maintenance as well as the scalability/clustering features that CouchDB offers suits our needs.

Web Server

We use Apache2, which is one of the most popular web servers around. It offers a lot of modules, features and options. We make use of the **mod_rewrite** module for URL rewrites, and of course **PHP**.

Bug Reporting

Bugzilla is a common tool for bug reporting in software projects. Because we know it is widely used, we wanted to try it out ourselves. Our instance can be accessed at bugzilla.hashtux.com.

Payment

Because we wanted to use a service that is widely used and trusted, we decided to use PayPal. PayPal offers an easy way to create payment buttons for websites.

External APIs

HashTux currently uses the following external APIs:

Twitter Search API
dev.twitter.com

YouTube Data API
developers.google.com

Instagram API
instagram.com/developer

You can find additional information about how hashtux.com interacts with these APIs in the section APIs Limitations and in Appendix.

10.2 Frameworks and Libraries

10.2.1 Erlang

Cowboy

Cowboy is a *"small, fast, modular HTTP server written in Erlang"*. We use it in our backend server to listen for requests from the PHP application.
github.com/ninenines/cowboy

JSX

JSON parser for Erlang.
github.com/talentdeficit/jsx

ibrowse

An Erlang HTTP client used to send requests to the Twitter API.
github.com/cmullaparthi/ibrowse

erlang-oauth

Used for Twitter API authentication.
github.com/tim/erlang-oauth/

10.2.2 PHP

cURL

A PHP library useful for making HTTP requests. Often comes with the PHP installation.
no.php.net/curl

TwitterOAuth

PHP library to connect to Twitter. We use this to be able to fetch trends from Twitter and display them on the first page of our client UI. Sure, it is a bit redundant to connect to Twitter even from PHP, but it is a very special case so we didn't want this feature to complicate our main application.
github.com/ricardoper/TwitterOAuth

PhpUserAgent

A PHP user-agent parser for PHP. We use it to see which browser, platform etc the user has.
github.com/donatj/PhpUserAgent

10.2.3 HTML, CSS and JavaScript

Tweet Linkify

The twitter display guidelines required us to link everything in all tweets to the corresponding page on twitter, i.e. "@hashtux" would lead to @hashtux's profile page. We used this external library for turning all links in the tweet content into usable links.
github.com/terenceponce/jquery.tweet-linkify

Twitter Intents

The display guidelines from twitter also required us to include the ability to perform twitter actions directly from the displayed tweet. We used their recommended *"Web Intents"* for this.
dev.twitter.com/web/intents

DataTables

JavaScript library used to create and edit the tables in the user statistics.
datatables.net

Bootstrap

As a base for the front-end design we decided to use Bootstrap as a framework to minimize the amount of basic CSS design we needed to write ourselves. Bootstrap provides you with a lot of basic design frameworks ranging from pre-designed button to responsive design.
getbootstrap.com

Chapter 11

Future

11.1 Short Term

In the short term we want to introduce more social media services on *HashTux*, for example Tumblr and/or Pinterest. Their content types seem like a good fit for our system. Because we kept the miner modules modifiable it should be easy to integrate those two services.

11.2 Long Term (longer than 1 year)

In the long term (*HashTux v2.0*) we are considering introducing user accounts, which would allow us to create a personal feed for each user. We would allow the users to connect their hashtux.com account with Instagram, Twitter and all the other services we are supporting.

We also would like to introduce Facebook, which is more relevant once there is a personal feed. Another interesting service is Twitch, for streaming video content.

From a technical point of view it may be beneficial to split up the backend server into several independent components communicating with each other over a protocol. This would make our application more distributed and more easy to deploy on more machines. We could also introduce other programming languages than Erlang where suitable.

For improving the daily development workflow, each of those component as well as the website could be in a separate git repository.

Up until now, the product has not been mobile friendly. This is because our current primary target audience would not benefit enough from a mobile friendly version to justify the effort of creating it. However, with the target audience also including the day-to-day social media consumer, the idea of a mobile friendly product becomes a lot more relevant. Preferably, this would be done as a separate mobile application, since mobile applications are generally more popular than mobile-friendly websites for this kind of product, in our eyes.

Appendix A

Reference to APIs Terms of Use

In our implementation we ensured hashtux.com to be compliant with the APIs Terms of Use and Limitations. Any modification or increment to the current implementation should be carried out within the limits defined by the legal documents listed below.

Twitter

[Twitter, Inc, 2015] Terms of Use.
Available at: dev.twitter.com/overview/terms
[Accessed: 16 December 2015]

[Twitter, Inc, 2015] Developer Policy.
Available at: dev.twitter.com/overview/terms/policy
[Accessed: 16 December 2015]

[Twitter, Inc, 2015] Developer Agreement.
Available at: dev.twitter.com/overview/terms/agreement
[Accessed: 16 December 2015]

[Twitter, Inc, 2015] Display Requirements.
Available at: about.twitter.com/company/display-requirements
[Accessed: 16 December 2015]

YouTube

[YouTube, Inc, 2010] Terms of Service.
Available at: www.youtube.com/t/terms
[Accessed: 12 December 2015]

[YouTube, Inc, 2010] Monetization Guidelines.
Available at: developers.google.com/youtube/creating_monetizable_applications
[Accessed: 12 December 2015]

[YouTube, Inc] Google Software Principles
Available at: http://www.google.com/about/company/philosophy/
[Accessed: 12 December 2015]

[YouTube, Inc] Branding Guidelines
Available at: developers.google.com/youtube/branding_guidelines
[Accessed: 12 December 2015]

Instagram

[Instagram, 2015] Platform Policy.
Available at: www.instagram.com/about/legal/terms/api/
[Accessed: 15 December 2015]

[Instagram, 2015] Terms of Use.
Available at: www.instagram.com/about/legal/terms/
[Accessed: 15 December 2015]

[Instagram, 2015] Privacy Policy.
Available at: www.instagram.com/about/legal/privacy/
[Accessed: 15 December 2015]