Linnæus University
Sweden

Softwork
Design Document

Author 1: Jon Cavallie Mester
Author 2: Karl Ekberg
Author 3: Elias Frigård
Author 4: Joel Salo
Author 5: Patrik Hasselblad

Linnæus University
Sweden

Table of contents

1. Introduction

1.1 Purpose and remainder of the document

The purpose of this document is to provide information about decisions concerned with the design and architecture of the application. The document will provide insight into the design decisions that have been made to realize this project. The document will provide the reader with knowledge of:

- Key abstractions.
- What the architectural goals are.
- How the application will be developed to meet certain requirements, such as
    - Developed with increased modularity.
    - Decisions on the overall architecture.
    - The responsibilities and symbiosis between the elements in the architecture

1.2 Definitions, acronyms, and abbreviations

Definitions

- Software Architecture refers to the fundamental structures of the software system
- Services refer to parts of the system that will act as standalone applications to feed the system with information.
- Assumptions refer to how we expect the different services will work
- Dependencies refer to the “must-have relationships” in the application, for example: how the application depends on external entities.
- System in this context refers to the software in its entirety, frontend, backend, API-services
- Application is the part of the system that runs as per user request. The application runs on the platform which is provided by the system software.
- API providers are the collective of job ad data sources, such as Platsbanken or Remotive, with which we can communicate through their API, to fetch data for our application.

Acronyms

- SLA - A Service Level Agreement is an agreement on the terms and conditions as well as the level of service, such as availability, of a service.

- API - The acronym for Application Programming Interface, which is a software intermediary that allows two applications to talk to each other [1].

2. Architectural goals and philosophy

The main goal of the application architecture is to facilitate scalability and modularity together with extensibility. In short, the architecture shall be designed with the mindset that the application will grow and expand.

The application will architecturally be designed using the client-server pattern. The backend will utilize the ideology of a service-oriented architecture, where the functionality of aggregating job ads will be abstracted to its service which in turn communicates with the backend. This Job Service will be responsible for fetching, filtering, and returning data from multiple different API providers. Furthermore, the architecture will include a database for storing user information and saving job ads. The backend server will act as a communication endpoint for the client application.

One of the most critical design decisions is how the application should behave in case any of the third-party API providers are unavailable. In a scenario where at least one API provider returns the requested data for the application, the end-user will receive this information. Should all of the API providers prove to be unavailable then no data will be returned and the application should state the unavailability of the service to the user in an appropriate manner.

Since data from different API sources might be structured in different ways, the Job Service will be designed to return data in a specific format. The service will abstract and filter the data from the different API sources to match this format used by the rest of the application.

The application should be deployed using well-known cloud services. This frees the application from any hardware concerns. It will also make the application highly scalable since it is possible to add power to and control the capacity of the application whenever needed. As the popularity of the application grows, it can be scaled to meet an ever-growing demand. It will thus be possible to keep the cost of deploying the application in line with customer demand.

Since data from different API sources might be structured in different ways, the Job Service will be designed to return data in a specific format. The service will abstract and filter the data from the different API sources to match this format used by the rest of the application.

The application should be deployed using well-known cloud services. This frees the application from any hardware concerns. It will also make the application highly scalable since it is possible to add power to and control the capacity of the application whenever needed. As the popularity of the application grows, it can be scaled to meet an ever-growing demand. It will thus be possible to keep the cost of deploying the application in line with customer demand.

3. Assumptions and dependencies

External APIs

The application is highly dependent on external API resources. However, we cannot fully guarantee that the API providers have 100% uptime and availability. While this is generally something that we cannot do anything about, we can acknowledge the risk and provide the end-user with relevant information as to if and why APIs are unavailable. The application should abstract the retrieval of API information so that multiple API services can be utilized, which increases redundancy and significantly reduces the risk of the end-user not receiving any relevant information at all. In the rare case that no API providers are available, the user shall be notified that this is the case.

Frameworks and technologies

While designing the architecture of this application, we have assumed that all developers have knowledge of Javascript, Node.js, Express.js, MongoDB, and are used to interacting with external API providers from previous work and educational experience. Thus, the decision has been made to utilize this knowledge in building the application. Furthermore, the decision has been made to include a Javascript front-end framework to build isolated, reactive, and reusable front-end components. This has made us design the application around using the MEVN (MongoDB, Express.js, Vue.js & Node.js) tech stack.

Libraries and tools

Generally, only libraries and tools with long-term support will be used when developing the application. This ensures the longevity and stability of the application and its structural dependencies. As a security feature, the application shall utilize the third-party library Bcrypt for the safe storage of user credentials. Developing such algorithms in-house is a far too large endeavour, prone to errors, and out of scope for this development process. To interface with our NoSQL MongoDB database, we make use of the NodeJS library Mongoose. It is continually developed and maintained by Automattic, the developers behind WordPress, and has 1,222,000 weekly downloads from the NPM[2] registry, indicating a certain level of stability and trustworthiness.

Deployment

Our application is also dependent on external cloud providers for deployment. Luckily, major cloud providers offer Service Level Agreements that usually state a guaranteed uptime of more than 99%. Should the availability of these cloud services diminish, cloud providers offer compensation. This means that whatever cloud provider we choose to use, we will have our backs covered in terms of uptime and availability. Should one cloud provider not meet expectations, it is possible to change. Thus, we have automatically incorporated redundancy in terms of this external dependency. Worth noting is that the SLAs are not valid if one chooses to use the providers free services, they are only applicable if their paid services are used.

Instead of locally hosting our deployment of MongoDB, we have decided to make use of the SaaS (Software as a Service) solution Mongo Atlas. This way we can delegate the management of our database system to a reliable third party.

SLA (Service Level Agreement) of two major cloud providers:

Google Cloud > 99% uptime [3]

Amazon Web Services > 99% uptime [4]

Link to the SLA (Service Level Agreement) for Mongo Atlas:

Mongo Atlas > 99% uptime [5]

4. Architecturally significant requirements

To decide which requirements are the most important from an architectural perspective, we have held a series of workshops in which all requirements listed in the requirements specification have been analyzed, discussed, and graded. During this process, the team concluded that four requirements primarily drive the architectural decisions of the system.

FR-1: Create an account & FR-5 Save job ad

Functional Requirement 1 and 5 together make a significant impact on the architecture of the application. Implementing these two requirements will drive the need for persistence and therefore having to adapt the architecture to include a database and interacting with this.

If these requirements were to be excluded, the architecture would change profoundly as a major part of the system (database service) would not be needed, thus enabling more and different choices regarding the overarching design choices.

By implementing persistence and saving the data correctly and securely the developers have to have the architecture in mind while developing the models and configuring the persistence to fit the needs of the end-users which most likely would like to have their user information saved to easily fetch the data which has been chosen to be saved by the end-user.

NFR-1: Source jobs from external or internal sources

By sourcing jobs from an external source, we have to use a kind of service-oriented architecture that does not clog the backend and promotes modularity and extensibility. This makes it necessary to remove the logic from the main backend server and place it into smaller services. This mostly affects the developers as they have to develop the software that sources jobs from multiple API providers while keeping modularity in mind.

NFR-2: Usable on desktop and mobile devices

Since this is a web application we must create some sort of user interface. The client-server architecture is a driving force of the system architecture as this is the easiest and most well-known pattern to use for two-way interaction between client and server.

This is by far the most important requirement for the end-users and the product owners because this is the fruits of the labor for which has been paid by the product owners. Therefore, if the end-user is happy and satisfied with the software and the usability of the user interface, the product owners will also be happy with it.

Rationale

By deriving these four requirements from the requirements documentation we have excluded several requirements such as those that demand the options to search by either keywords or pre-defined values. Those are functionalities that do not impose a threat to the chosen architecture nor create a need to prioritize other architectures higher than the client-server architecture.

5. Decisions, constraints, and justifications

Client-Server Architecture

We decided to utilize the client/server architecture pattern because the software will not be complex enough to justify other approaches such as fully blown microservice-oriented architecture. Having a dedicated frontend and backend will minimize unnecessary maintenance of multiple services.To see a visual representation of the architecture, in the form of a component diagram, see figure 1 in chapter 8.

Client/Frontend

The client is decomposed into several classes handling different types of responsibilities. The frontend is developed with reusable composable components and a client store.

- The search component is responsible for taking input from the user and extracting information from the different predefined filters (region, municipality and remote). The other major responsibility in the search component is that it bundles the information and sends it to the backend via an HTTP request in order, which the backend then uses to make a request to the different API-providers

- The authentication component cluster consists of three different classes that are responsible for: register, login and logout. The parts of the authentication cluster all communicate via an HTTP request to the backend in order to either register, sign in or sign out

- The ad component creates the visual representation of a job advertisement that is requested by the search component and assembles the data returned to the client.

- The favorite component is used to allow the user of the application to mark certain job advertisements that they find interesting in order to save them to be able to retrieve them from the database at their own convenience.

Service-Oriented Backend

With extensibility in mind, we decided to break out the component responsible for fetching job ads from external APIs and returning them according to a specific interface into its own, separate service. This will facilitate scalability, testability, and maintainability as well as relieving the main backend from having too many responsibilities. The development process will also become easier, as developers can divide their work with less risk of potentially working on the same things at the same time. This will place constraints on the development in the form of limiting the functionality/responsibility to be implemented in each component of the system.

Main Backend

The main backend will be responsible for dealing with tasks such as managing authentication, authorization, fetching data associated with registered users, and requesting job ads from the job service and then delivering them to the client. One could imagine the backend as the glue that keeps the application cohesive and cooperating.

The backend consists of two main classes:

- Main-controller is responsible for viewing ads. The objective of the class is to retrieve ads from the job service and send the data to the frontend to display. The main-controller also handles favorites, it communicates with the database to allow the user to save, retrieve and delete favorites from the graphical user interface.

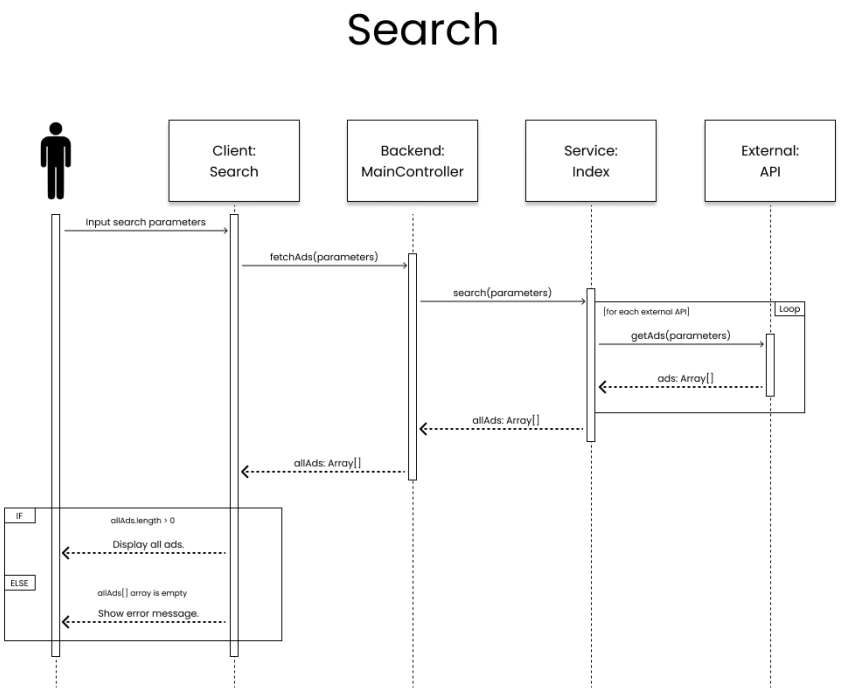
- User-controller is responsible for handling the user information. When an account is created, the user-controller makes sure that the email address provided is unique and in proper format before it hashes the password before saving it to the user collection in the database. The user-controller also takes care of sign in and sign out.

Job Service

The job service will be a service component that fetches job ads from all our different API providers according to supplied search criteria and formats their data according to one coherent schema and merges them all together before returning it as a response to the component that sent the request, in this case, our main backend.

The job service consists of four major classes: The classes that interface with the Platsbanken, Remotive, RemoteOK APIs, and a controller class which takes the returned data from the previously mentioned classes and sends them as a response to the main backend.

An overview of the search scenario to display the decomposed relationship between the different components:



6. Architectural Mechanisms

6.1. Persistence

Initial state

The purpose of having a persistence mechanism implemented is that we will allow the user to create user accounts to be able to store and access related user data.

Design

When deciding what storage solution to use for this application we need to take into account the type of database to use. For this type of simple application, we realized that a document database would be more suitable than a relational database because of the simplicity of implementation as well as the data handled by the application, which by and large is made up of reading and writing single documents from the database.

Implementation

All developers have previous experience working with NoSQL, which suits our demands perfectly. By using the third-party library Mongoose ODM together with MongoDB, we allow ourselves to abstract the integration of the database into our application. Models and queries will be easier to write and execute, thus simplifying the code and the development process.

6.2. Hashing and Encryption

Initial state

The purpose of hashing passwords is that if someone were to get access to the database, they would not be able to see users' passwords in plaintext and use that information to intrude on other accounts they have online. Encryption of the data being transmitted is also a key mechanism as security is not an optional feature, but something that's both expected and required.

Design

Since we are using Javascript as our language of choice, we made the design decision to use a third-party library. This would allow us to hash passwords using well-implemented, safe algorithms without having to implement them ourselves which in extension would expose our application to risks involving the use of subpar algorithms. HTTPS should be used to safely transfer user information between the client and server. Using old-fashioned HTTP would expose user information while in transit.

Implementation

We made the implementation decision to use one of the most popular and reliable third-party libraries for hashing passwords, namely Bcrypt. Many cloud providers offer built-in HTTPS, although setting it up in-house is also a possible alternative.

6.3. Front End Framework

Initial state

The purpose of using a modern front-end framework is to be able to create more readable, modular, testable, reusable, maintainable, and well-structured code. Using a framework will also make the code easier to read and understand because it creates an abstraction between the client and the DOM that is being manipulated.

Design

When deciding on what framework to use we were looking for a few specific attributes. We wanted to utilize lightweight, single-file components for simplified readability and modularity. We will also base a big portion of the decision on what developers want to use and learn.

Implementation

The decision fell on the Vue.js framework, it has the attributes we were looking for and has great features like reactive two-way data binding, and has very good documentation. It is a quite new and upcoming framework that has gained a lot of traction over the past few years.

6.4. Web Server

Initial state

The purpose of using a web server is that it is a crucial component of the client-server architecture. One could write a web server from scratch, but this would be inefficient and time-consuming when there are tried and tested implementations already available for use.

Design

We made the design decision to select one of the more popular and stable NodeJS server frameworks. Using a framework facilitates development of the server, as most encourage you to follow certain design patterns, giving developers a more structured way to work during the development process, and provide already implemented classes, middleware and so forth.

Implementation

The particular web server framework we have chosen, Express, is the de facto standard of web server frameworks in the Node.js ecosystem. It includes a vast number of middleware modules that can be used to perform additional tasks on the requests and responses and is very easy to configure.

6.5. Communication

Initial state

A large architectural mechanism of our system is network communication, between client and server as well as with third-party APIs. By implementing a way to get data from APIs we are allowing ourselves to get data from different API services spread out in smaller modules, communicating with a backend service to transmit the desired data to the user interface.

Design

There are a number of well-documented libraries for creating API requests, which are widely used and easy to implement into our codebase. One of these libraries should thus be used to abstract the underlying request objects. Communication between the client and server as well as to external resources utilize the HTTP protocol, which needs to be supported by the application and deployment infrastructure. Furthermore, all HTTP requests apart from internal server requests should be protected by the HTTPS protocol to encrypt data.

Implementation

By utilizing the package Axios in our implementation, we are able to fetch data from the APIs and send data between the backend and frontend in order to display the correct data in the user interface and create search queries by sending data from input fields in the frontend to the backend. As briefly mentioned in point 6.2, several of the deployment solutions that we have looked at offer built-in support for HTTPS. However, setting up own certificates is not time consuming or difficult to perform, should the deployment requirements change.

6.6. Deployment

Initial state

Deploying the application to the end-users is one of the major concerns of the architectural process. There are a few ways to solve this problem, either by setting up our own, on-premises server solution or letting a third party handle the deployment of the application. By creating our own solution we would have more control over the implementation and hardware, but it is also costly and prevents easy scalability. The third-party service for deploying the application is just the opposite, easy to use, highly scalable and reduces upfront cost.

Design

As acquiring our own hardware for deployment seems both cumbersome, impractical, and not feasible in terms of the budget of the project, we have made the decision to use a cloud-based deployment model.

Implementation

When selecting a cloud provider, there are several options like AWS, GCP, and Digital Ocean that require system administrators to configure virtual server instances or pay good money for pre-made solutions. Alternatively, especially when producing prototypes or small deployments, is to use Heroku and Netlify. These provide easy deployment of applications without the need for configuring server instances and have free plans that are suitable for deployment during the development process. These also have paid plans which support the number of users and reliability requirements specified in the initial requirements document.

We will make use of Netlify to host the frontend because we want to have continuous integration and continuous deployment during development, which Netlify provides in a very accessible manner. Netlify also allows us to easily enable the HTTPS protocol, so there will be no need to manually set up our certificates.

For our backend hosting, we will use Heroku because it is a well-known backend hosting platform and free to use for projects of this size. Both providers come with extensive documentation, support for continuous integration, and a minimal learning curve which will grant us more time to focus on the development of the application.

7. Key abstractions

7.1. Job Ads

Job ads are the main concept of the application. They contain information that end-users are interested in and are the main reason for anyone to use the application. Job ads are part of all the different aspects of our system, from what the user will view using the frontend to how data is handled on the backend and persistence.
While we have a quite clear idea of the implementation details of this abstraction at this point, there are many different ways to design a solution for it.

7.2. Users

Users are a critical concept of the system as they, together with job ads, shape the purpose and functionality of the application. The concept of a user in our application is an abstraction of an end-user that can store and retrieve personalized and related information from our application. A user has login credentials to gain access to his or her stored data.

8. Layers or architectural framework

The application shall utilize the well-known client-server architecture. The client will be responsible for displaying the UI that interacts with the end-user of the application and the server will be responsible for data management and authentication. The client will in this case run locally in the browser of each user, while the server is run on a centralized system. The backend infrastructure will borrow ideas from a service-oriented architecture. Concretely this means that the Job Service will be broken down into its own, independent application while the main backend service remains free of its implementation details. The figure below displays a graphical overview of the system architecture.

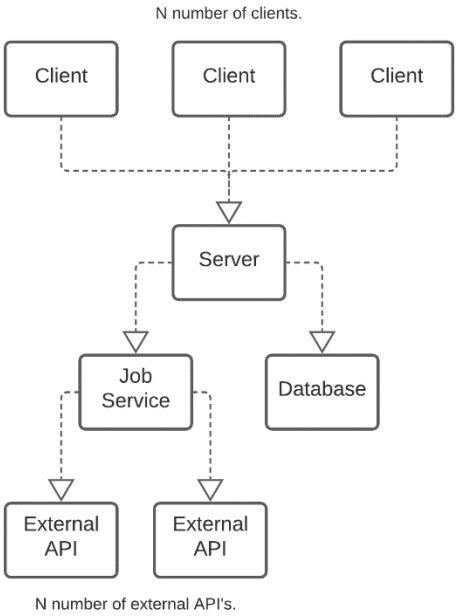


Figure 1: Graphical Overview of the System Architecture

9. Architectural views

9.1. Logical view

9.1.1 Architecture

The diagram below shows the overarching architecture of the system. It includes packages for the client and server in our architecture as well as displaying external resources. As the detail of the system grows, this diagram can be updated with additional information and classes to represent the inner workings of the system.

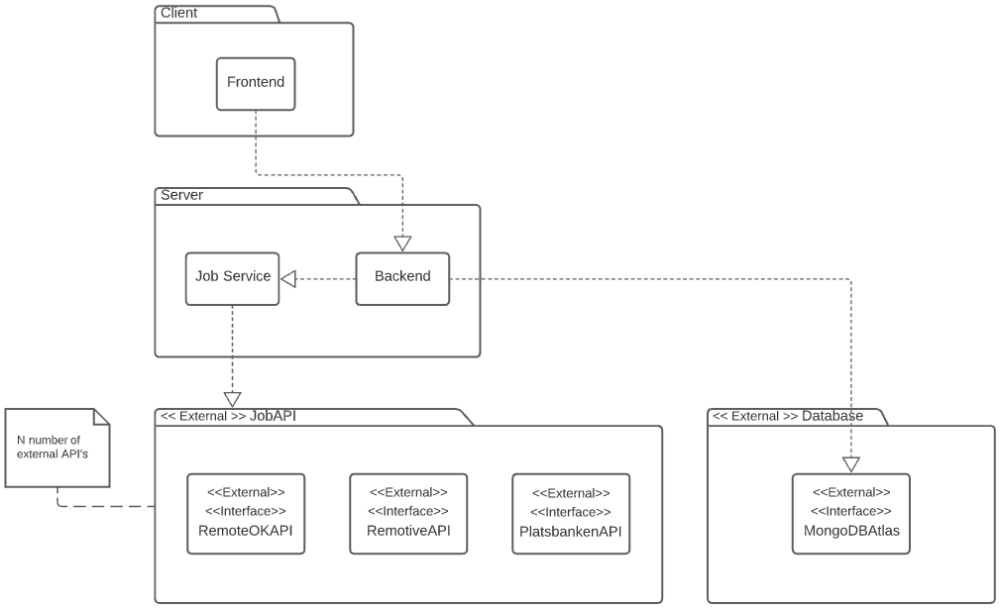


Figure 2: Package Diagram

9.1.2 API Abstraction

Description

One of the most important abstractions that we need to make while designing this application is what data should be returned and what parameters can be used to query data from the different services in our system. While working on our design, we have identified two major types of data abstractions; request and response for job ads. Response data specify in what format we expect data to be returned from the different services, while request data are the possible parameters that we expect to provide when sending a request.

We have used OpenAPI to specify and document these two abstractions. These can be seen as design contracts or interfaces that developers are bound to use in development.

Response Data

Figure 3: API Object Response Schema

The figure above describes the schema of the objects for response data, specified using OpenAPI [6]. Responses from the backend to the frontend and the job service to the backend will utilize this data format. Concrete responses from third-party APIs might contain redundant or misformed information in the context of this application. It is the responsibility of the Job Service to filter and match data for use in the rest of the application.

Request Data

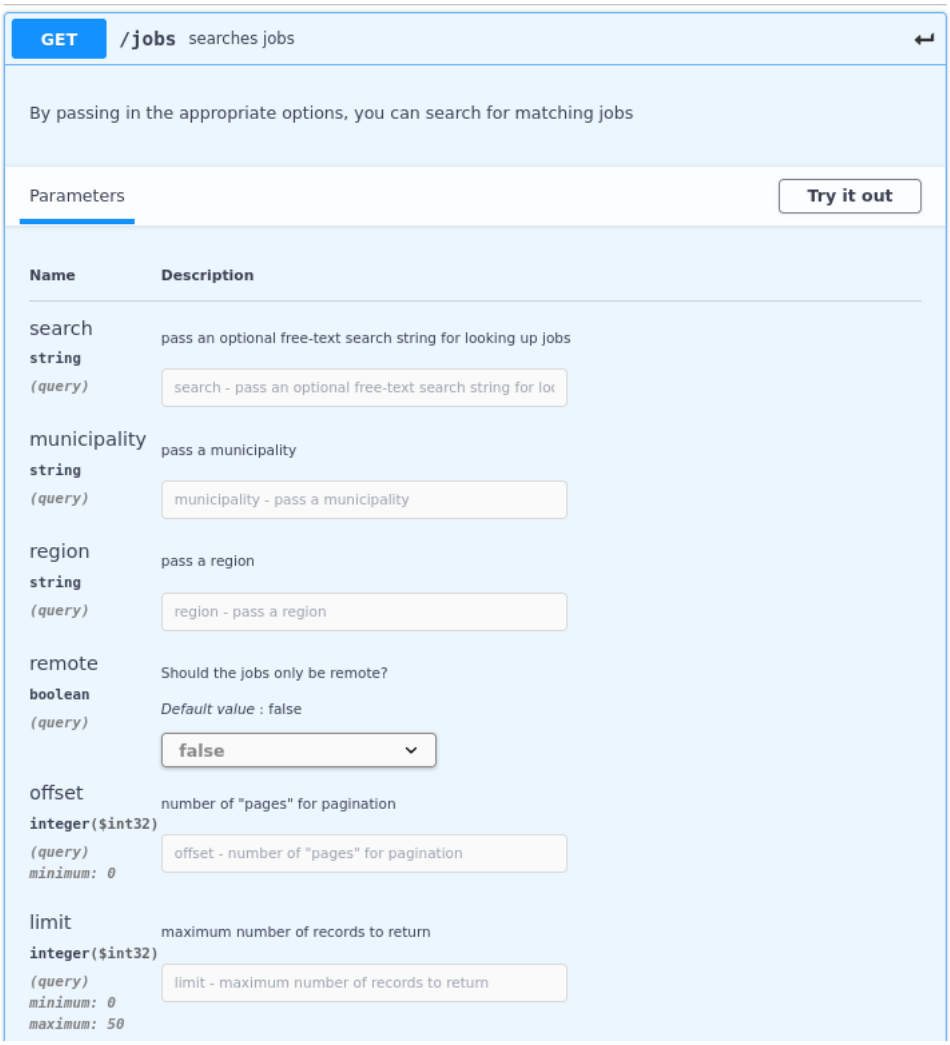


Figure 4: Request Parameters

The figure above shows the parameters that will be used for making requests. In contrast to response data, these parameters will flow in the opposite direction i.e. from the client or the backend to the job service. In essence, this is a design contract that should be adhered to when implementing classes that request job data from other parts of the application.

9.1.3 Persistence

The following section defines the user data that we use in our persistent storage solution. This is an overview of the attributes that are contained within a user document. Favorite job ads will be stored as embedded models within the user schema. This is one of the two ways of defining relationships in MongoDB and the most suitable for our use case because of faster queries to the database. MongoDB includes several attributes when creating documents, such as ID and time of creation. Hence, these attributes are not included in the model below, as including all attributes provided by mongo will clutter the schema.


```
{
   "email":"STRING",
   "passwordHash":"STRING",
   "jobs":[
      {
         "title":"STRING",
         "company":"STRING",
         "location":{
            "country":"STRING",
            "region":"STRING",
            "municipality":"STRING"
         },
         "description":"STRING",
         "url":"STRING",
         "remote":"BOOLEAN",
         "source":"STRING",
         "logoURL":"STRING"
      }
   ]
}
```


Figure 5: Persistence Schema

References

[1] MuleSoft, “What is an API?”, 2021, [Online]

https://www.mulesoft.com/resources/api/what-is-an-api#:~:text=API%20is%20the%20acronym%20for,you're%20using%20an%20API [Accessed May 11, 2021]

[2] W3Schools, “What is NPM”, [Online] https://www.w3schools.com/whatis/whatis_npm.asp [Accessed April 16, 2021]

[3]. Google, “Compute Engine Service Level Agreement (SLA), 2021, [Online] https://cloud.google.com/compute/sla [Accessed April 16, 2021]

[4]. Amazon, “Amazon Compute Service Level Agreement, 2020, [Online] https://aws.amazon.com/compute/sla/ [Accessed April 16 2021]

[5]. MongoDB, “MongoDB Atlas Service Level Agreement, 2020, [Online] https://www.mongodb.com/cloud/atlas/sla [Accessed April 16 2021]

[6]. SwaggerHub, “JobService API”, 2021, [Online]

https://app.swaggerhub.com/apis/jonmest/JobService/1.0.0 [Created by

Group 6 April 14 2021]