Server-side Web Development 2: Nodejs contd.

Backend Web Development

by: Salahuddin ElKazak

What is an API?

- **API (Application Programming Interface):** A way for different software to communicate.
- **CRUD API:** Handles Create, Read, Update, and Delete operations.
- Used for data management, integrating with other applications.
- A RESTful API typically uses HTTP methods like GET, POST, PUT, DELETE.

Simple API Structure

Create an index.js file to initialize the API.

Start a basic Express server:

```
const express = require('express');
const app = express();
const port = 3000;
app.listen(port, () => console.log(`Server running on port ${port}`));
```

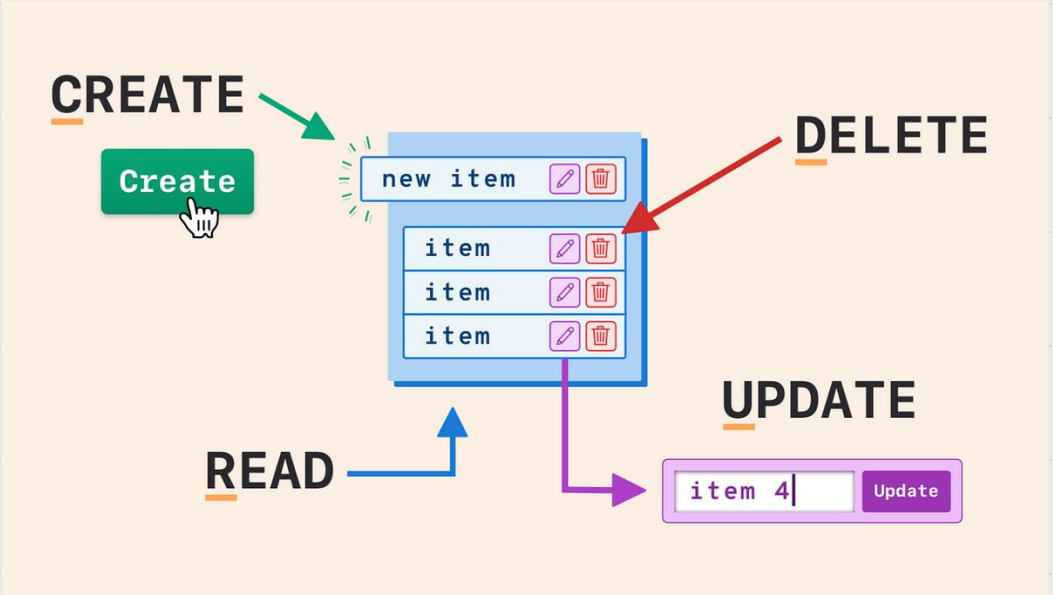
Add a simple test route:

```
app.get('/', (req, res) => res.send('Hello World'));
```

Adding Routes for CRUD Operations

- Define API routes for each CRUD operation:
  - **GET** – Retrieve data
  - **POST** – Create new data
  - **PUT** – Update existing data
  - **DELETE** – Remove data
- Example routes for a "Users" API:

```
app.get('/users', getUsers);
app.post('/users', createUser);
app.put('/users/:id', updateUser);
app.delete('/users/:id', deleteUser);
```



Creating Sample Handlers

- Define handler functions for each route:

```
const getUsers = (req, res) => { /* logic here */ };
const createUser = (req, res) => { /* logic here */ };
const updateUser = (req, res) => { /* logic here */ };
const deleteUser = (req, res) => { /* logic here */ };
```

- Use `req` and `res` to handle incoming requests and send responses.
- Example for `getUsers`:

```
const getUsers = (req, res) => {
  res.json([{ id: 1, name: 'Alice' }, { id: 2, name: 'Bob' }]);
};
```

Separating Routes into Modules

- **Why Separate Modules?**
  - Cleaner, organized code
  - Easier to maintain, test, and scale
- Create a `routes/users.js` file for user routes.
- Move routes to `routes/users.js` and export them:

```
const express = require('express');
const router = express.Router();
router.get('/', getUsers);
router.post('/', createUser);
module.exports = router;
```

Connecting Modules in the Main File

- Import the route module into `index.js`:

```
const userRoutes = require('./routes/users');
app.use('/users', userRoutes);
```

- This allows `/users` routes to be managed separately.

Adding Middleware for Parsing Data

- Use middleware to handle JSON and URL-encoded data:

```
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
```

- This allows processing of `req.body` in POST and PUT requests.

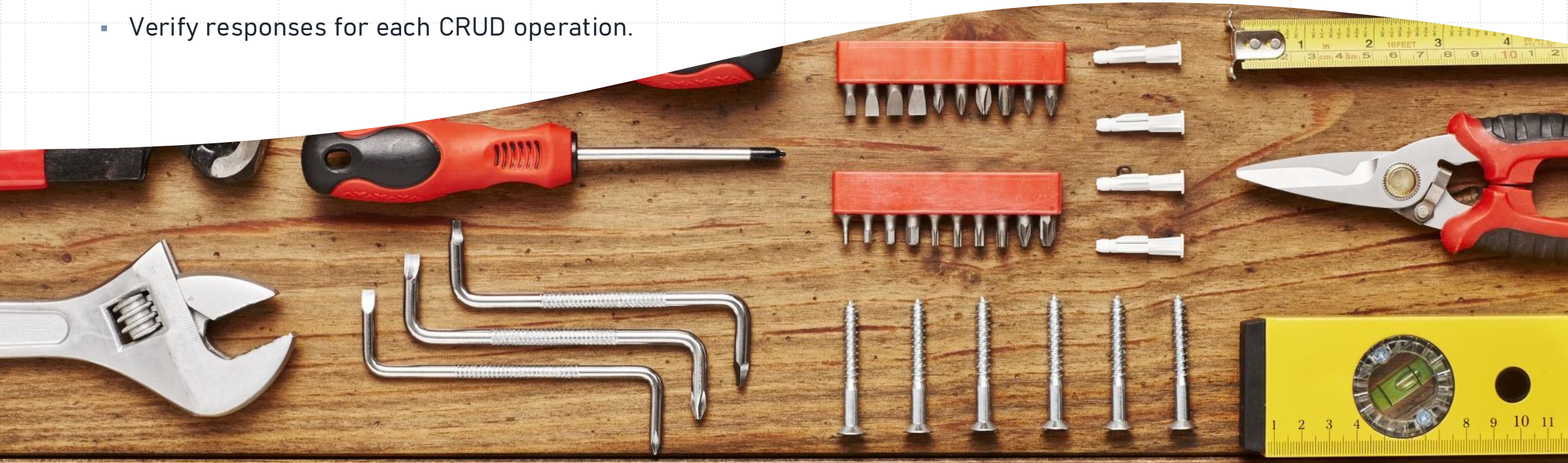
Error Handling

- Create error handling for invalid routes or bad requests.
- Basic 404 error:

```
app.use((req, res) => {
  res.status(404).send('Route not found');
});
```

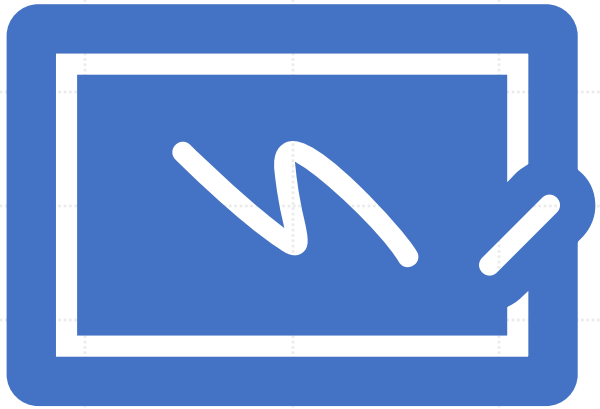
Testing the API

- Test endpoints using:
  - Browser (GET requests only)
  - Tools like *Postman* or *Insomnia* (GET, POST, PUT, DELETE requests) *or curl, if you are brave?!*
- Verify responses for each CRUD operation.



Lab 4 Exercise

Download the instructions to start this exercise!



Extending Node.js with View Engines

- **Why Use a View Engine?**
  - APIs return data, but view engines allow us to render dynamic HTML.
  - Ideal for server-rendered pages, where data is integrated into HTML before it reaches the client.
- **What is a View Engine?**
  - A template engine that lets us inject dynamic data directly into HTML templates.
  - Popular view engines for Express include **EJS**, **Pug**, and **Handlebars**.

Setting Up a View Engine

- **Install a View Engine**: For this example, we'll use EJS (Embedded JavaScript).

  ```
  npm install ejs
  ```

- Configure Express to use EJS:

  ```
  app.set('view engine', 'ejs');
  app.set('views', './views'); // Directory for view files
  ```

- EJS files will go inside the `views` folder.

Rendering Dynamic Data with EJS

- Add a route to render a view:

```
app.get('/photos', (req, res) => {
  res.render('photos', { photos: samplePhotosArray });
});
```

- **EJS Template Example** (`views/photos.ejs`):

```
<h1>Photo Gallery</h1>
<ul>
  <% photos.forEach(photo => { %>
    <li><%= photo.title %> - <img src="<%= photo.url %>" /></li>
  <% }) %>
</ul>
```

- Here, **<%= %>** syntax is used to inject JavaScript variables into HTML.

Real-Time Data with WebSockets

- **Why Use WebSockets?**
  - WebSockets allow for real-time, two-way communication between the server and client.
  - Ideal for live updates, notifications, chat applications, and collaborative tools.
- **How Do WebSockets Differ from HTTP?**
  - HTTP is request-response, but WebSockets maintain an open connection.
  - Allows the server to push data to the client whenever needed.

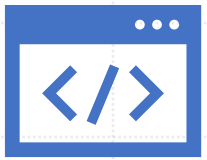
Adding WebSocket Support with Socket.io

- **Socket.io:** A popular library for handling WebSockets in Node.js.
- Install Socket.io:
  ```
  npm install socket.io
  ```
- Initialize Socket.io with Express:
  ```
  const http = require('http').Server(app);
  const io = require('socket.io')(http);

  io.on('connection', (socket) => {
    console.log('A user connected');
    socket.on('disconnect', () => console.log('User disconnected'));
  });

  http.listen(port, () => console.log(`Server running on port ${port}`));
  ```

Creating Real-Time Updates

Add a simple event to emit real-time updates to connected clients.

```
io.on('connection', (socket) => {
  socket.emit('message', 'Welcome to the live photo gallery!');
});
```



Frontend setup (HTML/JS):

```
<script src="/socket.io/socket.io.js"></script>
<script>
  const socket = io();
  socket.on('message', (msg) => console.log(msg));
</script>
```

Real-Time Data Example: Imagine adding comments to photos, where all users see new comments immediately.

Using WebSockets Alongside Your API

- WebSockets complement APIs for:
  - Notifications, live status updates
  - Collaborative features (e.g., shared to-do lists, multiplayer games)
  - Streaming data (e.g., live scores, stock prices)
- APIs provide structured data; WebSockets enable live interactivity.

From Traditional Servers to Serverless

- **What We've Done So Far:**
  - Built a CRUD API using Node.js and Express.
  - Added dynamic views with view engines.
  - Enhanced with WebSockets for real-time interactions.
- **Challenges with Traditional Server Models:**
  - Manual scaling to handle high traffic.
  - Server management and maintenance.
  - Higher costs if servers are idle or underused.

Introduction to Serverless Technologies

- **What is Serverless?**
  - Serverless doesn't mean "no servers"—it means we don't manage them directly.
  - Cloud providers handle scaling, patching, and availability.
  - We focus only on writing code; infrastructure management is abstracted away.
- **Benefits of Serverless:**
  - Automatic scaling to handle traffic spikes.
  - Cost-effective — pay only for what you use.
  - Faster development and deployment cycles.

Serverless Computing Components

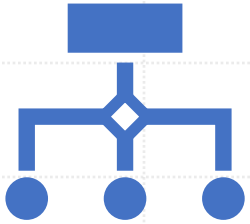
- **Function as a Service (FaaS):**
  - Deploy code as standalone functions (e.g., AWS Lambda, Google Cloud Functions).
  - Code executes in response to events (HTTP requests, file uploads, database triggers).
- **Backend as a Service (BaaS):**
  - Use ready-made services (e.g., Firebase, Auth0) to handle backend features.
  - Common for user authentication, databases, and file storage.

Using AWS Lambda to Deploy a Serverless CRUD API

Why AWS Lambda?

Scales automatically with demand.

Pay-per-use (billed per invocation and runtime).



How to Transition from Express API to Lambda:

Define each API route as a function in Lambda.

Use API Gateway to expose Lambda functions as HTTP endpoints.

Deploy only functions, not full servers.

Setting Up a Serverless API with AWS Lambda and API Gateway

- **Write the Lambda Function**:
  - Example: getPhotos function for retrieving photo data.

```
exports.handler = async (event) => {
  // process event and return a response
  return { statusCode: 200, body: JSON.stringify({ message: 'Hello from Lambda!' }) };
};
```

- **Connect to API Gateway**:
  - Create REST or HTTP API in API Gateway.
  - Map HTTP methods (GET, POST, etc.) to specific Lambda functions.

Serverless Databases and Storage

- **Serverless Database Options**:
  - **AWS DynamoDB**: NoSQL database with automatic scaling.
  - **Firestore**: Real-time NoSQL database, ideal for quick read/write operations.
  - **Aurora Serverless**: Managed SQL database with on-demand autoscaling.
- **Serverless Storage**:
  - **Amazon S3**: Store static files (images, videos, HTML, etc.) at scale.
  - Integrates with Lambda for real-time processing (e.g., processing images upon upload).

Using Serverless for Real-Time Applications

- **Real-Time Capabilities in Serverless**:
  - **AWS AppSync** (GraphQL API) and **Firebase Realtime Database** for real-time data sync.
  - Integrate WebSockets with API Gateway to allow serverless real-time updates.
  - Example: Real-time comments on photos using DynamoDB streams and Lambda.

Pros and Cons of Serverless for APIs

Pros:

- Cost-effective for variable loads.
- No server management or maintenance.
- Automatic scaling and high availability.

Cons:

- Cold starts: Initial delays in function invocation.
- Limited execution time (e.g., AWS Lambda's max 15 minutes).
- Debugging and local development can be more complex.

Recap & Choosing When to Use Serverless

- **When to Use Serverless**:
  - Applications with unpredictable or spiky traffic.
  - Microservices or event-driven architectures.
  - Projects where time to market is critical.
- **When Traditional Servers May Be Better**:
  - Applications needing consistent, long-running processes.
  - Complex APIs requiring heavy processing that exceed serverless limits.

Working with Databases

Fundamentals of Web Development III

Lecturer: Salahuddin ElKazak

salahuddin.elkazak@confederationcollege.ca

Transition to Databases: Why We Need Them

- **The Role of Databases in Web Development**:
  - Serverless functions and APIs perform data operations but need a place to store data.
  - Databases help persist and manage data effectively, supporting CRUD (Create, Read, Update, Delete) operations.
  - Allow us to manage relationships, run complex queries, and serve real-time data.
- **Data Without Servers**:
  - Serverless doesn't replace the need for a database; instead, we connect serverless APIs to cloud-based databases like AWS RDS (SQL) or DynamoDB (NoSQL) for scalable storage.

Types of Databases in Web Development

Relational Databases (SQL):

- Structure data in tables with relationships and defined schemas.
- Use SQL (Structured Query Language) to manage and query data.
- Examples: MySQL, PostgreSQL, SQLite, and Oracle.

NoSQL Databases:

- Flexible schema design, storing data as documents, key-value pairs, or graphs.
- Often chosen for scalability and handling unstructured data.
- Examples: MongoDB (document-based), DynamoDB (key-value), and Redis (key-value cache).

Choosing a Database: SQL vs. NoSQL

- **SQL (Relational)**:
  - Best for structured data and complex queries.
  - Enforces **ACID** (**Atomicity, Consistency, Isolation, Durability**) for transaction safety.
  - Use cases: Financial applications, inventory management, e-commerce.
- **NoSQL (Non-Relational)**:
  - Ideal for unstructured, rapidly changing data.
  - Often prioritizes scalability over strict consistency.
  - Use cases: Real-time analytics, content management, social media feeds.

Setting Up and Managing MySQL

- **What is MySQL?**
  - A popular open-source relational database known for reliability, speed, and ease of use.
  - Ideal for web applications that require structured data storage.
- **Basic Setup**: You can set it up standalone, but it comes with XAMPP!
- **Management Tools for MySQL**:
  - **MySQL Workbench**: GUI for creating databases, tables, and running queries.
  - **phpMyAdmin**: Web-based interface, often used in conjunction with PHP projects.

Setting Up and Managing MongoDB

- **What is MongoDB?**
  - A flexible, document-based NoSQL database using JSON-like documents.
  - Stores unstructured or semi-structured data with dynamic schema.
- **Basic Setup**:
  - **Install MongoDB on Linux**:

    ```
    sudo apt update
    sudo apt install -y mongodb
    ```
  - **Connecting to MongoDB**:
    - Command: `mongo` (or use MongoDB Atlas for cloud-based setup).
- **Management Tools for MongoDB**:
  - **MongoDB Compass**: A GUI to explore and visualize collections and documents.
  - **Robo 3T**: Lightweight MongoDB management tool with intuitive UI.

Understanding SQL and Its Types

- **What is SQL?**
  - SQL (Structured Query Language) is used to interact with relational databases.
  - Enables data querying, manipulation, and schema management.
- **Types of SQL Statements**:
  - **DDL (Data Definition Language)**: Defines database schema (`CREATE`, `ALTER`, `DROP`).
  - **DML (Data Manipulation Language)**: Operates on data (`SELECT`, `INSERT`, `UPDATE`, `DELETE`).
  - **DCL (Data Control Language)**: Manages permissions (`GRANT`, `REVOKE`).
  - **TCL (Transaction Control Language)**: Controls transactions (`COMMIT`, `ROLLBACK`).

Writing Efficient SQL Queries

- **Optimization Basics**:
  - Write only the data you need; avoid `SELECT *` in large tables.
  - Use **JOINs** strategically to fetch related data while minimizing redundancy.
  - Apply **WHERE** clauses to filter rows early and reduce workload.
- **Using Indexes**:
  - Indexes speed up data retrieval for columns that are frequently searched or filtered.
  - Only index columns that will improve query performance, as indexes can increase storage space.

Advanced SQL Optimization Techniques

Normalization:

- Organize data into related tables to avoid redundancy.
- Follow Normal Forms (1NF, 2NF, 3NF) to minimize duplicate data and ensure data consistency.

Query Caching:

- Caching can store the results of frequent queries, reducing database load.
- Use database caching tools like **Redis** for caching intensive queries.

Analyzing and Optimizing:

- Use tools like **EXPLAIN** in MySQL to understand query execution and detect bottlenecks.
- Optimize complex queries with subqueries and derived tables when needed.

Building a Database-Backed API

- **Integrating a Database with a Node.js API**:
  - Connect the CRUD API to a MySQL or MongoDB database.
  - **MySQL**: Use the `mysql` or `mysql2` library to connect and query MySQL databases.
  - **MongoDB**: Use the `mongodb` or `mongoose` library for MongoDB, especially useful for schemas in NoSQL.
- **Example Code**:
  - **MySQL**:

```
const mysql = require('mysql2');
const db = mysql.createConnection({ host: 'localhost', user: 'root', database: 'mydb' });
db.query('SELECT * FROM photos', (err, results) => { console.log(results); });
```

  - **MongoDB**:

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/mydb');
const Photo = mongoose.model('Photo', { title: String, url: String });
Photo.find({}, (err, photos) => { console.log(photos); });
```