Достъп до MongoDB с Express.js

I. Въведение

Express.js или само Express е сървърна рамка за Web приложения базирани на Node js. Тя позволява бързо изграждане на приложения с една страница - Single page Apps (SPA), приложения с множество страници, както и хибридни Web приложения. Express поддържа повечето популярни JavaScript стекове за разработка, например MEAN (MongoDB, Express, AngularJS, Node.js), MERN (MongoDB, Express, React, Node.js) и MEVN (MongoDB, Express, Vue, Node.js). Вижда се, че разликата е в рамката, реализираща потребителския интерфейс (AngularJS, React или Vue).

Express предоставя механизми за:

- Обслужване на различни видове HTTP(S) заявки (GET, POST и PUT) чрез пренасочването им към обслужващи функции чрез техните URL пътища (маршрути).
- Възможност за интегриране на двигател за визуализиране на "изгледи", за да се генерират отговори чрез вмъкване на данни в шаблони.
- Настройки на сървъра, обслужващ заявките.
- Добавяне на допълнителен междинен софтуер (middleware) за обработка на заявки във всяка точка от конвейера за обработка на заявки.

Макар че самият Express е с минималистичен дизайн, разработчиците са създали съвместими пакети за междинен софтуер, за да се справят с почти всеки проблем при разработката на Web приложения. Съществуват библиотеки за работа с бисквитки, сесии, следене на потребителски влизания, извличане на параметри, предавани към ресурс и много други. Функциите на междинния софтуер са функции, които имат достъп до обекта на заявката “request”, обекта на отговора “response”, както и до следващата функция на междинния софтуер в цикъла “заявка-отговор” на приложението. Функциите на междинния софтуер могат да изпълняват следните задачи:

- Изпълняват всякакъв код.
- Да правят промени в обектите на заявката и отговора.
- Да приключат цикъла “заявка-отговор”.
- Извикване на следващата функция на междинния софтуер в стека.

Едно Express приложение може да използва следните типове междинен софтуер:

- Междинен софтуер на ниво приложение.
- Междинен софтуер на ниво маршрутизатор.
- Междинен софтуер за обработка на грешки.
- Вграден междинен софтуер.
- Междинно програмно осигуряване от трети страни.

Междинното програмно осигуряване на ниво приложение се свързва с инстанцията на обекта, описващ Express (**app**), като се използват функциите app.**use** и app.**method**, където “method” е HTTP методът, използван при HTTP заявката, например **get**, **post** или **put**. Следва пример, при който се обслужват “сляпа” get заявки:

```
var express = require('express');
var app = express();
...
app.get('/', (request, response, next) => {
  response.send('Отговор на заявката, най-често HTML код.')
});
```

Зареждането на модул Express се реализира чрез метод **require**. Инстанция на обекта за достъп до Express **app** се получава чрез метод **express**. При сляпата GET заявка не се задава конкретен ресурс, който да обслужи заявката – първият аргумент е "/".

Обработка на грешки също се реализира чрез междинен софтуер. Междинният софтуер за обработка на грешки винаги приема четири аргумента, например:

```
app.use((error, request, response, next) => {
  console.error(error.stack)
  response.status(500).send(Информация за грешката.')
})
```

Код 500 за статус на отговора отговаря на вътрешна сървърна грешка.

Express има следните вградени функции за междинен софтуер:

- "express.static" обслужва статични ресурси, като HTML файлове и изображения.
- "express.json" анализира входящите заявки с JSON съдържание. Налична е при Express 4.16.0+.
- "express.urlencoded" анализира входящите заявки със съдържание кодирано чрез URL кодер. Налична е при Express 4.16.0+.

Много често се налага използване на междинен софтуер на трети страни с цел получаване на липсваща, но желана функционалност. Първо трябва да инсталирате модула за необходимата функционалност след което трябва да го заредете в приложението си на ниво приложение или на ниво маршрутизатор. Следва пример за използване на парсер на HTTP заявки:

```
const bodyParser= require('body-parser');
...
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
```

Информация за най-често използвания междинен софтуер на трети страни можете да получите от следната страница:

https://expressjs.com/en/resources/middleware.html

II. Последователност за създаване на приложения с използване на Express

1. Създайте папка в която ще създадете своя проект, например MongoWithExpress. Стартирайте cmd.exe така, че текущата папка да е проектната папка. Създайте нов проект чрез команда

npm init

Задайте последователно: име на проекта; версия на проекта; описание на проекта; входна точка (например index.js); git хранилище (не е задължително); ключови думи; автор и тип на лицензa. В резултат на командата ще бъде създаден файл **package.json**, който съдържа информацията за проекта, въведена до момент, например:

```
{
  "name": " MongoWithExpress",
  "version": "1.0.0",
  "description": "Generate CRUD requests using Express",
  "main": "server.js",
  "dependencies": {
    "mongodb": "^3.6.9"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js"
  },
  "author": "rs-soft-bg",
  "license": "ISC"
}
```

2. Инсталирайте Express. Express е програмна рамка за back-end Web приложения базирани на Node.js. По своята същност Express позволява стартиране на един или множество сървъри, обслужващи заявките на клиентите. Клиентите могат да бъдат както хора, така и smart модули. За да инсталирате Express изпълнете следната команда:

npm install express --save

Командата ще прехвърли в проекта ви всички файлове, които изграждат Express и ще обнови съдържанието на файл package.json, поле “dependencies”.

3. Създайте в проектната папка файл server.js. Той ще съдържа програмния код на сървъра чрез който ще обслужваме заявките на клиентите. Първо ще тестваме дали сървъра работи като въведем кода чрез който се стартира сървър, който подслушва зададен свободен порт и функция чрез която се обработват HTTP GET заявки:

```
const express = require('express');
const app = express();

app.listen(8080, () => {
  console.log('Start listening on port 8080...')
});
app.get('/', (request, response) => {
  response.send('Здравей!');
});
```

Първо, ще заредим Express като модул чрез метод **require**. Следва създаване на инстанция на програмната рамка с референция **app**. Стартирането на сървъра се реализира чрез метод **listen**. Методът изисква два аргумента: номера на порта, който се подслушва (8080) и ламбда функция, която се активира при нова заявка към сървъра. Всяка заявка, в зависимост от типа й (GET, POST, PUT), се пренасочва за обработка чрез съответния й метод. На този етап сървърът обслужва само “слепи” GET заявки (не е зададено име на ресурс) чрез метод **get**. Първият аргумент е URI (‘/’), а втория е ламбда функция, която обслужва GET заявките. На функцията се предават два обекта: request (достъп до входящия комуникационен канал - параметрите на заявката) и response (достъп до изходящия комуникационен канал). На този етап сървърът връща чрез метод **send** низа “Здравей!”.

4. Стартирайте сървъра чрез командата:

node server.js

След като на конзолата се появи низа “Start listening on port 8080...” стартирайте браузър и изпратете “сляпа” GET заявка към сървъра:

http://localhost:8080

Ако всичко е наред - трябва да видите низа “Здравей!”.

5. Ако промените кода на сървъра трябва да го спрете чрез Ctrl+C и рестартирате чрез node. За да автоматизирате този процес може да използвате инструмента **Nodemon**. Той следи промените в кой да е файл от сървърната страна и при всяка промяна рестартира сървъра автоматично. Инсталирайте Nodemon чрез следната команда:

npm install nodemon --save-dev

Тук използваме флаг “--save-dev”, защото ще използваме Nodemon само в процеса на разработка на приложението. Този флаг ще вмъкне информация за версията на Nodemon в поле “devDependency” във файл package.json, например:

```
"devDependencies": {
    "nodemon": "^2.0.12"
}
```

6. Ще създадем HTML форма в index.html за изпращане на заявки към сървъра. Нека заявката да проверява кои студенти имат определена оценка по зададена от потребителя дисциплина. Програмния код, който реализира това, е следния:

```
<form action="/query" method="POST">
    <input type="text" placeholder="Въведи дисциплина" name="subject">
    <input type="text" placeholder="Въведи оценка" name="grade">
    <button type="submit">Изпрати заявката</button>
</form>
```

Ще използваме HTTP POST заявка до ресурс “query”, за да изпращаме запитванията към базата данни. В тази форма се изисква въвеждане на две стойности: име на дисциплината “subject” и оценка “grade”.

7. За да формираме заявка към MongoDB сървъра трябва да прочетем стойностите на полета “subject” и “grade”, които се изпращат чрез POST заявката. За да реализираме това ще използваме модул “**body-parser**”. Инсталацията му се реализира чрез следната команда:

npm install body-parser –save

За да използваме този модул, ще въведем следната информация във файл server.js:

```
const bodyParser= require('body-parser');
...
app.use(bodyParser.urlencoded({ extended: true }));
// Следва обработка на GET и POST заявки.
...
app.post('/query', (request, response) => {
    let subject = request.body.subject;
    let grade = request.body.grade;
    response.send(`${subject}: ${grade}`);
});
```

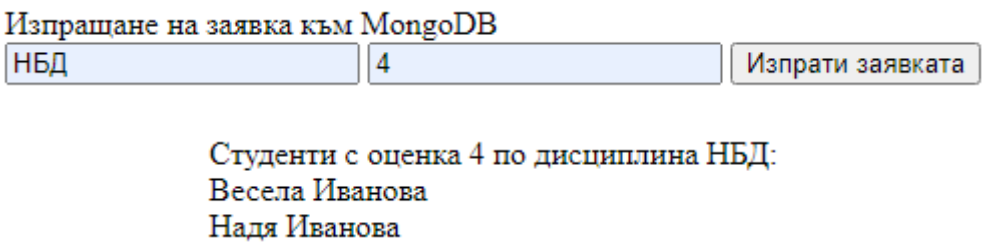
Чрез метод **urlencoded** се задава данните, предавани чрез HTML форми, да се декодират и добавят към свойството “body” в обекта на заявката (request). Чрез метод **post** се прехваща POST заявката към ресурс “query”. От тялото на ламбда функцията се извличат стойностите на двете полета от HTML формата (“subject” и “grade”). За да тестваме работата на метод post връщаме прочетените параметри на браузъра чрез метод send.

8. Вече можем да генерираме заявки към MongoDB сървъра. На практика се създават определен брой комуникационни канали към MongoDB (pool) и постъпващите заявки се разпределят да използват някои от тези канали. За да опростим задачата ще създадем само един комуникационен канал към сървъра. Това се реализира чрез метод **connect**. От тялото на функцията, която се активира при успешно свързване ще реализираме методите, които прехващат заявките от страна на браузъра (get и post). Чрез метод **get** ще обработваме слепите GET заявки, а чрез метод **post** – данните от HTML формата. На Фиг. 1 е показан примерен код, който реализира свързване с MongoDB сървъра и изпращане на заявка чрез метод **find**. В случая се търсят имената на студентите, които имат точно определена оценка по избрана дисциплина. Програмният код използва следните константи:

```
const url = 'mongodb://localhost:27017';
const dbName = 'students';
const collectionName = "data1";
const options = {
    serverSelectionTimeoutMS: 3000,
    connectTimeoutMS: 3000,
    socketTimeoutMS: 3000,
    useUnifiedTopology: true,
};
```

Ще работим с локално инсталиран сървър, база данни с име “students” и колекция “data1”. Чрез JSON обекта options се задава необходимите timeout времена. След изпълнение на метод connect се получава обект “client”. Първо, получаваме обект db за достъп до базата данни. След това получаваме обект “collection” за достъп до желаната колекция. Използваме метод **urlencoded** с цел достъп до параметрите, предавани от HTML формата. Следва програмния код на callback методи get и post. Метод **get** връща съдържанието на файла index.html. От тялото на метод **post** се получава стойностите на избраната дисциплина и оценка (“qSubject” и “qGrade”). Следва дефиниране на три константи, които се използват като аргументи на метод **find** (filter, projection и sort). За метод find е зададено резултатът да се преобразува до масив (метод **toArray**).

От тялото на **then** прочитаме резултата от сървъра (обект docs) и формираме резултата, който да върнем на браузъра. Това се реализира чрез метод **send**. Следва пример за конкретна заявка към базата данни и отговора, който приложението връща:



```
MongoClient.connect(url, options)
    .then(client => {
        const db = client.db(dbName);
        const collection = db.collection(collectionName);

        app.use(bodyParser.urlencoded({ extended: true }));

        app.get('/', (request, response) => {
          response.sendFile(__dirname + '/index.html')
        });
        app.post('/query', (request, response) => {
            let qSubject = request.body.subject;
            let qGrade = request.body.grade;
            const filter = {
              grade: {
                subject: `${qSubject}`,
                value: parseInt(`${qGrade}`)
              }
            };
            const projection = { _id: 0, name: 1 };
            const sort = { name: 1 };

            collection
                .find(filter, { projection: projection, sort: sort })
                .toArray()
                .then(docs => {
                    let result =
                    `Студенти с оценка ${qGrade} по дисциплина
                                                ${qSubject}:<br/>`;
                    if (docs.length == 0) {
                        response.send(result + 'Няма');
                    }
                    else {
                        docs.forEach((doc) => {
                            let name = doc.name;
                            result +=
                              `${name.firstName} ${name.lastName}<br/>`;
                        });
                        response.send(result);
                    }
            })
            .catch(error => {
                console.log(error.message);
            });

        });

        // listening on server port
        app.listen(SERVER_PORT, () => {
            console.log(`Start listening on port ${SERVER_PORT}...`)
        });
    })
    .catch(error => {
        console.log(error.message);
    });
```

Фиг. 1 Програмен код на метод connect

На Фиг. 2 е показан програмния код от файл server.js (без метод connect).

```
const express = require('express');
const bodyParser= require('body-parser');
const MongoClient = require('mongodb').MongoClient;
const app = express();

const url = 'mongodb://localhost:27017';
const SERVER_PORT = 8080;

const dbName = 'students';
const collectionName = "data1";
const options = {
    serverSelectionTimeoutMS: 3000,
    connectTimeoutMS: 3000,
    socketTimeoutMS: 3000,
    useUnifiedTopology: true,
};
//  Следва метод connect: свързване към базата данни, генериране на заявка и формиране на отговор (виж Фиг. 1).
```

Фиг. 2 Програмен код – файл server.js

Програмният код на файл index.html е показан на Фиг. 3.

```
<!DOCTYPE html>
<html lang="en">
    <head>
      <meta charset="UTF-8">
      <title>Query MongoDB</title>
    </head>
    <body>
        <div id="container">
            <header>
                Изпращане на заявка към MongoDB
            </header>
            <section data-position="query">
            <form action="/query" method="POST">
                <input type="text"
                        placeholder="Дисциплина" name="subject">
                <input type="text" placeholder="Оценка" name="grade">
                <input type="submit" value="Изпрати заявката"></input>
            </form>
            </section>
        <div>

    </body>
</html>
```

Фиг. 3 Програмен код – файл index.html

В този си вид приложението има следните по-важни недостатъци:

- При връщане на отговор се отваря нова HTML страница. Това налага връщане към страницата index.html, за да генерираме нова заявка.
- Липсва стилово форматиране.

Добре би било да можем да интегрираме отговора на MongoDB сървъра в страницата с която се генерира заявката. За целта е необходимо да можем да добавяме динамично

съдържание към HTML файл. Това се реализира чрез така наречените страници-шаблони. Шаблонът съдържа статичен HTML код, който не се променя и динамичен код, който се променя динамично. Съществуват множество двигатели за шаблони, например: Embedded JS (EJS), Handlebars, Nunjucks и др.

Ще използваме шаблон **EJS**, тъй като синтаксисът е много подобен на Java Server Pages (JSP). Чрез него ще може да комбинирате HTML и JavaScript код, както и да интегрирате стойностите на променливи и обекти в HTML страниците. За да инсталирате модул EJS използвайте следната команда:

```
npm install ejs --save
```

След това трябва да зададем изгледа да се формира от EJS:

```
app.set('view engine', 'ejs');
```

Вече можете да използвате EJS за динамично вмъкване на съдържание в HTML документ. Всички документи, които използват EJS трябва да са с разширение ejs. Създайте папка views и запишете в нея индексния файл index.htm. След това променете разширението на файла до ejs. Ще използваме метод **render**, който е част от Express, с цел визуализиране на съдържанието на ejs файла. Синтаксисът е следния:

```
response.render('index.ejs', { results: data });
```

Метод render предава на EJS модул index.ejs JSON обект. Може да извлечете стойността на полетата от JSON обекта чрез използване на EJS израз, например:

```
<%= results.title %>
```

На мястото на <%= … %> в HTML кода се вмъква стойността на поле “title” от обект “results”. Ако трябва да интегрирате JavaScript код в HTML страница използвайте <% код %>.

За да форматираме стилово HTML страницата ще напишем необходимия CSS код. Всички допълнителни модули ще запишем в папка public. Създайте тази папка и вмъкнете в нея файла style.css. Трябва да зададем на Express да направи тази папка достъпна:

```
app.use(express.static('public'));
```

Задайте в хедъра на index.html и index.ejs, че ще се използва файл style.css с цел стилово форматиране:

```
<link rel="stylesheet" href="/styles.css" />
```

На Фиг. 4 е показан програмния код на метод **connect**. Новият код е на жълт фон. При успешно получаване на резултат се формира JSON обект results, който съдържа следните полета: status (при стойност “ok” заявката е изпълнена успешно); title (информация какво представлява заявката) и docs (резултат от сървъра). Ако заявката не може да бъде изпълнена от тялото на клауза catch отново се създава обект results, но в този случай той съдържа следните полета: status със стойност “fail” и info (съдържа текстово описание на грешката).

```
MongoClient.connect(url, options)
    .then(client => {
        const db = client.db(dbName);
        const collection = db.collection(collectionName);

        app.set('view engine', 'ejs');
        app.use(bodyParser.urlencoded({ extended: true }));
        app.use(bodyParser.json());
        app.use(express.static('public'));

        app.get('/', (request, response) => {
          response.sendFile(__dirname + '/index.html')
        });
        app.post('/query', (request, response) => {
            let qSubject = request.body.subject;
            let qGrade = request.body.grade;
            console.log(`${qSubject}: ${qGrade}`);
            const filter = {
              grade: {
                subject: `${qSubject}`,
                value: parseInt(`${qGrade}`)
              }
            };
            const projection = { _id: 0, name: 1 };
            const sort = { name: 1 };
            collection
                .find(filter, { projection: projection, sort: sort } )
                .toArray()
                .then(docs => {
                    let title =
                    `Студенти с оценка ${qGrade} по дисциплината
                                                ${qSubject}:`;
                    let results = {title:title,docs:docs,status:"ok"};
                    response.render('index.ejs', { results: results });
            })
            .catch(error => {
                let results = {info:`Грешка:${error.message}`,
                                                status: "fail" };
                response.render('index.ejs', { results: results });
            });

        });
        // Listening on server port
        app.listen(SERVER_PORT, () => {
            console.log(`Start listening on port ${SERVER_PORT}...`)
        });
    })
    .catch(error => {
        console.log(error.message);
    });
```

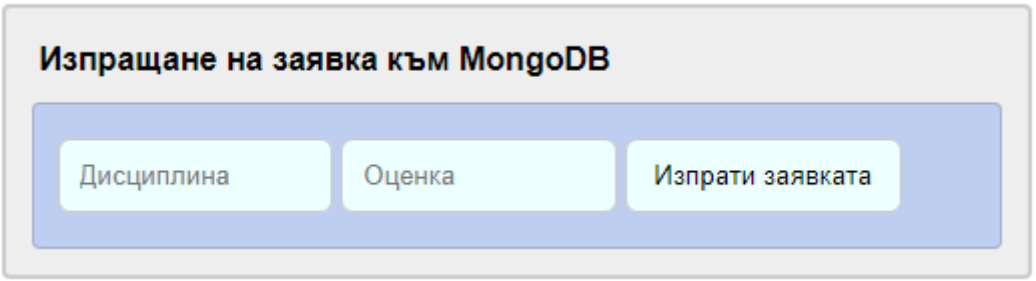
Фиг. 4 Програмен код на метод connect

На Фиг. 5 е показан програмния код от файл index.ejs. Съдържанието, което се визуализира, е в тялото на контейнер. При стойност на поле “status” е “ok” се обхождат всички документи от поле docs чрез метод **forEach**. При стойност “fail” на поле “status” се визуализира информация за грешката (параграф от клас “error”).

```
<!DOCTYPE html>
<html lang="en">
     <head>
        <meta charset="UTF-8">
       <title>Query MongoDB</title>
       <link rel="stylesheet" href="/styles.css" />
     </head>
     <body>
        <div id="container">
          <header>
            Изпращане на заявка към MongoDB
          </header>
          <section data-position="query">
            <form action="/query" method="POST">
              <input type="text" placeholder="Дисциплина"
                                                    name="subject">
              <input type="text" placeholder="Оценка" name="grade">
              <input type="submit" value="Изпрати заявката"></input>
            </form>
          </section>
          <section data-position="results">
            <% if(results.status == "ok") { %>
              <header><%= results.title %></header>
              <ul class="results">
                <% results.docs.forEach(doc => { %>
                  <li class="results">
                    <%= doc.name.firstName%> <%= doc.name.lastName%>
                  </li>
                <%}); %>
              </ul>
              <% }
              else { %>
                <p class="error"><%= results.info %></p>
              <%} %>
          </section>
        </div>
     </body>
</html>
```

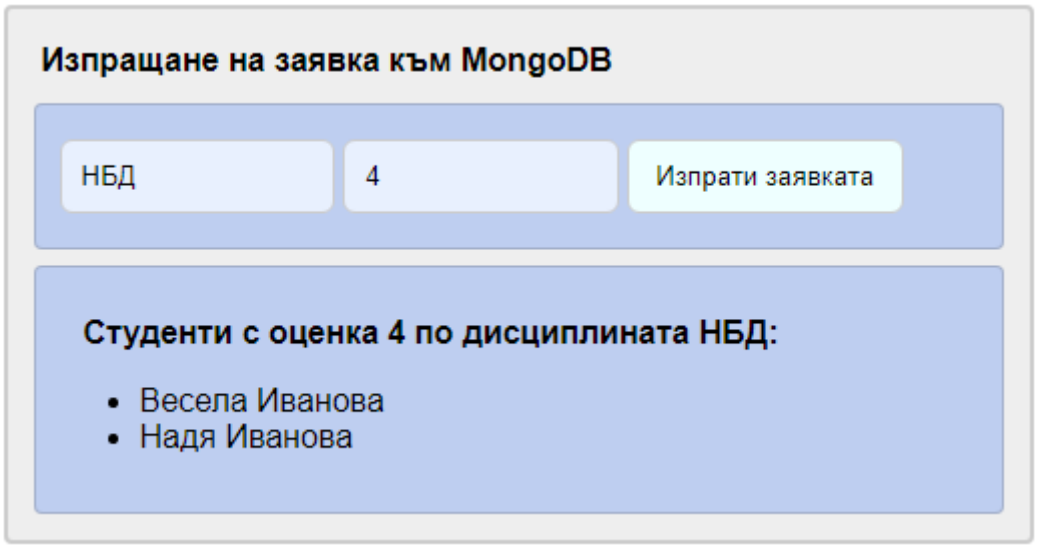
Фиг. 5 Програмен код– файл index.ejs

Стартирайте server.js и от браузъра изпълнете сляпа GET заявка (http://localhost:8080). Тя ще предизвика зареждане на съдържанието на файл index.html (виж Фиг. 6).



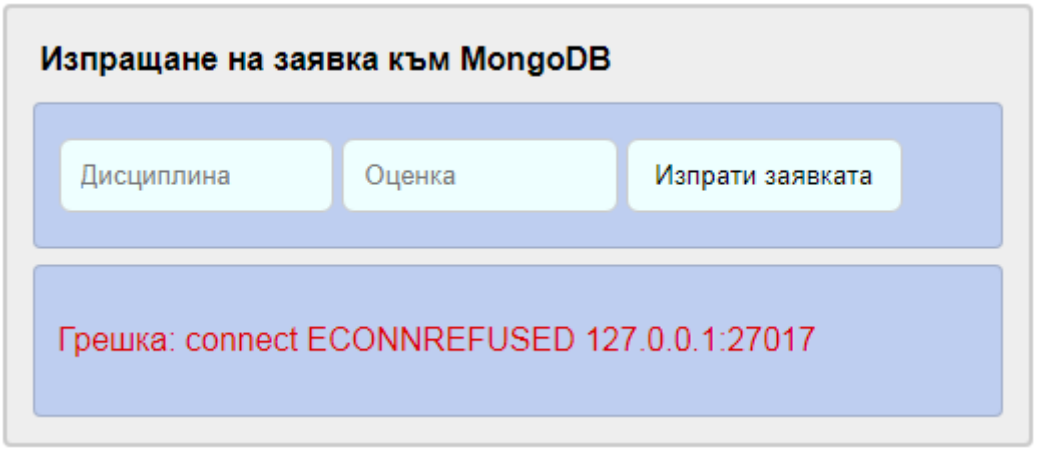
Фиг. 6 Изглед след сляпа GET заявка

Въведете двойката дисциплина и оценка и натиснете бутон “Изпрати заявката”. Трябва да бъдат намерени и разпечатани сортирано имената на студентите, които имат зададената оценка за зададената дисциплина (виж Фиг. 7).



Фиг. 7 Изглед след успешно изпълнена POST заявка

Спрете временно MongoDB сървъра и генерирайте отново заявката. Трябва да получите резултат, подобен на този, показан на Фиг. 8.



Фиг. 8 Изглед след неуспешно изпълнена POST заявка