لایه دسترسی به داده برای NodeJS به عنوان یک سرویس

گام سوم ، توسعه عملیات های CURD مطابق Assignment

علی عسگری

فروردین 1401

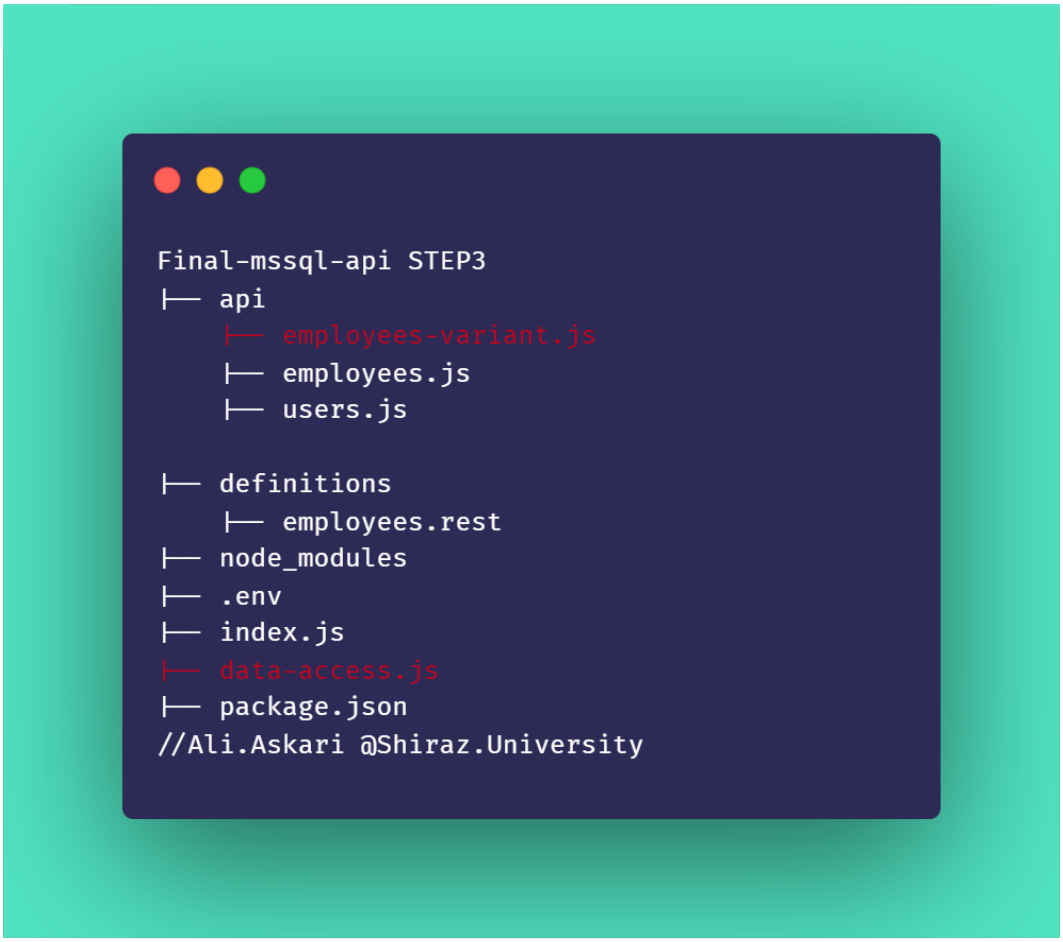
عنوان : لایه دسترسی به داده برای NodeJS به عنوان یک سرویس

گام سوم، توسعه عملیات های CURD مطابق Assignment

نگارش : علی عسگری

نام درس : معماری نرم افزار

استاد درس : دکتر مصطفی فخر احمد



تصویر O - Project Map

مقدمه :

همانطور که در تصویر بالا قابل مشاهده است ، دو فایل اصلی به رنگ قرمز در آمده اند ، فایل **data-access.js** سرویس توسعه داده شده توسط ماست و **employees-variant.js** فایل اصلی این پروژه می باشد که در حال استفاده از سرویس توسعه داده شده است. توجه کنید که employee.js در حال استفاده از سرویس توسعه داده شده نیست و خودش سعی میکند به DB متصل شود که مورد بحث ما نیست و صرفا یک پیاده سازیست. گام های طی شده در این گامِ سوم :

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

نحوه استفاده از سرویس ، در فایل employees-variant.js

توجه کنید که فایل employees.js بدون استفاده از سرویس لایه دسترسی ایجاد شده کار میکند و ما در فایل varaint هست که از سرویسمان استفاده کرده ایم، اطلاعات از طریق restfulApi رد و بدل میشود بین فایل variant و data-access.js ، پس در واقع ماژول طراحی شده برای دسترسی به دیتا واقعا یک سرویس است ، علاوه بر قابلیت حمل ، استفاده مجدد ، عدم وابستگی به اجزای دیگر پروژه و پایداری یک قابلیت فوق العاده دارد و آن این است که حتی میتوان سرور مورد استفاده برای دیتا اکسس را از سرور مورد استفاده از employees-variant.js جدا کرد .

1. فراخوانی به کمک ID موجود در URL

```
// First Attempt   employees-variant.js

router.get('/:id', async (req, res) => {
    try {
        const result = await dataAccess.querySelectById( Table = 'Employee' ,[
            { name: 'Id', value: req.params.id ,operator : '=' }
        ]);
        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
```

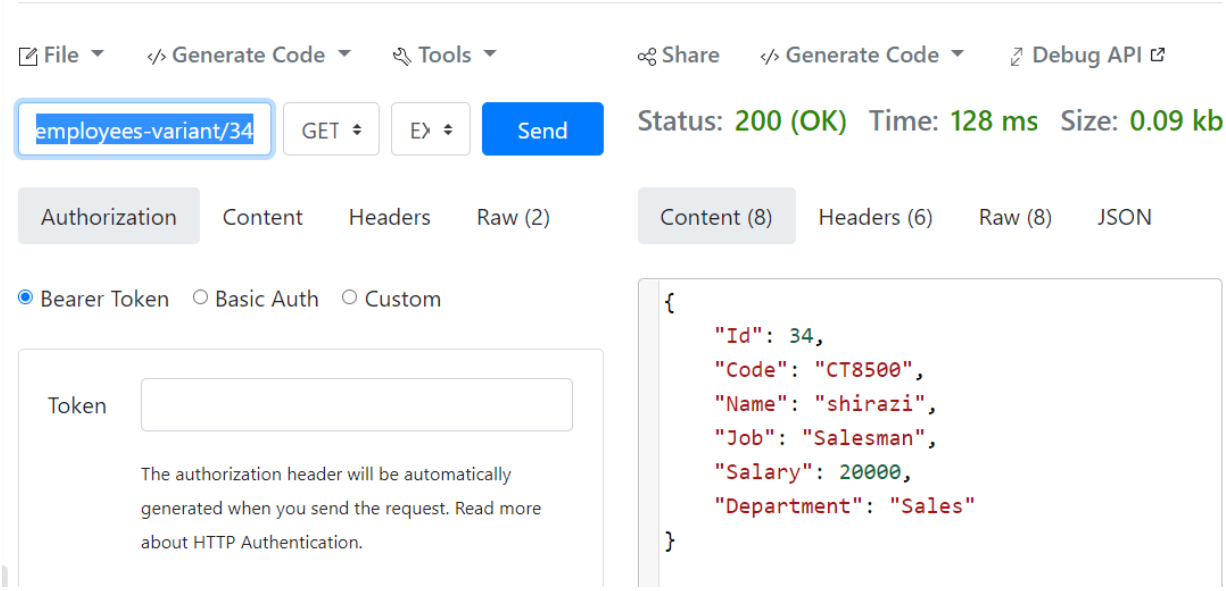
تصویر1 - تلاش اول ، استفاده از querySelectedById

همانطور که در لاین 6 ام قابل مشاهده است ورودی های تابع querySelectedById که از dataAccess فراخوانی شده است ، نام Table در کنار یک Condition از جنس Dictionary میباشد. درون این dict سه متغیر یا شیئ یا آبجکت نهادینه شده است.

```
1 // First Attempt   employees-variant.js
2
3 router.get('/:id', async (req, res) => {
4     try {
5         const result = await dataAccess.querySelectById( Table = 'Employee' ,[
6             { name: 'Id', value: req.params.id ,operator : '=' }
7
8 //Ali.Askari @Shiraz.University
```

تصویر 2 -Condition Dictionaty

متغیر name درون این دیکشنری Id متعلق به هر رکورد تعیین شده است ، value مقدار عددی Id میباشد که از URL به کمک req.params بیرون کشیده میشود و همچنین operator با مساوی یا برابر مقدار دهی شده است که در ساخت query SQL به کار میرود.



تصویر 3 - دریافت دیتای رکورد با Id مساوی با 34

- GET http://localhost:3000/api/employees-variant/34

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

تصویر 4 - فانکشن querySelectById در فایل data-access.js ، تلاش اول

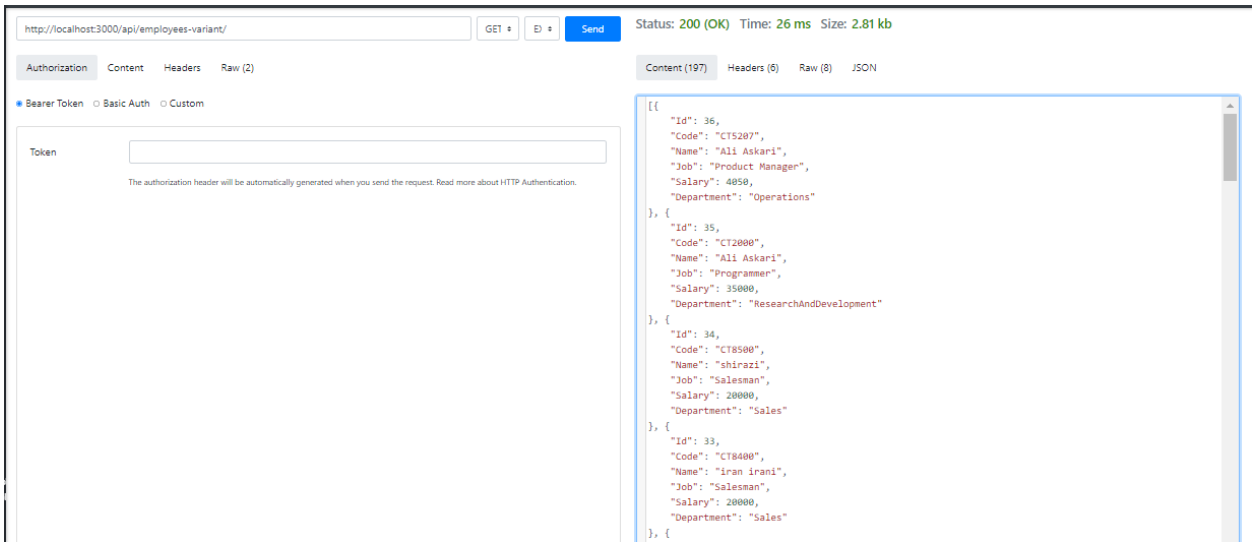
تصویر بالا در واقع نحوه هندل کردن querySelectById را نشان میدهد ، همچنین نحوه ایجاد کوئری به صورت داینامیک هم آشکار است ، امیدوارم توضیحات مربوط به فانکشن run را فراموش نکرده باشید.
لازم به ذکر است که inputes در واقع یک آرایه است که دیکشنری ارسال شده از فایل employee-variants را در بر گرفته است. همچنین دقت کنید که Table و = به صورت داینامیک هندل شده اند و مقادیرشان همانند inputs از فایل variants به اینجا ارسال شده است.

2. فراخوانی کل رکورد ها بر اساس ORDER دلخواه

```
1 //second Attempt  orderby ID DESC
2 router.get('/', async (req, res) => {
3     try {
4         const result = await dataAccess.querySelectByORDER('Employee', order='DESC');
5         const employees = result.recordset;
6 
7         res.json(employees);
8     } catch (error) {
9         console.log(error)
10        res.status(500).json(error);
11    }
12 });
13 //Ali.Askari @Shiraz.University
```

تصویر5-نحوه استفاده از فانکشن querySelectByORDER

دقت کنید که در این مورد ما ORDER را هم هندل کردیم و از فایل variant میشود برای کوئری ای که در فایل data-access.js تشکیل میشود ORDER را تنظیم کرد. همچنین مطابق قبل نام Table را ارسال میکنیم اما در این مرحله به علت اینکه در عملیات select شرط خاصی نداریم ، inputs خالی خواهد بود.



تصویر 6 - خروجی به کمک api

- #### Get All Employees
- GET http://localhost:3000/api/employees-variant

هدف برگرداندن تمام دیتا ها بود.

```
//Second Attempt data-access.js
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 7 - querySelectByORDER درون data-access.js ، تلاش دوم

3. افزودن رکورد

```
///start  new 3  variant.js
router.post('/', async (req, res) => {
    try {
        // passing input as entity
        const result = await dataAccess.queryEntityInpute(
         'Employee',
          req.body
        );

        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 8 - استفاده از queryEntityInpute درون فایل variant

خب به این مرحله رسیدیم که افزودن رکورد به دیتابیس به کمک لایه ایجاد شده یا در واقع سرویس ایجاد شده را بررسی کنیم ، ابتدا توسط کدهای بالا درخواست خودمان برای تشکیل کوئری را از فایل variant به فایل data-access.js ارسال میکنیم که توسط فانکشن مورد استفاده به نام queryEntityInpute کوئری دلخواهمان و درج رکورد دلخواهمان ساخته و انجام شود.

یک نکته بسیار مهم در این روشی که برای درج رکورد استفاده کردیم وجود دارد و آن هم این است که باید بدانیم در حال استفاده کردن از JS هستیم و قرار هست دیتایمان را از طریق req.body از کاربر بگیریم ، این دیتا میتواند از طریق یک فرم یا هر چیزی وارد شود. در واقع باید مطابق تصویر زیر inputs یا دیکشنری condition را ایجاد کنیم.

خوب به تصویر زیر نگاه کنید ، حال اگر به تصویر بالا نگاه کنید میبینید که ما مطابق تصویر زیر عمل نکرده ایم. بله ما از یکی از قابلیت های JS استفاده کرده ایم و در فایل varaint که در حال استفاده از سرویس data-access.js هست خلاقیت به خرج داده ایم ، تنها عبارت req.body خودش تمامی مقادیر وارد شده توسط کاربر را برایمان میگیرد و نیازی نیست ما به صورت دستی دیکشنری و کاندیشن را بنویسیم و برای سرویس ارسال کنیم ، تنها req.body در کنار نام Table کفایت میکند. همچنین پر واضح است که از متد POST استفاده کرده ایم.دقت کنید تماما منظور ما از variant ، فایل employee-variant.js است.

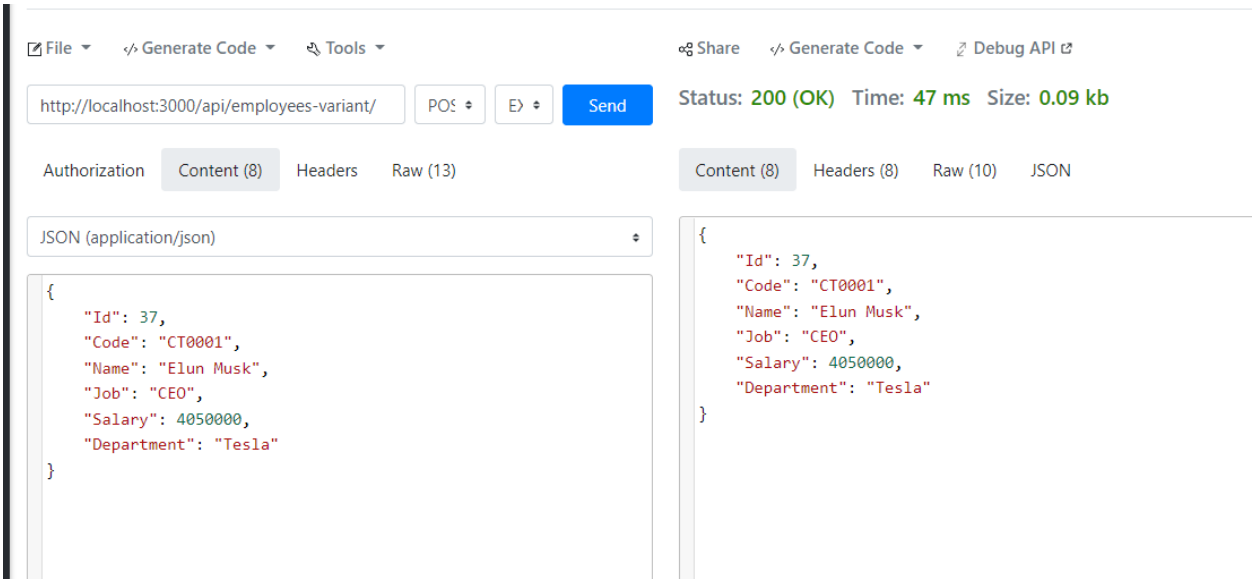
```
///start  new 3  variant.js
            [
                { name: 'Code', value: req.body.Code },
                { name: 'Salary', value: req.body.Salary },
                { name: 'Job', value: req.body.Job },
                { name: 'Department', value: req.body.Department },
                { name: 'Name', value: req.body.Name },
            ]
//Ali.Askari @Shiraz.University
```

تصویر9 - شیوه ای که در تلاش دوم ازآن استفاده نکرده ایم

بحث ما در تصویر بالا ، دریافت مقادیر از کاربر در فایل employees-variant.js است که باید آنها را هندل کنیم و از دو راه میشود برای سرویس توسعه داده شده این ها را ارسال کنیم ، اصلی ترین راه و نرمال ترین راه مانند تصویر بالاست که بیاییم برای هر شرط یا input یا مقدار وارد شده توسط کاربر یک آرایه تشکیل دهیم که هر خانه از این آرایه را یک dictionary پر کرده است و هر کدام از این dict ها حاوی یک condition یا input هستند که به هر حال مقادیریست که توسط کاربر وارد میشود ، دقت کنید که ما توسط api در این پروژه با کاربر تعامل داریم و دیتای وارد شده توسط کاربر را در فایل employees-variant.js هندل می کنیم ، سپس در این فایل variant از فانکشن های سرویسمان استفاده میکنیم به عنوان ورودی به این فانکشن ها مانند تصویر بالا یک آرایه از دیکشنری ها ارسال میکنیم که در واقع condition ها یا input های ماست و ما در سرویسمان بر این آرایه inputs نام نهادیم.

```
1 //3d Attempt   data-access.js
2 onst queryEntityInpute = async (Table, entity, outputs = []) => {
3     command =     `
4     INSERT INTO ${Table}  (Code, Salary, Job, Department, Name)
5     OUTPUT inserted.Id
6     VALUES (@Code, @Salary, @Job, @Department, @Name);
7 `
8     const inputs = fetchParams(entity);
9     return run('query', command, inputs, outputs);
10 };
11 //Ali.Askari @Shiraz.University
```

تصویر 10 - عملیات درج رکورد جدید توسط data-access.js



تصویر 11 - نتیجه درج رکورد جدید ، تلاش سوم

حال اگر به تصویر 10 با دقت بیشتر نگاه کنید ، در میابید که به نوعی وابستگی به مقادیر Employee درون command یا کوئری در سرویس ما یعنی در فایل data-access.js وجود دارد .

من حتما پس از پی بردن به این وابستگی سعی در برطرف کردنشان خواهم کرد ، منظورم لاین های 4 تا 6 تصویر 10 یعنی فانکشن queryEntityInpute میباشد .
پس برای حل این مشکل شیوه استفاده سنگین از req.body یعنی آن خلاقیت جاوااسکریپتی خود را کنار میگذاریم و به فانکشن queryInput می رویم و آن را به صورت کاملا داینامیک به کمک Inputs راه می اندازیم ، پس همانطور که مشخص است ، از Entity به inputs مهاجرت میکنیم . تصویر زیر گویای همه چیز است و داینامیک بودن کوئری درون command فانکشن queryInput زار میزند. همچنین فانکشن قبلی را پایین همین فانکشن قرار میدهم که بتوانید به راحتی مقایسه کنید.
حال با این فانکشن جدید که مورد استفاده قرار میگیرد ، اگر key ها در رکورد ها مورد تغییر قرار گیرند ، باز هم سرویس ما به درستی کار خواهد کرد ، مثلا فرض کنید یک key به نام Salary به Debt تغییر کند ، این تغییر دیگر باعث از کار افتادن سرویس ما نمیشود و وابستگی سرویس ما به DB مربوط به رکورد های employee ها را از بین میبرد .

```
// Add Record   data-access.js
//3d Attempt
//used
const queryInput = async ( Table , inputs = [], outputs = []) => {
    command = `
    INSERT INTO ${Table} (${inputs[0].name}, ${inputs[1].name}, ${inputs[2].name}, ${inputs[3].name},
${inputs[4].name})
    OUTPUT inserted.Id
    VALUES (@${inputs[0].name}, @${inputs[1].name}, @${inputs[2].name}, @${inputs[3].name}, @${inputs[4].name});
    `
    return run('query', command, inputs, outputs);
};
//not used
const queryEntityInpute = async (Table, entity, outputs = []) => {
    command =    `
    INSERT INTO ${Table}  (Code, Salary, Job, Department, Name)
    OUTPUT inserted.Id
    VALUES (@Code, @Salary, @Job, @Department, @Name);
`
    const inputs = fetchParams(entity);
    return run('query', command, inputs, outputs);
};
//Ali.Askari @Shiraz.University
```

تصویر 12- فانکشن جدید و قدیمی برای درج در یک نگاه ، اولی جدید است

تغییر آخر به نوعی در variant.js صورت میگیرد. به جای استفاده از req.body سنگین به جای همه مقادیر ، به استفاده از دیکشنری برای کاندیشن ها بر میگردیم .در واقع به روشی که آن را کنار گذاشته بودیم و در تصویر 9 به آن اشاره کردیم بر میگردیم، به کد های کامنت شده دقت کنید.

```
// Add Record   variant.js
//3d Attempt
///start  new 3
router.post('/', async (req, res) => {
    try {

        // passing input as arrays
        const result = await dataAccess.queryInput(
            'Employee',
            [
                { 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 },
            ]
        );
       /*
        // passing input as entity
        const result = await dataAccess.queryEntityInpute(
         'Employee',
          req.body
        );
         */
        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 - نحوه استفاده از فانکشن queryInput در فایل variant.js

مشاهده میکنید که به سادگی یک روش پایدار ایجاد کردیم.

4. ویرایش رکورد

خب به بحث ویرایش رکورد رسیدیم که این کار در دو بخش انجام میشود ، در فایل variant.js که ابتدا باید متد PUT را به کمک ROUTER هندل کنیم ، سپس از فانکشنی که در سرویسمان ایجاد کردیم باید در variant استفاده کنیم یا به نوعی صدایش بزنیم و آن را مقدار دهی کنیم ، اما ما در این مرحله از دو فانکشن استفاده میکنیم. ابتدا از فانکشن querySelectById برای دریافت Id رکورد از URL استفاده میکنیم، سپس از فانکشن queryEntityUpdate برای اعمال آپدیت بر روی رکورد انتخاب شده استفاده میکنیم.
تابع queryEntityUpdate به عنوان ورودی، نام Table و اطلاعات جدید که باید جایگزین اطلاعات قبلی شوند را میگیرد که همانند مرحله قبل از req.body استفاده میکنیم.
دقت کنید استفاده از req.body باعث میشود تا حدی ورودی های فانکشن سرویس ما مشابه با Assignment به نظر نرسد ، اما دقیقا همان است و تنها از یک خلاقیت جاوا اسکریپتی استفاده شده است.

اما مشکلی که اینجا گریبان ما را میگیرد باز هم مشابه داستان قبلیست، با همان شیوه قبلی ابتدا پیدا سازی ها را بر مبنای req.body انجام میدهیم و سپس به سمت داینامیک سازی و حذف وابستگی ها قدم بر خواهیم داشت.

```
// Update  4rth Attempt variant.js
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.querySelectById('Employee' , [
            { name: 'Id', value: req.params.id ,operator : '=' }
        ]);

        let employee = result.recordset.length ? result.recordset[0] : null;
        if (employee) {
            await dataAccess.queryEntityUpdate(
            'Employee',
             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
```

تصویر 14 - پیاده سازی اولیه ی آپدیت به کمک فانکشن های querySelectById و queryEntityUpdate
کد بخش variant.js

اما همانطور که ذکر کردیم ، این شیوه ما را در سرویس به مشکل وابستگی دچار میکند ، همین حالا این مشکل را در این فایل بر طرف میکنیم و تصویر زیر گویای ROUTER جدید ما در همین فایل variant.js خواهد بود.

```
// Update  4rth Attempt variant.js
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.querySelectById('Employee' , [
            { name: 'Id', value: req.params.id ,operator : '=' }
        ]);

        let employee = result.recordset.length ? result.recordset[0] : null;
        if (employee) {
            await dataAccess.queryUpdate(
            'Employee',
            [
                { 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 },
                { name: 'Id' , value : req.body.Id}
            ]
            );

            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
```

تصویر 15 – آپدیت ، فایل variant.js ، بهبود کدها

همانطور که قابل مشاهده است ، فانکشن queryUpdate در خط 17 جایگزین فانکشن قبلی که با Entity کار میکرد شده است. این فانکشن هم مانند فانکشنی که در قسمت add Record به طور بهینه و داینامیک و بدون وابستگی به بقیه اجزای پروژه و مقادیر رکورد ها ایجاد کردیم جدید است و از همان ایده و خط فکری پیروی کرده است. در اینجا هم به جای ارسال مقادیر رکورد هایی که نیاز به آپدیت دارند به صورت یک req.body واحد ، به صورت دیکشنری واقع در inputs مقادیر را به سرویسمان ارسال میکنیم که یعنی مقادیر را به همین شیوه ای که گفتم به ورودی فانکشنی که از سرویسمان در حال فراخوانی و استفاده هستیم نسبت میدهیم.

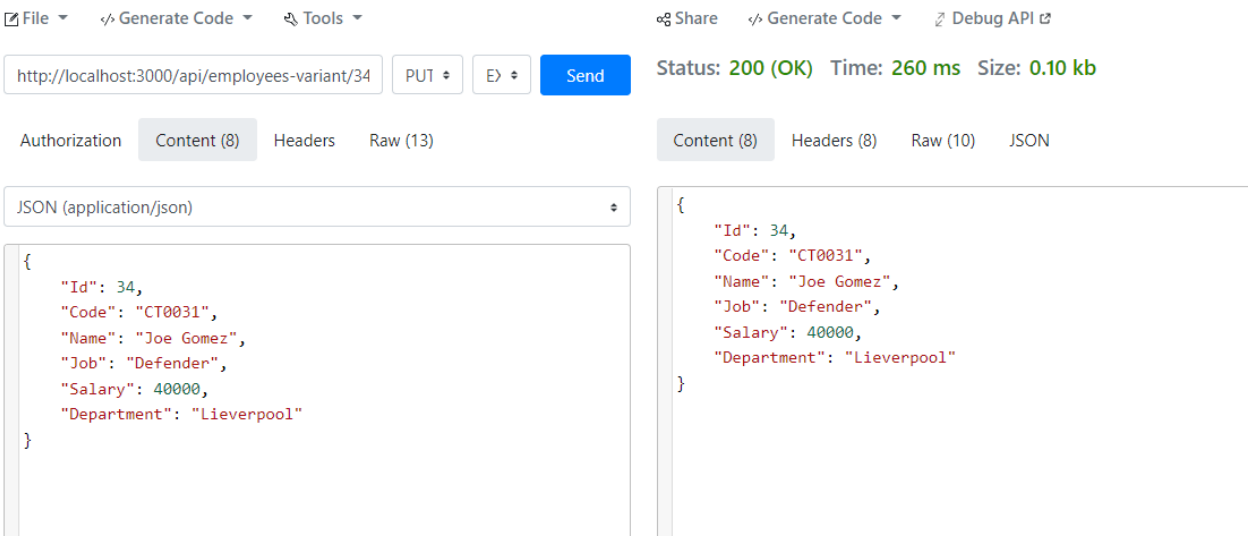
در ادامه کد های مربوط به سرویسمان را خواهیم دید ، در تصویر 17 ، هر دو فانکشن قدیمی و جدید را در کنار هم مشاهده می کنید که درک بهتری نسبت به تغییر داشته باشید :
فانکشن جدید queryUpdate و فانکشن قدیمیِ مشکل ساز queryEntityUpdate نام دارد. به تصویر 16 دقت کنید که دو فانکشن در طرز تنظیم command است.

```
//4th Attempt data-access.js
//not used Old
command =       `
    UPDATE ${Table} SET
        Code = @Code,
        Salary = @Salary,
        Job = @Job,
        Department = @Department,
        Name = @Name
    WHERE Id = @Id;
`
//Used for Update New
    command =     `
    UPDATE ${Table} SET
    ${inputs[0].name} = @${inputs[0].name},
    ${inputs[1].name} = @${inputs[1].name},
    ${inputs[2].name} = @${inputs[2].name},
    ${inputs[3].name} = @${inputs[3].name},
    ${inputs[4].name} = @${inputs[4].name}
    WHERE ${inputs[5].name} = @${inputs[5].name};
`
//Ali.Askari @Shiraz.University
```

تصویر 16 - تفاوت command ها در آپدیت جدید و قدیمی

```
//4th Attempt data-access.js
const queryEntityUpdate = async ( Table ,entity, outputs = []) => {
    command =       `
    UPDATE ${Table} SET
        Code = @Code,
        Salary = @Salary,
        Job = @Job,
        Department = @Department,
        Name = @Name
    WHERE Id = @Id;
`
    const inputs = fetchParams(entity);
    return run('query', command, inputs, outputs);
};


//Used for Update
const queryUpdate = async ( Table ,inputs = [], outputs = []) => {
    command =     `
    UPDATE ${Table} SET
    ${inputs[0].name} = @${inputs[0].name},
    ${inputs[1].name} = @${inputs[1].name},
    ${inputs[2].name} = @${inputs[2].name},
    ${inputs[3].name} = @${inputs[3].name},
    ${inputs[4].name} = @${inputs[4].name}
    WHERE ${inputs[5].name} = @${inputs[5].name};
`
    return run('query', command, inputs, outputs);
};
//Ali.Askari @Shiraz.University
```

تصویر 17 - فانکشن های جدید و قدیمی آپدیت در سرویس



تصویر 18 - نحوه اجرا و کارکرد سرویس برای آپدیت رکورد

- #### Update Employee
- PUT http://localhost:3000/api/employees-variant/34
- content-type: application/json

{ "Id": 34,
"Code": "CTOO31",
"Name": "Joe Gomez",
"Job": "Defender",
"Salary": 4OOOO,
"Department": "Lieverpool" }

5. حذف رکورد

حذف هم مانند آپدیت در دو مرحله صورت میگیرد ، ابتدا Id رکورد را از URL به کمک params میگیریم، سپس آن رکورد را به کمک یک فانکشن دیگر حذف میکنیم.

برای دریافت Id ، از querySelectById استفاده میکنیم و سپس از queryDelete برای حذف استفاده میکنیم ، این فانکشن ها متعلق به سرویس ایجاد شده توسط خودمان هستند .

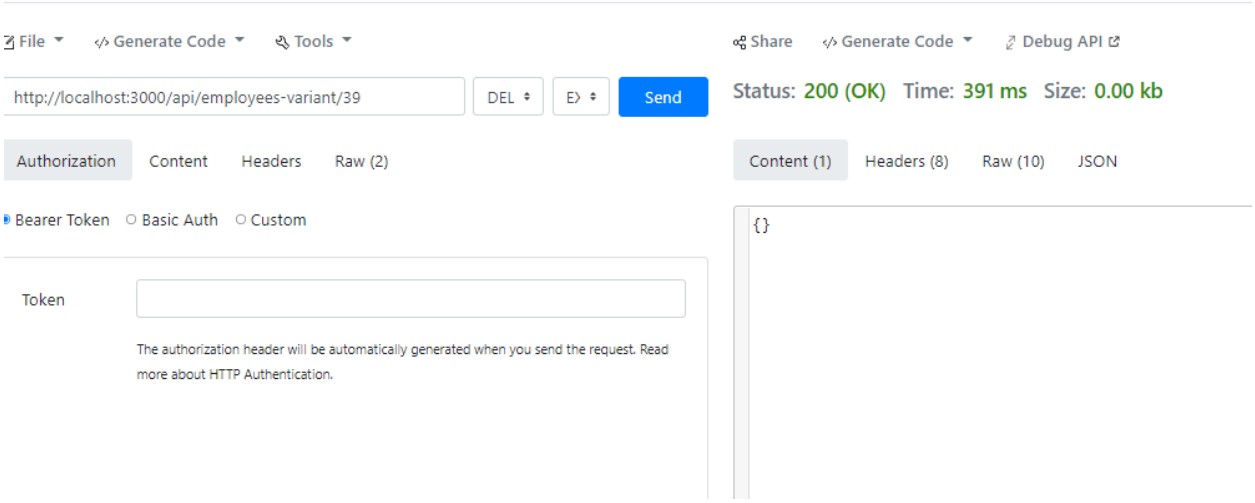
```
// Delete 5th Attempt variant.js
router.delete('/:id', async (req, res) => {
    try {
        const result = await dataAccess.querySelectById( 'Employee' ,[
            { name: 'Id', value: req.params.id ,operator : '=' }
        ]);

        let employee = result.recordset.length ? result.recordset[0] : null;
        if (employee) {
            await dataAccess.queryDelete('Employee' , [
                { 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 19 - استفاده از فانکشن حذف رکورد در employees-variant.js

نکته خاصی برای بیان در مورد تصویر بالا وجود ندارد ، اما به تصویر 20 که تصویر زیر است دقت کنید که مرتکب این اشتباه نشوید که فکر کنید Id اینجا در این رکورد وابستگی به بقیه پروژه ایجاد کرده است ،قطعه کد بالا مربوط به فایل سرویس ماست ، وابستگی وجود ندارد چرا که Id در همه پایگاه های داده و در همه رکورد ها بالاخره Id است و به رکورد مربوط است و خود DB آن را ایجاد میکند. پس وابستگی وجود ندارد ، هر چند که می توانم همین Id را به روش قبلی یعنی از inputs بگیرم اما تنها به کدها پیچیدگی اضافه میشود. در ضمن ما حذف به واسطه دریافت Id را پیاده سازی کردیم ، می توان گسترش داد این فانکشن را ولی به افزودن پیچیدگی به فایل سرویسمان نمی ارزد.

```
1 //5th Attempt delete , data-access.js
2 const queryDelete = async ( Table , inputs = [], outputs = []) => {
3     command = `DELETE FROM ${Table} WHERE Id = @Id;`
4
5     return run('query', command, inputs, outputs);
6 };
7
8 //Ali.Askari @Shiraz.University
```

تصویر 20 - فانکشن حذف رکورد در سرویس

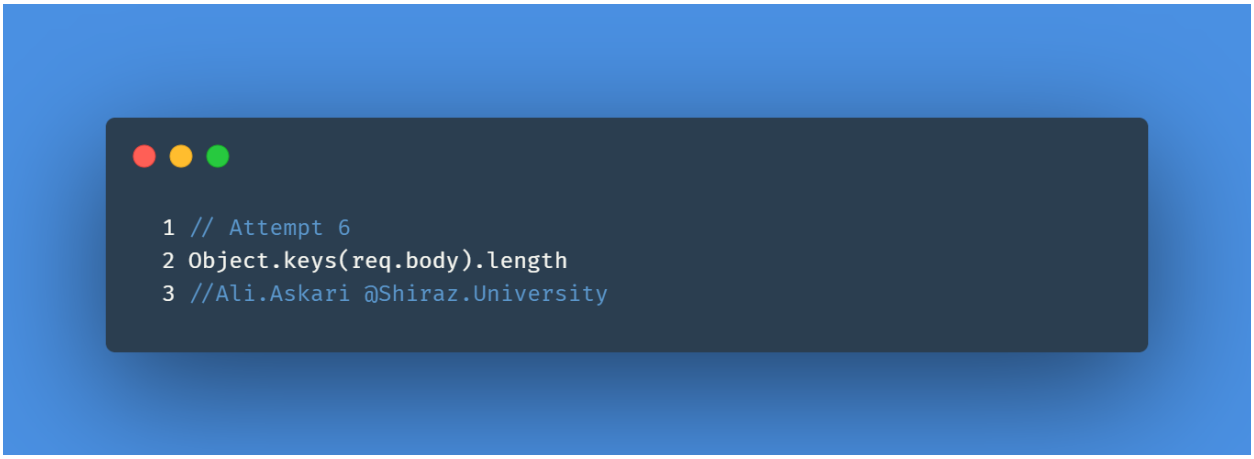


تصویر 21 - حذف موفقیت آمیز رکورد 39

- #### Delete Employee
- DELETE http://localhost:3000/api/employees-variant/39

6. برگرداندن رکورد با استفاده از بیش از یک شرط

در نهایت دست به کار جالبی میزنیم ، برای اینکه بتوانیم تعداد input ها را برای یک سرچ ، هم از سمت کاربر ، یعنی employees-variant.js و هم از سمت سرویس طراحی شده کاملا داینامیک کنیم ، باید در هر دو سمت دو منطق قابل توجه را پیاده سازی کنیم ، باید تعداد input ها یا در واقع req.body ها که در سمت سرویس طراحی شده تعداد دیکشنری ها میشوند را پیدا کنیم و به تعداد مناسب برای درخواست کاربر مقادیرش را بگیریم و کارش را هندل کنیم. در تصویر زیر نحوه فهمیدن تعداد مقادیر ارسالی از سمت کاربر را میفهمیم ، همچنان در variant هستیم.



تصویر 22 - تعداد req.body ها از سمت کاربر

حالا یک منطق به کمک if ,else پیاده سازی میکنیم که بتوانیم تعداد دیتای دریاقتی از کاربر را به طور داینامیک هندل کنیم. هدف ما این است که تنها با یک Router ، درخواست های متعدد و متفاوت کاربر را برای تشکیل یک کوئری با شروط بسیار و از نوع SELECT را هندل کنیم ، پس ابتدا در variant منطق خود را پیاده سازی میکنیم که تنها یک Router برای حالت زیر داشته باشیم .

SELECT ... AND ... AND ...

در تصویر زیر اولین شرط را مشاهده میکنید ، شرط های بیشتر در قطعه کد موجود است ، یعنی else if ها و یک else هم وجود دارد اما در تصویر زیر تنها if را مشاهده میکنید چرا که قصد شلوغ تر شدن فایل گزارش را ندارم.

```
// Attempt 6 Variant.js router.post('/custom')
if (Object.keys(req.body).length == 2) {
            const result = await dataAccess.CustomquerySelectById( Table = 'Employee',[
                { name: 'Id', value: req.body.Id, operator : '='},
              { name: 'Department', value: req.body.Department, operator : '='}

               /// next
 else if (Object.keys(req.body).length ==3) {
            const result = await dataAccess.CustomquerySelectById( Table = 'Employee',[
                { name: 'Id', value: req.body.Id, operator : '='},
                { name: 'Job', value: req.body.Job, operator : '='},
                { name: 'Department', value: req.body.Department , operator : '='}
            ]);
          //there is so much more in employee-variant.js
//Ali.Askari @Shiraz.University
```

تصویر 23- بخشی از شروط در custom router ، فایل varaiant

همانطور که در تصویر بالا مشخص است ، به کمک تعداد زیادی if,else حالات مربوط به مقادیر ورودی و تشکیل dict برای ارسال به سرویسمان هندل شده است. در واقع ما اول چک میکنیم ببینیم تعداد مقادیری که کاربر می خواهد وارد کند و وارد کرده است چند تاست ، مطابق آن عدد تعداد dictionary تشکیل میدهیم و آن را برای سرویس توسعه داده شده میفرستیم تا کوئری ایجاد شود. تصویر بالا ، کدهای سرویس توسعه داده شده نیست پس حتی اگر ازین منطق استفاده نشود و هارد کد شود مقادیر ورودی به راندامان سرویس ما لطمه وارد نمیکند اما به هر حال ما این منطق را در اینجا هم مورد پیاده سازی قرار دادیم.

بخش های بیشتر از این منطق در فایل variant موجود است و هنوز هم جای کار دارد می توان خلاقیت را بیشتر کرد ، اما من به همین بسنده میکند و به سراغ سرویسمان میروم.

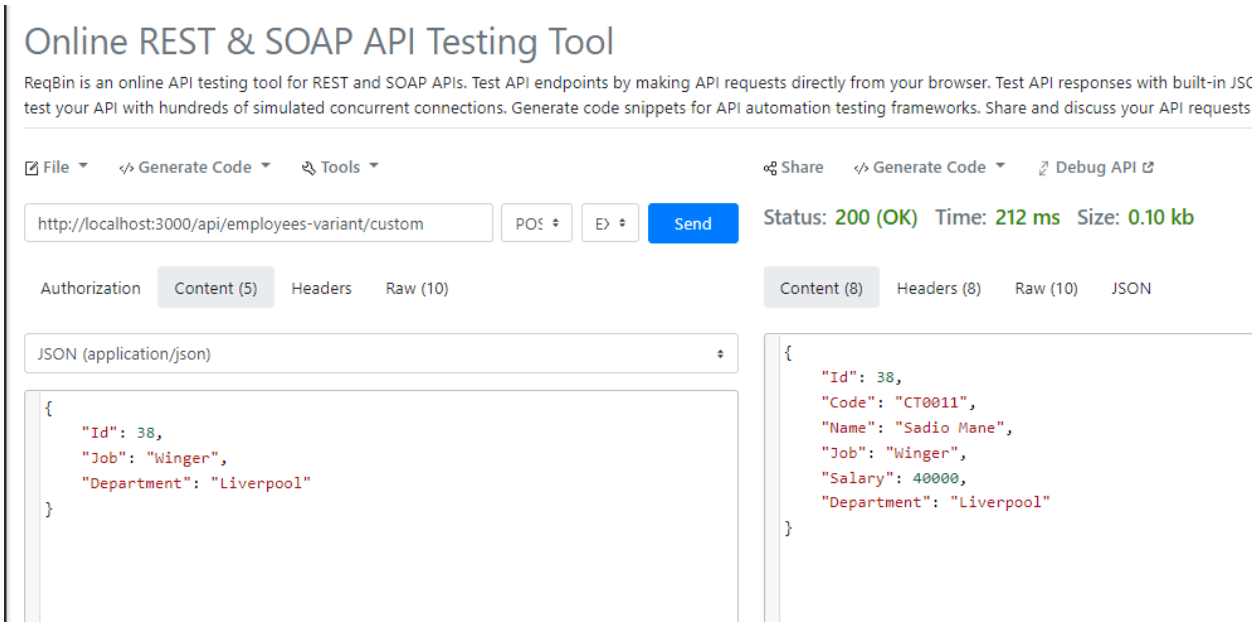
در سرویس توسعه داده شده هم در کل چیزی مشابه با همین منطق باید پیاده سازی شود ، ابتدا چک شود تعداد dictionary های ارسال شده به سرویس از طریق و درونِ inputs چندتاست ، این عدد نشان دهنده شروط ما ، در واقع Condition ها و البته مقادیریست که از طرف کاربر وارد شده است که به عنوان شرط مورد استفاده قرار گیرد و ابتدا varinat به عنوان یک استفاده کننده از سرویس ما این ها را دریافت کرده و سپس برای سرویس ارسال کرده است. حال سرویس باید پس از پردازش نتیجه را برگرداند . همانطور که در تصویر زیر مشخص است ، این منطق به این شکل است که تعداد dict ها را چک می کند و مطابق با تعداد سعی می کند شروط کوئری را به صورت داینامیک هندل کند ، برای مثال وقتی می بیند 4 شرط داریم ، یعنی کاربر 4 ورودی یا متغیر وارد کرده ، پس سرویس ما خودش را برای ساخت یک کوئری SELECT با 4 شرط AND آماده میکند.

همچنین میشود پا را ازین مرحله فراتر نهاد و نوع ترکیب شرط ها ، مثلا AND یا OR شدنش را هم به صورت داینامیک هندل کرد یعنی آن را به کاربر یا variant سپرد و البته همانطور که ذکر کردم میتوان به داینامیک سازی و توسعه سرویس دائما ادامه داد. حال تنها مانده نمایش همین منطق در سرویس زیبای ما و البته خروجی .

```
// Attempt 6 data-access.js
const CustomquerySelectById = async (Table ,  inputs = [], outputs = []) => {

    if(inputs.length == 1){
        command = `SELECT * FROM ${Table} WHERE ${inputs[0].name} ${inputs[0].operator} @${inputs[0].name} `
    }
    if(inputs.length == 2){
        command = `SELECT * FROM ${Table} WHERE ${inputs[0].name} ${inputs[0].operator} @${inputs[0].name}
        And ${inputs[1].name} ${inputs[1].operator} @${inputs[1].name};`
    }
    else if (inputs.length == 3 ) {
        command = `SELECT * FROM ${Table} WHERE ${inputs[0].name} ${inputs[0].operator} @${inputs[0].name}
        And ${inputs[1].name} ${inputs[1].operator} @${inputs[1].name}
        And ${inputs[2].name} ${inputs[2].operator} @${inputs[2].name};`
    }
    else if (inputs.length == 4 ) {
        command = `SELECT * FROM ${Table} WHERE ${inputs[0].name} ${inputs[0].operator} @${inputs[0].name}
        And ${inputs[1].name} ${inputs[1].operator} @${inputs[1].name}
        And ${inputs[2].name} ${inputs[2].operator} @${inputs[2].name}
        And ${inputs[2].name} ${inputs[2].operator} @${inputs[2].name};`
    }
    else {
        command = `SELECT * FROM ${Table} WHERE ${inputs[0].name} ${inputs[0].operator} @${inputs[0].name} `
    }

    return run('query', command, inputs, outputs);
};

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

تصویر 24 - منطق پیاده سازی شده برای هندل کردن شروط و Condition های مختلف به صورت داینامیک

همانطور که مشخص است ، همه چیز به صورت داینامیک و بدون هارد کد شدن هندل شده است ، شروط مختلفی را میتوان برای ایجاد یک کوئری SELECT به وسیله این فانکشن به طور کامل هندل کرد.
در ادامه خروجی را خواهیم دید که وقتی کاربر همزمان چند شرط را برای برگرداندن دیتا درخواست می کند ، سیستم و در واقع سرویس ما به درستی کار میکند. در ضمن باید گفت این منطق که در تصویر 24 میبینید با توجه به تعداد dict های موجود درون inputs کار میکند و شروط خود را متناسب با آن تنظیم کرده و سعی میکند بخش قابل توجه ای از احتمالات را پوشش دهد، همچنین قابلیت پوشش احتمالات بیشتر و تبدیل شدن به یک سرویس بسیار بسیار کامل برای این فانکشن وجود دارد و همچنین فانکشن های قبلی ولی فعلا متناسب با زمان این تمرین به همین اکتفا میکنم و سعی کردم سیستم دچار پیچیدگی و سنگینی نباشد.



تصویر 25 - نحوه کار ویژگی ششم ، multiple condition SELECT

گام های طی شده در این گامِ سوم :

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

پایان گام سوم

با احترام و تشکر به خاطر وقتی که برای مطالعه گذاشتید.

همچنین از کارکردِ سیستم ویدیوی کوتاهی در فایل پروژه قرار گرفته است.

علی عسگری - فروردین 1400