



لایه دسترسی به داده برای NodeJS به عنوان یک سرویس  
گام چهارم ، بهینه سازی ، دینامیک سازی و راه اندازی وب سرویس

علی عسگری

اردیبهشت 1401



عنوان : لایه دسترسی به داده برای NodeJS به عنوان یک سرویس  
گام چهارم ، بهینه سازی و داینامیک سازی و راه اندازی وب سرویس

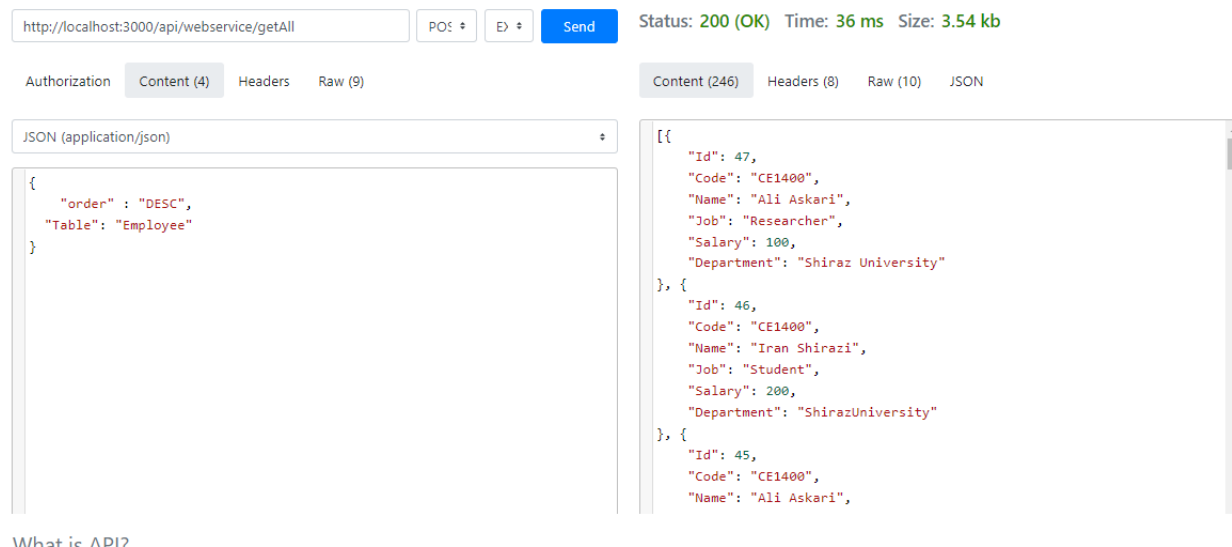
نگارش : علی عسگری

نام درس : معماری نرم افزار

استاد درس : دکتر مصطفی فخر احمد

## مقدمه :

حذف وابستگی ها و یا در واقع داینامیک سازی به طور گسترده در دستور توسعه گام چهارم قرار گرفت ، همچنین بحث بهینه سازی کدها به طرز مناسبی انجام شد و در نهایت وب سرویس ما در حال کار کردن است.



تصویر 1 - کارکرد وب سرویس

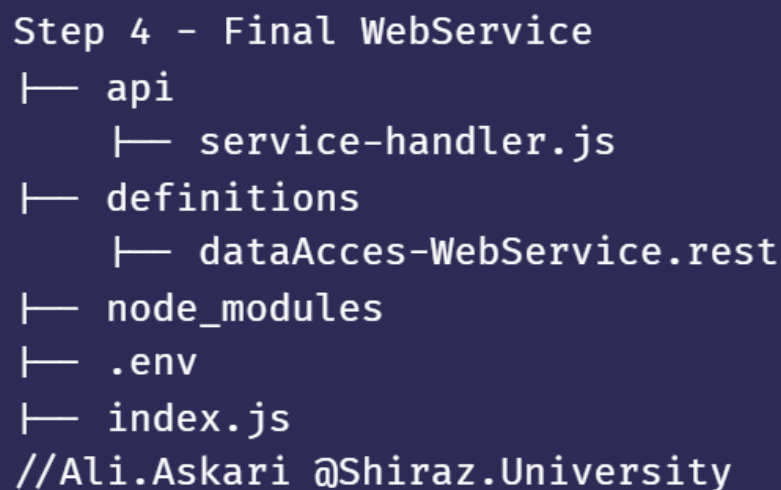
در تصویر بالا یکی از عملیات ها تست شده است، همانطور که مشاهده میکنید در بلاک سمت چپ دو مقدار مربوط به order و نام Table در چارچوب json به سمت وب سرویس ما پست شده است و وب سرویس بلاک سمت راست را برگردانده است ، عملیات در حال اجرا می باشد که وظیفه اش دریافت تمام رکورد های Table درخواستی با order درخواستی از سمت کاربر است.

همچنین مشاهده status برابر با 200 نشان دهنده اجرای بی نقص وب سرویس ماست، حال در ادامه باید بررسی کنیم که بهینه سازی ها و داینامیک سازی های انجام شده چه بوده اند و از عملیات های بهینه شده وب سرویسمان با نام جدید رونمایی کنیم.

وب سرویس ما در این مرحله هیچ وابستگی به یک Table خاص، یک عملیات خاص و یا یک operation خاص ندارد و در ادامه خواهید دید در عملیات هایی که operation کاربرد دارد آن را هم از سمت کاربر یا کلاینت دریافت میکند و سپس پاسخ مناسب می دهد.

عملیات های بهینه داینامیک سازی شده در گام چهارم :

- بهینه سازی فراخوانی به کمک ID موجود در URL
- بهینه سازی فراخوانی کل رکورد ها بر اساس ORDER دلخواه
- بهینه سازی افزودن رکورد جدید
- بهینه سازی ویرایش رکورد
- بهینه سازی حذف رکورد
- بهینه سازی ویژگی ششم ، `multiple condition SELECT`



```
Step 4 - Final WebService
├─ api
│   └─ service-handler.js
├─ definitions
│   └─ dataAcces-WebService.rest
├─ node_modules
├─ .env
├─ index.js
└─ //Ali.Askari @Shiraz.University
```

در گام نخست فایل `service-handler` را ایجاد کردیم که با `data-access.js` ما گفت و گو میکند ، در واقع `service-handler.js` هست که با دنیای بیرون و سطح و لایه ی `network` تعامل دارد ، سپس هر بار نتیجه تعاملش را با `data-access.js` در میان میگذارد ، جالب توجه است که در هیچ یک از این دو فایل ، در هیچ یکی از عملیات های اصلی `CURD` و `multiple Condition Select` هیچ بخشی هارد کد نشده و وابستگی به دیتابیس خاص ، جدول خاص ، عملیات و یا `operator` خاصی وجود ندارد و همگی به جز دیتابیس از کلاینت پس از پشت سر نهادن لایه `network` به کمک `restFul Api` دریافت می شود. لازم به ذکر است که نام و مشخصات دیتابیس که `microsoft SQL server` می باشد هم به کمک یک فایل `env` به صورت داینامیک هندل شده است.

همچنین از نتیجه گفت و گوی `service-handler` با دنیای بیرون و تعامل داینامیک و سبکش با `data-access.js` است که در فایل `data-access.js` ، کوئری به صورت داینامیک و با در نظر گرفتن احتمالات بسیار ساخته می شود. پس ما به سادگی می توانیم بگوییم یک وب سرویس داریم که از کاربر جزئیات و شرط دریافت میکنیم و کوئری مناسبش را خودمان ساخته و سپس پردازش دیتا و برگرداندن نتیجه را هم هندل می کنیم.

## 1. بهینه سازی فراخوانی به کمک ID موجود در URL

```
// 1-First Attempt of CURD for Dynamic web Service
router.post('/getOne/:id', async (req, res) => {
  try {
    const result = await dataAccess.querySelectById( Table = req.body.Table , [
      { name: 'Id', value: req.params.id , operator : req.body.operator }
    ])
  }
})
//Ali.Askari @Shiraz.University
```

تصویر 3 - بهینه سازی عملیات اول `CURD` در سرویس هندلر

تصویر بالا مربوط به سرویس هندلر است ، البته این بخشی از کد مربوط به این `router` است اما طبیعتاً بخش اصلی است ، خب کاری که در این مرحله انجام داده ایم دریافت نام `Table` و `operator` از خود کاربر

است ، کاربر سرویس ما باید نوع عملیات خودش را مثلا '=' انتخاب کند و سپس در کنار نام جدول به صورت json برای سرویس ما پست کند ، همچنین مانند سابق در این مرحله Id مربوط به رکورد مدنظر کاربر را از طریق URL دریافت میکنیم ،همانطور که ذکر کردیم سرویس هندلر برای سرویس ما گفت و گو ها و تبادل اطلاعات با کلاینت در سطح network را انجام می دهد ، حال پس از این مقادیر دریافتی از کاربر برای data-access.js ارسال می شود تا کوئری در آنجا ساخته شود و جوابش دوباره برای نمایش به همین router در سرویس هندلر برگردد.

```
//1-First Attempt data-access.js
const querySelectById = async (Table , inputs = [], outputs = []) => {
  command = `SELECT * FROM ${Table} WHERE Id ${inputs[0].operator} @Id`
  return run('query', command, inputs, outputs);
};
//Ali.Askari @Shiraz.University
```

تصویر 4 - بهینه سازی عملیات اول CRUD در دیتا اکسس

همچنین در تصویر بالا وارد فایل دیتا اکسس شدیم ، تنها تفاوتی که اینجا ایجاد کردیم این است که علاوه بر Table ، در کوئری ای که در حال ایجاد است ، operator را هم به صورت داینامیک قرار دادیم و سرویس هندلر پس از دریافت مقدارش در چارچوب آرایه ی inputs به سمت دیتا اکسس مقدار operator را ارسال میکند.

## 2. بهینه سازی فراخوانی کل رکورد ها بر اساس ORDER دلخواه

خب در این مرحله هم تفاوتی که برای فایل سرویس هندلر ایجاد کردیم ، دریافت نام Table و order دلخواه به کمک api از کلاینت است ، دقت کنید که فایل سرویس هندلر ما همان employee-variant سابق است که دستخوش تغییرات بسیار گردیده است.

```
//2- second Attempt service-handler orderby ID DESC
router.post('/getAll', async (req, res) => {
  try {
    const result = await dataAccess.querySelectByORDER(req.body.Table,
                                                         order = req.body.order);

    const response = result.recordset;
    res.json(response);
  }
});
//Ali.Askari @Shiraz.University
```

تصویر 5- بهینه سازی فراخوانی کل رکورد ها بر اساس ORDER دلخواه - گام دوم CURD در سرویس هندلر

دقت کنید که **order** همان ترتیب نمایش از ابتدا یا از انتهاست. تصویر بالا مربوط به سرویس هندلر است ، و در تصویر پایین به سراغ همین عملیات در **data-access.js** میرویم.

```
//2- second Attempt data-access.js order by ID
const querySelectByORDER = async ( Table, order, inputs = [], outputs = []) => {
  command = `SELECT * FROM ${Table} ORDER BY Id ${order}`;
  return run('query', command, inputs, outputs);
};
//Ali.Askari @Shiraz.University
```

تصویر 6- بهینه سازی فراخوانی کل رکورد ها بر اساس ORDER دلخواه - گام دوم CURD در دیتا اکسس

همانطور که در حال مشاهده هستید ، بهینه سازی ما به کمک دریافت داینامیک **Table** و **order** در تشکیل کوئری انجام شده است. پس کوئری ایجاد شده ، به جدول خاصی وابستگی ندارد و با دیتایی که کلاینت از طریق **api** به وب سرویس ما میفرستد و سرویس هندلر آن را دریافت کرده و سپس به همین دیتا اکسس ارسال می کند ، کوئری را ایجاد میکنیم.

### 3. بهینه سازی افزودن رکورد جدید

```
1 // 3rd Attempt post / insert to Table service-handler
2 router.post('/AddRecord', async (req, res) => {
3   try {
4     countConitions = Object.keys(req.body).length-2
5     operator = req.body.operator
6     let inputs = []
7     function Display(number ,arr) {
8       for (let i = 0; i < number ; i++) {
9         arr.push({ name: Object.keys(req.body)[i],
10                   value: Object.values(req.body)[i], operator})
11       }
12     }
13     Display(countConitions,inputs)
14     const result = await dataAccess.queryInput(
15       req.body.Table,
16       inputs
17     );
18
19     const record = req.body;
20     record.Id = result.recordset[0].Id;
21     res.json({message: 'Record Added Successfully'});
22 //Ali.Askari @Shiraz.University
```

تصویر 7 - درج رکورد جدید ، کد مربوط به سرویس هندلر ، عملیات سوم CURD

خب در اینجا دست به دامان خلاقیت بیشتر و تغییر بیشتری شده ایم، اول اینکه متغیری داریم با نام countConiditions که وظیفه اش شمارش تعداد مقادیر است که کلاینت به واسطه api قصد دارد به جدول ما به عنوان یک رکورد جدید تحمیل کند ، این عدد از شمارش همه object key ها منهای دو به دست آمده است ، این عدد دو به این خاطر است که نام Table و operator را جزو مقادیری که به دیتابیس و جدول وارد میکنیم نمی شماریم.



سپس یک فانکشن طراحی میکنیم که تعداد دیکشنری‌هایی که در مراحل و گام‌های قبلی به صورت استاتیک طراحی کرده بودیم را، به صورت داینامیک محاسبه کند و در واقع به صورت داینامیک دیکشنری‌های قرار گرفته در آرایه `inputs` که برای ارسال به `data-access.js` به کار می‌رود را ایجاد کرده و سپس این تابع که `Display` نام دارد، نتیجه‌اش را با نام `inputs` که یک آرایه متشکل از تعداد دیکشنری تشکیل شده از مقادیر دریافتی از کاربر هست را درون ورودی عملیات و فانکشن مربوط به `data-access.js` که در سرویس هندلر آن را فراخوانی کردیم قرار می‌دهیم.

```
1 // 3rd Attempt post / insert to Table service-handler
2 function Display(number ,arr) {
3     for (let i = 0; i < number ; i++) {
4         arr.push({ name: Object.keys(req.body)[i], value: Object.values(req.body)[i], operator})
5     }
6 }
7 Display(countConitions,inputs)
8 const result = await dataAccess.queryInput(
9     req.body.Table,
10    inputs
11 );
12 //Ali.Askari @Shiraz.University
```

تصویر 8 - فانکشن `Display` و نحوه استفاده، در سرویس هندلر

این فانکشن بسیار مهم هست چرا که در ادامه باز هم از این فانکشن استفاده خواهیم کرد. کاری که انجام می‌دهد ساخت داینامیک آرایه زیر است و ما را از هارد کد کردن آرایه زیر برای ارسال به `data-access.js` نجات می‌دهد.

```
1 // 3rd Attempt post / insert to Table service-handler
2 [
3     { name: 'Code', value: req.body.Code },
4     { name: 'Salary', value: req.body.Salary },
5     { name: 'Job', value: req.body.Job },
6     { name: 'Department', value: req.body.Department },
7     { name: 'Name', value: req.body.Name },
8 ]
9 //Ali.Askari @Shiraz.University
```

تصویر 9- آرایه تقریباً هارد کد شده که قبلاً مورد استفاده بود و دیگر مورد استفاده نیست

البته لازم به ذکر است به هر کدام از این دیکشنری ها مقدار داینامیک **operator** هم افزوده شده است.

```
1 // 3rd Attempt post / insert to Table service-handler
2 {name: Object.keys(req.body)[i], value: Object.values(req.body)[i], operator}
3 //Ali.Askari @Shiraz.University
```

تصویر 10 - به لاین بالا که اساس ایده اجرا شده است دقت کنید

خب از به سراغ دیتا اکسس جی اس برای عملیات درج رکورد میرویم ، ببینیم فانکشنی که این مقادیر را دریافت میکند چگونه دچار تغییر و بهینه شدن شده است.

خب به تصویر 11 دقت کنید ، ما در حال ایجاد کوئری به صورت گسترده و بدون وابستگی به تعداد مقادیر یا ستون های وارد شده توسط کاربر هستیم ، به نام Table هم وابستگی نداریم و همه این ها به صورت داینامیک هندل شده است.

چالش اینجا تعداد مقادیر وارد شده توسط کاربر است که میخواهد با توجه به پذیرش Table مد نظرش ، مقادیر مد نظرش را به دیتابیس وارد کند ، مشکل این است که Table ها مختلف میتوانند تعداد ستون ها و مقادیر متفاوتی را پذیرا باشند، پس این یک چالش است.

کاری که ما انجام دادیم این است که به کمک if و else ها تنظیم شده ، ساخت کوئری برای جدول هایی که از 1 تا 7 مقدار برای یک رکورد میپذیرند را به صورت داینامیک هندل کردیم. در واقع این مشکل حل شده است چرا که وب سرویس طراحی شده می تواند با توجه به دانشی که نسبت به جدول هایی در دیتابیس نهاده شده ، تا جایی که نیاز است بر این else if ها بیفزاید و به سادگی با تبدیل عدد از 7 به 8 یا بیشتر ، احتمالات بیشتری را هندل کند.

ما میدانیم که جدول فعلی مان 6 مقدار برای یک رکورد میگیرد اما به این دانسته بسنده نکردیم و حالت های بین 1 تا 7 رکورد را هندل کردیم ، همانطور که ذکر کردم ، با کپی و پیست یک else if برای هندل کردن این عملیات متناسب با جدول فرضی با بیش از 7 ستون، میتوانیم داینامیک سازی را توسعه دهیم. در تصویر زیر شرط 1 ستونه بودن و 7 ستونه بودن جدول را مشاهده میکنید، بقیه شرط ها در تصویر زیر قرار داده نشده اند.

```

1 // 3rd Attempt post / insert to Table service-handler
2 const queryInput = async ( Table , inputs = [], outputs = []) => {
3     if(inputs.length =1){
4         command = `
5         INSERT INTO ${Table} (${inputs[0].name})
6         OUTPUT inserted.Id
7         VALUES (@${inputs[0].name});
8         `
9
10
11
12
13
14         else if(inputs.length =7){
15             command = `
16             INSERT INTO ${Table} (${inputs[0].name}, ${inputs[1].name},
17             ${inputs[2].name}, ${inputs[3].name},    ${inputs[4].name},
18             ${inputs[5].name} ,${inputs[6].name} )
19             OUTPUT inserted.Id
20             VALUES (@${inputs[0].name}, @${inputs[1].name},
21             @${inputs[2].name}, @${inputs[3].name}, @${inputs[4].name}
22             @${inputs[5].name},${inputs[6].name});
23         `
24
25         //Ali.Askari @Shiraz.University

```

تصویر 11 - data-access.js و بحث درج رکورد

#### 4. بهینه سازی ویرایش رکورد

ابتدا به سراغ مذاکره کننده مان با لایه network میرویم ،خب در اینجا هم از همان فانکشن Display استفاده میکنیم ، operator و نام Table و همچنین مقادیری که میخواهد ویرایش کند از کاربر به کمک api دریافت می شود.

```

1 // Update 4rth Attempt service-handler
2 router.put('/update/:id', async (req, res) => {
3   try {
4     if (+req.params.id !== req.body.Id) {
5       res.status(400).json({
6         message: 'Mismatched identity'
7       });
8       return;
9     }
10
11     const result = await dataAccess.querySelectById( req.body.Table , [
12       { name: 'Id', value: req.params.id ,operator : '=' }
13     ]);
14
15     let record = result.recordset.length ? result.recordset[0] : null;
16     if (record) {
17       countConitions = Object.keys(req.body).length-2
18       operator = req.body.operator
19       let inputs = []
20       function Display(number ,arr) {
21         for (let i = 0; i < number ; i++) {
22           arr.push({ name: Object.keys(req.body)[i], value:
23             Object.values(req.body)[i], operator})
24         }
25       }
26       Display(countConitions,inputs)
27       console.log(inputs)
28       await dataAccess.queryUpdate(
29         req.body.Table,
30         inputs
31       );
32
33       response = { ... record, ... req.body };
34
35       res.json(response);
36
37       //Ali.Askari @Shiraz.University

```

تصویر 12 - router مربوط به update در سرویس هندلر

ابتدا رکورد را به واسطه آی دی که از URL میگیریم پیدا میکنیم ، سپس در صورت موجود بودنش به کمک req.body نام Table و operator مد نظر کاربر را دریافت میکنیم،

شروط یا مقادیر وارد شده برای اصلاح رکورد توسط کاربر ، متناسب با تعدادی متغیر یا آبجکتی که کاربر وارد کرده است ، توسط Display function به صورت داینامیک و در لحظه ایجاد میشود ، پس ما نگران ارسال اطلاعات توسط inputs به data-access نیستیم چرا که inputs اتوماتیک مقدار دهی میشود و برای هر آبجکت اولیه ای که کاربر وارد کرده یک دیکشنری متشکل از {name , value , operation} ایجاد میشود و در هر خانه آرایه inputs یکی از این دیکشنری ها قرار می گیرد.

```
1 //This code is Used for Update
2 const queryUpdate = async ( Table ,inputs = [], outputs = []) => {
3   if(inputs.length === 2){
4     command = `
5       UPDATE ${Table} SET
6       ${inputs[1].name} ${inputs[1].operator} @${inputs[1].name}
7       WHERE ${inputs[0].name} ${inputs[0].operator} @${inputs[0].name};
8     `
9   }
10  .
11  .
12  .
13  else if (inputs.length === 8){
14    command = `
15      UPDATE ${Table} SET
16      ${inputs[1].name} ${inputs[1].operator} @${inputs[1].name},
17      ${inputs[2].name} ${inputs[2].operator} @${inputs[2].name},
18      ${inputs[3].name} ${inputs[3].operator} @${inputs[3].name},
19      ${inputs[4].name} ${inputs[4].operator} @${inputs[4].name},
20      ${inputs[5].name} ${inputs[5].operator} @${inputs[5].name},
21      ${inputs[6].name} ${inputs[6].operator} @${inputs[6].name},
22      ${inputs[7].name} ${inputs[7].operator} @${inputs[7].name}
23      WHERE ${inputs[0].name} ${inputs[0].operator} @${inputs[0].name};
24    `
25  }
26  //Ali.Askari @Shiraz.University
```

تصویر 13 - عملیات آپدیت در data-access.js

خب مانند مرحله قبل عمل کردیم و همچنان از if , else برای پوشش دادن احتمالات متناسب با تعداد ستون ها استفاده کردیم ، توجه کنید که ما در حال ساخت یک ORM نیستیم ، پس قطعا ما میدانیم وب

سرویس ما متشکل از چه Table هایی با چه تعداد ستون هایی خواهد بود ، اما در هر حال تا اینجا پیش آمده ایم :)  
قطعه کد بالا تنها دو تا از شرط ها را نمایش میدهد و بقیه در فایل کد موجود هستند ، به علت تشابه درج نشده اند.

## 5. بهینه سازی حذف رکورد

```
1 // Delete 5th Attempt
2 router.post('/delete/:id', async (req, res) => {
3   try {
4     const result = await dataAccess.querySelectById( req.body.Table , [
5       { name: 'Id', value: req.params.id , operator : req.body.operator }
6     ]);
7     let record = result.recordset.length ? result.recordset[0] : null;
8     if (record) {
9       await dataAccess.queryDelete(req.body.Table , [
10        { name: 'Id', value: req.params.id , operator : req.body.operator }
11      ]);
12      res.json({message:'Record Deleted'});
13    }
14  }
15 }
16 //Ali.Askari @Shiraz.University
```

تصویر 14 - سرویس هندلر و router مربوط به حذف رکورد

خب ما در اینجا چالش خاصی نداریم ، آی دی رکورد را از URL میگیریم ، نام Table و operation را هم همینطور ، سپس فانکشن مربوط به عملیات حذف که توسط data-access.js توسعه داده شده است را صدا میزنیم ، با حفظ ارجاع ورودی های مناسب به این فانکشن.

```
1 //5th Attempt data-access.js
2 const queryDelete = async ( Table , inputs = [], outputs = []) => {
3   command = `DELETE FROM ${Table} WHERE Id ${inputs[0].operator} @Id`;
4   return run('query', command, inputs, outputs);
5 };
6 //Ali.Askari @Shiraz.University
```

تصویر 15 - data-access.js و ایجاد کوثری حذف رکورد

نکته خاصی در مورد تصویر 15 وجود ندارد جز اینکه نام Table و operation که مثلا میتواند '=' باشد به صورت داینامیک هندل شده است.

## 6. بهینه سازی فانکشن multiple Condition Select یا انتخاب با شروط چندگانه

```
1 // Custom Big Query Multiple Condition SELECT
2 router.post('/multiCond', async (req, res) => {
3   try {
4     // mines 1 is because we dont count req.body.Table
5     countConitions = Object.keys(req.body).length-2
6     operator = req.body.operator
7     let inputs = []
8     function Display(number ,arr) {
9       for (let i = 0; i < number ; i++) {
10         arr.push({ name: Object.keys(req.body)[i],
11                   value: Object.values(req.body)[i], operator})
12       }
13     }
14
15     Display(countConitions,inputs)
16     console.log(inputs)
17     if (countConitions>1) {
18       const result = await dataAccess.CustomquerySelectById(
19         Table = req.body.Table,inputs );
20       const record = result.recordset.length ? result.recordset[0] : null;
21
22       if (record) {
23         res.json(record);
24       } else {
25         res.status(404).json({
26           message: 'Record not found'
27         });
28       }
29     }
30     else {
31       order = req.body.order
32       //default order = DESC
33       if(!req.body.order) order = 'DESC'
34
35       const result = await dataAccess.querySelectByORDER(req.body.Table, order);
36       const response = result.recordset;
37       if (response) {
38         res.json(response);
39     //Ali.Askari @Shiraz.University
```

تصویر 17 - هندل کردن یک SELECT با شروط چندگانه ، سرویس هندلر

در گام قبلی برای پیاده سازی این بخش از if و else های بسیار استفاده کردیم ، ابتدا باید بگویم همچنان operator و نام Table از کاربر گرفته میشود، از تابع Display که پیش تر توضیح داده شد برای پر کردن آرایه inputs با مقادیر و شروطی که کاربر وارد میکند استفاده شده است.

در بخشی از کد در شرایط خاص اگر کاربر order را وارد نکند ، دیفالت آن را با DESC برابر میکند ، این شرط برای برگرداندن تمام دیتاهای یک Table کار میکند.

بقیه بخش ها در مراحل پیشین در همین داکيومنت توضیح داده شده است ، مانند countConitions و کاربردش در فانکشن Display. در واقع تعداد مقادیری که کاربر وارد میکند در حلقه به تشکیل یک دیکشنری برای هر آبجکت کمک میکند که این دیکشنری ها یکی پس از دیگر در همان فانکشن Display در آرایه inputs پوش میشوند. برای هر آبجکتی که کاربر وارد کرده ، مثلاً Age:20 ، اینجا یک دیکشنری متشکل از موارد زیر ایجاد میشود .

{Name : 'Age' , value : 20 , operator : '='}

```
[
  { name: 'Code', value: 'CE1400', operator: '=' },
  { name: 'Name', value: 'Ali Askari', operator: '=' },
  { name: 'Job', value: 'Researcher', operator: '=' },
  { name: 'Salary', value: 100, operator: '=' },
  { name: 'Department', value: 'Shiraz University', operator: '=' }
]
[nodemon] restarting due to changes...
[nodemon] starting `node index index.js`
Server started running on 3000 for development
```

تصویر 18 - یک لاگ از inputs array ایجاد شده توسط Display پس از اجرای عملیات

در تصویر زیر ، تصویر 19 خواهید دید که در data-access.js چگونه هندل کرده ایم بحث انتخاب با شروط چندگانه و همچنین ساخت کوئری به صورت داینامیک .

مشابه با عملیات های قبلی برای پوشش تعداد متغیر ورودی توسط کاربر و تشکیل کوئری به صورت داینامیک از یک سری if و else ها استفاده کردیم که از وارد شدن یک آبجکت توسط کاربر را تا عدد 7 پوشش میدهند ، همانطور که پیش تر ذکر شد، این محدوده به سادگی قابل افزایش است و در حال حاضر وب سرویس ما کاملاً داینامیک از کاربر نام جدول را میگیرد ، عملیات مدنظر را نیز کاربر انتخاب می کند و عملیاتش انجام شده و پاسخ می گیرد .



```

1 // Multi Condition Query USEING
2 const CustomQuerySelectById = async (Table , inputs = [], outputs = []) => {
3   if(inputs.length === 1){
4     command = `SELECT * FROM ${Table}
5     WHERE ${inputs[0].name} ${inputs[0].operator} @${inputs[0].name} `
6   }
7   .
8   .
9   .
10  else if (inputs.length === 7 ) {
11    command = `SELECT * FROM ${Table}
12    WHERE ${inputs[0].name} ${inputs[0].operator} @${inputs[0].name}
13    And ${inputs[1].name} ${inputs[1].operator} @${inputs[1].name}
14    And ${inputs[2].name} ${inputs[2].operator} @${inputs[2].name}
15    And ${inputs[3].name} ${inputs[3].operator} @${inputs[3].name}
16    And ${inputs[4].name} ${inputs[4].operator} @${inputs[4].name}
17    And ${inputs[5].name} ${inputs[5].operator} @${inputs[5].name}
18    And ${inputs[6].name} ${inputs[6].operator} @${inputs[6].name}
19    ;`
20  }
21  //Ali.Askari @Shiraz.University

```

تصویر 19 - انتخاب با شروط چندگانه - data-access.js

خب در اینجا به پایان ساخت سرویس با امکانات مد نظر طراح سوال رسیدیم. فیچرهای بیشتر هم که به کمک رویه‌ها توسعه داده شده بود نیز در گام دوم بررسی شده است. با توجه به عوض شدن api های وب سرویسمان نسبت به گام قبلی، در ادامه به بررسی آنها میپردازیم.

سرویس ما اکنون آماده است

هر چند که api ها در dataAccess-WebService.rest آمده اند ، واقع در فایل definitions ، اما من در ادامه آن ها را مطرح میکنم.

```
1 ##### Get All Records
2 POST http://localhost:3000/api/webService/getAll
3 content-type: application/json
4 {
5     "Table": "Employee",
6     "order" : "DESC"
7 }
8
9 //Ali.Askari @Shiraz.University
```

<http://localhost:3000/api/webService/getAll>

مهم : آنچه در لاین چهارم تا هفتم میبینید را شما به عنوان کلاینت یا کاربر این سرویس باید برایش ارسال کنید.

باز هم تاکید میکنم ، آنچه به رنگ سبز در چارچوب یک دیکشنری میبینید ، ورودی های سرویس ماست که کاربر باید ارسال نماید.

```

1 ##### Get One Record
2 POST http://localhost:3000/api/webService/getOne/1
3 content-type: application/json
4 {
5     "Table": "Employee",
6     "operator" : "="
7 }
8 //Ali.Askari @Shiraz.University

```

<http://localhost:3000/api/webService/getOne/1>

دقت کنید که Id به صورت عدد در URL وارد میشود، **getOne** یعنی یک رکورد برمی گرداند و ربطی به Id ندارد. مهم : متد پست است ،نام جدول و **operator** مد نظر به شکل بالا برای سرویس ارسال میگردد.

```

1 ##### Add Record
2 POST http://localhost:3000/api/webService/AddRecord
3 content-type: application/json
4 //We dont USE Id for adding new record through this URL
5 {
6     "Code": "CT2007",
7     "Name": "Tom Holland",
8     "Job": "Actor",
9     "Salary": 1000000,
10    "Department": "Hollywood",
11    "operator" : "=",
12    "Table": "Employee"
13 }
14 //Ali.Askari @Shiraz.University

```

<http://localhost:3000/api/webService/AddRecord>

مهم هست که بدانیم برای کار کردن با api مربوط به درج رکورد ، نباید Id رکورد را وارد کنیم ، Id اتوماتیک توسط سیستم تولید میگردد . همچنان دیکشنری هایی که در تصویر میبینید ورودی هستند و اغلب توسط پست یا برای آپدیت با put باید ارسال شوند ، خواهید دید که حتی متد حذف هم با پست اجرا شده است.

```
#### Update Record
PUT http://localhost:3000/api/webservice/update/44
content-type: application/json
{
  "Id": 44,
  "Code": "CT2004",
  "Name": "David Beckham",
  "Job": "Winger",
  "Salary": 3500000,
  "Department": "ManchesterUnited",
  "operator" : "=",
  "Table": "Employee"
}
//Ali.Askari @Shiraz.University
```

<http://localhost:3000/api/webservice/update/44>

دقت کنید که برای مثال ، در بحث ویرایش رکورد بالا می توانید هر کدام از آبجکت ها که میخواهید را تغییر دهید ، یا هر تعدادی که می خواهید در یک رکورد می توانید همزمان تغییر ایجاد کنید ، سپس با متد put به سرویس ما ارسال کنید تا عملیات انجام شود.

دقت نمایید که Id وارد شده در URL باید با Id موجود در دیکشنری ارسالی به سرویس برابر باشد ، برخلاف api قبلی که مربوط به درج رکورد جدید بود ، اینجا Id بسیار مهم است و هم در URL و هم در دیکشنری json باید وارد شود ، البته با یک سرچ ساده میتوان Id هر رکورد را یافت.

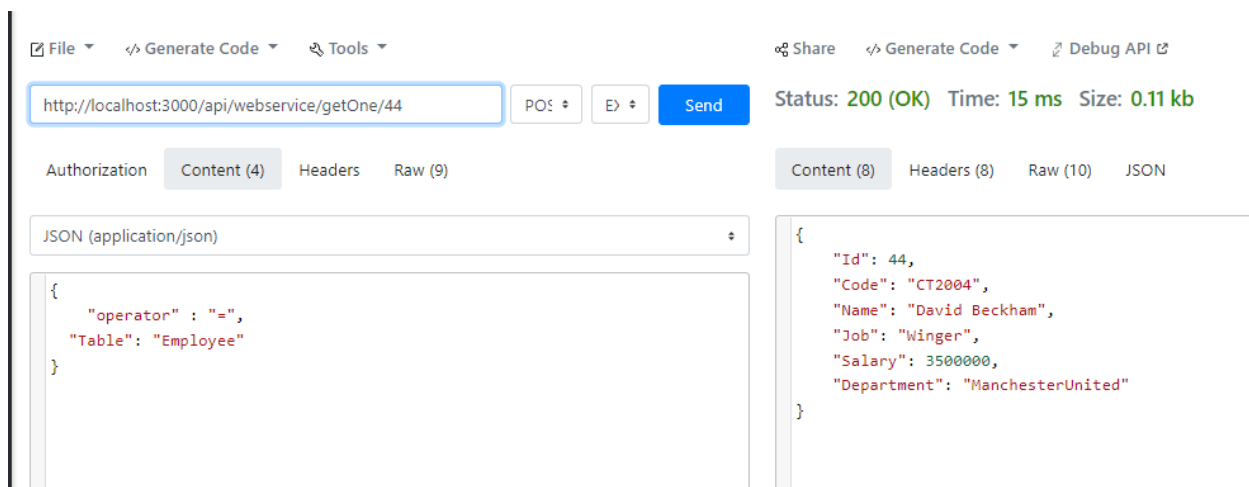
دقت کنید تمام آی دی هایی که در URL وارد میکنید باید در دیتابیس موجود باشد ، اگر موجود نبود سرچ کنید یا getAll را فراخوانی کنید تا بتوانید رکورد مد نظرتان را پیدا کنید. برای سرچ هم از multiCond

میتوان استفاده کرد و هم از خود search که یک فیچر افزون بر تمرین است و پیاده سازی شده است و مثلاً بر اساس نام میتواند جست و جو کند.

```
1 ##### Delete Record
2 POST http://localhost:3000/api/webservice/delete/42
3 content-type: application/json
4 {
5     "Table": "Employee",
6     "operator" : "="
7 }
8 //Ali.Askari @Shiraz.University
```

<http://localhost:3000/api/webservice/delete/42>

خب در تصویر بالا باز هم می بینید برای حذف هم باید operator و نام Table را مطابق تصویر برای سرویس ارسال نمایید. مسئله مهم دیگری وجود ندارد اما به ملزومات هر api توجه کنید تا سرویس به درستی کار کند، نتایج سرویس هم به شکل json برگشت داده میشوند.



نتیجه اجرای <http://localhost:3000/api/webservice/getOne/44>

○ ○ ○

```
# MULTI CONDINTIOAL SELCETION
POST http://localhost:3000/api/webservice/multiCond
content-type: application/json
{
  "Table": "Employee"
}
content-type: application/json
{
  "order" : "DESC",
  "Table": "Employee"
}
content-type: application/json
{
  "Id": 44,
  "operator" : "=",
  "Table": "Employee"
}
content-type: application/json
{
  "Id": 44,
  "Code": "CT2004",
  "Name": "Green Wood",
  "Salary": 30000,
  "operato" : "=",
  "Table": "Employee"
}
content-type: application/json
{
  "Id": 44,
  "Code": "CT2004",
  "Name": "Green Wood",
  "Job": "forward",
  "Salary": 30000,
  "Department": "Manchester United",
  "operator" : "=",
  "Table": "Employee"
}
//Ali.Askari @Shiraz.University
```

<http://localhost:3000/api/webservice/multiCond>

این api به زیبایی کار میکند، کاربر میتواند تنها نام Table را وارد کند تا این api تمام رکورد ها را برگرداند ، کاربر میتواند ترتیب را تعیین کند، operator تعیین کند و همچنین از همه مهم تر با شروط بسیار اقدام به سرچ کند ، مثلا همزمان میتواند در سرچ Department و Job را دخیل کند ،همانطور که در تصویر بالا مشاهده میکنید تمامی دیکشنری های موجود در تصویر شروطی هستند که کاربر برای سرویس میفرستد تا سرویس خروجی مناسب کاربر را برگرداند. برای مثال به یکی از خروجی های اجرا این api دقت کنید.



```
# MULTI CONDINTIOAL SELCETION
POST http://localhost:3000/api/webservice/multiCond
one response for this api
[ {
  "Id": 47,
  "Code": "CE1400",
  "Name": "Ali Askari",
  "Job": "student",
  "Salary": 100,
  "Department": "Shiraz University"
}, {
  "Id": 46,
  "Code": "CE1400",
  "Name": "reza Shirazi",
  "Job": "Student",
  "Salary": 200,
  "Department": "Shiraz University"
}, {
  "Id": 45,
  "Code": "CE1400",
  "Name": "ehsan Askari",
  "Job": "Student",
  "Salary": 200,
  "Department": "Shiraz University"
},
.
.
.
//Ali.Askari @Shiraz.University
```

همانطور که در تصویر بالا میبینید ، خروجی سرویس نیز فرمت json دارد و به صورت آرایه ای از دیکشنری ها تمام رکورد های متناسب با شروط را برمیگرداند.

حال که به اتمام کار رسیدیم ، بار دیگر api های اضافه و فیچر های اضافه ای که در گام دوم توسعه دادیم را هم یک بار دیگر اینجا به طور خلاصه مطرح میکنم و کار به اتمام می رسد ، دقت کنید که فیچر های اصلی CURD و multiple Condition Selection کاملاً مطابق Assignment و استانداردهایش پیاده سازی شده است اما این فیچر های اضافه چندان سفت و سخت پایبند به Assignment نیستند ، مثلاً operator از کاربر گرفته نمیشود.

## بررسی فیچر های اضافه

### #### Get Employees Status

```
response GET http://localhost:3000/api/webservice/Employee/status
{
  "Count": 35,
  "Max": 4050000,
  "Min": 0,
  "Average": 287702,
  "Sum": 10069570
}
//Ali.Askari @Shiraz.University
```

GET <http://localhost:3000/api/webservice/Employee/status>

آنچه در تصویر می بینید خروجی و نتیجه استفاده از api بالاست.



### #### Search Employees

```
response of GET http://localhost:3000/api/webservice/Employee/search?name=Ali Askari
[{"Id": 35,
  "Code": "CT2000",
  "Name": "Ali Askari",
  "Job": "Programmer",
  "Salary": 35000,
  "Department": "ResearchAndDevelopment"}, {"Id": 36,
  "Code": "CT5207",
  "Name": "Ali Askari",
  "Job": "Product Manager",
  "Salary": 4050,
  "Department": "Operations"}, {"Id": 45,
  "Code": "CE1400",
  "Name": "Ali Askari",
  "Job": "Student",
  "Salary": 200,
  "Department": "Shiraz University"}, {"Id": 47,
  "Code": "CE1400",
  "Name": "Ali Askari",
  "Job": "Researcher",
  "Salary": 100,
  "Department": "Shiraz University"}]
//Ali.Askari @Shiraz.University
```

GET <http://localhost:3000/api/webservice/Employee/search?name=Ali Askari>

آنچه در تصویر می بینید خروجی و نتیجه استفاده از api بالاست.

### #### Employee Summary

```
response of GET http://localhost:3000/api/webservice/Employee/search?name=Ali Askari
{
  "Department": [{
    "Department": "Tesla",
    "EmployeeCount": 1,
    "Salary": 4050000,
    "Annual": 48600000
  }, {
    "Department": "ManchesterUnited",
    "EmployeeCount": 1,
    "Salary": 3500000,
    "Annual": 42000000
  }, {
    "Department": "Hollywood",
    "EmployeeCount": 1,
    "Salary": 1000000,
    "Annual": 12000000
  }, {
    "Department": "Lecter City",
    "EmployeeCount": 1,
    "Salary": 500000,
    "Annual": 6000000
  },
  .
  .
  .
}
//Ali.Askari @Shiraz.University
```

GET <http://localhost:3000/api/webservice/Employee/summary>

آنچه در تصویر می بینید خروجی و نتیجه استفاده از api بالاست.

و در آخر یک api داریم که کمک میکند به کاربر تا بتواند به سرویس ما بگوید که بیش از یک رکورد را همزمان ایجاد کند. دقت کنید تمام آبجکت ها و دیکشنری های زیر content-type ، تاکنون و هم اکنون مربوط به آن چیز است که کاربر و کلاینت باید از طریق متد پست برای سرویس ما ارسال کند ، در اینجا هم برای مثال دیتای مربوط به دو رکورد را همزمان وارد می کند تا در دیتابیس و جدول مربوطه اضافه شوند.

#### #### Add Many Employees

POST <http://localhost:3000/api/webService/Employee/many>

content-type: application/json

```
[
  {
    "Id": 0,
    "Code": "CT8400",
    "Name": "iran irani",
    "Job": "Salesman",
    "Salary": 20000,
    "Department": "Sales"
  },
  {
    "Id": 0,
    "Code": "CT8500",
    "Name": "shirazi",
    "Job": "Salesman",
    "Salary": 20000,
    "Department": "Sales"
  }
]
```

دو رکورد بالا به جدول و دیتابیس اضافه می شوند اگر از قبل موجود نباشند. هم چنین بیش از دو رکورد را هم می شود همزمان به دیتابیس افزود. دقت کنید که فانکشنالیتی نسبت به گام دوم برای این فیچر ها تغییر نکرده ولی api ها تغییر کرده اند ، برای مثال Employee به آدرس api افزوده شده است که ما بدانیم بر روی کدام جدول در حال عملیات هستیم.

## پایان گام چهارم ، سرویس ما اکنون آماده است

با احترام و تشکر به خاطر وقتی که برای مطالعه گذاشتید.  
همچنین از کارکردِ سیستم ویدیوی کوتاهی در فایل پروژه قرار گرفته است.  
برای تست باید از ابزار هایی مثل POSTMAN استفاده نمایید، و یا ابزار های آنلاین تست api.  
علی عسگری - اردیبهشت 1400