HASHSCATTER

DIT029 H15 Project: Software Architecture for Distributed Systems

Oscar Bergström
Benedikt Einarsson
Eyþór Atli Einarsson
Tobias Lindell
Kai Salmon
Petronela Petre
Lena Vartanian

INDEX

1.Introduction

This document describes the architectural layout for the Hashscatter system from multiple perspectives. The system's structure and its constraints are described in terms of how it is deployed, how its processes run, and from the perspective of a potential user. This document captures many of the architectural decisions that were made during the system's development, as well as why we made those decisions.

1.2 Purpose

Hashscatter is a useful marketing tool that many companies can use; especially media companies. The Hashscatter web page allows a user to input two different words into text fields. A graph will be produced which shows all the hashtags that appear together with either of the two inputs and to what degree. The companies in question can use this graph to understand the interests of its customer base. For example, if a company inputs topics that their products cover, they can see what other things are tagged when talking about these topics.

1.3 Scope

Although the document mostly shows architectural decisions, it is worth mentioning that the project was developed with a SCRUM development methodology. This means some time will be spent giving an overview of its interaction with the creation of this system, including the obstacles that were created and which had to be overcome with the help of SCRUM.

2. Overview

Use-Case View (Scenarios):
The view goes through the sequence of events that the user experiences and the goal that the user wants out of it. Used to validate the system's architecture.

Logical View:
Through inspecting the domain of the system, the logical view identifies the functional requirements of the system.

Process View:
Mostly focuses on the runtime behaviour of the system. This includes the process it executes, as well as the lines of communication between components.

Deployment View:
The deployment view focuses on the static organization of the software. This includes what hardware we are using and what physical structure the system is in.

3. Architectural Goals and Constraints

The development team met some considerable constraints when using server space. In hindsight, it might have been a good idea to pay a small amount of money for a higher capacity server, but at the time we did not know where to acquire one or how high of a capacity we needed. We decided to get server space on a free server that was owned by a student association (SKIP). This allowed us to move on with development very quickly.

Further into development we noticed that the free server we were using had a much smaller capacity than any normal server you would rent out. First of all, the server only allowed two people to be on it at a time, which meant we had to plan the tasks each sprint in a way that many people didn't need to make changes to the actual server. Additionally, in the late stages of development, we wanted to combine the information we received from the hashtag evaluator and the database by first adding the evaluator results to the database. This wasn't possible in the end because the server couldn't handle that much traffic. This was a huge constraint because we had chosen the noSQL database solely because it could handle a huge amount of traffic.

In most cases, using a web page was an obvious choice for our system. It is very easily deployed, and its capabilities were enough for our core functionality. However, there are some interesting features we could have added if we were using a different medium. For example, it would have been helpful if the points on the scatter graph were interactive to reveal more information. Perhaps they could have been chosen to be viewed as a third axis. The canvas object we used in our script was not capable of these things.

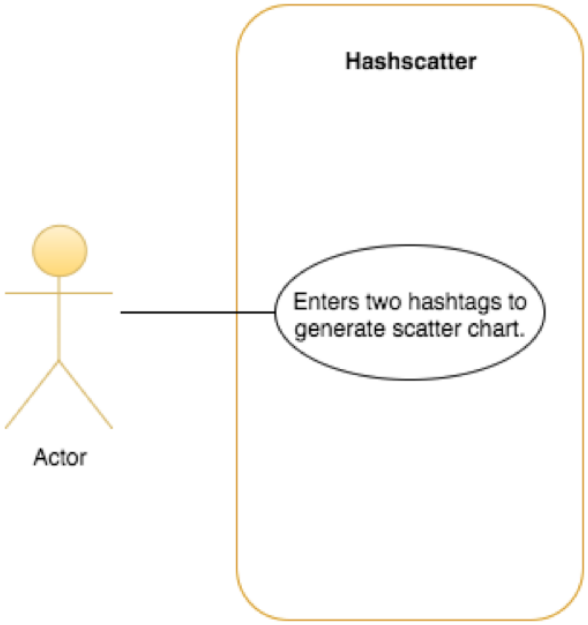
One of the functional requirements we started with needed to be modified because of the restraints on using a web page. If we had used an android application, for example, we would have been able to make the data change dynamically as more live tweets are fed into the database. Additionally, storing local data for individual users became too difficult of a task for the minimal product value it would have given. At one point, we tried to store the data involved in making the graph as cookies for each user, but there was not enough space available to make that feature possible.

4. Use-case View

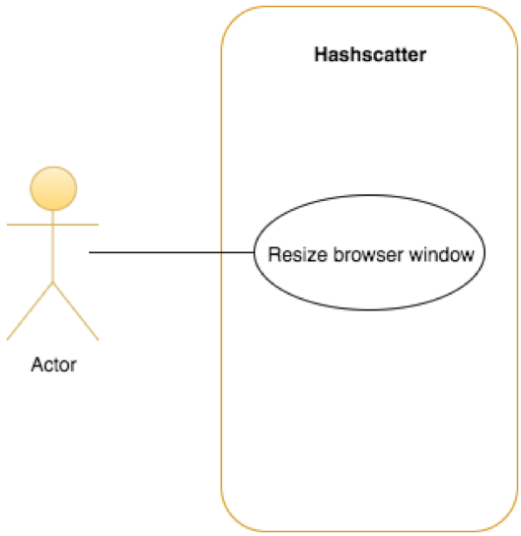
The core purpose of Hashscatter is providing its users with the possibility of creating a scatter graph based on two hashtags. The graph displays correlating hashtags and enables its users to determine which related hashtags are most commonly used among Twitter's users. The frequency of a word's occurrence can be determined based on its position in the chart.

The typical flow of events for a user would be the following:

1. User accesses hal2000.skip.chalmers.se,
2. User enters two hashtags,
3. User generates chart by clicking button "Create Chart",
4. User views generated chart and may use the acknowledged information as desired.

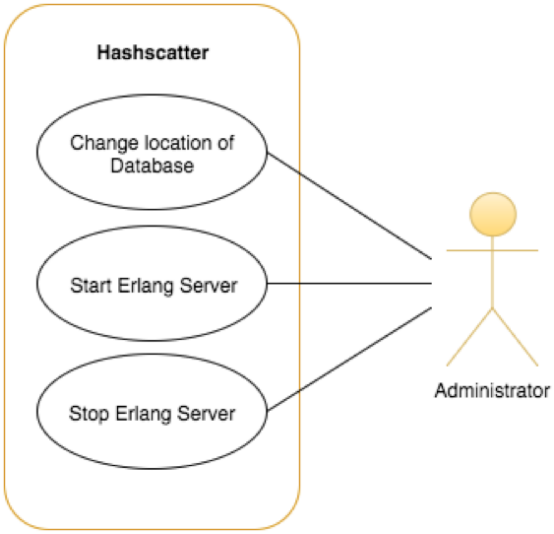


Although it doesn't provide significant change in the website's functionality, users have the possibility resize their browser window. To enable them to view all features in the best, possible way, the front-end has been designed to slightly rearrange when the window is minimized to ensure a user-friendly interface on all devices.



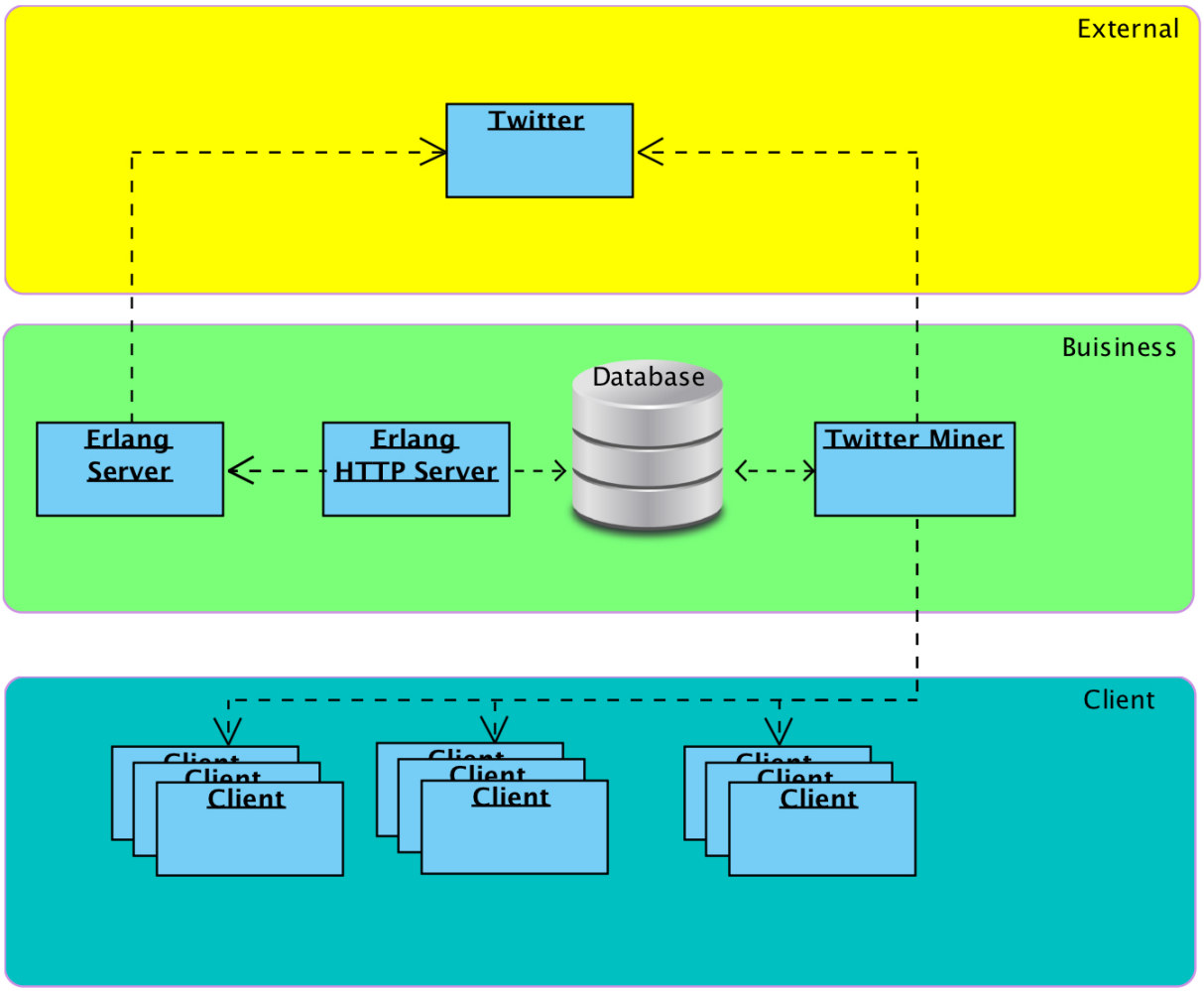
Actors other than the users should be taken into consideration. The system is currently maintained by seven administrators who have access to the server and authority to make significant changes in the system. Changing the location of the database is one major change that can be made by administrators. Moving the database from one physical location to another could easily be done by redirecting where information is fetched from and stored to by changing the url of the location in the code and moving the database to that same location.

An administrator also has the authority to turn the erlserver on and off. When it is switched off, users are disabled from using the functions of Hashscatter. Apart from changes and adjustments in code, these three use cases present the changes with the largest impact on functionality for users.



5. Logical View

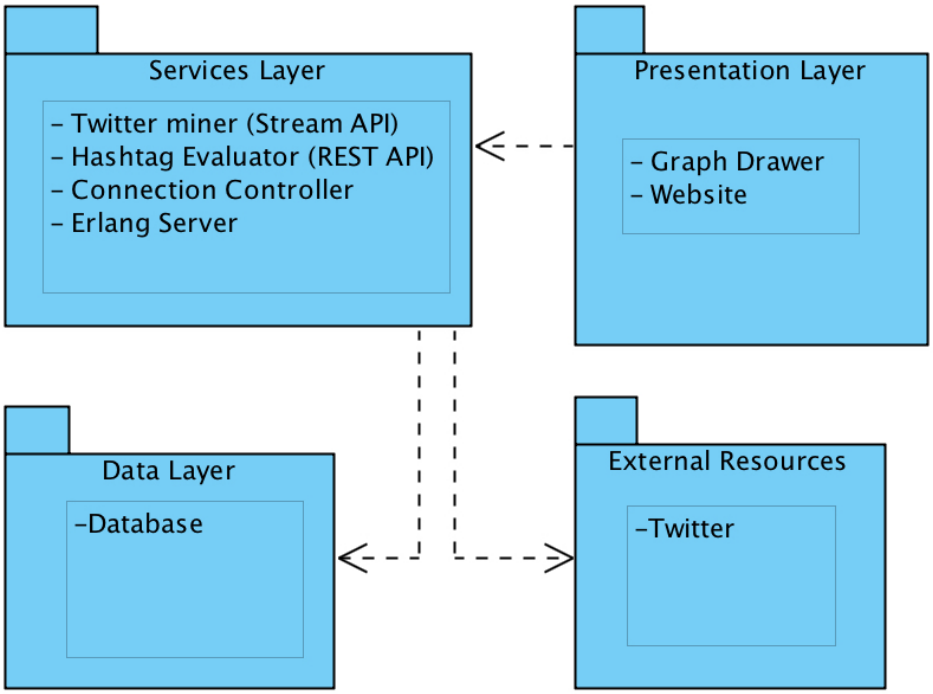
The online hashtag comparison page is layered by basing it on client-server model. The figure below illustrates the model in Tiers.



The layering model of Hashscatter is based on a responsibility layering strategy that associates each layer and components within them with a particular responsibility.

This strategy was chosen because it isolates some of the components responsibilities from one another, which improves both maintenance and development.

Overview



Presentation Layer

- **Graph Drawer**
- **Web Browser**

The Graph Drawer is the module that calculates the results from a search and draws a graph onto the webpage, which is the user interface for the system.

Service Layer

- **Twitter miner (Stream API)** — opens an HTTP-stream to Twitter which runs on the Erlang Server. The module is getting tweets via the stream which then gets filtered.

- **Hashtag Evaluator (REST API)** — queries Twitter's own database. Instead of connecting to Twitters live feed it uses Twitter's own search function and receives the results straight from Twitter.

- **Connection** — handles the connection to the database.

- **Erlang Server** — initiates both the Twitter miner and the Hashtag Evaluator and "talks" to the different components.

Data Layer

- **Database** — Stores hashtag pairs with their frequency.

External Resources
Twitter — an online social networking service that enables users to send and read short 140-character messages called "tweets".

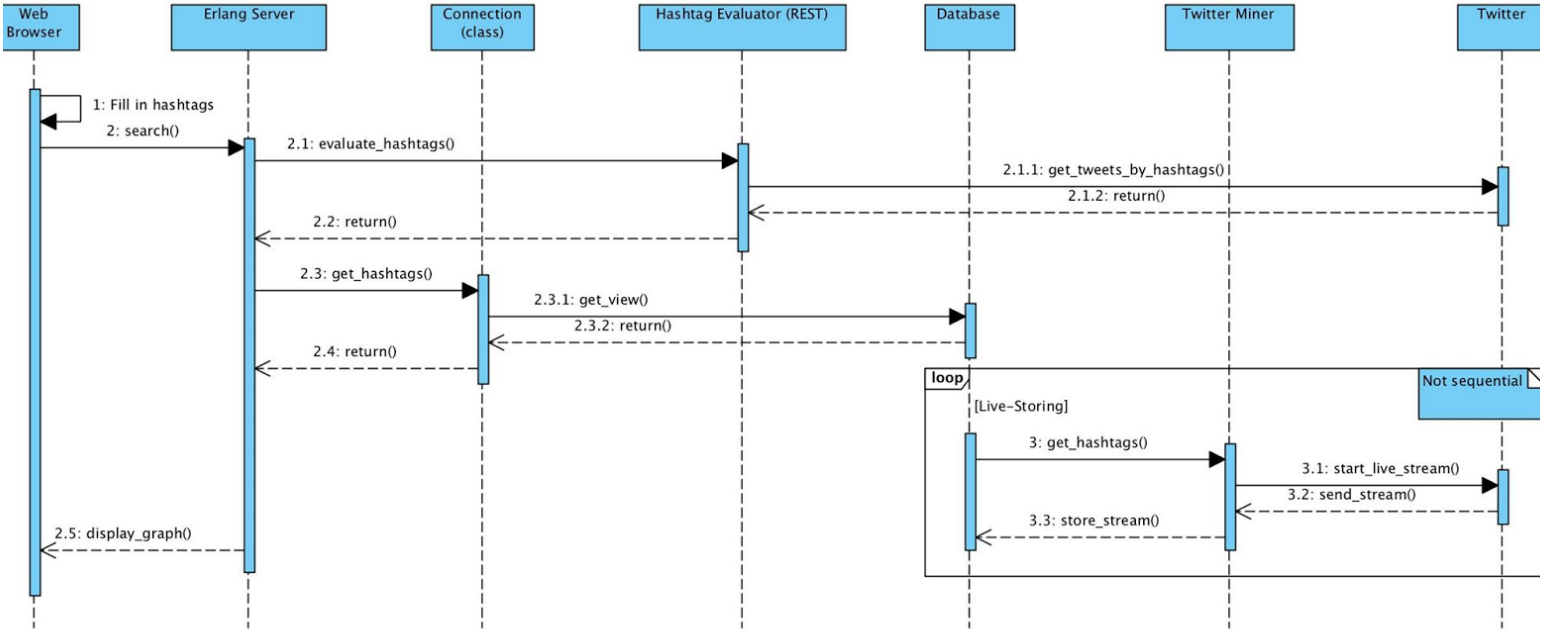
Functional Requirements
The following table identifies and discusses the functional requirements of Hashscatter that have architectural significance.

Requirement	Description	Significance
Search	The system will provide two separate search bars where the user should be able to fill in text and press a button for the site to process search results.	Without this requirement there would be no project.
Display a graphical representation of recurring hashtags.	The system will be able to display a two dimensional graphical representation of how often hashtags are recurring in relation to two specifically selected hashtags.	This is the main-feature of the software and the most important requirement in regard to the system.
Real time update	The system will be able to display real time changes in the graph representing recurring hashtags.	This feature would add value to the end user in means of interaction possibilities.
Switch data source	The system should be able to switch from searching Twitter to searching in its own database.	This feature would ensure availability since it always would have a data source.
Interface for multiple data sources	The system could have an interface so that multiple data sources could be implemented in the future.	This feature would ensure modifiability.

The requirements identified were partially fulfilled. “Search” and “Display a graphical /…/“ represented the main functionality, so naturally we couldn't change them. “Real time update” was one of the requirements that had to be changed. Initially the requirement would add value to the user but as we made progress in the project we found a different solution satisfying our need but not fulfilling the requirement. Having an interface for multiple data sources was initially a requirement that was never fulfilled, the reason being time constraints and lack of necessity.

6. Process View

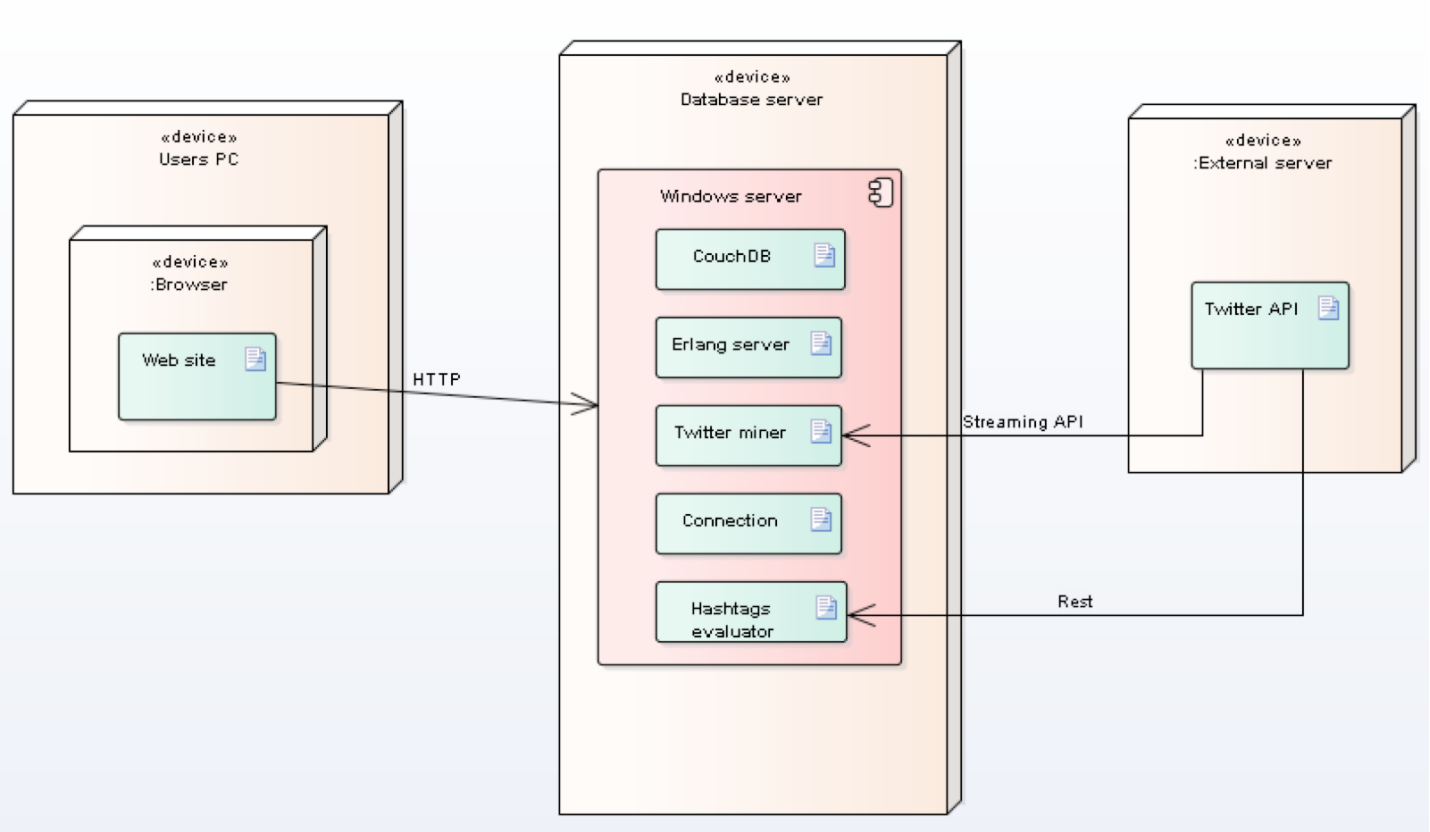
There will be two parallel processes: One which is continually running which adds new tweets to the database as they are submitted to Twitter, and one which fetches the data from both the database and Twitter's servers and displays them visually to use when a user submits a query.



When the user inputs the desired hashtags in the web browser and submits, the browser sends an HTTP GET request to the Erlang Server. The Erlang server then sends a request to Twitter Evaluator class, which forwards that request to Twitter using the REST API. Twitter responds with a sample of tweets that have one of the two desired hashtags (between one hundred and five hundred depending on configuration). These tweets are mapped and reduced into list of hashtags and their frequencies. Next the Server uses the Connection component to access the database, and fetches all of the hashtag-pairs that share one of the two desired hashtags. The two hashtag-frequency lists are then combined into one and returned to the web browser. Finally the browser draws the data in the form of a scatter graph.

7. Deployment View, Size and Performance

Hashscatter is designed as a web page application that is useable on every device with an Internet web browser and internet connection. There are not many devices needed to run the program. A computer or a mobile device connected to the Internet that has a web browser should be able to run the program. The web browser connects to "http://hal2000.skip.chalmers.se" where it's taken to the Hashscatter website. The user then writes in two values on the website which connects to the database, CouchDB, to fetch the data shown on the graph.



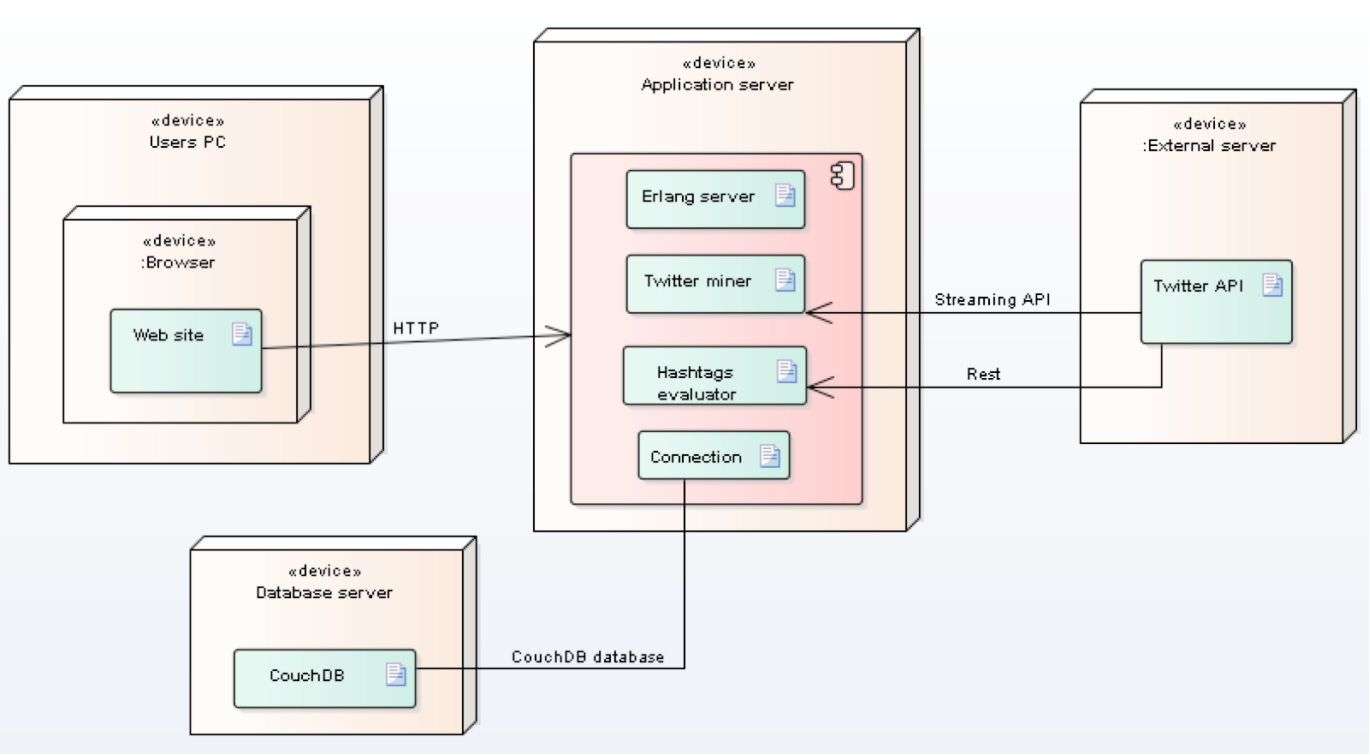
Deployment diagram of Hashscatter.

In the early stages of development, we had all our files stored on different local machines, which made it difficult for us to see the process of other members of the group. When we put it up on the server the work become clearer and from that point the progress was a lot better than before.

We got a server from SKIP (The Association for Informatics and Programming). As members of Göta Studentkår we were eligible for getting the server. Before we acquired that server we temporarily stored on a server that one group member had access to.

We made a decision to use a noSQL database since, based on our research, it should be faster than SQL and could handle a lot more data. We started out with a database called MongoDB and spent a long time trying to get that working but never managed to. We knew some other groups were using CouchDB and had managed to connect it to their server and ended up using that ourselves.

We have a live feed that fetches all hashtags from twitter in real time and when on it slows down our server significantly and makes it problematic working on the server at the same time. Because of that we have had that turned off unless when we were working the actual live feed it or seeing real time results. To resolve this issue somewhat we have made a hypothetical deployment view of how easy it would be to move our database to a different server.



"Hypothetical" deployment view. The database has been moved to its own server.

Having the live feed turned on while someone was working on the server was a problem, because it affected the performance significantly. We also had a problem with accessibility to the server, a restriction on how many users could use it simultaneously and when two admins were on the server and another group member had to get on the server he either kicked someone off or one of the two already on had to sign off.

9. Security

Hashscatter is an interesting system when it comes to security. At the start of development, we had added security in the top five architectural drivers, but in the end we did very little to make the data in our system secure. The only part of our system that needed some security was the source code. We are not running any part of the system on our personal machines. This means all the code that is responsible for what the system does is on the SKIP server. When we allocated server space with SKIP, we received some login usernames as well as passwords. This is the only layer of security our system contains.

If this system was to be shipped as a product on the market, I think it would need to have some higher level of security even though the source code for is the only sensitive data on the server. Perhaps the code could be kept on a separate physical server, which is completely closed off if not accessed directly, while the remote server only has the database on it.

Although we could have separated parts of the system onto different physical servers to improve the security of the source code, this would have opened us up to different security risks. One reason we had for putting it all on the same server space was that if we had the server and the database separate, many of the connections would be packets which are sent through the internet, rather than just accessing methods locally. For example, the data sent between the erlang server and the database would be very susceptible to outside manipulation if they were on separate physical servers.

The database itself does not have any information on it that requires a security layer. At some point we had considered making a login system for the users so their searches could be locally stored. If this were the case, we would need to protect their personal information. More specifically, in our source code, the two developers who were responsible for making the connections with the external Twitter API have their own usernames and passwords with the API. This information can be considered sensitive and is currently in the code unprotected. If further development were made on the system, this data would need to be encrypted somehow. However, as the system is now, the only data that is on the database are hashtag pairs that are publically available from twitter.