System overview/intro

We have designed a clone to the website “Hacker news”, which allows the guests to browse and view posts. As a guest you can’t do that much apart from just browsing, however, if you sign up, various new features will be unlocked to your account.
The features each user will get, is the following: Allowed log in, make new posts and/or comments, upvoting and downvoting each post and comments or flag posts and comments (this feature is to report certain content that doesn’t have its place on the website).
When you register on the website, each user will receive a profile, in which you can browse your “karma” and all the posts and comments the user has made.

System architecture

Our Hackernews clone is hosted in droplets on digitalocean, and consists of three separate docker images running on our *production server*.

Production server applications:

- Image 1, database: A MongoDB instance for communicating with our backend
- Image 2, backend: The backend, running in the Node environment.
- Image 3, frontend: The frontend, written in HTML, CSS and javascript (using various libraries such as: Angular.js, jQuery etc).

The frontend requests data for creating pages from the backend, which exposes its functionality through a series of REST endpoints, such that the frontend and database never have to communicate directly with one another.

Build server

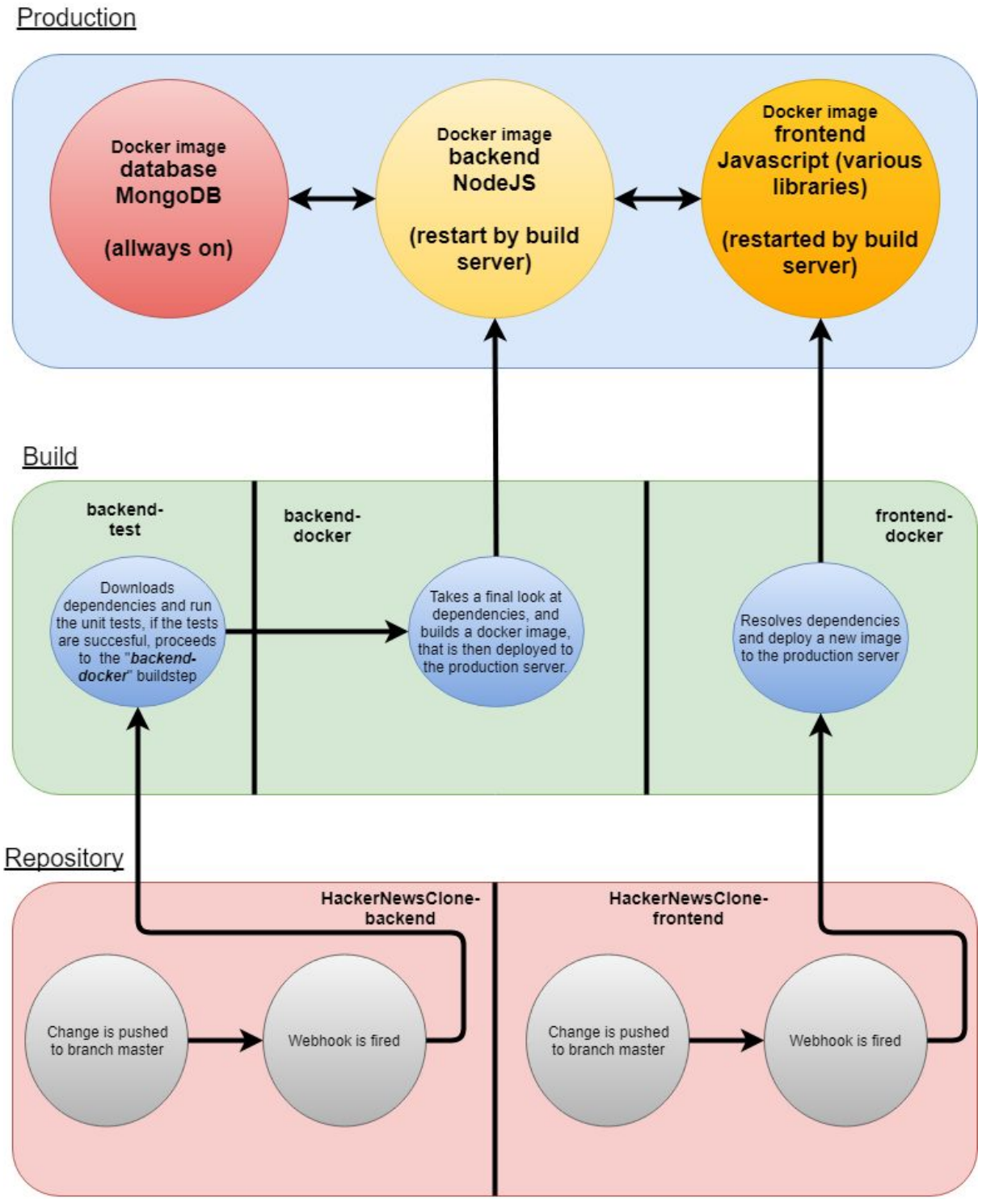
Besides the production server, we also make use of a *build server* in our CI chain.
This build server runs on a droplet (hosted on digital ocean).
The server runs an instance of *Jenkins*, a source automation server that automates the entire deployment process.

Jenkins manages this by listening to two *webhooks,* in two specific github repositories, on two specific branches.
Whenever any change is registered on the given branch, a request is fired, from github, that is picked up by our server, which proceeds to pull the new code, run unit tests, manage dependencies and lastly deploy the new code to the production server.

Code hosting

We use github for all of our code hosting needs. We've split up the project such that the frontend- and backend code resides in different repositories.

Whenever the *master branch* in any of the two aforementioned repositories is updated, the build server will automatically update the production server with the new changes.



The above diagram describes the entire system
and the CI chain that keeps it running.

Information

Github:

Frontend code is hosted on: https://github.com/ElDuderino420/HackerNewsClone-frontend

Backend code is hosted on: https://github.com/ElDuderino420/HackerNewsClone-backend

Digital ocean

Our digital ocean droplets have the following information:

Jenkins droplet: 146.185.172.211:
- Automatic Builds and Tests

Production droplet: 188.226.152.93:
- Docker containers:
- Nginx: port 80
- Frontend
- Nginx: port 8080
- Load Balancer
- Backend: port 3000 (3030)
- Uptime monitor: port 3001

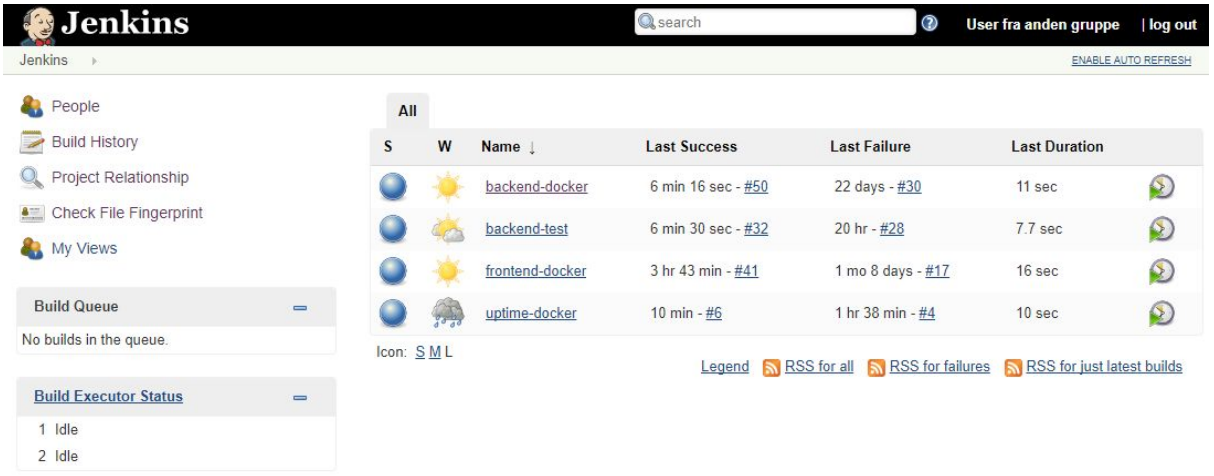
How to access the various machines, with which credentials, keys, etc and how to restart components?

Jenkins user:

We have added a user on our Jenkins server:
http://146.185.172.211:8080/login?from=%2F

1. username: user
2. password: user

With it you can login to the jenkins server and see if the backend, frontend and the status is online, and if they are not you can schedule a build for it. you can also see the build in the console for each of them.
Though it should be noted that this is the only thing you can do with the user



How to get information about the inner state of your system, such as databases, log files, etc.

In Jenkins we have set up a project that you can run to check the docker logs for each of our running containers on the production server.

To see the log, run the get_docker_logs task and read the console output.

Dataflow within your system, within and across components.

Sequence diagram

We have made a sequence diagram documenting how the different activities gets processed both on the frontend and the backend.
https://go.gliffy.com/go/share/s5ei0aw20l2mwrv20qjp
The picture is quite big so it can be viewed in better quality through the link above.
It documents the scenario of:

1. Browsing the home page
2. Registering
3. Post a story
4. Comment
5. Upvote
6. Downvote
7. Flag post
8. Log out
9. Getting a loan via loanbroker

This scenario takes the user through almost all features of the site. excluding viewing ones profile page, viewing the list of users, viewing the highest rated stories.
The last two can be done though the home page, where one can choose what to view.

Code run through

Rest api og docs (ligger på github)

You can find the api calls on the following link:
https://github.com/ElDuderino420/HackerNewsClone-backend/blob/master/HackerNewsApiDocumentation_V3.pdf

Backend

Database

This module consists of nothing but an instance of MongoDB running in a separate docker container. We connect the server to the Mongo container by finding it's IP with the command:

"docker inspect $CID | grep IPAddress | cut -d '"' -f 4"

Where $CID represents the container ID obtained through the command:

"docker ps"

Besides the above, nothing really exciting is going on in this part of the system.

Server

This module consists of a Node.js/Express http server, that contains all our API endpoints.

```
191  ////////////////////////////////////////////////////////////////////////////////
192  //                  Retrieves all posts made by the given user
193  // Takes a user name as a request param
194  // Responds with either 888 (no posts found) or 200 and the posts
195  ////////////////////////////////////////////////////////////////////////////////
196  router.get("/all/:userName", function (req, res, next)
197  {
198
199      if (typeValidator.isString(req.params.userName))
200      {
201          postFacade.getAllPostsByUserName(req.params.userName, function (posts)
202          {
203              if (posts === false)
204              {
205                  res.status(888).send();
206              }
207              else
208              {
209                  res.end(JSON.stringify(posts));
210              }
211          });
212      }
213      else
214      {
215          res.status(400).send();
216      }
217  });
```

An example of one of our endpoints in the API.

To make for seamless integration between the API and the frontend (that consumes the responses from the API), we use a combination of two strategies. Firstly we make sure to heavily comment on all the methods in our API. Below is an example of this practice

```
219  ////////////////////////////////////////////////////////////////////////////////////////////
220  //                  Edits a post
221  //  Takes the hanesstId of the post in question, a new text and a new post title.
222  /*
223  {
224      "hanesst_id": 1,
225      "new_post_title": null, //Den er den samme
226      "new_post_text": "jeg er ændret" //denne er ændret
227  }
228   */
229  // BEMÆRK AT UÆNDREDE FIELDS SKAL SÆTTES TIL NULL (se above example)
230  // responderer med 888, hvis den ikke kan finde posten eller 200
231  ////////////////////////////////////////////////////////////////////////////////////////////
232
233  router.put("/edit", function (req, res, next)
234  {...});
254
255
256  ////////////////////////////////////////////////////////////////////////////////////////////
257  //                  Upvotes the post with the given ID
258  // Takes the ID of the post to be upvoted.
259  // Responds with either 888 (if no post found with given ID) or 200
260  ////////////////////////////////////////////////////////////////////////////////////////////
261  router.put("/upvote", function (req, res, next)
262  {...});
283
284  ////////////////////////////////////////////////////////////////////////////////////////////
285  //                  Downvotes the post with the given ID
286  // Takes the ID of the post to be downvoted.
287  // Responds with either 888 (if no post found with given ID) or 200
288  ////////////////////////////////////////////////////////////////////////////////////////////
289  router.put("/downvote", function (req, res, next)
290  {...});
310
311
312  ////////////////////////////////////////////////////////////////////////////////////////////
313  //                  Flags the post with the given ID
314  // Takes the ID of the post to be flagged.
315  // Responds with either 888 (if no post found with given ID) or 200
316  ////////////////////////////////////////////////////////////////////////////////////////////
317  router.put("/flag", function (req, res, next)
318  {...});
338
339  ////////////////////////////////////////////////////////////////////////////////////////////
340  //                  Gets the total karma (upvotes - downvotes) for the post with the given ID
341  // Takes the ID of the post in question.
342  // Responds with either 888 (if no post found with given ID) or 200 and the total karma
343  // the response looks as follows:
344  /*
345      {
```

Secondly we make use of a continuously updated document, that resides in the git repositories root folder. This document, called “HackerNewsApiDocumentation” contains documentation and examples for every single endpoint in the API. Here follows an example:

Description	Gets the karma of a specific user
REQUEST TYPE	GET
PATH	/api/user/karma/:**username**
REQUEST BODY	No body, just the request/url param
RESPONSE BODY	Status 200 {"totalKarma":2} **OR** Status 777

POST ENDPOINTS

Description	Get the post matching the given ID
REQUEST TYPE	GET
PATH	/api/post/: **hanesstId**
REQUEST BODY	No body, just the url param

RESPONSE BODY	Status 200
	{
	"_id": "59e2571e0a83ac2e889c58f9",
	"total_score": 0,
	"created_at": "2017-10-14T18:27:42.817Z",
	"updated_at": "2017-10-14T18:27:42.817Z",
	"user_name": "asdf",
	"post_type": "story",
	"post_title": "Jeg kan godt lide Lamaer2222",
	"post_parent": 0,
	"hanesst_id": 65,
	"post_text": "Cafeteria Fraiche er guddommeligt2222",
	"post_upvotes": 0,
	"post_downvotes": 0,
	"post_flagged": 0,
	"__v": 0
	}
	OR
	Status 888

Together the above practices assures we always have up-to-date and easily accessible documentation at hand for developing up against the REST API.

Frontend

Description of Modules

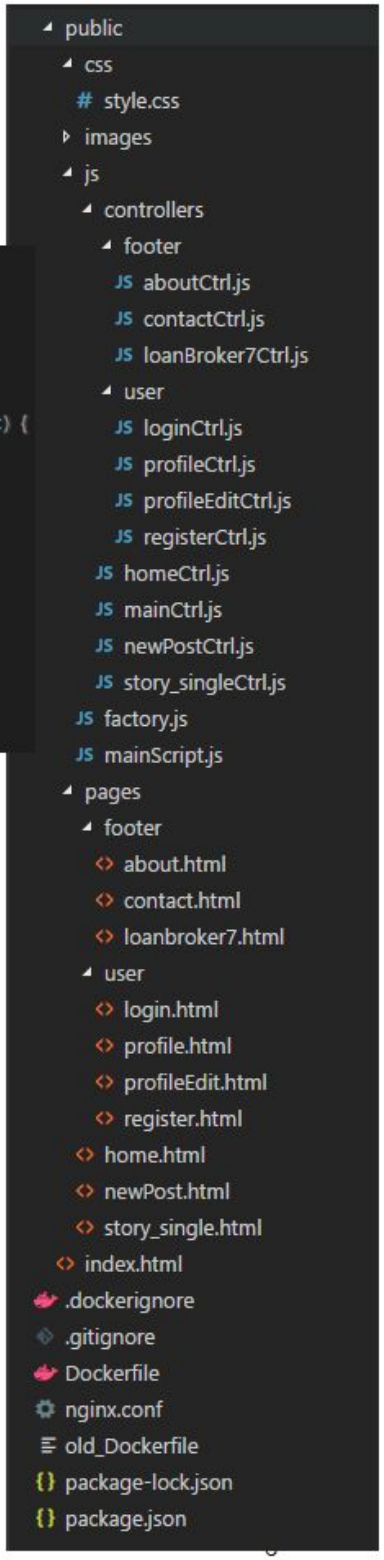
The frontend is made with AngularJS, where each page on the site has its own controller which gets the necessary information from the backend to work. The controllers get this information via the AuthService factory, below you can see an example of the register rest call:

```
angular.module('haxorNews')
    .factory('AuthService', function (API_ENDPOINT, $http, $location) {
        var currentUser = {};
        return {
            register: function (user, callback) {

                $http.post(API_ENDPOINT.url + "/api/user/new", user).then(function (result) {
                    if (result.data != null) {
                        //console.log("yay")
                        callback(result);
                    }
                }, function (err) {
                    if (err != null) {
                        //console.log("nooo")
                        callback(err);
                    }
                })

            },
```

So in a controller all one has to do is use the AuthService and use this method, supply it with a user and a callback function and its good to go.



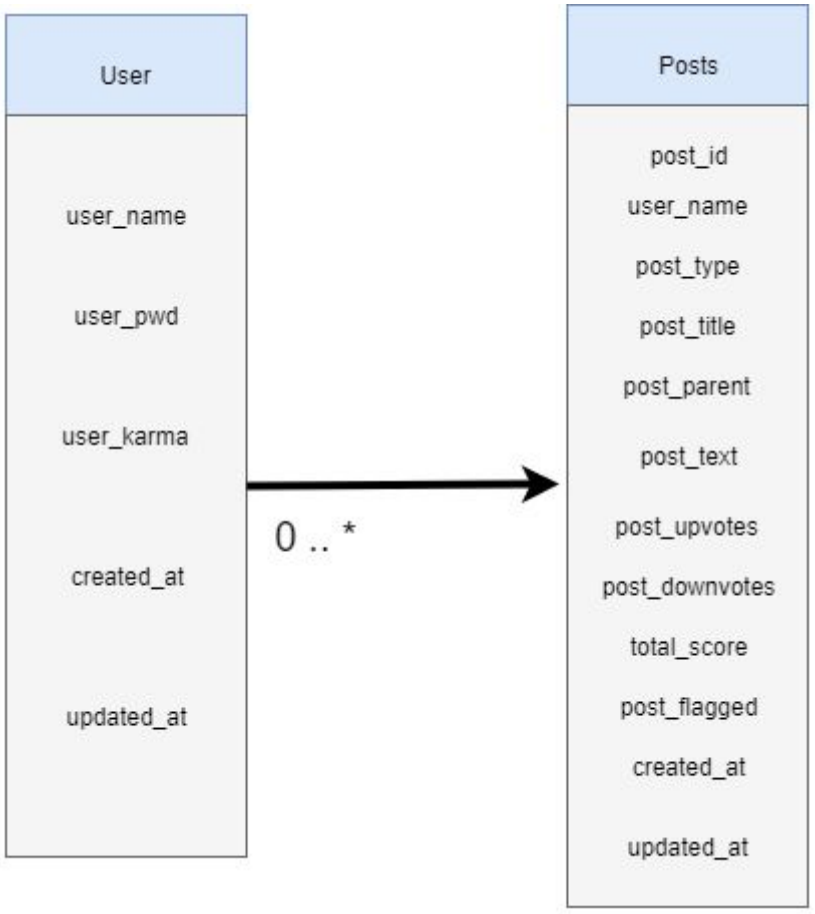
Nginx

This is a load balancing feature which is used to have multiple instances of backend running so that even if one goes down the site continuous. As we are using continuous integration with Jenkins, this also means that when updating the backend, it is down sequentially. So that one of the the backend instances also operates, while the other is updating.

```
events {
}
http {
    include       /etc/nginx/mime.types;
    upstream hackernews {
        server 188.226.152.93:3000;
        server 188.226.152.93:3030;
    }
    server {
        listen 8080;
        #server_name 188.226.152.93;
        server_name localhost;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $remote_addr;
        location /status {
            proxy_pass http://188.226.152.93:3001;
        }
        location /loanrequest {
            proxy_pass http://188.226.152.93:3002;
        }
        location / {
            proxy_pass http://hackernews;
        }
    }
    server {
        listen 80;
        root /usr/share/nginx/html;
        index index.html index.htm;
        #server_name 188.226.152.93;
        server_name localhost;
        location ~ ^/(images|javascript|js|css|flash|media|static)/  {
            expires 30d;
        }
        location / {
            #expires -1;
            #add_header Pragma "no-cache";
            #add_header Cache-Control "no-store, no-cache,
            #must-revalidate, post-check=0, pre-check=0";
            try_files $uri$args $uri$args/ $uri $uri/ /index.html =404;
        }
    }
}
```

Logical Data Model

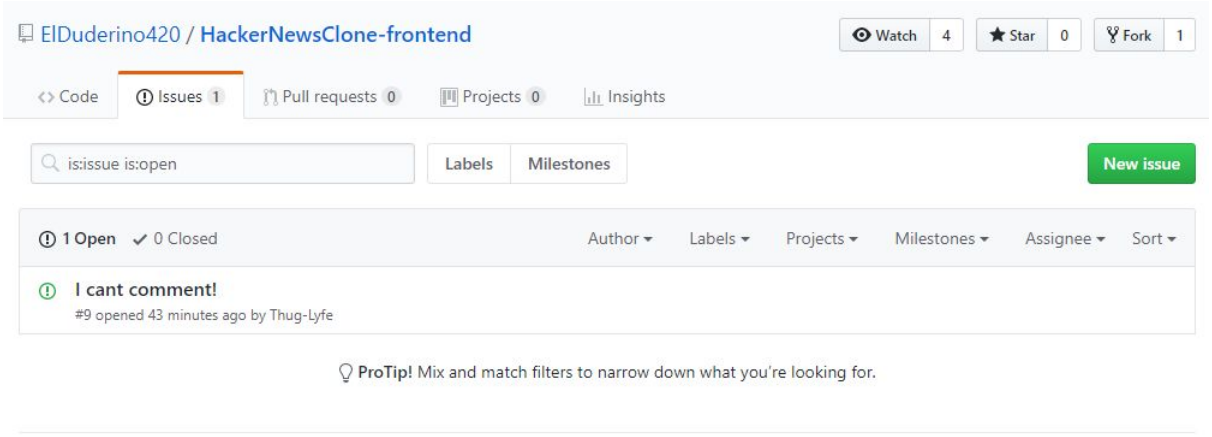
The above is our logical data model. It deviates quite a bit from the usual SQL based models. Since we use Mongo DB we have seen no sense in dividing the model into more than two parts.



How to file bug reports and issues?

Brug af github issues.

To report an issue to the developers you have to have an account on the github.com, when you have this, you can go to either
https://github.com/ElDuderino420/HackerNewsClone-frontend/issues or
https://github.com/ElDuderino420/HackerNewsClone-backend/issues
to report an issue regarding the frontend or the backend respectively.
on those pages you will see something similar to this:



Here you can click the green button labelled “New Issue” and get redirected to a form where the the issue at hand can be documented and the developers will be notified.