AWS Developer Essentials — Node.js Integration Guide

Goal:
By the end of this document, you'll understand the most commonly used AWS services for backend and full-stack developers, along with small working Node.js examples, NPMs, and best practices.

---

1. Amazon S3 — Simple Storage Service

Description:

Amazon S3 is used to store and retrieve files (like images, videos, PDFs, and backups).
It's a **secure, scalable, and cost-effective** object storage service.

Key Features:

- Stores static files (images, videos, JSON, backups).
- Provides public/private file access via URLs.
- Versioning and lifecycle management (auto-delete old files).
- Used with **CloudFront** for CDN delivery.

Useful NPM Packages:

- `aws-sdk` *(legacy)*
- `@aws-sdk/client-s3` *(modern modular SDK)*

Node.js Example:

```
import { S3Client, PutObjectCommand } from "@aws-sdk/client-s3";
import fs from "fs";

const s3 = new S3Client({ region: "ap-south-1" });

const uploadFile = async () => {
  const fileStream = fs.createReadStream("./image.png");
  const params = {
    Bucket: "my-demo-bucket",
    Key: "uploads/image.png",
    Body: fileStream,
    ContentType: "image/png",
  };
  await s3.send(new PutObjectCommand(params));
  console.log("✅ File uploaded successfully!");
};

uploadFile();
```

---

2. Amazon SQS — Simple Queue Service

Description:

SQS is a **message queue** that helps applications communicate asynchronously.
It's ideal for background jobs, order processing, and event-driven systems.

Key Features:

- Decouples microservices.
- Handles high traffic safely.
- Supports standard and FIFO queues.
- Integrates easily with Lambda and EventBridge.

Useful NPM Packages:

- `@aws-sdk/client-sqs`
- `bullmq` *(for local dev queue simulation)*

Node.js Example:

```
import { SQSClient, SendMessageCommand } from "@aws-sdk/client-sqs";

const sqs = new SQSClient({ region: "ap-south-1" });
const sendMsg = async () => {
  const params = {
    QueueUrl:
"https://sqs.ap-south-1.amazonaws.com/123456789012/myQueue",
    MessageBody: JSON.stringify({ userId: 42, task:
"email_notification" }),
  };
  await sqs.send(new SendMessageCommand(params));
  console.log("📨 Message sent to SQS!");
};

sendMsg();
```

---

3. Amazon SNS — Simple Notification Service

Description:

SNS is used to **send notifications** to multiple subscribers (emails, SMS, HTTP endpoints, or Lambda).
It's great for alert systems, status updates, and broadcasting messages.

Key Features:

- Pub/Sub model (Publish once, deliver to many).
- Sends SMS, Email, or HTTP requests.
- Can trigger Lambda functions automatically.

Useful NPM Packages:

- `@aws-sdk/client-sns`

Node.js Example:

```
import { SNSClient, PublishCommand } from "@aws-sdk/client-sns";

const sns = new SNSClient({ region: "ap-south-1" });

const sendNotification = async () => {
  const params = {
    Message: "🚀 New event is live on our platform!",
    TopicArn: "arn:aws:sns:ap-south-1:123456789012:MyTopic",
  };
  await sns.send(new PublishCommand(params));
  console.log("✅ Notification sent!");
};

sendNotification();
```

---

4. Amazon SES — Simple Email Service

Description:

Amazon SES is a **highly reliable email-sending service** used for transactional and marketing emails.

Key Features:

- Send emails via API or SMTP.
- Email tracking, reputation monitoring.
- Cheaper and scalable compared to third-party services.

Useful NPM Packages:

- `@aws-sdk/client-ses`
- `nodemailer` *(can be configured with SES SMTP)*

Node.js Example:

```
import { SESClient, SendEmailCommand } from "@aws-sdk/client-ses";

const ses = new SESClient({ region: "ap-south-1" });

const sendMail = async () => {
  const params = {
    Source: "noreply@myapp.com",
    Destination: { ToAddresses: ["user@example.com"] },
    Message: {
      Subject: { Data: "Welcome to MyApp 🎉" },
      Body: { Text: { Data: "Thank you for joining our platform!" } },
    },
  };
  await ses.send(new SendEmailCommand(params));
  console.log("📧 Email sent successfully!");
};

sendMail();
```

---

5. Amazon Pinpoint / SNS SMS — Send SMS

Description:

Use AWS SNS or Amazon Pinpoint to send **SMS messages** worldwide.
Commonly used for OTPs, alerts, or promotional messages.

Key Features:

- Fast SMS delivery.
- Supports multiple countries.
- Integrated with SNS for unified messaging.

Node.js Example (via SNS):

```
import { SNSClient, PublishCommand } from "@aws-sdk/client-sns";

const sns = new SNSClient({ region: "ap-south-1" });

const sendSMS = async () => {
  await sns.send(new PublishCommand({
    Message: "Your OTP is 246810 ✅",
    PhoneNumber: "+919876543210",
  }));
  console.log("📱 SMS sent successfully!");
};

sendSMS();
```

---

6. AWS Lambda — Serverless Compute

Description:

AWS Lambda lets you run code **without managing servers**.
Just upload your function — AWS handles scaling and execution.

Key Features:

- Event-driven architecture (trigger from S3, API, SQS, etc).
- Pay only for execution time.
- Great for microservices and automation.

Useful NPM Packages:

- `aws-lambda`
- `serverless` *(for deploying easily)*

Sample Handler:

```
export const handler = async (event) => {
  console.log("📦 Lambda event received:", event);
  return { statusCode: 200, body: "Hello from Lambda!" };
};
```

---

7. AWS API Gateway

Description:

A **fully managed service** to create, publish, and secure APIs at scale.

Key Features:

- Acts as a bridge between frontend and Lambda/EC2.
- Supports rate-limiting, caching, and authorization.
- Easy integration with Cognito and Lambda.

Common Use:

- Use API Gateway → Lambda → DynamoDB stack for **serverless apps**.
- Create REST or WebSocket APIs.

---

8. Amazon DynamoDB — NoSQL Database

Description:

A **fully managed NoSQL** database that's lightning fast and scales automatically.

Key Features:

- Key-Value + Document model.
- Auto-scaling and serverless.
- Perfect for chat apps, caching, user sessions.

Useful NPM Packages:

- `@aws-sdk/client-dynamodb`
- `@aws-sdk/lib-dynamodb`

Node.js Example:

```
import { DynamoDBClient, PutItemCommand } from
"@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({ region: "ap-south-1" });

const addUser = async () => {
  const params = {
    TableName: "Users",
    Item: {
      userId: { S: "U123" },
      name: { S: "Pooja" },
    },
  };
  await client.send(new PutItemCommand(params));
  console.log("👤 User added to DynamoDB!");
};

addUser();
```

---

9. AWS IAM — Identity and Access Management

Description:

IAM controls **who can access what** within your AWS account.
It's the backbone of AWS security.

Key Concepts:

- Create **users, groups, and roles.**
- Assign **least privilege policies.**
- Use **access keys** for SDK authentication.
- Rotate credentials regularly.

Pro Tip:

Never hardcode credentials — use **`.env` + IAM roles** instead.

---

10. Other Handy AWS Services for Developers

- **EventBridge** — Event-based communication between services (connect SQS, Lambda, etc).
- **CloudWatch** — Logs and monitors all AWS activity.
- **Cognito** — User authentication and token management.
- **Route53** — Domain and DNS management.
- **ECS/Fargate** — Run Docker containers easily.
- **Elastic Beanstalk** — Deploy Node.js apps quickly without managing infrastructure.

Best Practices

- Store credentials using `.env` and `dotenv`.
- Use IAM roles over access keys.
- Always handle errors using try/catch in SDK calls.
- Configure AWS SDK region globally.
- Use async/await for cleaner async calls.
- Log all AWS actions with timestamps.

Mini Real-World Project Challenge

Goal: Combine S3 + SQS + SES + SNS in one workflow.

Scenario:

1. User uploads a profile image → stored in **S3**.
2. Upload event triggers a message in **SQS**.
3. A background worker reads SQS → sends email via **SES**.
4. A success SMS is sent via **SNS**.

Outcome: You'll understand real-time AWS integration and event-driven architecture.

.

Calling Third-Party APIs in Node.js

Goal:
Learn how to connect your Node.js backend to **external APIs** (payment gateways, weather data, SMS services, social platforms, etc.) in **different ways** — safely, efficiently, and professionally.

---

1. Using Native `fetch()` (Built-in from Node 18+)

Simple, modern, and promise-based.
Great for most REST API calls.

```
// Example: Fetch GitHub user info

const getGitHubUser = async (username) => {

  const res = await fetch(`https://api.github.com/users/${username}`);

  const data = await res.json();
  console.log("👤 GitHub User:", data.login);

};

getGitHubUser("octocat");
```

Why use it?

- No dependency required.
- Cleaner async/await syntax.
- Works like frontend fetch API.

---

2. Using `axios` (Most Popular & Powerful)

Feature-rich HTTP client with interceptors, automatic JSON handling, and better error management.

```
import axios from "axios";

const getWeather = async (city) => {

  const { data } = await
axios.get("https://api.open-meteo.com/v1/forecast", {

    params: { latitude: 12.97, longitude: 77.59, hourly:
"temperature_2m" },

  });

  console.log("🌤️ Weather Data:", data.hourly.temperature_2m[0]);

};

getWeather("Bangalore");
```

Why use it?

- Cleaner syntax.
- Easy interceptors for logging / JWT.

- Auto-converts responses to JSON.
- Works well for APIs with headers or tokens.

---

3. Using `node-fetch` (For Older Node Versions)

A polyfill for browsers' fetch API — helpful if your environment is < v18.

```
import fetch from "node-fetch";

const callAPI = async () => {

  const response = await fetch("https://api.publicapis.org/entries");

  const data = await response.json();

  console.log("📚 APIs List:", data.count);

};

callAPI();
```

Why use it?

- Lightweight.
- Familiar syntax if you know fetch.

---

4. Using `https` Core Module (No External NPM)

Low-level native approach — good for learning how APIs work internally.

```
import https from "https";

https.get("https://api.github.com/users/octocat", {

  headers: { "User-Agent": "node-app" },

}, (res) => {

  let data = "";
  res.on("data", (chunk) => (data += chunk));

  res.on("end", () => console.log("🐱 User:",
JSON.parse(data).login));

});
```

Why use it?

- No dependencies at all.
- Good for learning how requests and streams work.

---

5. Using `request` or `got` (Simplified APIs)

`got` — modern replacement for `request`

```
import got from "got";

const getJoke = async () => {

  const { body } = await got("https://icanhazdadjoke.com/", {

    headers: { Accept: "application/json" },

  });

  console.log("🤣 Joke:", JSON.parse(body).joke);

};

getJoke();
```

Why use it?

- Great for retries, timeouts, and advanced HTTP configs.
- Supports streams.

---

6. With Headers, Tokens & POST Body

Example: Calling an Authenticated API

```
import axios from "axios";

const createUser = async () => {

  const { data } = await axios.post(

    "https://jsonplaceholder.typicode.com/users",

    { name: "Pooja", email: "pooja@example.com" },

    { headers: { Authorization: "Bearer your_api_token" } }

  );

  console.log("✅ User Created:", data);

};

createUser();
```

What You Learn:

- Sending body data (POST/PUT).
- Passing tokens (headers).
- Handling API responses and errors gracefully.

---

7. Error Handling + Retry Logic

Always wrap API calls with `try/catch`:

```
try {

  const res = await axios.get("https://api.fakeendpoint.com/data");

  console.log(res.data);

} catch (err) {

  console.error("❌ API Error:", err.message);
}
```

Pro Tip:
Use npm like `axios-retry` to automatically retry failed calls due to network issues.

---

8. Real-World Task — Third-Party API Integration

Challenge:
Create a Node.js script that:

1. Fetches user data from a **public API** (e.g., JSONPlaceholder).
2. Stores user details in **S3 bucket (as JSON)**.
3. Sends an email via **SES** after successful upload.
4. Publishes a **success message to SNS**.

Outcome:
You'll master real-world integrations combining external + AWS APIs — the way real backend systems work!