Ontwikkeldocument-lite.

# Onderzoeksfase.

## Casus.

Bouw een headless RESTful api met toegang tot een database. Creëer een Client die secure kan communiceren met de database. De client kan een gebruiker inloggen, en een gebruiker kan records bekijken en aanmaken.

## Requirements:

Must haves:

* Als gebruiker wil ik een lijst van contacten kunnen inzien in een web applicatie.
* Als gebruiker wil ik de lijst met contacten kunnen bewerken.
* Als gebruiker wil ik de lijst met contacten kunnen verwijderen.
* Als opdrachtgever wil ik een API back-end zodat de client data kan opvragen.
* Als opdrachtgever wil ik een authenticatie systeem zodat dat mijn klanten secure kunnen inloggen.
* Als opdracht gever wil ik dat de client scaleable is zodat deze in de toekomst makkelijk uitgebreid kan worden.

Should haves:

* Als opdrachtgever wil ik dat de API secure communiceert met de database, om datalekken te voorkomen.
* Als opdrachtgever wil ik dat de database scaleable is zodat deze in de toekomst makkelijk uitgebreid kan worden.

Could haves:

* Als gebruiker wil ik dat de web applicatie er leuk uit ziet, zodat het plezierig is om mee te werken
* Als gebruiker wil ik een zoek functie om snel bij een bepaald contact te kunnen komen.

Won’t haves:

* Als gebruiker wil ik verschillende categorieën aan contacten kunnen toevoegen, zodat mijn contacten boek goed georganiseerd blijft

## Technieken:

### MongoDB

Ik heb gekozen voor mongoDB omdat het *document-model* een krachtige manier is om data op te slaan en op te halen. Ook is het opzetten erg snel en kan mongDB goed horizontaal schalen, ook support mongoDB grote volumes aan object en traffic.

### NodeJS

NodeJS is ontworpen om real-time data te versturen, wijzigen en op te halen. Ook is het een pre dat het in javascript is geschreven en ik zo hete hele project hierin kan schrijven.

React  
React is een Javascript framework en leent zich goed voor applicaties die veel data moeten verwerken. Ook is het in react mogelijk om objecten te voorzien van nieuwe data zonder de pagina te moeten refreshen. React is snel, schaalbaar en de basis is makkelijk.  
Ook kan er, mocht dit nodig zijn gebruik worden gemaakt van Redux en wordt er een MVC opgezet of gekozen voor Onion programming.

### PassportJS

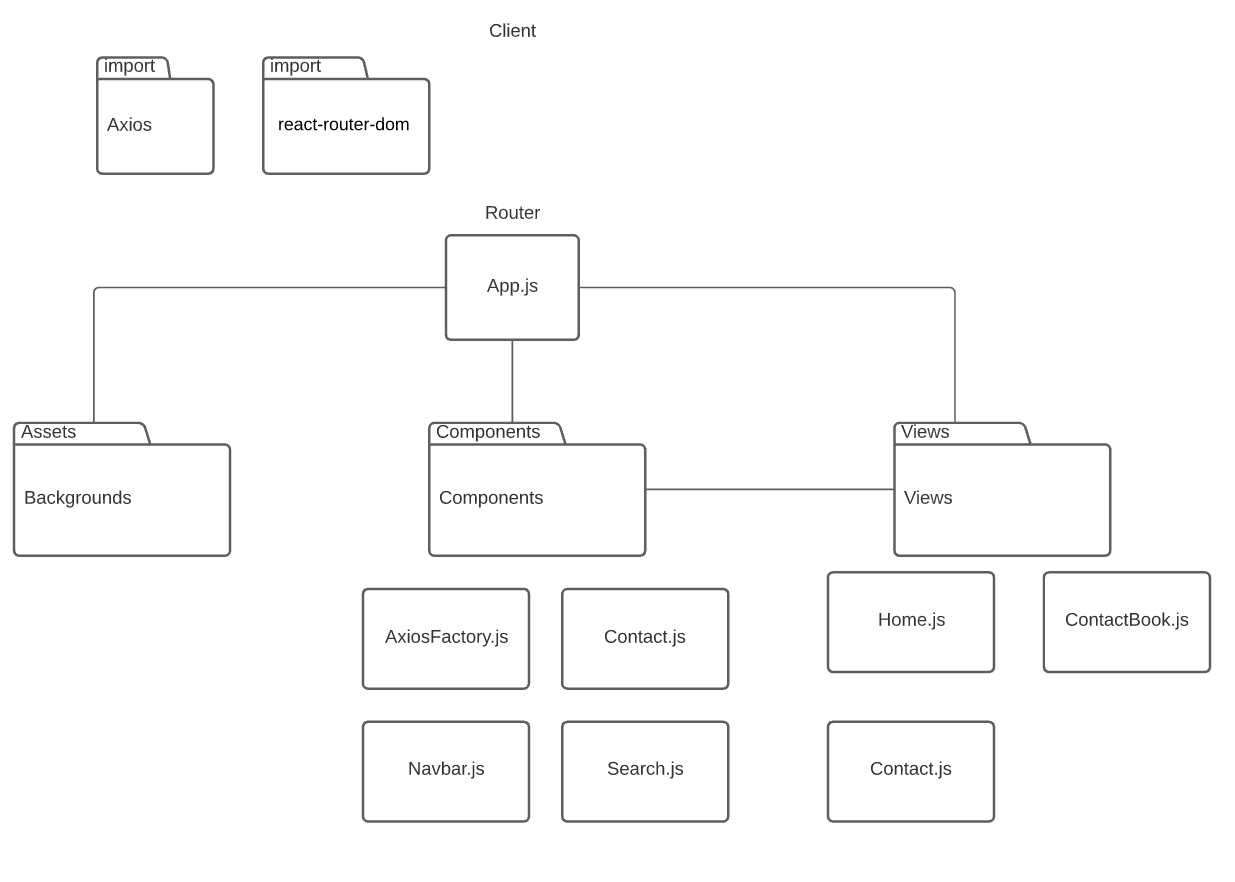
PassportJS is een flexibel en modulair. Het kan dus makkelijk in een bestaande Api gedropt worden. Er zijn meerdere strategieën waarmee passportJS kan werken: zo kun je een username en password strategie opzetten of juist gebruik maken van de strategieën van bijvoorbeeld google.com of facebook.com. Dit zorgt er voor dat de gebruiker zelfs zonder een account aan te maken direct gebruik kan maken van jou app door in te loggen met de *provider* van hun keuze.

### Axios

Ik heb er voor gekozen om Axios te gebruiken ipv node-fetch, omdat axios gewoonweg beter is.   
Het is sneller, minder zwaar, meer secure en minder code. Zie [HTTP Requests Compared: Why Axios Is Better Than Node-Fetch (Automatic Transformations, More Secure, Can Handle Errors Better, Interceptor Support, And More Browser Friendly) | by Jeff Lewis | Medium](https://medium.com/@jeffrey.allen.lewis/http-requests-compared-why-axios-is-better-than-node-fetch-more-secure-can-handle-errors-better-39fde869a4a6)

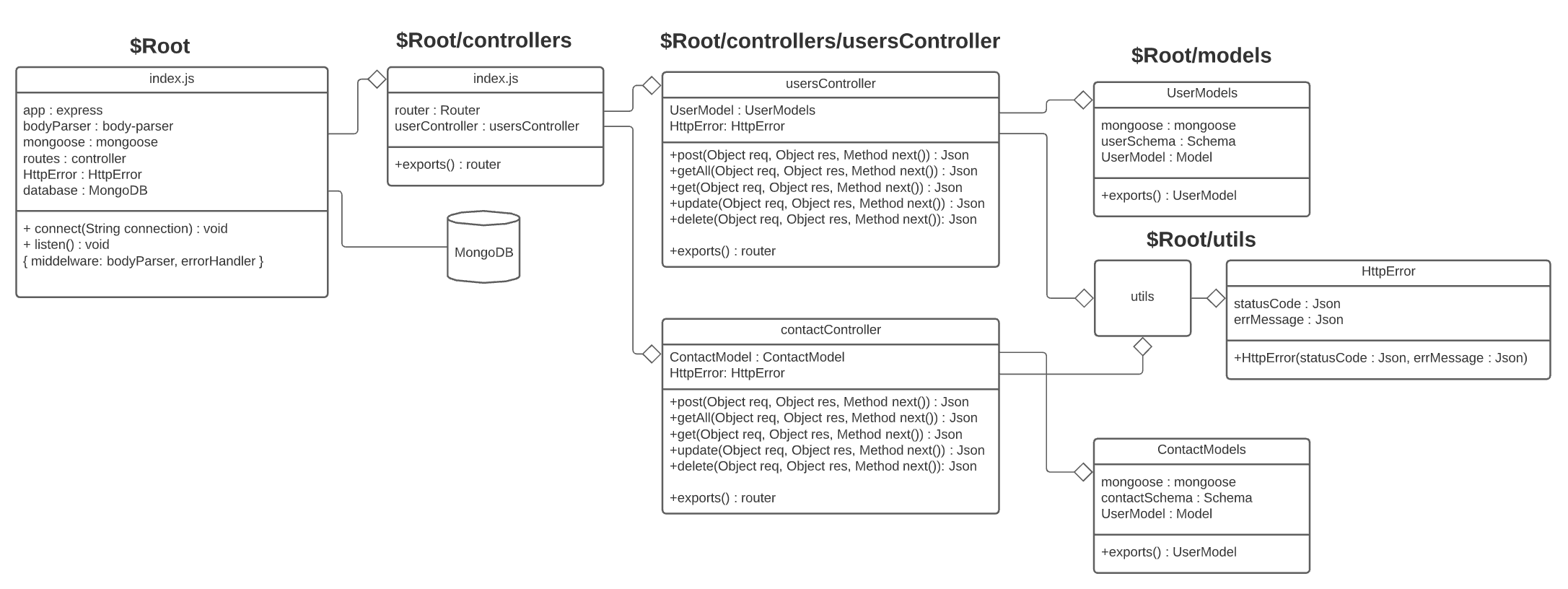
# Ontwerpfase.

## Client



Om niet te veel tijd kwijt te zijn heb ik het ontwerp van de client vrij simpel gehouden.  
Wel heb ik bij het ontwerpen gelet op de verantwoordelijkheden en gekeken welke logica in componenten verwerkt kon worden.

## Api



Het diagram van de Api heb ik redelijk nauwkeurig uitgewerkt. Dit is de eerste keer geweest dat ik een Api heb opgezet en ik heb kort onderzoek gedaan naar best practise.  
Zo heb ik er voor gekozen om gebruik te maken van een controller die requests per route doorstuurt naar de juiste controller.  
Ik heb gebruik gemaakt van modellen en schema’s om de objecten in de database een format te geven. Door het gebruik van deze modellen kan er niet afgeweken worden van de datastructuur. Dit is met name belangrijk in een NoSql database..

Ook heb ik er voor gekozen om een HttpError object te maken waaraan een status code en een bericht kunnen worden meegegeven. Een voordeel hiervan is natuurlijk dat er minder code herhaalt wordt. Ook kan wordt er nu in Root/index.js gecontroleerd of een error een instantie is van HttpError voordat deze verstuurd wordt naar de user. En zo is er een extra laag gecreëerd tussen de gebruiker en de werking van de api.

## Login

Ik heb er voor gekozen om de Login niet te implementeren. Dit omdat het niet haalbaar is binnen de 8 uur die gesteld is. Wel heb ik, omdat ik het leuk vindt om nieuwe technieken te leren in mijn eigen tijd een simpele login front-end en Api opgezet. Deze moet beschouwd worden als separate van dit project maar zou eventueel makkelijk geïmplementeerd kunnen worden. Het uitzoeken en bouwen van een secure login front-end en Api hebben mij nog eens 5 uur gekost en dit kost wellicht nog een paar uur om te implementeren in de huidige Api.

# Ontwikkelfase

<<Beschrijf het proces>>

* Trello

<https://trello.com/invite/b/Xi2sIczk/0719e51b0fa04d18c234c6e56a0168b0/ois>

* Ontwerp
* Uren
* Moeilijkheden

<<Beschrijf het resultaat>>

<<Reflectie op het proces>>

<<Vertaalslag naar de toekomst>>