



لایه دسترسی به داده برای **NodeJS** به عنوان یک سرویس
گام اول

علی عسگری

فروردین 1401



عنوان : لایه دسترسی به داده برای NodeJS به عنوان یک سرویس ، گام اول

نگارش : علی عسگری

نام درس : معماری نرم افزار

استاد درس : دکتر مصطفی فخر احمد

مقدمه :

ساخت یک لایه دسترسی به داده برای NodeJS و MSSQL به صورت کمینه و ساده برای برنامه‌هایی که قصد سادگی دارند در واقع هدف ما از انجام این تمرین است. باید افزود که برنامه‌ها در این نوع معماری از لایه های مختلفی تشکیل شده اند از جمله ، لایه دسترسی به داده (DAL)، لایه دسترسی کسب و کار (BAL)، لایه ارائه / (PL)، لایه های API و غیره. در این تمرین ما سعی میکنیم لایه دسترسی به داده خودمان برای NodeJS و MS SQL Server ایجاد کنیم .

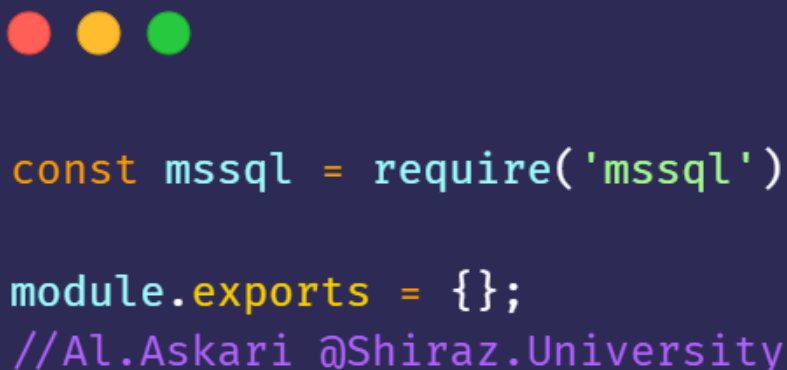
هدف این است که بیشتر بر منطق کسب و کار خود تمرکز کنیم نه زیباسازی برخی منطق دسترسی به داده ها. بنابراین اول و مهم تر از همه، رایج ترین عملیات را شناسایی میکنیم و آنها را در توابع مفیدی که قرار است تشکیل دهیم قرار میدهم.

برای رسیدن به هدف خود مراحل زیر را دنبال می کنیم:

- ایجاد ماژول دسترسی به داده
- اتصال به پایگاه داده
- انتزاع اجرای پرس و جو
- برخورد با entities
- گذاشتنش برای آینده

ایجاد سرویس دسترسی به داده

ابتدا باید یک فایل ایجاد کنیم ، پس یک ماژول جاوا اسکریپت ایجاد می کنیم که به عنوان یک لایه دسترسی به داده عمل می کند. با توجه به اینکه ما قبلا بسته mssql را با استفاده از **npm i mssql** نصب کرده ایم. کد فایل data-access.js در تصویر زیر نشان داده شده است:



```
const mssql = require('mssql')

module.exports = {};

//Al.Askari @Shiraz.University
```

تصویر 1 - ماژول mssql / فراخوانی اولیه

اتصال به پایگاه داده

برای اتصال باید پیکربندی اتصال (رشته اتصال) را تنظیم کنیم. با توجه به اینکه در آینده از متغیرهای محیطی (env) استفاده خواهیم کرد، بنابراین تابعی ایجاد خواهیم کرد که این پارامترهای پیکربندی را از فایل محیط ما (env) واکنشی می کند.

```
const poolConfig = () => ({
  driver: process.env.SQL_DRIVER,
  server: process.env.SQL_SERVER,
  database: process.env.SQL_DATABASE,
  user: process.env.SQL_UID,
  password: process.env.SQL_PWD,
  options: {
    encrypt: false,
    enableArithAbort: false
  }
});

//Al.Askari @Shiraz.University
```

تصویر 2 - اتصال به پایگاه داده

MySQL یک سیستم مدیریت پایگاه داده رابطه ای منبع باز (RDBMS) است، در حالی که MSSQL Server یک RDBMS توسعه یافته توسط مایکروسافت است. شرکت ها می توانند بین چندین نسخه MSSQL Server متناسب با نیازها و بودجه خود یکی را انتخاب کنند.

پس از این، بگذارید روی ایجاد ارتباط بین برنامه خود و پایگاه داده تمرکز کنیم. mssql یک اتصال pool را ارائه می دهد که با استفاده از آن می توانیم یک اتصال به یک پایگاه داده ایجاد کنیم و عملیات بعدی را انجام دهیم. ما یک آبجکت pool در فایل خود ایجاد می کنیم و یک تابع به نام connect ایجاد می کنیم که اگر قبلاً مقدار دهی نشده باشد، ارتباط با پایگاه داده مسئول مقداردهی اولیه این آبجکت pool خواهد بود. کد زیر را بررسی کنید:

```

let pool;

const connect = async () => {
  if (!pool) {
    pool = new mssql.ConnectionPool(poolConfig());
  }
  if (!pool.connected) {
    await pool.connect();
  }
};

//Al.Askari @Shiraz.University

```

تصویر 3 - pool و connect

در اینجا، ابتدا مطمئن می شویم که آیا شی pool تعریف شده است یا نه، اگر نه، با کمک کلاس `ConnectionPool` بسته `mssql`، شی pool را مقداردهی اولیه می کنیم. بعداً بررسی می کنیم که آیا اتصال قبلاً باز است یا نه، اگر نه، با استفاده از روش `ConnectionPool` به پایگاه داده متصل می شویم. ما همچنین الگوی `async/await Promise` را مورد استفاده قرار می دهیم.

انتزاع اجرای پرس و جو (query)

برای اجرای پرس و جو با استفاده از `mssql`، ابتدا باید یک اتصال به پایگاه داده باز کنیم، سپس به یک شی درخواست نیاز داریم که در صورت نیاز بتوانیم ورودی/خروجی را به آن متصل کنیم، سپس پرس و جو یا کوئری را اجرا می کنیم (پرس و جو برای پرس و جوهای درون خطی و اجرا برای رویه ها/توابع است). برای هر دوی این عملیات وظایف مشترک اتصال و تخصیص ورودی/خروجی شیء درخواستی است. بنابراین برای این عملیات رایج، ما یک متد اجرا می سازیم که کار فوق را انجام می دهد، اما متد `query` را بر اساس مقدار ارائه شده فراخوانی یا اجرا می کند.

متد اجرا، ورودی/خروجی هایی را که برای پرس و جو مورد نیاز است، می پذیرد. ورودی ها و خروجی ها را به صورت آرایه می پذیریم. اما باید آنها را تجزیه کنیم و با استفاده از روش ورودی و خروجی شی درخواست، این ورودی ها و خروجی ها را به شی درخواست متصل کنیم.

برای این کار ما یک تابع ایجاد می کنیم که به ما کمک می کند تا این ورودی ها و خروجی ها را تجزیه کنیم و آنها را به شی درخواست مربوطه متصل کنیم. تابع assignParams زیر این کار را با پذیرش شی درخواست و آرایه های ورودی/خروجی انجام می دهد:

```
const assignParams = (request, inputs, outputs) => {
  [inputs, outputs].forEach((params, index) => {
    const operation = index === 0 ? 'input' : 'output';
    params.forEach(param => {
      if (param.type) {
        request[operation](param.name, param.type, param.value);
      } else {
        request[operation](param.name, param.value);
      }
    });
  });
};
//Ali.Askari @Shiraz.University
```

تصویر 4 - Abstract مربوط به query

در اینجا، ما درخواست، ورودی ها و خروجی ها را به عنوان پارامتر ارائه می کنیم و با استفاده از تکرارهای نشان داده شده در بالا، روش های درخواست مربوطه را فراخوانی می کنیم.

بازگشت به تابع اجرا؛ اکنون باید به پایگاه داده متصل شویم، یک شی درخواست ایجاد کنیم، تابع assignParams را فراخوانی کنیم و متد درخواست مربوطه را یا query یا براساس نام ارائه شده اجرا کنیم:

```
const run = async (name, command, inputs = [], outputs = []) => {
  await connect();
  const request = pool.request();
  assignParams(request, inputs, outputs);
  return request[name](command);
};
//Ali.Askari @Shiraz.University
```

تصویر 5 - run function

نام عبارت است از query یا execute، فرمان درخواست ارائه شده در قالب رشته یا نام رویه/تابع و ورودی/خروجی ها پارامترها هستند. تابع run این کار را انجام می دهد، اما در نگاه اول پیچیده به نظر می رسد، زیرا ما باید نام تابع را به صراحت ارائه کنیم و اگر مجبور باشیم این کار را بارها و بارها از روی منطق تجاری خود انجام دهیم، بسیار خطرناک است، زیرا یک نام کمی اشتباه منجر می شود. ما به اشتباه بنابراین به جای نمایش تابع run، کوئری جداگانه ایجاد می کنیم و تابعی را اجرا می کنیم که به صورت داخلی فقط تابع run را فراخوانی می کند اما از لایه دسترسی به داده ما. کد زیر را بررسی کنید:

```
const query = async (command, inputs = [], outputs = []) => {
  return run('query', command, inputs, outputs);
};

const execute = async (command, inputs = [], outputs = []) => {
  return run('execute', command, inputs, outputs);
};

//Ali.Askari @Shiraz.University
```

تصویر 6 - query و execute

هر دو تابع بالا، کاری که انجام میدهند فراخوانی تابع `run` است و این مسئله بسیار صریح است.

برخورد با `entity` ها

پرس و جو و اجرا می تواند عدالت را برای لایه دسترسی به داده ما انجام دهد، اما در دنیای جاوا اسکریپت، ما داده ها را بر حسب اشیاء دریافت می کنیم و نه به صورت آرایه ای با فرمت خاص که توسط پارامترهای ورودی (نام، مقدار، نوع) لازم است. بنابراین اجازه دهید تابعی ایجاد کنیم که بتواند به ما در تبدیل شیء دریافتی به یک آرایه ورودی کمک کند. برای این تبدیل، یک تابع به نام `fetchParam` ایجاد کرده ایم که پارامترهای ورودی را از موجودیت شیء داده شده دریافت می کند.

```
const fetchParams = entity => {  
  const params = [];  
  for (const key in entity) {  
    if (entity.hasOwnProperty(key)) {  
      const value = entity[key];  
      params.push({  
        name: key,  
        value  
      });  
    }  
  }  
  return params;  
};  
  
//Ali.Askari @Shiraz.University
```

تصویر 7- `fetchParam`

به سادگی با استفاده از حلقه `for in` majestic می توانیم شیء داده شده را به آرایه ای از نوع خود (نام و مقدار) تبدیل کنیم.

اکنون می توانیم به سادگی یک `wrapper` جدید برای توابع اجرا برای سناریوی موجودیت ها مانند شکل زیر ایجاد کنیم:

```

const queryEntity = async (command, entity, outputs = []) => {
  const inputs = fetchParams(entity);
  return run('query', command, inputs, outputs);
};

const executeEntity = async (command, entity, outputs = []) => {
  const inputs = fetchParams(entity);
  return run('execute', command, inputs, outputs);
};

//Ali.Askari @Shiraz.University

```

تصویر 8 - query , execute

فقط یک گام ساده قبل از فراخوانی تابع اجرای واقعی هم برای تابع کوئری و هم برای اجرا باقی مانده است.

همانطور که در بالا نوشته شد، یک لایه حداقل دسترسی به داده حداقل کارها را انجام می دهد، و بقیه را باید برای منطق کد سفارشی با ایجاد لایه دسترسی به داده خود برای تغییرات بعدی در صورت نیاز بسط دهیم. برای چنین سناریویی، موردی را در نظر بگیرید که در آن باید پارامترهای با ارزش جدول را به یک رویه ذخیره شده ارسال کنیم. تابع `execute` ما این کار را به خوبی انجام می دهد تا هر رویه ارائه شده را فراخوانی کند، اما فاقد پشتیبانی برای تولید ورودی به عنوان جدول است، بنابراین برای این منظور می توانیم تابعی ایجاد کنیم که وظیفه ایجاد جدول `SQL` را برای ما انجام دهد و بعداً می توانیم این جدول را برای دسترسی به داده های ما ارائه می کنیم و تابع را اجرا می کنیم.

```

const generateTable = (columns, entities) => {
  const table = new mssql.Table();

  columns.forEach(column => {
    if (column && typeof column === 'object' && column.name &&
column.type) {
      if (column.hasOwnProperty('options')) {
        table.columns.add(column.name, column.type,
column.options);
      } else {
        table.columns.add(column.name, column.type);
      }
    }
  });

  entities.forEach(entity => {
    table.rows.add(... columns.map(i => entity[i.name]));
  });

  return table;
};

//Ali.Askari @Shiraz.University

```

تصویر 9- generateTable

در اینجا، ما ستون ها و موجودیت ها را به تابع generateTable منتقل می کنیم. ستون ها آرایه ای از ستون ها خواهند بود که حاوی ویژگی های نام، نوع و گزینه ها هستند و با تکرار آن ها، موجودیت ها به ردیف ها اضافه می شوند. ما باید نام موجودیت و ستون را یکسان نگه داریم تا بتوانیم تکرار سطر مان را عملی کنیم.

اکنون اجازه دهید فانکشن هایی را که باید در معرض دید خارجی قرار دهیم، export کنیم.

```
module.exports = {  
  pool,  
  mssql,  
  connect,  
  query,  
  queryEntity,  
  execute,  
  executeEntity,  
  generateTable  
};  
  
//Ali.Askari @Shiraz.University
```

تصویر 10 - export functions

ما شی pool و حتی شی mssql را نیز در معرض نمایش قرار می دهیم. افشای شی mssql عمل سالمی و امنی نیست، اما فقط برای اینکه لایه را حداقل و سبک نگه داریم و اجازه دهیم کد خارجی مستقیماً روی آن کار کند، آن را در بخش export قرار می دهیم.

وقت آن است که لایه دسترسی به داده های خود را در کد واقعی خود مورد استفاده قرار دهیم.

خلاصه آنچه تاکنون پیمودیم

تا کنون ما یک لایه ایجاد کرده ایم که به ما کمک می کند تا عملیات پایه و رایج پایگاه داده را به سرعت انجام دهیم. در بخش پیش رو، چنین عملیات و سناریوهای مختلف را بررسی خواهیم کرد و آزمایش خواهیم کرد که سرویس دسترسی به داده ما واقعاً چگونه عمل می کند. این دسترسی به داده نمونه ای از این است که چگونه می توانیم ابزارهای خود را برای حل مشکلات بزرگتر به دست آوریم یا بسازیم، اما نکته مهم این است که خود نباید به یک مشکل تبدیل شود. بنابراین بسیار ظریف است و قطعاً به بهبود مستمر نیاز دارد تا بتواند به تمام نیازهای پروژه ما رسیدگی کند.

عملیات های CRUD

- Read All Employees
- Read Single Employee
- Create Employee
- Update Employee
- Delete Employee

```
router.get('/', async (req, res) => {
  try {
    const result = await dataAccess.query(`SELECT * FROM Employee ORDER BY Id DESC`);
    const employees = result.recordset;

    res.json(employees);
  } catch (error) {
    res.status(500).json(error);
  }
});

//Ali.Askari @Shiraz.University
```

تصویر 11- Read All Employees

در اینجا، تابع `query` دسترسی به داده، شیء نتیجه را که حاوی مجموعه رکوردها، مجموعه رکوردها، خروجی ها و خصوصیات `rowsAffected` است را برمی گرداند. مجموعه رکورد در مورد فوق، سوابق کارمند را برمی گرداند.

```

router.get('/:id', async (req, res) => {
  try {
    const result = await dataAccess.query(`SELECT * FROM Employee WHERE Id = @Id`, [
      { name: 'Id', value: req.params.id }
    ]);
    const employee = result.recordset.length ? result.recordset[0] : null;

    if (employee) {
      res.json(employee);
    } else {
      res.status(404).json({
        message: 'Record not found'
      });
    }
  } catch (error) {
    res.status(500).json(error);
  }
});

//Ali.Askari @Shiraz.University

```

تصویر 12 - Read Single Employee

در اینجا، تابع پرس و جو دسترسی به داده، آرایه پارامترهای پرس و جو و ورودی را می پذیرد. پرس و جو شیء کارمند را بر اساس شناسه کارمند برمی گرداند.

```

router.post('/', async (req, res) => {
  try {
    const result = await DataAccess.query(`
      INSERT INTO Employee (Code, Salary, Job, Department, Name)
      OUTPUT inserted.Id
      VALUES (@Code, @Salary, @Job, @Department, @Name);
    `, [
      { name: 'Code', value: req.body.Code },
      { name: 'Salary', value: req.body.Salary },
      { name: 'Job', value: req.body.Job },
      { name: 'Department', value: req.body.Department },
      { name: 'Name', value: req.body.Name },
    ]
    );
    const employee = req.body;
    employee.Id = result.recordset[0].Id;

    res.json(employee);
  } catch (error) {
    res.status(500).json(error);
  }
});

//Ali.Askari @Shiraz.University

```

تصویر 13 - Create Employee

در اینجا تابع query یک رکورد کارمند در پایگاه داده ایجاد می کند.

حتی یک راه بهتر از روش بالا برای مقابله با عملیات insert وجود دارد که به وسیله آن می توانیم مستقیماً شیء را که ویژگی ها را به ستون نگاشت می کند، ارسال کنیم و بر اساس آن پرس و جو را با پارامترهای ورودی ایجاد کنیم. اما نام ویژگی باید با نام ستون یکی باشد. برای این ما تابع queryEntity داریم، کد زیر را بررسی کنید:

```
const result = await dataAccess.queryEntity(`
    INSERT INTO Employee (Code, Salary, Job, Department, Name)
    OUTPUT inserted.Id
    VALUES (@Code, @Salary, @Job, @Department, @Name);
`, req.body);

//Ali.Askari @Shiraz.University
```

تصویر 14 - Update Employee

حال در ادامه به واسطه قطعه کد زیر، تابع `queryEntity` دسترسی به داده پرس و جو و شی ورودی را می پذیرد. تابع `queryEntity`، فرآیند `mapping` را بر اساس شیء کارمند ارائه شده انجام می دهد.


```

router.put('/:id', async (req, res) => {
  try {
    if (+req.params.id !== req.body.Id) {
      res.status(400).json({
        message: 'Mismatched identity'
      });
      return;
    }

    const result = await dataAccess.query(`SELECT * FROM Employee WHERE Id = @Id`, [
      { name: 'Id', value: req.params.id }
    ]);

    let employee = result.recordset.length ? result.recordset[0] : null;
    if (employee) {
      await dataAccess.queryEntity(`
        UPDATE Employee SET
          Code = @Code,
          Salary = @Salary,
          Job = @Job,
          Department = @Department,
          Name = @Name
        WHERE Id = @Id;
      `, req.body
    );

    employee = { ...employee, ...req.body };

    res.json(employee);
  } else {
    res.status(404).json({
      message: 'Record not found'
    });
  }
} catch (error) {
  res.status(500).json(error);
}
});

//Ali.Askari @Shiraz.University

```

```

router.delete('/:id', async (req, res) => {
  try {
    const result = await dataAccess.query(`SELECT * FROM Employee WHERE Id = @Id`, [
      { name: 'Id', value: req.params.id }
    ]);

    let employee = result.recordset.length ? result.recordset[0] : null;
    if (employee) {
      await dataAccess.query(`DELETE FROM Employee WHERE Id = @Id`, [
        { name: 'Id', value: req.params.id }
      ]);
      res.json({});
    } else {
      res.status(404).json({
        message: 'Record not found'
      });
    }
  } catch (error) {
    res.status(500).json(error);
  }
});

//Ali.Askari @Shiraz.University

```

تصویر 16 - Delete Employee

در اینجا، ابتدا بررسی می‌کنیم که آیا سابقه کارمند بر اساس شناسه کارمند وجود دارد یا خیر. برای عملیات حذف، شناسه کارمند را به عنوان پارامتر ورودی برای انجام درخواست حذف ارائه می‌کنیم.

خب در این نقطه کار ما در گام اول به پایان می‌رسد اما در ادامه این پروژه پیاده سازی شده را توسعه می‌دهیم.

پایان گام اول

با احترام و تشکر به خاطر وقتی که برای مطالعه گذاشتید.
 همچنین از کارکرد سیستم ویدیوی کوتاهی در فایل پروژه قرار گرفته است.
 علی عسگری - فروردین 1400