Contents

Introduction

Welcome to NodeOps: A Data Odyssey.

All of these coding examples are done on a Windows computer.

Overview

Data Sources from Databases

SQL Server

Azure SQL Services

Oracle DB

Postgre-SQL

MySQL and MariaDB

Snowflake

Cassandra

Apache Hive

MongoDb

AWS Redshift

Querying Redis

Databricks APIs and other tools

Azure Data Lake

Apache Spark

Data Sources from APIs

Twitter

Facebook

LinkedIN

Next part of the book is on building ETLs or ELTs. Both are the same

Nextract
Datapumps
ETL
Empujar
Extraload
Proc-that

Data Lake

Building a Data Lake

Data Warehouse

Building a Datawarehouse

Building a Data Lakehouse

Reporting and Dashboards

Data Sources

Node.js Connect to SQL Server

The SQL Server and Azure SQL Services are two very popular RDBMS' (Relational Database Management System). Your data can be stored in these systems as a source or even as a destination or source for your data to be used for reporting purposes or other analytical purposes, including even to feed AI models.

Depending on the version you are using, you can use standard relational tables or graph tables as well. Certain version like HDInsight, which incorporates Hadoop and clustered base supercomputer versions for big data applications are also available.

I am presenting two drivers that can be used with these various these various versions of the SQL Server platform as well as Sybase based systems. The first example is a port of the TDS driver TDS (Tabular Data Stream) protocol. The second example uses the msnodesqlv8 driver and the mssql npm package for APIs. The third example uses the mssql alone to connect and interact with the databases.

Solution 1 SQL Server using the Tedious driver:

There are many ways to connect to SQL Server from Node.js, depending on the library or framework you use. The first example is with the Tedious driver. Information on the driver can be found in the npm registry at https://www.npmjs.com/package/tedious

Install

You can install using npm:.

```
npm install tedious –save-dev
```

Basic Example

Using the tedious driver and the sequelize ORM12:

The following example uses the Tedious driver and Sequelize, a modern ORM. Of course, you can use Tedious without the ORM module

```
var Connection = require('tedious'.).Connection;
var config = {
  server: 'your_server.database.windows.net',
  authentication: {
    type: 'default',
    options: {
      userName: 'your_username', //update me
      password: 'your_password' //update me
```

```
    }
  },
  options: {
    // If you are on Microsoft Azure, you need encryption:
    encrypt: true,
    database: 'your_database'
  }
};
var connection = new Connection(config);
connection.on('connect', function(err) {
  // If no error, then good to proceed.
  console.log("Connected");
});
```

Solution 2 using msnodesqlv8 driver and the mssql package:

Information can be on these packages are available on their respective npm registry web page by following these urls:

https://www.npmjs.com/package/msnodesqlv8

https://www.npmjs.com/package/mssql

To use the mssql client, you can use either the msnodesql8 driver or the Tedious driver as in the previous example. For accessing an Azure SQL database, you need to specify the encrypt=true options (see above). You can get the server's name from the Azure SQL blade page on Azure.

Install

You can install both the driver and driver using npm as usual:

```
npm install msnodesqlv8 –save-dev
```

```
npm install mssql –save-dev
```

Basic Example

```
const sql = require("mssql/msnodesqlv8");
const conn = new sql.ConnectionPool({
  database: "db_name",
  server: "server_name",
  driver: "msnodesqlv8",
  options: {
    trustedConnection: true
  }
});
conn.connect().then(() => {
  //... sproc call, error catching, etc
});
```

Solution 3 using mssql alone:

Install

```
npm install mssql –save-dev
```

Basic Example

```
var mssql = require("mssql");
var dbConfig = {
  server:"server_name",
  database:"my_database",
  user:"username",
  password:"secret_password",
};
var connection = new mssql.Connection(dbConfig, function (err) {
  var request = new mssql.Request(connection);
  request.query('select * from Users', function (err, recordset) {
    if (err) //... error checks
      console.log('Database connection error');
    console.dir("User Data: "+ recordset);
  });
});
```

Working with Social Media Sources

Node.js Data Extraction from Twitter (X) example code

To work with a various popular social network, you have a series of npm modules at you disposal or you can use the REST APIs directly if you prefer.

Solution 1-Using the twit module

The twit module for example is a Twitter API client for Node.js that supports REST and streaming APIs. This module allows you to create and manage connections, execute requests, and handle responses using Node.js.

Installation

```
npm i twit --save
// Load the twit module
const Twit = require('twit');
```

Basic Examples

```
// Create a Twit object with your API credentials
const T = new Twit({
  consumer_key: 'your consumer key', // replace with your consumer key
  consumer_secret: 'your consumer secret', // replace with your consumer
secret
  access_token: 'your access token', // replace with your access token
  access_token_secret: 'your access token secret', // replace with your
access token secret
});

// Make a GET request to the Twitter API
T.get('search/tweets', { q: 'nodejs', count: 10 }, function(err, data,
response) {
  if (err) {
    // Handle the error
    console.error('Error getting tweets', err);
  } else {
    // Handle the data
    console.log('Got tweets');
    console.log(data);
  }
});
```

Data Extraction and Loading using an ETL

Develop ETL Flows using Node.js

There are several npm modules that are suited for ETL (or ELT) operations. I will provide working examples of some of the more popular.

- Nextract
- Datapumps
- ETL
- Empujar
- Extraload
- Proc-that

Solution 1- using Nextract:

Nextract is a platform for extracting, transforming, and loading data from various sources using Node.js streams. It is an alternative to Java-based ETL tools that are more rigid and complex. Nextract allows you to write scripts that perform common ETL operations using standard npm packages and plugins.

The following is a simple example on how to load data from a JSON file, transform it and rewrite it back to a JSON file. This example can be easily expanded to work with other data sources and transformation depending on your requirements. Also, using the same logic, you can write the transformed data to a data warehouse.

Installation

```
npm i nextract path --save
```

Basic Example

```
**
 * Example: JSON input and sort...
 */
const path     = require('path'),
      Nextract = require(path.resolve(__dirname, '../nextract'));
//Define our input and output files
```

```
const sampleEmployeesInputFilePath = path.resolve(process.cwd(),
'data/employees.json'),

      sampleEmployeesOutputFilePath = path.resolve(process.cwd(),
'data/employees_output.json');

//Tranforms always start with instance of the Nextract base class and a
tranform name

const transform = new Nextract('jsonAndSort');

//We load the core plugin and then other additional plugins our transform
requires

transform.loadPlugins('Core', ['Input', 'Output', 'Sort', 'Logger'])

  .then(() => {

    return new Promise((resolve) => {

      //STEP 1: Read data in from a JSON file (we specify the object path we
care about)

      transform.Plugins.Core.Input.readJsonFile(sampleEmployeesInputFilePath,
'data.employees.*')

        //STEP 2: Pass data in to be sorted (1 element is pushed back and it
is the expected input

        //for a new stream read call to sortOut)

        .pipe(transform.Plugins.Core.Sort.sortIn(['last_name'], ['asc']))

        .on('data', (sortInDbInfo) => {

          if (sortInDbInfo !== undefined) {

            resolve(sortInDbInfo);

          }

        });

    });

  })

  .then((sortInDbInfo) => {

    transform.Plugins.Core.Sort.sortOut(sortInDbInfo)
```

```
      //STEP 3: We want to write the sorted data back out to a new JSON file
so first we use

      //toJsonString to stringify the stream.

      .pipe(transform.Plugins.Core.Output.toJsonString(true))

      //STEP 4: Write out the new file


.pipe(transform.Plugins.Core.Output.toFile(sampleEmployeesOutputFilePath))

      .on('finish', () => {

        //Just logging some information back to the console

        transform.Plugins.Core.Logger.info('Transform finished!');

        transform.Plugins.Core.Logger.info(sampleEmployeesOutputFilePath,
'has been written');

      })

      .on('end', () => {

        transform.Plugins.Core.Logger.info('Transform ended!');

        process.exit();

      });

  })

  .catch((err) => {

    transform.Plugins.Core.Logger.error('Transform failed: ', err);

  });
```

Data Lakehouse

The other end of the data engineering process is the data lake, data warehouse or data marts, which is used a source for data analytics, reporting and data analysis.

Data lakes are centralized repositories that allow you to store all your structured and unstructured data at any scale. They enable you to ingest, store, process, and analyze large volumes of data from various sources. Here are some popular data lake solutions:

Solution 6. Apache Iceberg Lakehouse Platform

Building a data lakehouse with Apache Iceberg in TypeScript involves several steps. Here's a high-level overview to get you started:

1. Set Up Your Environment

- **Install Node.js and TypeScript**: Ensure you have Node.js and TypeScript installed on your machine.
- **Initialize Your Project**: Create a new TypeScript project using `npm init` and configure TypeScript with `tsconfig.json`.

2. Install Required Packages

- **Apache Iceberg**: While Iceberg itself is not a TypeScript library, you can interact with it using REST APIs or through a compatible query engine.
- **Other Dependencies**: Install necessary packages like `axios` for making HTTP requests, and any other libraries you might need for your project.

```
npm install axios
```

3. Set Up Apache Iceberg

- **Data Storage**: Use a storage solution like MinIO (S3 compatible) for your data lake.
- **Catalog**: Set up a catalog service like Apache Hive or AWS Glue to manage your Iceberg tables.

4. Create Data Lakehouse Components

- **Data Ingestion**: Write TypeScript code to ingest data into your Iceberg tables. This can be done by interacting with your storage and catalog services.

```
import axios from 'axios';

const ingestData = async (data: any) => {
    const response = await axios.post('http://your-catalog-
service/ingest', data);
    return response.data;
};

// Example usage
```

```
const data = { /* your data */ };
ingestData(data).then(response => console.log(response));
```

5. Querying Data

- **Query Engine**: Use a query engine like Apache Spark or Dremio to query your Iceberg tables. You can interact with these engines using their REST APIs.

```
const queryData = async (query: string) => {
    const response = await axios.post('http://your-query-engine/query', {
query });
    return response.data;
};

// Example usage
const query = 'SELECT * FROM your_table';
queryData(query).then(response => console.log(response));
```

6. Data Management

- **Schema Evolution**: Manage schema changes and partitioning using Iceberg's capabilities.
- **Versioning**: Utilize Iceberg's data versioning features to handle data updates and rollbacks.

7. Deployment and Monitoring

- **Deploy**: Deploy your application using a containerization tool like Docker.
- **Monitor**: Use monitoring tools to keep track of your data lakehouse's performance and health.

8.Example Project Structure

```
my-datalakehouse/
├── src/
│   ├── index.ts
│   ├── ingest.ts
│   └── query.ts
├── package.json
└── tsconfig.json
```

Building a Data Warehouse

Why Json database

Using a JSON database for business intelligence (BI) can offer several advantages, especially in handling diverse and large datasets. Here are some key reasons why JSON databases are beneficial for BI:

1. Flexibility and Schema-less Design

JSON databases, such as MongoDB and Couchbase, are schema-less, meaning they can store data without a predefined schema. This flexibility allows you to easily adapt to changing data requirements and structures without needing to modify the database schema. There are many options available, including AWS DocumentDB, Azure CosmosDB and GCP BigTable to name a few.

2. Handling Unstructured and Semi-structured Data

Business intelligence often involves analyzing unstructured or semi-structured data, such as logs, social media feeds, and sensor data. JSON databases are well-suited for storing and querying this type of data due to their ability to handle nested and complex data structures.

3. Scalability

JSON databases are designed to scale horizontally, making them capable of handling large volumes of data and high query loads. This scalability is crucial for BI applications that need to process and analyze big data efficiently.

4. Integration with Modern BI Tools

Many modern BI tools and platforms, such as Tableau and Power BI, support JSON data sources. This integration allows for seamless data visualization and analysis, enabling businesses to gain insights from their JSON-stored data quickly.

5. Real-time Data Processing

JSON databases often support real-time data processing and querying, which is essential for BI applications that require up-to-date information. This capability allows businesses to make timely decisions based on the latest data.

6. Ease of Use and Developer-Friendly

JSON is a lightweight and human-readable data format, making it easier for developers to work with. This ease of use can speed up the development and deployment of BI applications, reducing time-to-insight.

Building a Data Warehouse with a NoSQL (Json) database

Solution 1 – building a data warehouse with mongodb

Building a data warehouse with MongoDB involves leveraging its flexible schema, scalability, and powerful querying capabilities to store and analyze large volumes of data. Here's a step-by-step guide to help you get started:

Step-by-Step Guide

Set Up MongoDB

First, you need to set up a MongoDB instance. You can use MongoDB Atlas, a fully managed cloud database service, or set up MongoDB on your own server.

- **MongoDB Atlas**: Sign up for MongoDB Atlas and create a new cluster.
- **Self-Hosted MongoDB**: Download and install MongoDB from the official website.

Design Your Data Model

Design your data model based on the requirements of your data warehouse. MongoDB's flexible schema allows you to store data in JSON-like documents, which can be nested and complex.

- **Collections**: Group related documents into collections.
- **Documents**: Store data in BSON format, which is a binary representation of JSON.

Solution 6 – GCP BigQuery

Building a data warehouse with Google Cloud Platform (GCP) BigQuery using TypeScript involves several steps. Here's a high-level overview to guide you through the process:

1. Set Up Your Environment

- **Install Node.js and TypeScript**: Ensure you have Node.js and TypeScript installed on your machine.
- **Initialize Your Project**: Create a new TypeScript project using npm init and configure TypeScript with tsconfig.json.

2. Install Required Packages

- **Google Cloud SDK**: Install the Google Cloud SDK to interact with GCP services.
- **BigQuery Client Library**: Install the BigQuery client library for Node.js.

Installation

```
npm install @google-cloud/bigquery dotenv
```

Basic Example

3. Set Up GCP BigQuery

- **Create a GCP Project**: Use the GCP Console to create a new project.
- **Enable BigQuery API**: Enable the BigQuery API for your project.
- **Service Account**: Create a service account and download the JSON key file. Store this key securely.

4. Connect to BigQuery

- **Environment Variables**: Store your GCP credentials in a .env file for security.

```
GOOGLE_APPLICATION_CREDENTIALS=path/to/your/service-account-file.json
```

- **Connection Code**: Use the BigQuery client library to connect to your BigQuery instance.

```
import { BigQuery } from '@google-cloud/bigquery';
import dotenv from 'dotenv';

dotenv.config();

const bigquery = new BigQuery();

async function connectToBigQuery() {
    const [datasets] = await bigquery.getDatasets();
    console.log('Datasets:');
    datasets.forEach(dataset => console.log(dataset.id));
}
connectToBigQuery().catch(err => {
    console.error('Error connecting to BigQuery', err);
});
```

5. Data Ingestion

- **ETL Processes**: Implement Extract, Transform, Load (ETL) processes to load data into BigQuery. You can use custom scripts or tools like Google Cloud Dataflow.
- **Batch and Streaming Data**: Use Google Cloud Storage for batch data loading and Pub/Sub for streaming data ingestion.

6. Data Storage and Management

- **Schema Design**: Design your table schema to efficiently store and query data.
- **Partitioning and Clustering**: Use partitioned and clustered tables to optimize query performance and manage large datasets.

7. Data Querying

- **BigQuery SQL**: Use SQL to query your BigQuery tables.

```
async function queryData() {
    const query = 'SELECT * FROM `your-project.your-dataset.your-table` LIMIT 10';
    const [rows] = await bigquery.query(query);
    console.log('Rows:');
    rows.forEach(row => console.log(row));
}

queryData().catch(err => {
    console.error('Error querying data', err);
});
```

8. Scalability and Performance

- **Optimizing Queries**: Use best practices for optimizing your queries, such as avoiding SELECT * and using appropriate filtering.
- **Cost Management**: Monitor and manage your BigQuery costs by setting up budget alerts and using cost-effective storage options.

9. Security

- **Access Control**: Implement IAM roles and policies to manage access to your BigQuery datasets.
- **Encryption**: Use encryption for data at rest and in transit to ensure data security.

10. Monitoring and Maintenance

- **Monitoring Tools**: Use Google Cloud Monitoring and Logging to monitor your BigQuery instance's performance and health.
- **Backup and Recovery**: Set up automated backups and have a recovery plan in place.

Example Project Structure

my-data-warehouse/

├— src/

```
|   ├── index.ts
|   ├── ingest.ts
|   └── query.ts
├── package.json
├── tsconfig.json
└── .env
```

Reporting and Dashboards

Reporting

Reporting is focused on providing detailed information while dashboards are great at providing consolidated views of data. Also, dashboards usually offer drilldown capabilities to allow analyst to dig deeper.

I will focus on Reporting in this first part of the chapter, Dashboards will be covered afterwards. My objective is to provide free solutions. There are many commercial options like ActiveReportsJS available, and I will leave those up to you if your project is focused on commercially available tools. There are many libraries and frameworks available for building reports, both open sourced and commercial. Although it is impossible to cover all options in this space, I want to demonstrate building a report and a reporting application with JavaScript.

Some of the popular libraries and frameworks are for building data tables with or without pagination. Of course, these can be combined with charts and other components. As previously mentioned, I'll focus on visualization in the next section:

Solution 1 – Nuxt DataTable

https://nuxt3-primevue-starter.netlify.app/prime/datatable

Installation

```
npm install nuxt datatables.net-vue3 datatables.net-bs5
```

Basic Example

Here's an example using vue3-easy-data-table:

```
<template>
  <div>
    <EasyDataTable :headers="headers" :items="items" />
  </div>
</template>


<script setup lang="ts">
  import { Header, Item } from "vue3-easy-data-table";


  const headers: Header[] = [
    { text: "PLAYER", value: "player" },
    { text: "TEAM", value: "team" },
    { text: "NUMBER", value: "number" },
    { text: "POSITION", value: "position" },
```

```
    { text: "HEIGHT", value: "indicator.height" },
    { text: "WEIGHT (lbs)", value: "indicator.weight", sortable: true },
    { text: "LAST ATTENDED", value: "lastAttended", width: 200 },
    { text: "COUNTRY", value: "country" },
  ];

  const items: Item[] = [
    {
      player: "Stephen Curry",
      team: "GSW",
      number: 30,
      position: "G",
      indicator: { height: "6-2", weight: 185 },
      lastAttended: "Davidson",
      country: "USA",
    },
    // Add more grocery data here
  ];
</script>
```

Conclusion

In this book I set out with the objective of creating Data Engineering and Business Intelligence stack using Node.js and JavaScript and TypeScript. I think I have delivered on that promise. I have covered some of the most popular data sources, showing you through simple example how to connect and extract or perform CRUD operations using the above-mentioned technology. I also provided examples of some of the different libraries and frameworks for ETL development. I also explored cloud solutions on the three main cloud providers and showed how to build and interact with popular data lakes, data lakehouses like Apache Iceberg and to finish off, I showcased various reporting and dashboarding open-sourced technology.

This stack is by no means a definite reference, but it least shows you what is possible with the Node.js platform.

Of course, this book is a work in progress as the technological landscape is constantly evolving and I hope my book will evolve through different editions.

As I said in the beginning, this is a blueprint and tool for you, so please provide feedback to help me make it better for you.

Thanks

Kevin

References

Index

Appendix