Next JS - 2

Mongo DB

MongoDB is a flexible, document-based NoSQL database that stores data in JSON-like documents (BSON) within collections, rather than traditional tables with rigid rows and columns, making it great for modern apps needing scalability and agile data models because it easily handles structured, semi-structured, and unstructured data with a dynamic schema. It's popular for its developer-friendliness, allowing data to map directly to application objects and supporting features like ad-hoc queries, indexing, and horizontal scaling for high availability.

Installation

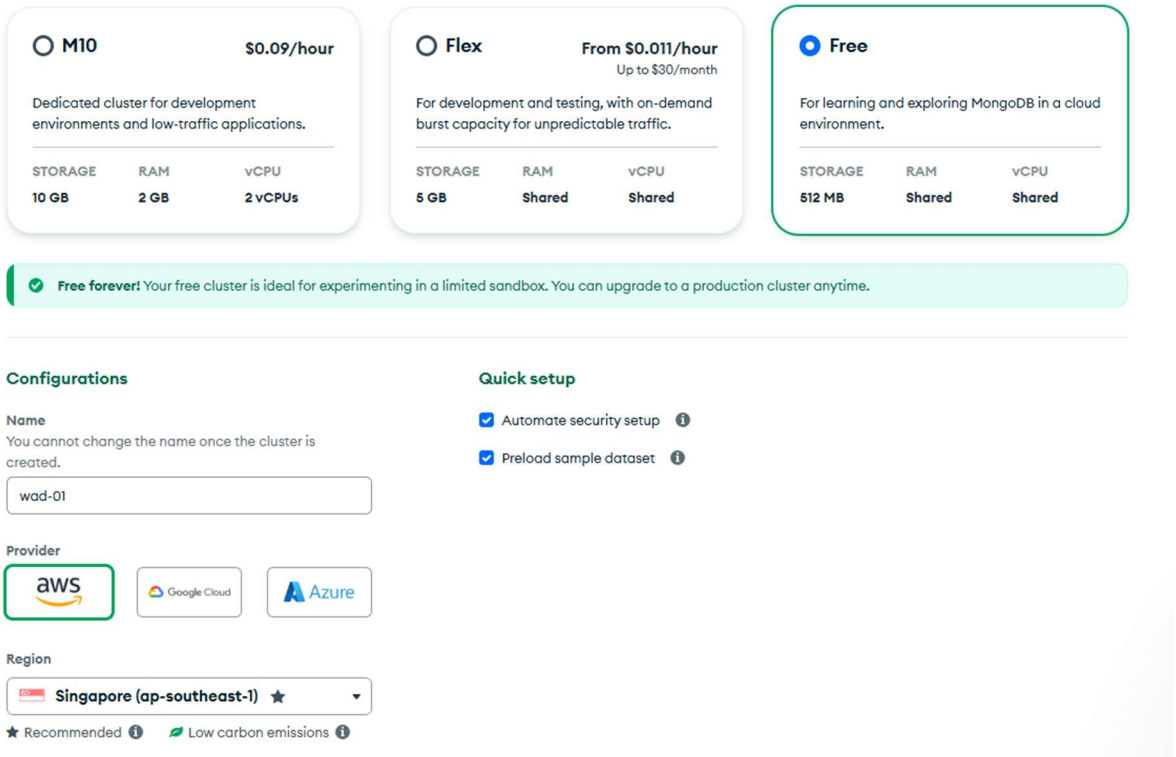
For teaching and learning purposes, you can create your free Mongo DB cluster on cloud.

Visit the https://www.mongodb.com/cloud/atlas/register to register as a new user, if you don't have ones.

Once you can access to the Mongl Atlas, create your first cluster.

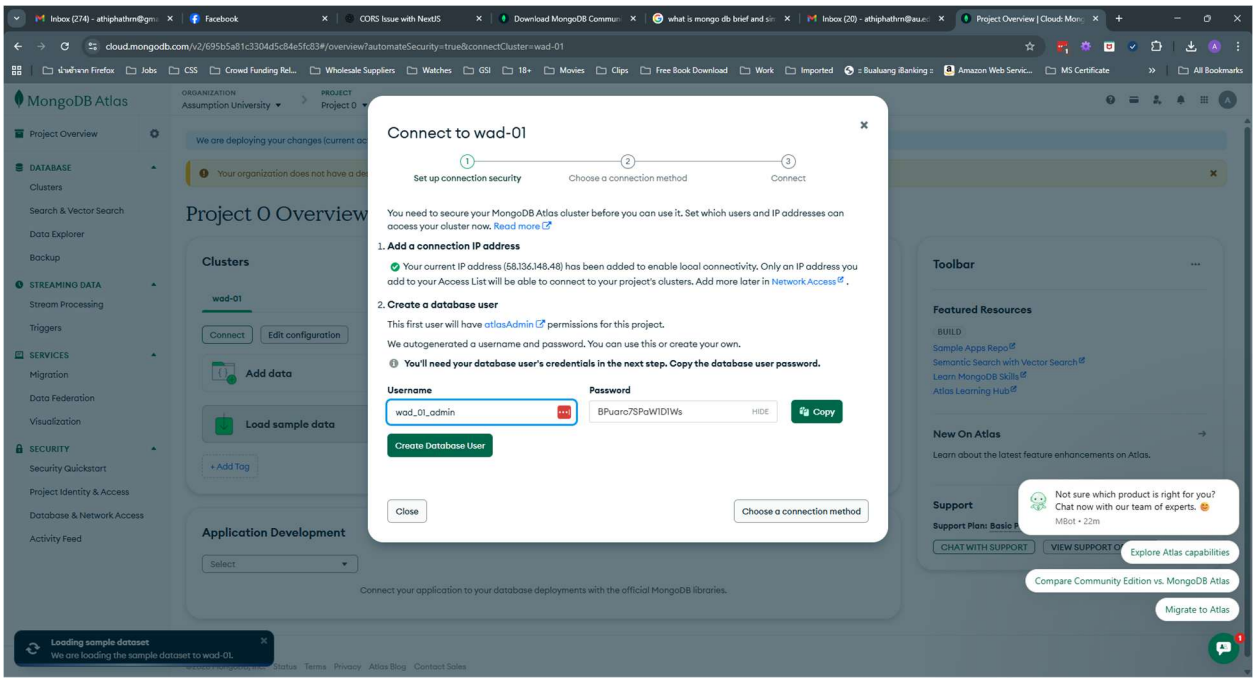
Name the cluster as you want, you cannot change it later.

Don't forget to change to use Free option.

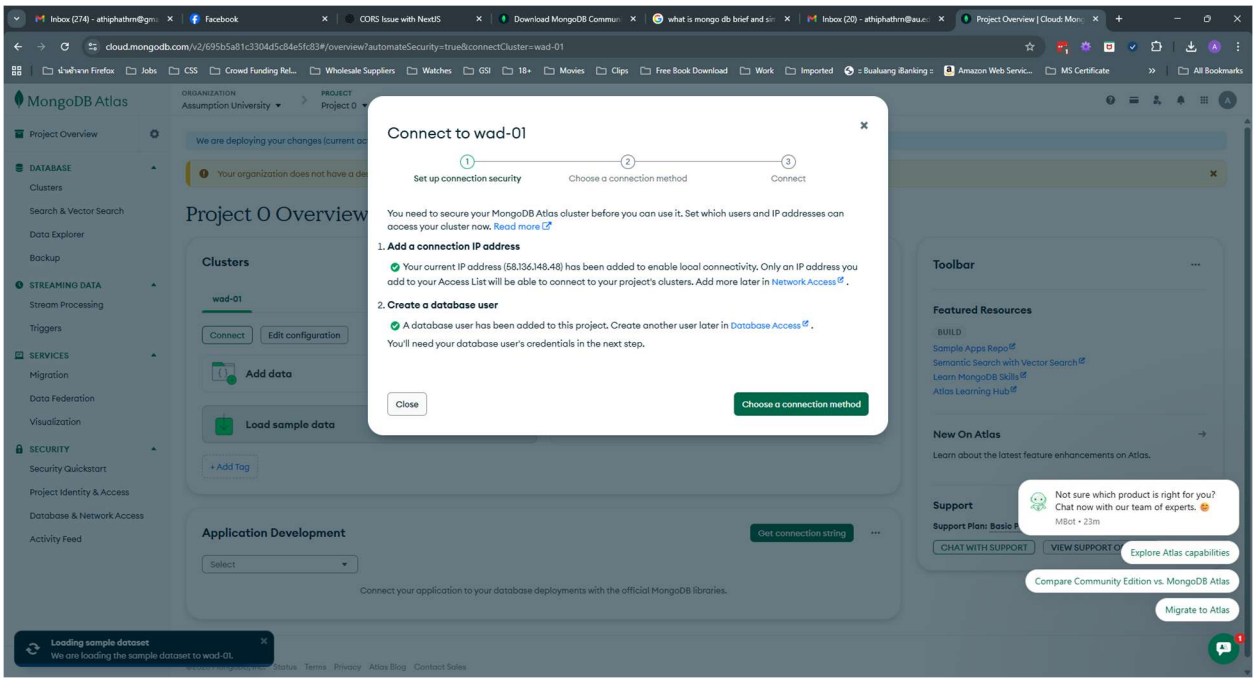


Click “Deploy” to start deployment your first cluster.

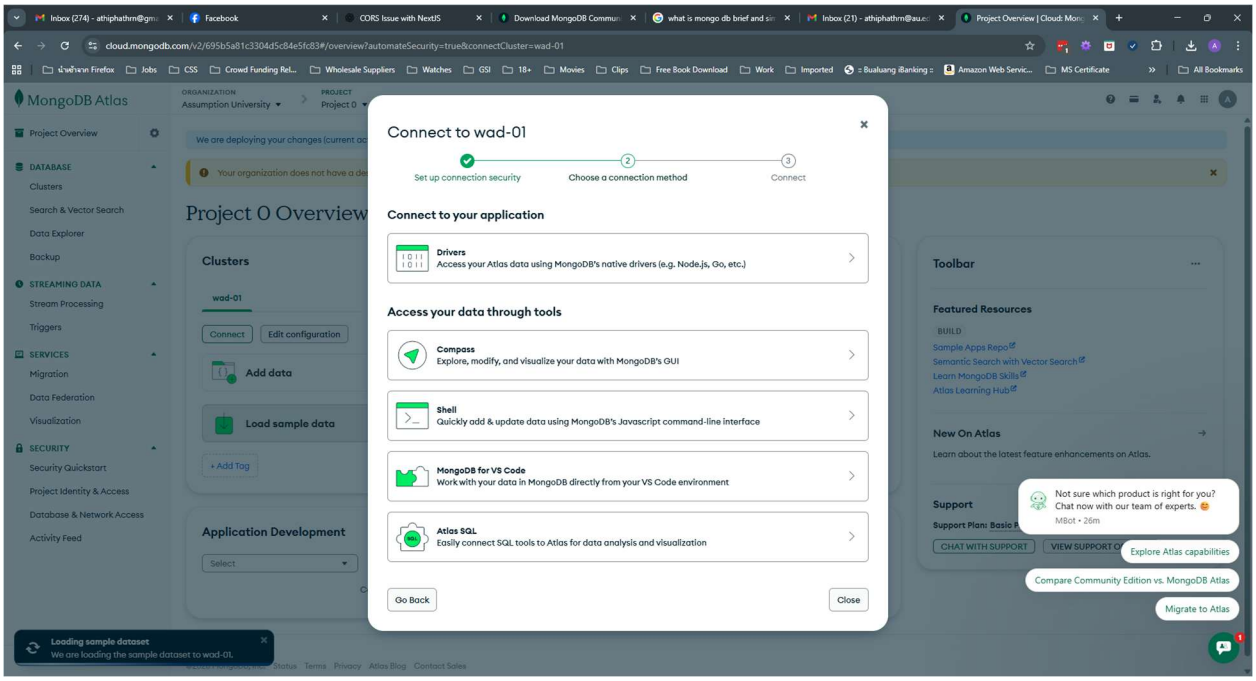
Then, the system will ask you about username and password. Copy to somewhere safe since you need to use them in further step, then click Create database user.



Continue with Choose connection method



The connection method is the part that will demonstrate to you how to connect to the Mongo DB from each client.



You can choose “Node JS” to see how to connect to Mongo DB, it can show you the full code with password bundled.

If you use VSCode, then you can connect to the Mongo DB directly with its extension, click “MongoDB for VS Code” and follow the instructions.

At this step, you can choose whatever you want.

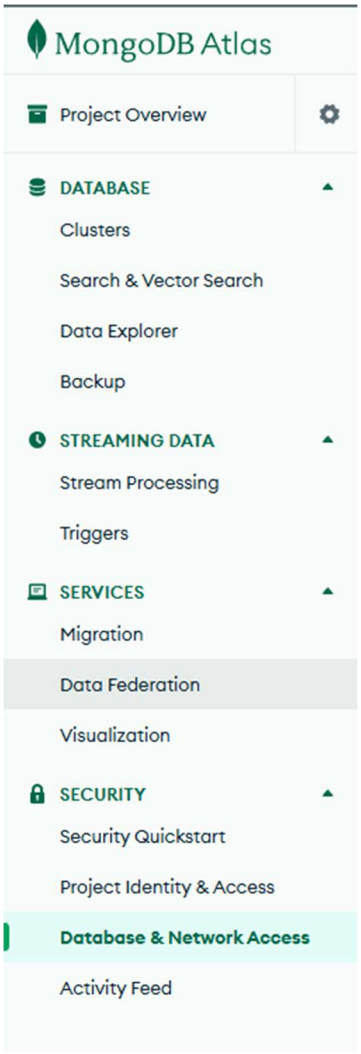
Once you select “Done” from one of the connection types, it will bring you to the cloud UI dashboard.

Also, if you add extension into your VS Code and input the connection string successfully, you can see new menu icon on the left panel named Mongo DB, click on it and you can see your data too.

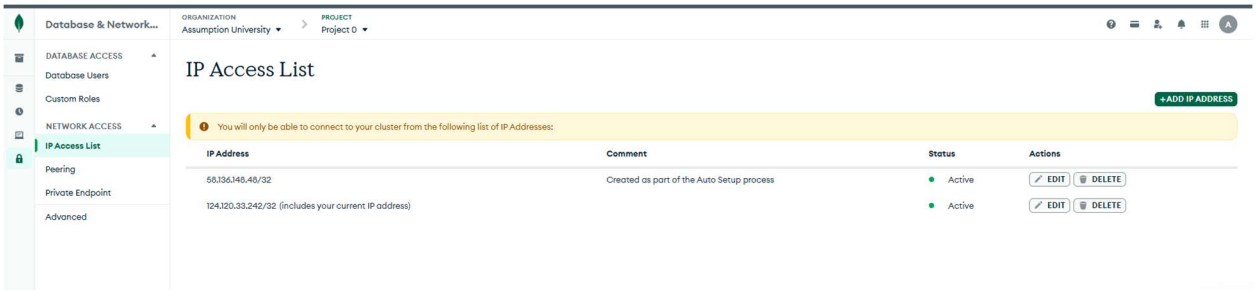
Network Access

When you complete your first cluster, it will allow your current IP address only.

You can add more IP addresses later by accessing its security menu.



Then go to IP Access List



You can add your current IP address to the list by clicking at “Add IP Address”. Some Network access configurations are only available for paid tiers. If you are using public WiFi or Mobile

network, your IP address will be changed over time, and it may be a bit difficult to work. However, when you deploy your code to a VM, the IP is then fixed and there shall be no problem using with cloud free tier.

Please be aware that if you use public WiFi or Mobile network, don't forget to remove the IP address after use. However, if you consider that it is just for playing around, it should be fine.

The other solution during development is to install Mongo DB into your system directly. You can download and install it from the Mongo DB official website.

Connection from Next JS

Next JS does not create any connection to Mongo DB at start up, it only creates connections when needed.

To create a connection, you need to export a connection state from any file.

Once the connection state is used, it will be initiated and connected to Mongo DB.

Importantly, do not hardcode the connection credentials into your code, use environment instead.

Implementation:

1. Install Mongo DB dependencies:

```
npm install mongodb
```

2. Create a file named "mongodb.js" in "src/lib" (like cors.js) with the following code:

```
import { MongoClient } from "mongodb";

const options = {};
let globalClientPromise;

export function getClientPromise() {
  const uri = process.env.MONGODB_URI;
  if (!uri) {
    throw new Error("Please add your Mongo URI to .env.local or set MONGODB_URI
env variable");
  }
  if (process.env.NODE_ENV === "development") {
    if (!globalClientPromise) {
      const client = new MongoClient(uri, options);
      globalClientPromise = client.connect();
    }
    return globalClientPromise;
  } else {
```

```
    const client = new MongoClient(uri, options);
    return client.connect();
  }
}
```

3. Create a new file named ".env.local" at project root directory with the following content using your given connection string from previous topic.

```
MONGODB_URI= <your-connection-string>
```

The ".env.local" is a special file treated by Next JS as the local environment data.

Creating a testing API.

Once you create a free tier in Mongo DB Atlas, there should be a sample database named "sample_mflix" to play around.

We will try to connect and fetch the collection named "comments" from the database.

1. Create a new directory named "mongo_test" under "src/app/api"
2. Create route.js file with the following code:

```
import corsHeaders from "@/lib/cors";
import { getClientPromise } from "@/lib/mongodb";
import { NextResponse } from "next/server";

export async function GET() {

  const client = await getClientPromise();

  const db = client.db("sample_mflix");
  const result = await
db.collection("comments").find({}).skip(0).limit(10).toArray();
  return NextResponse.json(result, {
    headers: corsHeaders
  });
}
```

3. Run the Next server, and observe the result from: http://localhost:3000/api/mongo_test

Assignment 3 – Your first Mongo DB

Create your own Collection

There is a crucial detail you need to understand. You can access non-existed database or collection without any error, and nothing happens on the Mongo DB as well. If you try to search anything from the non-existed database or collection, you just get empty data. Once you insert data to the Mongo DB, then the database and associated collection will be created altogether with the first data. Alternatively, you can explicitly create a collection by calling createCollection().

Item API

1. Create a folder “item” under “api”.
2. Create a file named “route.js” inside “item” with the following code:

```
import corsHeaders from "@/lib/cors";
import { getClientPromise } from "@/lib/mongodb";
import { headers } from "next/headers";
import { NextResponse } from "next/server";

export async function OPTIONS(req) {
  return new Response(null, {
    status: 200,
    headers: corsHeaders,
  });
}

export async function GET () {

  try {
    const client = await getClientPromise();
    const db = client.db("wad-01");
    const result = await db.collection("item").find({}).toArray();
    console.log("==> result", result);
    return NextResponse.json(result, {
      headers: corsHeaders
    });
  }
  catch (exception) {
    console.log("exception", exception.toString());
    const errorMsg = exception.toString();
    return NextResponse.json({
      message: errorMsg
    }, {
      status: 400,
      headers: corsHeaders
    })
  }
```

```
}

export async function POST (req) {
  const data = await req.json();
  const itemName = data.name;
  const itemPrice = data.price;
  const itemCategory = data.category;

  try {
    const client = await getClientPromise();
    const db = client.db("wad-01");
    const result = await db.collection("item").insertOne({
      itemName: itemName,
      itemCategory: itemCategory,
      itemPrice: itemPrice,
      status: "ACTIVE"
    });
    return NextResponse.json({
      id: result.insertedId
    }, {
      status: 200,
      headers: corsHeaders
    })
  }
  catch (exception) {
    console.log("exception", exception.toString());
    const errorMsg = exception.toString();
    return NextResponse.json({
      message: errorMsg
    }, {
      status: 400,
      headers: corsHeaders
    })
  }
}
```

3. Test creating item with Item API using any API Testing Tool
   Url: http://localhost:3000/api/item
   Method: POST
   Body:
   {
    "name":"Pen",
    "category":"Stationary",
    "price":"10.99"
   }

4. Test Get item with Item API
   Url: http://localhost:3000/api/item
   Method: GET

Update Data

The key to updating data is identifying an existing resource using a unique key, commonly an id.

Once identified, the data can be updated either entirely or partially.

In standard RESTful design, POST is used to create new data, while PUT or PATCH is used to update existing data.

PUT typically replaces the entire resource, whereas PATCH is used to update only specific fields.

Dynamic Route in Next JS with JavaScript

So far, we do have just static route path by creating physical folders.

To handle dynamic paths, use square braces and follow by parameter name.

Get Detail, Update and Replace API

1. Create a directory named “[id]” under “item” directory you created above.
2. Create a file named “route.js” under “[id]” directory with the following code:

```
import corsHeaders from "@/lib/cors";
import { getClientPromise } from "@/lib/mongodb";
import { headers } from "next/headers";
import { NextResponse } from "next/server";
import { ObjectId } from "mongodb";

export async function OPTIONS(req) {
  return new Response(null, {
    status: 200,
    headers: corsHeaders,
  });
}

export async function GET (req, { params }) {

  const { id } = await params;

  try {
    const client = await getClientPromise();
```

```
    const db = client.db("wad-01");
    const result = await db.collection("item").findOne({_id: new ObjectId(id)});
    console.log("==> result", result);
    return NextResponse.json(result, {
      headers: corsHeaders
    });
  }
  catch (exception) {
    console.log("exception", exception.toString());
    const errorMsg = exception.toString();
    return NextResponse.json({
      message: errorMsg
    }, {
      status: 400,
      headers: corsHeaders
    })
  }
}

export async function PATCH (req, { params }) {

  const { id } = await params;
  const data = await req.json(); //assume that it contain part of data...
  const partialUpdate = {};
  console.log("data : ", data);
  if (data.name != null) partialUpdate.itemName = data.name;
  if (data.category != null) partialUpdate.itemCategory = data.category;
  if (data.price != null) partialUpdate.itemPrice = data.price;

  try {
    const client = await getClientPromise();
    const db = client.db("wad-01");
    const existedData = await db.collection("item").findOne({_id: new
ObjectId(id)});
    const updateData = {...existedData, ...partialUpdate};

    const updatedResult = await db.collection("item").updateOne({_id: new
ObjectId(id)}, {$set: updateData});
    return NextResponse.json(updatedResult, {
      status: 200,
      headers: corsHeaders
    })
  }
  catch (exception) {
```

```
    const errorMsg = exception.toString();
    return NextResponse.json({
      message: errorMsg
    }, {
      status: 400,
      headers: corsHeaders
    })
  }
}

export async function PUT (req, { params }) {
  const { id } = await params;
  const data = await req.json(); //assume that it contain whole item data...

  try {
    const client = await getClientPromise();
    const db = client.db("wad-01");

    const updatedResult = await db.collection("item").updateOne({_id: new
ObjectId(id)}, {$set: data});
    return NextResponse.json(updatedResult, {
      status: 200,
      headers: corsHeaders
    })
  }
  catch (exception) {
    console.log("exception", exception.toString());
    const errorMsg = exception.toString();
    return NextResponse.json({
      message: errorMsg
    }, {
      status: 400,
      headers: corsHeaders
    })
  }
}
```

3. Test the API with the following details, replace the id with one of your item's _id property:
   Url: http://localhost:3000/api/item/<id>
   Method: GET

   Method: PATH
   Body:
   {
    "price":"6.99"

}

Method: PUT
Body:
{
 "itemName":"EV Pan",
 "itemCategory":"Appliance",
 "itemPrice":"50.99"
}

Using with React Frontend

Up to this point, your Next backend provided item management API that supports:

- Create service
- Read service
- Update service

Let's try using React front end to utilize the APIs.

The example below demonstrates how to use the API url: http://localhost:3000/api/item to get all items and create (insert) a new item.

Assume that you understand well how to use React Router at this point.

```
import { useEffect, useRef, useState } from "react";

export function Items() {

  const [items, setItems] = useState([]);
  const itemNameRef = useRef();
  const itemCategoryRef = useRef();
  const itemPriceRef = useRef();

  async function loadItems () {
    try {
      const response = await fetch("http://localhost:3000/api/item");
      const data = await response.json();
      console.log("==> data : ", data);
      setItems(data);
    } catch (err) {
      console.log("==> err : ", err);
      alert("Loading items failed")
    }
  }
```

```
async function onItemSave () {
  const uri = "http://localhost:3000/api/item";
  const body = {
    name: itemNameRef.current.value,
    category: itemCategoryRef.current.value,
    price: itemPriceRef.current.value
  }
  const result = await fetch (uri, {
    method: "POST",
    body: JSON.stringify(body)
  })
  const data = await result.json();
  console.log("==> data: ", data);

  loadItems();
}

useEffect(()=>{
  console.log("==> Init...");
  loadItems();
},[]);

return (
  <>
  <table>
    <thead>
      <tr>
        <th>ID</th>
        <th>Name</th>
        <th>Category</th>
        <th>Price</th>
        <th></th>
      </tr>
    </thead>
    <tbody>
      {
        items.map((item, index) => {
          return (
            <tr key={index}>
              <td>{item._id}</td>
              <td>{item.itemName}</td>
              <td>{item.itemCategory}</td>
              <td>{item.itemPrice}</td>
              <td><a href={`/items/${item._id}`}>Edit</a></td>
            </tr>
```

```
            )
          })
        }
        <tr>
          <td> - </td>
          <td><input type="text" ref={itemNameRef}/></td>
          <td><select ref={itemCategoryRef}>
            <option>Stationary</option>
            <option>Kitchenware</option>
            <option>Appliance</option>
            </select></td>
          <td><input type="text" ref={itemPriceRef}/></td>
          <td><button onClick={onItemSave}>Add Item</button></td>
        </tr>
      </tbody>
    </table>
    </>
  )
}
```

Note that when you use POST method with JSON body, you must "stringify" the object, this is a common pitfall!

In this example, it reloads immediately when insert a new item, which has no problem for small data set, but this is not practical approach for large data set.

However, it depends on the UI design about how to handle the loading process or what the process is next after inserting new data.

You may notice that there is a link in every row, it navigates us to the edit page.

The example code below demonstrates how to use the API http://localhost:3000/api/item/[id] to update (PATH) the data.

```
import { useEffect, useState, useRef } from "react";
import { useParams } from "react-router-dom";

export function ItemDetail () {

  const { id } = useParams();
  const itemNameRef = useRef();
  const itemCategoryRef = useRef();
  const itemPriceRef = useRef();

  async function loadItem () {
    const uri = `http://localhost:3000/api/item/${id}`;
```

```
    console.log("==> uri: ", uri);
    const result = await fetch (uri);
    const data = await result.json();
    console.log("==> data :", data);
    itemNameRef.current.value = data.itemName;
    itemCategoryRef.current.value = data.itemCategory;
    itemPriceRef.current.value = data.itemPrice;
  }

  async function onUpdate () {
    const body = {
      name: itemNameRef.current.value,
      category: itemCategoryRef.current.value,
      price: itemPriceRef.current.value
    }
    const uri = `http://localhost:3000/api/item/${id}`;
    console.log("==> uri: ", uri);
    const result = await fetch (uri, {
      method: "PATCH",
      body: JSON.stringify(body)
    })
    if (result.status == 200) {
      loadItem();
    }
  }

  useEffect(()=>{
    loadItem();
  },[]);

  return (
    <div>
      <table>
        <tbody>
          <tr>
            <th style={{textAlign: "left"}}>Name</th>
            <td style={{textAlign: "left", paddingLeft: "20px"}}><input
type="text" ref={itemNameRef}/></td>
          </tr>
          <tr>
            <th style={{textAlign: "left"}}>Categoery</th>
            <td style={{textAlign: "left", paddingLeft: "20px"}}>
              <select ref={itemCategoryRef}>
                <option>Stationary</option>
                <option>Kitchenware</option>
```

```
                <option>Appliance</option>
              </select>
            </td>
          </tr>
          <tr>
            <th style={{textAlign: "left"}}>Price</th>
            <td style={{textAlign: "left", paddingLeft: "20px"}}><input
type="text" ref={itemPriceRef}/></td>
          </tr>
        </tbody>
      </table>
      <hr/>
      <button onClick={onUpdate}>update</button>
    </div>
  )
}
```

In this example, it uses PATCH method even if it submits the whole item data. Commonly, PATCH is frequently used more than PUT since it can handle both partially and full modifications when PUT only does replacement. However, this depends on what BE provided.

If you take a look carefully, you may see that, even after you modify the data successfully, when you press “back” button to the main listing page, the data you fetched at initialization step is still the old version. This is because of caches that are stored inside React.

To fix this bug, you need to strictly inform the React (or any frontend) not to keep the cache. This can be done on backend side by adding some headers.

The following snippet demonstrates to add the no-cache headers along with CORS header in GET method.

```
export async function GET () {

  const headers = {
    "Cache-Control": "no-store, no-cache, must-revalidate, proxy-revalidate",
    "Pragma": "no-cache",
    "Expires": "0",
    ...corsHeaders
  }

  try {
    const client = await getClientPromise();
    const db = client.db("wad-01");
    const result = await db.collection("item").find({}).toArray();
    console.log("==> result", result);
    return NextResponse.json(result, {
```

```
      headers: headers
    });
  }
  catch (exception) {
    console.log("exception", exception.toString());
    const errorMsg = exception.toString();
    return NextResponse.json({
      message: errorMsg
    }, {
      status: 400,
      headers: corsHeaders
    })
  }
}
```

WARNING!

Please be reminded that these examples are very simple and they did not provide complete data validation check logic and these shall not be implemented in the real application. For the real application all the data and results from external APIs must be verified and handled properly.

Assignment 4 – Item CRUD