HOW TO MAKE A C.R.U.D APPLICATION IN THREE DAYS OR LESS

A PRACTICAL GUIDE TO BUILDING DATA-DRIVEN APPLICATIONS

A PRACTICAL GUIDE TO BUILDING DATA-DRIVEN APPLICATIONS

- Welcome to our workshop on building a CRUD Application in three days. In this series of presentations, we will guide you through the essential steps to create, read, update, and delete data in your web application.

A PRACTICAL GUIDE TO BUILDING DATA-DRIVEN APPLICATIONS

- All Data driven applications use the same fundamentals:
  1. Connection Object
  2. Record Setting
  3. CRUD

A PRACTICAL GUIDE TO BUILDING DATA-DRIVEN APPLICATIONS

- Creating a CRUD application essentially involves two main steps:
- 1. Establish a connection to your database - this is the bridge that links your application to the database, enabling communication.
-
- 2. Use an object that alters the records - this is your tool for manipulating the data. It allows you to perform the fundamental operations of:

  - Creating new data
  - Reading existing data
  - Updating current data
  - Deleting unnecessary data

- These two steps form the cornerstone of any CRUD application. It's as straightforward as that.

UNDERSTANDING CRUD

UNDERSTANDING CRUD

- CRUD is an acronym for Create, Read, Update, and Delete. These operations form the backbone of many software systems, particularly those dealing with persistent storage like databases. Here's what each operation does:

UNDERSTANDING CRUD

- Create: The process of adding new data to our application. For example, registering a new user account involves a create operation.
- Read: This operation fetches and displays existing data. If you're viewing a user profile on a social media app, you're using a read operation.
- Update: When existing data needs to change, an update operation is used. Changing your user settings on a website involves an update operation.
- Delete: This operation removes data. For instance, when you remove a photo from a social media platform, a delete operation is performed.

UNDERSTANDING CRUD

- These operations are vital as they form the foundation of data interaction in software systems, enabling users and systems to manage and manipulate data effectively.

INTRODUCTION TO APIS

INTRODUCTION TO APIS

- An API, or Application Programming Interface, is a set of rules that allow different software applications to communicate with each other. It's like a menu in a restaurant. You, the customer, can see a list of all the dishes you can request from the kitchen. Similarly, an API provides a list of all the operations that can be requested from a software component.

- In the context of CRUD operations, APIs provide the necessary endpoints to perform these operations.

INTRODUCTION TO APIS

- RESTful APIs are a type of API that follow the principles of Representational State Transfer (REST). They use standard HTTP methods to perform CRUD operations:

    - POST: Corresponds to the 'Create' operation in CRUD. This method adds new data.
    - GET: Corresponds to the 'Read' operation. It retrieves data.
    - PUT or PATCH: Corresponds to the 'Update' operation. It changes existing data.
    - DELETE: As the name suggests, it corresponds to the 'Delete' operation.

INTRODUCTION TO APIS

- For instance, a RESTful API for managing users in an application might have endpoints like "/users" (to create a new user or fetch all users) and "/users/{id}" (to update or delete a specific user).

- In the next slides, we'll delve deeper into how these operations are carried out in the context of a web application.

CONNECTION OBJECTS

CONNECTION OBJECTS - NODE.JS VS C#

- Connection Objects are essentially a bridge between your application and the database. They handle the communication, allowing you to execute commands and retrieve results.

- In Node.js, the mysql library provides a createConnection function that takes an object with properties such as host, user, password, and database. You can then call connect on the returned object to establish the connection.

CONNECTION OBJECTS - NODE.JS VS C#

- Connection Objects are essentially a bridge between your application and the database. They handle the communication, allowing you to execute commands and retrieve results.

- In Node.js, the mysql library provides a createConnection function that takes an object with properties such as host, user, password, and database. You can then call connect on the returned object to establish the connection.

- In C#, you create a SqlConnection object by passing a connection string to its constructor. The connection string contains information like the data source (server location), the initial catalog (database name), and the type of security used (Integrated Security in the case of Windows authentication). You open the connection by calling the Open method on this object.

CONNECTION OBJECTS - NODE.JS VS C#

```javascript
//NodeJS
const mysql = require("mysql");
const connection = mysql.createConnection({
  host: "localhost",
  user: "user",
  password: "password",
  database: "my_db",
});
connection.connect();
```

```csharp
//C#
string connectionString = "Data Source=.;Initial
  Catalog=my_database;Integrated Security=True";
SqlConnection connection = new SqlConnection(connectionString);
connection.Open();
```

SET UP EXPRESS

NPM COMMANDS

- If you would like to follow along with the node portion first run these commands:
  - npm i express
  - npm install mysql –save
- Install Xampps control pannel https://www.apachefriends.org/
- Run Mysql and apache
- Press admin on mysql
- Create a new database
- Update your connection object

```
const db = mysql.createConnection({
  host: "localhost",
  user: "root",
  database:"raincity",
});
```

SET UP EXPRESS - NODE.JS VS C#

```
const express = require("express");
const bodyParser = require("body-parser");
const mysql = require("mysql");

const app = express();
app.use(bodyParser.json());

const db = mysql.createConnection({
  host: "localhost",
  user: "root",
  database:"raincity",
});

db.connect((err) => {
  if (err) throw err;
  console.log("Connected to database");
});

global.db = db;

app.listen(3000, () => {
  console.log("Server started on port 3000");
});
```

CRUD OPERATIONS

CREATE OPERATIONS - NODE.JS VS C#

- Creating records in a database usually involves executing an INSERT SQL statement. This statement tells the database to add a new row with the specified data to a table.

- In Node.js, you can call the query function on the connection object to execute SQL statements. This function takes the SQL command and a callback function that gets executed when the operation is complete. It's important to check for errors in this callback and handle them accordingly.

- In C#, the SqlCommand class is used to execute SQL commands. You create an instance by passing the SQL command and the connection object to its constructor. You can then add values to the SQL command using the AddWithValue method on the Parameters property of the SqlCommand object. Finally, you call the ExecuteNonQuery method to execute the command.

CREATE OPERATIONS - NODE.JS VS C#

```
using SqlCommand command = new SqlCommand("INSERT INTO Customers (FirstName, LastName, DOB,
  StreetNo, StreetName, City, Province, PostalCode, Country, PhoneNo, Email) Values
  (@FirstName, @LastName, @DOB, @StreetNo, @StreetName, @City, @Province, @PostalCode,
  @Country, @PhoneNo, @Email)", connObj);
command.Parameters.AddWithValue("@FirstName", customer.FirstName);
command.Parameters.AddWithValue("@LastName", customer.LastName);
command.Parameters.AddWithValue("@DOB", customer.DateOfBirth);
command.Parameters.AddWithValue("@StreetNo", customer.HomeAddress.StreetNumber);
command.Parameters.AddWithValue("@StreetName", customer.HomeAddress.StreetName);
command.Parameters.AddWithValue("@City", customer.HomeAddress.City);
command.Parameters.AddWithValue("@Province", customer.HomeAddress.Province);
command.Parameters.AddWithValue("@PostalCode", customer.HomeAddress.PostalCode);
command.Parameters.AddWithValue("@Country", customer.HomeAddress.Country);
command.Parameters.AddWithValue("@PhoneNo", customer.ContactDetail.Phone);
command.Parameters.AddWithValue("@Email", customer.ContactDetail.Email);
```

```
// Add a students
app.post('/student', (req, res) => {
  let { FirstName, LastName } = req.body;
  let sql = `INSERT INTO students (FirstName, LastName) VALUES ('${LastName}', '${FirstName}')`;
  db.query(sql, (err, result) => {
    if (err) throw err;
    res.send('students added');
  });
});
```

READ OPERATIONS - NODE.JS VS C#

- Reading data from a database involves executing a SELECT SQL statement. This statement tells the database to return data that meets the specified criteria.
- In Node.js, you can execute the SELECT statement using the query function on the connection object. This function takes the SQL command and a callback function. The callback function gets called with the results of the operation, which can be accessed via the second argument.
- In C#, you use the SqlCommand class to execute the SELECT command. After creating the SqlCommand object, you call its ExecuteReader method to get a SqlDataReader object. This object allows you to read the returned rows one by one.

READ OPERATIONS - NODE.JS VS C#

```
// Get all student
app.get('/student', (req, res) => {
  let sql = 'SELECT * FROM student';
  db.query(sql, (err, result) => {
    if (err) throw err;
    res.send(result);
  });
});
```

```
sqlQuery = "SELECT * FROM Customers WHERE CustomerID = @CustomerID";
command.Parameters.AddWithValue("@CustomerID", id);
command.CommandText = sqlQuery;
```

UPDATE OPERATIONS (RECORD SETTING)

- Updating records involves executing an UPDATE SQL statement. This statement tells the database to modify the existing data that meets the specified criteria.
- In Node.js, you can call the query function on the connection object to execute the UPDATE command.
- In C#, you use the SqlCommand class to execute the UPDATE command. After creating the SqlCommand object, you add the parameters for the placeholders in the SQL command using the AddWithValue method on the Parameters property, and then call the ExecuteNonQuery method to execute the command.

UPDATE OPERATIONS (RECORD SETTING)

```
using SqlCommand cmd = new SqlCommand("UPDATE Customers SET FirstName = @FirstName, LastName =
  @LastName, BranchID = @BranchID, DOB = @DOB, StreetNo = @StreetNo, StreetName = @StreetName,
  City = @City, Province = @Province, PostalCode = @PostalCode, Country = @Country, PhoneNo =
  @PhoneNo, Email = @Email WHERE CustomerID = @CustomerID", connObj);
cmd.Parameters.AddWithValue("@FirstName", customer.FirstName);
cmd.Parameters.AddWithValue("@LastName", customer.LastName);
cmd.Parameters.AddWithValue("@BranchID", 2);
cmd.Parameters.AddWithValue("@DOB", customer.DateOfBirth);
cmd.Parameters.AddWithValue("@StreetNo", customer.HomeAddress.StreetNumber);
cmd.Parameters.AddWithValue("@StreetName", customer.HomeAddress.StreetName);
cmd.Parameters.AddWithValue("@City", customer.HomeAddress.City);
cmd.Parameters.AddWithValue("@Province", customer.HomeAddress.Province);
cmd.Parameters.AddWithValue("@PostalCode", customer.HomeAddress.PostalCode);
cmd.Parameters.AddWithValue("@Country", customer.HomeAddress.Country);
cmd.Parameters.AddWithValue("@PhoneNo", customer.ContactDetail.Phone);
cmd.Parameters.AddWithValue("@Email", customer.ContactDetail.Email);
cmd.Parameters.AddWithValue("@CustomerID", currentID);
```

```
// Update a students
app.put('/student/:id', (req, res) => {
  let { FirstName, LastName } = req.body;
  let sql = `UPDATE student SET name = '${FirstName}', LastName = '${LastName}' WHERE id = ${req.params.id}`;
  db.query(sql, (err, result) => {
    if (err) throw err;
    res.send('students updated');
  });
});
```

DELETE OPERATIONS (RECORD SETTING)

- Deleting records involves executing a DELETE SQL statement. This statement tells the database to remove the existing data that meets the specified criteria.

- In Node.js, you call the query function on the connection object to execute the DELETE command.

- Both examples emphasize the importance of parameterized queries. This is a best practice to avoid SQL Injection attacks. By using placeholders for variables in your SQL commands, you ensure that user-supplied values can't alter the structure of the SQL command.

DELETE OPERATIONS (RECORD SETTING)

```
// Delete a students
app.delete('/student/:id', (req, res) => {
  let sql = `DELETE FROM student WHERE id = ${req.params.id}`;
  db.query(sql, (err, result) => {
    if (err) throw err;
    res.send('students deleted');
  });
});
```

```
const string sqlQuery = $"DELETE FROM Customers WHERE CustomerID = @CustomerID;";
using (SqlConnection connection = new SqlConnection(connectionString))
{
    connection.Open();
    using (SqlCommand command = new SqlCommand(sqlQuery, connection))
    {
        command.Parameters.AddWithValue("@CustomerID", currentID);
        int rowsAffected = command.ExecuteNonQuery();
        if (rowsAffected > 0)
        {
            MessageBox.Show("Delete Successful");
            ClearForm();
        }
        else
        {
            MessageBox.Show("Could Not Delete Customer Data");
        }
    }
}
```