Team seeker

Project documentation

Gabriela Chmelařová, Petr Kohout

We wanted to simplify ways in which people can search through iGEM teams from different years of the competition and make it easier to look for team wikis covering a particular topic. We would also like to make a quick and easy way for the database to be expanded by adding iGEM teams from future years, ensuring that our teams seeker would be kept up to date.

Architecture

The project consists of three main parts, which are shown in Figure 1. The first part is a Scraper script, which is responsible for extracting data from iGEM pages (a list of pages from which information is collected can be found in the README.md file). After extracting the data, the script sends them to the server, which stores them in the ElasticSearch database after checking for the absence of the necessary attributes. The server continues to provide the data to the web application through the API. The web application displays information to the user and allows the user to enter search words that are sent to the server, which processes the request and queries the database. The result of the query is sent back to the application, which displays these results.

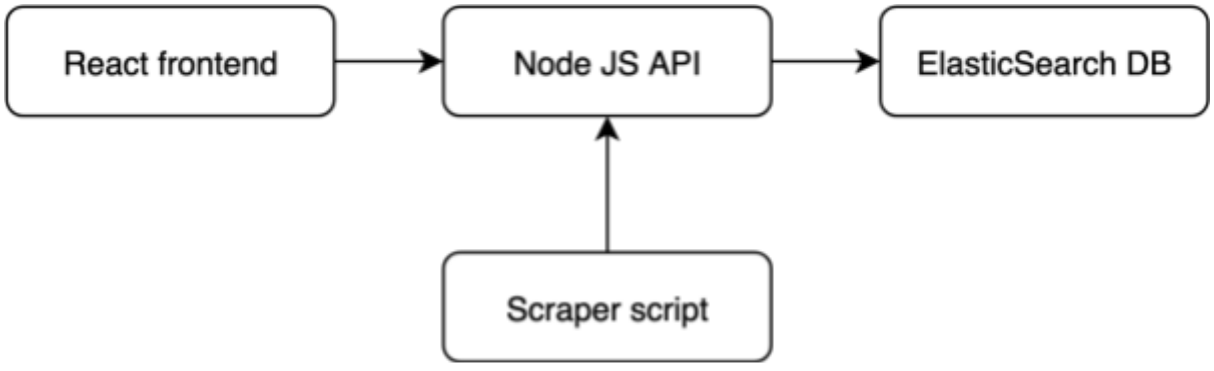

Figure 1: project architecture

React frontend

The web application was built using the JavaScript library React, which was chosen mainly due to its ability to use so-called components, which allow easier manipulation of often repeated elements. The basic knowledge needed to work with this library was acquired from the Learn React JS tutorial.

The application communicates with the API by sending GET and POST requests. The application itself is based on three main components. The first is the App component, which first uses the fetch method to create a GET request for team and biobricks data and then prepares them for display. It also renders one of two versions of another component called the SearchBar, which allows the user to search and display results. It is possible to choose between searching for teams and biobricks. According to the selected search engine, the obtained data are loaded, which form is modified by components that are created for each searched object.

The search works as follows. The entered term is searched with pre-specified parameters, which can be changed using the radio buttons below the search field. It is possible to choose in which category you want to search and whether to select the results that contain the term or those that do not.

The method of obtaining search results is based on sending POST requests, which content is built on the basis of selected parameters. First is obtained the name of the currently selected category in which the entered text will be searched. Next, a choice of whether or not the resulting text should contain this expression is obtained, and finally the entered text from the search field is saved as a value. In the case of direct text entry without prior selection of categories, the default settings are used to compile the request.

Node JS API

Used libraries

- bcrypt - API - creates a hash value of the password, which is stored in the database
- body-parser - API - parsing requests directed to the server
- elasticsearch - API - ElasticSearch database client
- express -API - framework for Node.js
- jsonwebtoken -API - user authentication (jwt)
- morgan -API - server logging
- nodemailer -API - sending e-mail notification to the user (account creation)
- randomstring -API - generating a unique and random link for user authentication when creating an account

Node.js was used to implement the application API. This solution was chosen on the basis of its high popularity and extensive support in the form of teaching materials and existing libraries.

To gain basic knowledge in the field of API technology, I used a series of videos about creating a REST API on YouTube on the Academind channel.

The server accepts GET, POST and DELETE requests and mediates all communication with the ElasticSearch database. The request handling works as follows. Requests are received by the server and distributed to the responsible query processors. In the case of GET requests, there is immediate communication with the database and the result is returned. In the case of POST requests, in some cases the user who entered the request is authenticated and the correct data structure is subsequently verified. When a DELETE

request is processed, the user is authenticated and the action is performed. In the event of any error, the corresponding error message is returned.

The application processes three main types of data: user data, biological data (so-called biobricks) and information about teams. A separate handler was created for each category. The structure of API requests is described below.

GET
By default, the server runs on port 3001.

`localhost: 3001 / teams`
Returns a list of teams that are contained in the database.

`localhost: 3001 / teams / structure`
Returns a list of attributes contained in team records.

`localhost: 3001 / biobricks`
Returns a list of biological structures that are contained in the database and have been registered within iGEM.

`localhost: 3001 / biobricks / structure`
Returns a list of attributes contained in biological structure records.

POST

`localhost:3001/teams/match`
`localhost:3001/biobricks/match`
Returns data obtained using a query that is specified in the request body with the structure and logic described below.

```
{
      NAME:[{contain: Bool, value: String}]
      .
      .
      .
}
```

The `name` position contains the name of the attribute being searched. The value of this attribute is the list with the result. The `contain` item indicates whether or not the value of the `value` attribute should be included in the result. The individual requests are then used to compile a query, which is forwarded to the database.

Below is described the requirement for teams registered in 2020 and whose name does not contain the number 2, but contains the value `team`.

```
POST: localhost:3001/teams/match
{
      "title":[{"contain":false, "value":"2"},
               {"contain":true, "value":"team"}],
      "year":[{"contain":true, "value":"2020"}]
}
```

from this structure a query is compiled that uses a list of values that should or may not be true. The example above will be converted to the following interpretation.

```
{
      "query":{"bool":{
                  "should": [{"match":{"title":"team"}},
                             {"match": {"year": 2020}}],
                  "must_not": [{"match": {"title": "2"}}]
               }}
}
```

Inserting
POST requests can be used to insert data into the application at the following addresses.

`localhost:3001/teams`
`localhost:3001/biobricks`
After the data is received by the server and the user is authenticated, an appropriate error message is sent to the user in the case of an error.

For the possibility of adding more actual records in the future, the possibility of inserting records using a script that extracts this information from iGEM pages has been created. To control the insertion, a user authentication process was established, which will be used primarily when inserting new data. The average user will not need it to fully use the application.

`localhost:3001/users/signup` - sends data to the user who requires registration
`localhost:3001/users/login` - will log in
When logging in and registering, it is necessary to send the account e-mail address and password as part of the POST request. When creating a user account, it is verified whether the account with the given address has already been set up. In the case of an existing account, the user is informed. Then, the password is converted to a hash using the *bcrypt* library, and the account is created as an account that is waiting for confirmation. Then the user is sent an email with a link and after clicking on it, a real account is created for the user and it is possible to log in. Below is the body structure of the registration and login request as well as the response to a successful login.

```
POST: localhost:3001/user/signup
{
      "email": "test@email.com",
```

```
      "password": "testPassword"
}


POST: localhost:3001/user/login
{
      "email": "test@email.com",
      "password": "testPassword"
}

response = {
                  "message": "Auth successful",
                  "code": 200,
                  "token": "eyJhbGciOiJIUzI1..."
            }
```

The response contains a token that is used to authenticate the user. The request must be sent in the header in the Authorization attribute in Bearer format with the following token.

```
{
      Authorization: “Bearer eyJhbGciOiJIUzI1…”
}
```

To delete all temporary accounts, it is necessary as an authenticated user to send to the url
`DELETE: localhost:3001/user/confirm`

`DELETE: localhost:3001/teams/:teamCode`
to delete a team using *teamCode*

`DELETE: localhost:3001/biobricks/:title`
to delete biobricks using the *title* item

`DELETE: localhost:3001/user`
to delete the user (user itself) using the item. A token and credentials must be sent in the body of the request.

ElasticSearch DB

As a place to store data about iGEM teams and biobricks, we chose the ElasticSearch database, thanks to its specialization in full-text search.

ElasticSearch allows the creation of text indexes over individual fields, which allow you to search in the text only the word base, inflected expressions or synonyms. It is therefore suitable for our use, where the user enters a (inaccurate) search string from the frontend of the website and the backend actually works as a search engine.

Scraper script

Data on iGEM teams and biobricks is currently not accessible in an easy-to-download form or through an API. They are located on the iGEM website in the form of "wiki" pages. To obtain this data for subsequent indexing, we used the scraping technique.

When scraping iGEM teams, the script first downloads a spreadsheet CSV file with a list of teams, which contains partial information about historically all teams participating in the iGEM competition. Each team has (separately for each year) its own more detailed page, which contains additional information, such as the name of the school, a list of team members or a division. For us, the most important thing for searching is the name and abstract of the project of the given team.

There is a large number of biological parts called biobricks on the iGEM website and their complete list is not available. They are found in different collections by category. The script scans a number of such categories and finds pages describing individual biobricks. The description of biobricks is in the form of a wiki page. It often has a fragmented form (paragraphs, pictures). However, the script retrieves only the textual content of all paragraphs from it.

The obtained data is sent via a POST request to the Node JS backend. When sent, the script authenticates the user using a bearer token, which it obtains, when the script is run, from the Node JS backend, using the email address and password obtained from the script's arguments.

The script is written in the Kotlin language and uses the skrape.it library, which is clear and suitable for easy and fast implementation into this project.

Future use

We find the use of our project mainly in the practical use of the iGEM competition teams. The idea for this web application was created mainly thanks to the requirements of the participants of the competition, who use the data on the past topics of the teams for inspiration and possible take over them. We have expanded the search with the possibility of searching for created substances called biobricks, which are essential for teams and information about them can be found in several places. According to users, current tools for searching for similar data are confusing or contain only outdated data. For this reason, we have created our tool with the possibility of future data updates.

How to run

npm run start-both

There is also a docker-compose file that runs all project folders (frontend, backend, database) in Docker.