**Project Overview: University Data Analysis with Node.js + Express**

Welcome to your full-stack **data analysis project** — where we combine **real-world data**, **server-side programming**, and **frontend interactivity** to build a complete application!

In this project, we’ll explore, process, and visualize a dataset of **1000+ universities worldwide** using modern JavaScript tools.

**What We Will Learn**

**✅ 1. Real-World Data Analysis**

* Work with **actual university ranking data** from a CSV file
* Analyze patterns such as the number of universities per country, top-ranked schools, and score distributions

**✅ 2. Node.js & the Event-Driven Model**

* Understand how Node handles operations **asynchronously**
* Learn how **event-driven architecture** makes file handling fast and efficient

**✅ 3. CSV Parsing in Backend**

* Use the csv-parser package to read and convert CSV data into usable JSON
* Handle large datasets with **streaming** (not loading all at once)

**✅ 4. Filtering and Searching**

* Build **filtering functions** in the frontend to:
  + Search by country
  + Get top/bottom ranked universities
  + Filter by score or year

**✅ 5. Data Aggregation**

* Generate dynamic summaries like:
  + Number of universities per country
  + Country-specific statistics
  + Score-based groupings

**✅ 6. Frontend & Backend Communication**

* Learn how the frontend **fetches** data from an Express backend
* Use JavaScript to manipulate and work with JSON responses

**✅ 7. Graphical Visualization (Optional/Next Phase)**

* Use charting libraries like **Chart.js** or **Google Charts**
* Create visual dashboards (bar charts, pie charts, etc.) from the analyzed data

**Skills You'll Practice**

* Full-Stack JavaScript (Node.js + EJS + browser JS)
* Working with external datasets
* Writing reusable frontend functions
* Client-server interaction via fetch
* Handling data with efficiency and accuracy
* Thinking like a data analyst!

**Tools & Tech Used**

* Node.js – server runtime
* Express.js – web framework
* csv-parser – CSV to JSON parser
* EJS – for rendering frontend pages
* fetch – to get backend data from frontend
* HTML/JS/CSS – for user interface

**📊 Final Goal**

By the end of this project, you’ll have built a **working mini dashboard** that:

* Displays university data
* Lets users search/filter by country or score
* Optionally visualizes the data in charts

And most importantly — you’ll understand how to think like a **developer AND data analyst**

**University Rankings Project – Full Classroom Guide**

**🟢 Project Concept**

You’re given a dataset containing **1000+ university rankings** from around the world in CSV format. Your goal is to build a **Node.js + Express** app that reads this data, sends it to the frontend, and displays it with filtering options like **country**, **rank**, and **score**.

**❓ Question**

“You have a CSV file with over 1000 university records. Would you send all of them to the frontend and let it filter, or filter them on the backend before sending? What are the pros and cons of each? Which one is more scalable?”

**💬 Classroom Activity:**

Split the class into two teams:

* One defends **frontend filtering**
* One defends **backend filtering**

Justify your approach and debate **efficiency**, **speed**, **security**, and **scalability**.

**📦 Dataset**

**Link to download**:  
[Kaggle – World University Rankings](https://www.kaggle.com/datasets/mylesoneill/world-university-rankings?resource=download)  
  
<https://www.kaggle.com/datasets/mylesoneill/world-university-rankings?resource=download>

**🛠️ Project Setup: Step-by-Step**

**1️ Create an Express App**

Create a folder and inside that folder, run:

npx express-generator . --view=ejs

**2️ Add csv-parser**

In order to read CSV files, install csv-parser:

npm install csv-parser

**3️ Add nodemon for Automatic Updates**

Install:

npm install --save-dev nodemon

Update package.json:

{

  "name": "worduni1",

  "version": "0.0.0",

  "private": true,

  "scripts": {

    "start": "node ./bin/www",

    "dev": "nodemon ./bin/www --watch app --watch routes --watch views"

  },

  "dependencies": {

    "cookie-parser": "~1.4.4",

    "csv-parser": "^3.2.0",

    "debug": "~2.6.9",

    "ejs": "^3.1.10",

    "express": "^4.21.2",

    "http-errors": "~1.6.3",

    "morgan": "~1.9.1",

    "nodemon": "^3.1.9"

  }

}

**4️ Create university Route**

Inside the routes folder, create a new file:

var express = require("express");

var router = express.Router();

//add path

var path = require("path");

//we use csv-parser

var csv = require("csv-parser");

//add file

var fs = require("fs");

router.get("/", function (req, res, next) {

  res.render("university", { title: "University" });

});

//show all universities ranking

//this route reads a csv file and returns its content to JSON

router.get("/getuniversities", (req, res) => {

  //define an empty space

  const results = [];

  const filePath = path.join(\_\_dirname, "..", "Data", "cwurData.csv");

  //createREADSTREAM

  fs.createReadStream(filePath)

    .pipe(csv())

    .on("data", (data) => results.push(data))

    .on("end", () => {

      res.json(results);

    })

    .on("error", (err) => {

      res.status(500).json({ error: "failed to read csv file" });

    });

});

module.exports = router;

**📄 As You May Notice: We Use fs.createReadStream**

**STREAMING – Understanding fs.createReadStream and CSV Handling in Node.js**

**🧩 What is fs.createReadStream?**

fs.createReadStream is a method in Node.js used to **read files in small chunks** rather than loading the entire file into memory all at once.  
This is known as **streaming**, and it's especially useful when dealing with **large files** such as CSV datasets.

**🧠 Why Streaming Is Important**

When we have a large CSV file (e.g., 1000+ university rankings), reading the full file at once can consume a lot of memory and slow down the server. Streaming reads the file **bit by bit**, making the application more **efficient and scalable**.

**Question: Does async/await Matter in Our CSV Route?**

We are currently using a **stream-based, event-driven** approach (fs.createReadStream with csv-parser). This style uses **callbacks and event listeners**, not Promises — so async/await doesn’t really apply directly.

**What is the Event-Driven Approach?**

An **event-driven approach** is when your program **waits for certain events to happen** (like a file being read, a button being clicked, or data arriving), and then **responds** to those events by running specific code — called **event handlers**.

You don’t tell the program “do this, then this, then that.”  
Instead, you say:

“When this event happens, run this function.”

**🔁 How It Works in Node.js**

Node.js runs on a **single thread** (just one line of execution). So instead of blocking or waiting for things like file reads or HTTP requests to finish, it **registers event listeners**, and **moves on**.

When something completes (like a file read), Node triggers the **event**, and the code listening for that event runs.

**🧠 Example in Concept (No Code)**

1. You start reading a big CSV file.
2. You attach an event listener: “When a line of data arrives, do this.”
3. While waiting, Node.js keeps doing other work.
4. When a row is ready, it triggers the **data event**.
5. Your function runs and adds that row to the array.

Same with:

* end event → means all data is read
* error event → something went wrong while reading

**🕹 Real-Life Analogy**

Think of a **restaurant**:

* The chef doesn’t stand waiting at the oven for food to cook.
* Instead, a **timer** goes off when the food is ready (an event), and the chef reacts.
* Meanwhile, the chef keeps working on other things.

Node.js works the same way — it doesn’t block, it waits for **events** to happen and responds.

**💡 Why This Matters**

* This makes Node.js **super fast** and **efficient**, especially for I/O-heavy tasks like reading files or handling web requests.
* It also avoids the problem of “freezing” the server when something takes time.

**🧠 So, why not async/await here?**

* createReadStream() doesn’t return a Promise — it returns a stream and uses **event listeners** like on("data") and on("end").
* Because of that, wrapping it in an async function or using await would **not help or change behavior**.
* **It’s already non-blocking and efficient due to Node.js’s event loop**.

**🔍 Would using async/await make it cleaner?**

Only **if** you used a **different method** like:

* Reading the entire file with fs.promises.readFile() (which you shouldn’t do with large files)
* Or wrapping the whole stream logic into a custom Promise

But in your case — since you’re using streaming for **performance and scalability**, async/await wouldn’t be useful or improve it.

**✅ Conclusion**

* Your current approach is already **asynchronous**, just **not promise-based**.
* Using async/await **wouldn’t make a difference** in this specific route.
* You’re doing it right by sticking with stream + events for large CSV data.

**🧪 How It Works with csv-parser**

1. fs.createReadStream reads the file line by line.
2. We connect it to csv-parser, which **converts each line of CSV into a JavaScript object**.
3. These objects are **collected into an array**.
4. Once the reading is complete, the array is sent to the frontend as JSON.

**🔄 Events in the Process**

* **data** – Triggered every time a new row is parsed.
* **end** – Triggered when the file is fully read.
* **error** – Triggered if something goes wrong (e.g., file not found).

**🧰 Tools Used**

* fs – Node.js file handling
* csv-parser – CSV to JSON parser
* Express.js – Routing and server logic
* EJS – Template rendering engine

**5️ Bind Route to app.js**In app.js:

//add university route

var universityRouter = require("./routes/university");

Add this line to your route mounting section:

app.use("/", indexRouter);

app.use("/users", usersRouter);

//university

app.use("/university", universityRouter);

Now, your route is active at:

<http://localhost:3000/university>

**6️ Create View**

Inside views, create:

views/university.ejs

This view will be used to display university data.

**Sample Data Response from Backend**

When visiting:  
http://localhost:3000/university/getuniversities

You’ll receive a JSON like:

[

{

"world\_rank": "1",

"institution": "Harvard University",

"country": "USA",

"national\_rank": "1",

"score": "100",

"year": "2012"

},

...

]

You now have access to 1000+ rows on the frontend.

**Backend vs Frontend Filtering – What’s More Efficient?**

**Option 1: Frontend Filtering**

* Backend sends **all data** to the browser.
* Frontend handles **filtering, sorting, searching**.

✅ Pros:

* Very fast interactivity
* Simpler to implement for small projects

❌ Cons:

* Large initial data load
* Bad for large datasets
* Possible data leakage

**Option 2: Backend Filtering**

* Frontend sends **query** like ?country=USA
* Backend sends **only filtered** results

✅ Pros:

* Scalable
* Efficient resource usage
* More secure

❌ Cons:

* More backend code and logic
* Requires more routes or query handling

**🧭 Teacher’s Hint to Students**

Both approaches work.  
**Frontend filtering** is easier for learning and small datasets.  
**Backend filtering** is ideal for real-world applications with scalability in mind.

**🔍 Filter Ideas to Implement (Frontend for Now)**

Once you send all data to frontend, implement:

1. **Filter by Country**
   * Create a dictionary: { "USA": 300, "UK": 100 }
   * Use a dropdown to filter
2. **Top 100 Universities**
   * Filter by world\_rank <= 100
3. **Bottom 100 Universities**
   * Filter by world\_rank >= 900
4. **Score Range**
   * Add a slider to filter by score range (e.g., 80 to 100)
5. **Search by Institution Name**
   * Text search box (e.g., "Oxford")
6. **Group by Year**
   * If you include more years later

**Function: getAllData()**

This function retrieves the **raw university data** from the backend.

async function getAllData() {

        try {

            const url = "http://localhost:3000/university/getuniversities"

            const response = await fetch(url);

            const data = await response.json()

            console.log(data);

            return data;

        } catch (error) {

            console.error("error", error)

            return error;

        }

    }

**Purpose**:  
To fetch and return all university data from the server for further **filtering** and **analysis** on the frontend.

**Function: filterByCountry(countryName)**

This function is used to **filter universities by country name**. It is a **helper function** that operates on the client side.

//filter by country

    async function filterByCountry(countryName) {

        try {

            const dataSource = await getAllData();

            const filtered = dataSource.filter(item =>item.country && item.country.toLowerCase().trim() == countryName.toLowerCase().trim());

            console.log(`Universities in ${countryName}, filtered`)

            //return

            return filtered;

        } catch (error) {

            console.error("ERror", error);

            return [];

        }

    }

Used in showByCountry() to process input from the user and return results.

**Function: showByCountry()**

This function is triggered when a user enters a country name and clicks a button. It uses filterByCountry() to retrieve and display the relevant data.

//this show filterbycountry

    async function showbyCountry() {

        const countryName = document.getElementById("countryname").value;

        console.log("country", countryName);

        try {

            const dataByCountry = await filterByCountry(countryName);

            console.log(dataByCountry);

        } catch (error) {

            console.log("error", error)

        }

    }

**Function: countByCountry()**

The countByCountry() function returns a summary of how many universities each country has in the dataset.

//count uni by Country

    async function countByCountry() {

        //we create an empty object

        let countryStats = {}

        try {

            const dataSource = await getAllData();

            dataSource.forEach(item => {

                const country = item.country?.trim(); //check if it has country

                if (country) {

                    if (countryStats[country]) {

                        countryStats[country] += 1;

                    } else {

                        countryStats[country] = 1;

                    }

                }

            });

            console.log("university count by country:", countryStats);

            //for graphic export

            return countryStats;

        } catch (error) {

            console.error("error", error)

        }

    }

**Purpose:  
To generate a dictionary object where the keys are country names and the values represent the number of universities in each country.**

**Final scripts**

<script>

    //since we will need for filtering we write a client side function which returns raw data from server

    async function getAllData() {

        try {

            const url = "http://localhost:3000/university/getuniversities"

            const response = await fetch(url);

            const data = await response.json()

            console.log(data);

            return data;

        } catch (error) {

            console.error("error", error)

            return error;

        }

    }

    //this show filterbycountry

    async function showbyCountry() {

        const countryName = document.getElementById("countryname").value;

        console.log("country", countryName);

        try {

            const dataByCountry = await filterByCountry(countryName);

            console.log(dataByCountry);

        } catch (error) {

            console.log("error", error)

        }

    }

    //filter by country

    async function filterByCountry(countryName) {

        try {

            const dataSource = await getAllData();

            const filtered = dataSource.filter(item => item.country && item.country.toLowerCase().trim() == countryName.toLowerCase().trim());

            console.log(`Universities in ${countryName}, filtered`)

            //return

            return filtered;

        } catch (error) {

            console.error("ERror", error);

            return [];

        }

    }

    //count uni by Country

    async function countByCountry() {

        //we create an empty object

        let countryStats = {}

        try {

            const dataSource = await getAllData();

            dataSource.forEach(item => {

                const country = item.country?.trim(); //check if it has country

                if (country) {

                    if (countryStats[country]) {

                        countryStats[country] += 1;

                    } else {

                        countryStats[country] = 1;

                    }

                }

            });

            console.log("university count by country:", countryStats);

            //for graphic export

            return countryStats;

        } catch (error) {

            console.error("error", error)

        }

    }

</script>

**PART 2- ADD GRAPHIC**

To make our university data more **visual and interactive**, we will use [**Chart.js**](https://www.chartjs.org/) — a powerful and easy-to-use charting library for the web.

**📦 Step 1: Include Chart.js**

We will use a CDN to include Chart.js in our HTML file. Add the following inside your <head> tag:

<script src="https://cdn.jsdelivr.net/npm/chart.js@3.9.1/dist/chart.min.js"></script>

Or visit the official links:

* [Chart.js Documentation](https://www.chartjs.org/docs/latest/getting-started/)
* [CDN Links via cdnjs](https://cdnjs.com/libraries/Chart.js)

**📊 Step 2: Understand Our Data**

Our countByCountry() function returns a **JavaScript object** representing the number of universities per country. For example:

1. *{USA: 573, United Kingdom: 144, Japan: 159, Switzerland: 26, Israel: 22, …}*
   1. **Argentina**: 7
   2. **Australia**: 58
   3. **Austria**: 24
   4. **Belgium**: 20
   5. **Brazil**: 36
   6. **Bulgaria**: 2
   7. **Canada**: 72
   8. **Chile**: 8
   9. **China**: 167
   10. **Colombia**: 4
   11. **Croatia**: 2
   12. **Cyprus**: 2
   13. **Czech Republic**: 10
   14. **Denmark**: 12

We will extract:

* **Labels** → Country names (keys of the object)
* **Data** → University counts (values of the object)

These two arrays will be passed into Chart.js to draw a **bar chart**, **pie chart**, or **doughnut chart**.

**Step 3: Using countByCountry() with Chart.js**

Create a function that:

1. Calls countByCountry()
2. Extracts the keys and values into labels and data
3. Passes them into a Chart.js config to render the chart

You can draw:

* 📊 Bar Chart – for comparison between countries
* 🥧 Pie or Doughnut Chart – for proportion of universities
* 📈 Line Chart – if comparing over time (e.g., by year)

**🖼️ Step 4: Add a Canvas Element**

In your HTML, add a canvas where the chart will be drawn:

<canvas id="countryChart" width="600" height="400"></canvas>

Then your JS function will select this canvas and draw the chart using Chart.js.

**✅ Step 5 – Create a Bar Chart with Chart.js**

We will now write a function that:

1. Gets the data from countByCountry()
2. Extracts the country names (labels) and university counts (data)
3. Uses Chart.js to render a **bar chart** in a <canvas>

**📄 Step-by-Step Instructions**

**🖼️ 1. Add a <canvas> in your HTML**

Make sure this is inside your <body>:

<!--canvas-->

    <canvas id="countryChart" style="max-width: 100%; height: 500px;"></canvas>

**2. Add the Chart-Drawing Function**

Place this let chartInstance;

        async function drawCountryBarChart() {

            const countryStats = await countByCountry();

            const sorted = Object.entries(countryStats).sort((a, b) => b[1] - a[1]);

            const topCountries = sorted.slice(0, 10);

            const labels = topCountries.map(entry => entry[0]);

            const data = topCountries.map(entry => entry[1]);

            const ctx = document.getElementById('countryChart').getContext('2d');

            // Destroy previous chart if it exists

            if (chartInstance) {

                chartInstance.destroy();

            }

            chartInstance = new Chart(ctx, {

                type: 'bar',

                data: {

                    labels: labels,

                    datasets: [{

                        label: 'Number of Universities per Country',

                        data: data,

                        backgroundColor: 'rgba(54, 162, 235, 0.6)',

                        borderWidth: 1

                    }]

                },

                options: {

                    responsive: true,

                    maintainAspectRatio: true,

                    scales: {

                        y: {

                            beginAtZero: true

                        }

                    }

                }

            });

        }

<script> block after your existing JS (or in the same <script>):

3 **Trigger It**

<button onClick="drawCountryBarChart()">Draw Country Chart</button>

A screenshot of a graph

AI-generated content may be incorrect.