NodeJS Starter Guide

This guide will walk you through the process of launching a barebones NodeJS app on Aptible. See it deployed live here.

📘 Get this starter template now and start building your app in minutes! This is the easiest and quickest way to get started with NodeJS. Start building your app without having to worry about the details of deployment.

H1: Table of Contents

H1: Deploy a NodeJS App on Aptible

In this guide, we'll deploy a basic NodeJS app on Aptible. In just three CLI commands, you'll be running an entire application environment in Aptible.

H2: Prerequisites

Deploying a NodeJS app on Aptible requires the following:

a. Create an Aptible account (it's free!)
b. Install Git
c. Install pip
d. Install the Aptible CLI
e. Add an SSH public key to your Aptible user account
f. Install Docker and Docker Compose

g. Install aws-cli (you'll need S3 admin access to your AWS account)

📘 You might also want to read about Apps, Environments, and Stacks on Aptible.

H2: Download our basic NodeJS template app (or use your own)

We're providing you with our starter template app, but you're welcome to bring your own NodeJS app to test with Aptible. We'll assume you know what you're doing and you can skim most of the instructions in this guide. But if you're going to use our app, follow the rest of the instructions in this step.

To get a copy of the NodeJS app's source code on your local machine, use the git clone command:

```
Unset
git clone https://github.com/aptible/template-express.git
```

The code above will create a local folder named `template-express` containing all the files from the repository.

H2: Optional: Media File Handling

Our `template-express` project doesn't contain any images, videos, or audio files, but you might have some in your custom app.

Due to the ephemeral nature of containers in platforms like Aptible, it's best practice to use cloud storage solutions like Amazon S3 for serving media files.

H3: Set up File Storage on Amazon S3

Create an S3 bucket using the AWS Console or AWS CLI. Take note of your S3 bucket name because we'll use it in a later step.

Note your AWS credentials for later use: `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`.

H3: Install Required Packages

To integrate Amazon S3 with your NodeJS app, you'll need some packages to enable S3 functionality:

```
Unset
npm install aws-sdk multer multer-s3
```

Refer to multer's documentation for sample configurations.

H3: Set Aptible Environment Variables

Update the environment variables on Aptible to securely include your AWS credentials (replace `your_access_key` below with your **`AWS_SECRET_ACCESS_KEY`**) and the S3 bucket name (replace `your_bucket_name` below):

```
Unset
aptible config:set --app your-app-name AWS_ACCESS_KEY_ID=your_access_key
AWS_SECRET_ACCESS_KEY=your_secret_key AWS_STORAGE_BUCKET_NAME=your_bucket_name
```

H3: Secure Media Files

If you want private media files, use Amazon S3's signed URLs. You'll need to change your NodeJS views and models to generate signed URLs that expire after a given time. You can use AWS SDK for JavaScript to interact with S3 for secure media delivery.

H2: Custom Configuration

Custom configs help make sure a Dockerized NodeJS app on Aptible allows for tailored performance and enhanced security measures. We use Configuration variables and a special YAML file called .aptible.yml.

NOTE: Most of the instructions below are optional, such as database configuration, media file handling, email configuration, third-party integrations, etc. We're providing them for your benefit, should you choose to deploy a more feature-rich app than the `template-express` example project.

H3: Environment Variables

Aptible loves environment variables. Keep sensitive information (e.g. sensitive keys and configurations) out of your code and make it easy to adjust configurations. See our best practices guide for more.

```
Unset
aptible config:set --app your-app-name NODE_ENV=production
JWT_SECRET=myjwtsecret PORT=3000
```

- `NODE_ENV:` Set to ‘production’
- `JWT_SECRET:` The secret key for JSON Web Tokens.
- `PORT:` The port your app listens on

H3: Database Configuration

To use a database with your NodeJS app on Aptible, you’ll need a `DATABASE_URL`. You can find the URL in the Aptible Dashboard under your app's settings or retrieve it using the Aptible CLI with a command like `Aptible config --app <ENVIRONMENT> | grep DATABASE_URL`. Use it in your app, typically in a `config.js` or `database.js` file.

```
Unset
aptible config:set --app your-app-name DATABASE_URL=your-database-url
```

H3: Media Files

If you’re serving media files via Amazon S3 (as we described above), set the AWS keys and bucket as environment variables.

```
Unset
aptible config:set --app your-app-name AWS_BUCKET=mybucket AWS_REGION=myregion
AWS_ACCESS_KEY_ID=myaccesskey AWS_SECRET_ACCESS_KEY=mysecretkey
```

H3: Email Configuration

If you're using a transactional email service like SendGrid:

```
Unset
aptible config:set --app your-app-name SMTP_HOST=smtp.sendgrid.net
SMTP_USER=apikey SMTP_PASS=sendgrid-api-key SMTP_PORT=587 USE_TLS=True
```

H3: Third-party Integration

For software like Stripe, keep API keys in environment variables:

```
Unset
aptible config:set --app your-app-name STRIPE_KEY=my-stripe-key
OTHER_API_KEY=my-other-api-key
```

H3: Caching

If you're using Redis for caching, it's a good idea to set the Redis URL as an environment variable.

This makes it easy to change the Redis instance your app connects to without altering the code.

Set up the Environment Variable

In your `.env` file, add the following line:

```
Unset
REDIS_URL=redis://localhost:6379 #assuming you are running Redis locally
```

Used the Environment Variable in Your NodeJS App

Install the `redis` npm package if you haven't already:

```
Unset
npm install redis
```

In your NodeJS code

```
Unset
const redis = require('redis');
const client = redis.createClient(process.env.REDIS_URL);

client.on('connect', function() {
  console.log('Connected to Redis');
});
```

By doing this, you can easily switch Redis instances by just updating the REDIS_URL in your .env file. No code changes needed.

H3: Custom Domains

If you have a custom domain for your app:

```
Unset
aptible config:set --app your-app-name CUSTOM_DOMAIN=my-domain.com
```

H3: Security Settings

Enhance security with more environment variables (e.g., `SECRET_KEY`, `DEBUG`, `ALLOWED_HOSTS`).

```
Unset
aptible config:set --app your-app-name SECURE_REDIRECT=True CSRF_TOKEN=mytoken
```

H3: Scaling and Performance

Based on your app's needs, set environment variables for performance.

```
Unset
aptible config:set --app your-app-name WEB_CONCURRENCY=4
```

H3: Periodic Tasks

If you use something like `node-cron` for scheduled tasks, these will run within your Aptible app container. Make sure to handle failures and retries appropriately, as Aptible doesn't provide built-in support for cron jobs.

H2: Deploy Your App on Aptible

Before deploying an app to Aptible, you have to create it within your Aptible account. (Make sure you're signed in with the Aptible CLI before you start.)

H3: Create your App in Aptible

Navigate to your `template-express` app's root directory and execute the aptible apps:create command:

```
Unset
aptible apps:create --environment=cre-dev test-app
```

You should see a success message: "App test-app created!" For the purpose of this demonstration, we named our app test-app and deployed it to our cre-dev environment, but you can name your app and environment whatever you like.

📘Learn how to create an environment on Aptible.

H3: Deploy your code

Now we're going to execute a git-based deploy to push the `template-express` app to your Aptible environment. Run the following commands to tell git where the deploy will occur and then push the code to Aptible:

```
Unset
git remote add aptible git@beta.aptible.com:cre-dev/test-app.git
git push aptible main
```

After a few seconds of verbose CLI output, you should see the following:

```
Unset
remote: (8≅ | INFO: Deploy succeeded.
To beta.aptible.com:cre-dev/test-app.git
 * [new branch]      main -> main
```

Congratulations! You've deployed your first app to Aptible. Now go test it using the link available in your Aptible account.

H3: Optional: Automate your deploy with CI/CD

You've just completed a very simple template app install, but Aptible is built for much more sophisticated use cases. Integrate your CI/CD workflow using either git- or Docker-based deploys.

H2: Testing and Quality Assurance

Aptible specializes in secure and compliant hosting solutions and takes quality assurance seriously. We conduct thorough testing, covering not just app-specific checks but also the hosting environment and platform-specific nuances.

The following steps will walk you through setting up a parallel test environment specifically for QA functions.

H3: Set up a Test Environment on Aptible

Create a new testing environment so your main app remains unaffected during testing. Use the `aptible apps:create [APP_HANDLE] --environment=[ENVIRONMENT_HANDLE]` command to set up a separate testing environment. The [APP-HANDLE] we used above was "test-app".

H3: Deploy to the Test Environment

After setting up your tests locally, push your Dockerized NodeJS app to your Aptible test environment. Make sure your Dockerfile is correctly configured for whatever tests you're going to run. Then execute the git commands we did earlier:

```
Unset
git remote add aptible git@beta.aptible.com:cre-dev/test-app.git
git push aptible main
```

H3: Run Your Tests

Aptible supports executing commands via SSH, so you should be able to run nearly any test scenario with any performance, security, and functionality testing tools you've deployed. Simply SSH into your app and run your tests:

```
Unset
aptible ssh --app [APP_HANDLE] --command "npm test" #To run your NodeJS test within the Aptible environment
```

H3: Clean Up Your Environment

Once testing is complete and you're satisfied, you might wish to remove the test environment to save on costs.

```
Unset
aptible apps:deprovision --app [APP_HANDLE] #To remove the app from the corresponding Aptible environment
```

Tips

- Continuously review and improve your test suite as your app evolves.
- If your NodeJS app has a frontend component, consider using browser testing platforms that integrate with Aptible to test across different browsers and devices.
- See our Best Practices Guide for more information on setting up your Aptible account with a production ready environment.

H2: Scaling and Maintenance

Applications deployed on the Aptible platform may experience variable demand levels. To accommodate such fluctuations, users have the option to either allocate supplementary resources to their existing environment or to expand their deployment through the addition of additional containers.

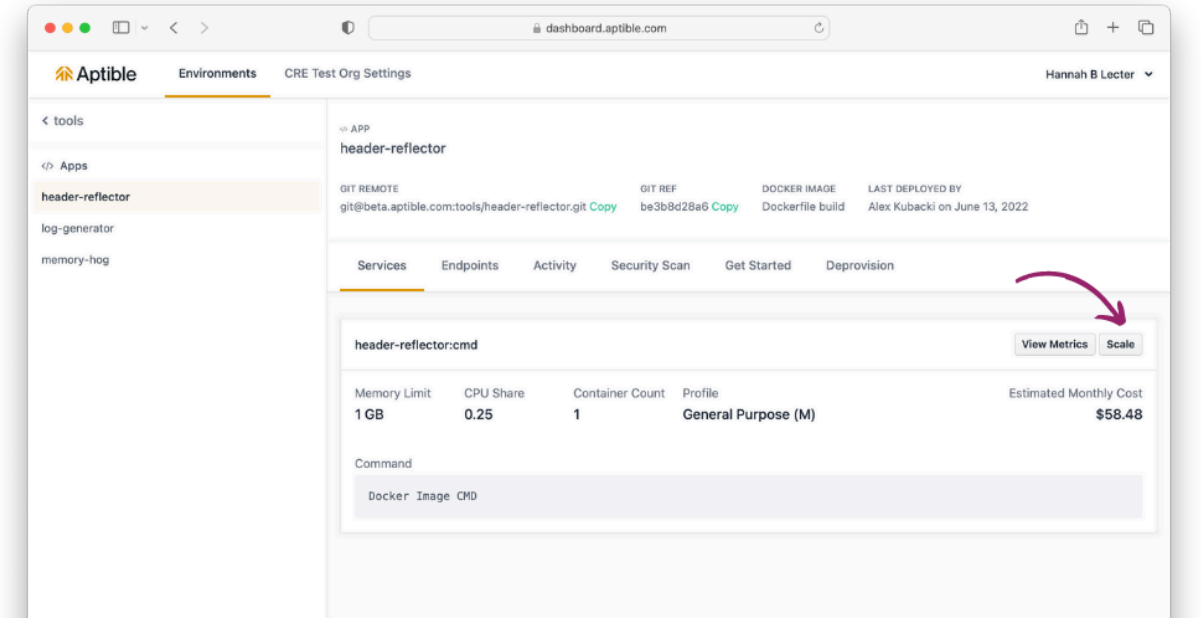
To manage network traffic across multiple containers, Aptible employs a load balancing mechanism. This ensures the efficient distribution of incoming network requests. Various methods for scaling configurations are supported by Aptible and are elaborated upon in our scaling documentation.

H3: Manual Scaling

Within the Aptible platform, manual scaling affords users the opportunity to deliberately modulate resource allocation in accordance with forecasted requirements. This enables the pre-emptive

expansion of container or resource availability in anticipation of increased traffic, as well as the reduction of such resources during periods of diminished demand to conserve resources.

To execute manual scaling, proceed to the deployment settings accessible through the Aptible dashboard. From there, navigate to the specific environment housing your application, select the 'Apps' tab, choose the NodeJS application, and then opt for the 'Scale' option.



To scale your Aptible app, you can use the aptible apps:scale command, the Aptible dashboard, or the Aptible Terraform Provider.

From this interface, you are afforded the latitude to designate either the quantity of containers or the magnitude of resources to allocate. While this method grants enhanced command over the resources at your application's disposal, it may entail increased expenditures. It is advisable to meticulously evaluate your application's needs and rigorously oversee its performance to circumvent excessive resource allocation.

Beyond horizontal scaling, vertical scaling is another option, whereby the resources allocated to an individual container are augmented. This can be accomplished by escalating the CPU, memory, or storage resources assigned to the container. Although vertical scaling can often prove to be a more economically efficient manner in which to scale your application, it does present complexities that necessitate careful management.

H3: Step-by-Step Instructions

Note: Remember to replace placeholders like **`[APP_HANDLE], [DATABASE_HANDLE], [DESIRED_SIZE], and [NEW_VERSION]`** with the appropriate names or values for your application and environment.

1. **Monitoring Performance Metrics:**
    a. Before scaling, it's important to know the current performance of your app.
    b. Use Aptible's built-in metrics or metric drains.

2. **Scale Containers Vertically (Resources per Container):**
    a. Scale your NodeJS apps vertically by changing the size of Containers, i.e., changing their Memory Limits and CPU Limits.
    b. The available sizes are determined by the Container Profile. Aptible offers a variety of container profiles: general purpose, CPU optimized, and RAM optimized.
    c. If your app requires more resources, you can increase the container size.

```
Unset
aptible apps:scale --app "$APP_HANDLE" SERVICE \
    --container-size SIZE_MB
```

3. **Scale Containers Horizontally (Number of Containers):**
    a. Scale your NodeJS apps horizontally by adding more Containers to a given Service. Services scale up to 32 Containers.
    b. Note that apps scaled to 2 or more Containers are automatically deployed in a high-availability configuration, with Containers deployed in separate AWS Availability Zones.
    c. If you need to handle more concurrent connections or distribute traffic, add more containers. For example, to scale the `web` Service to 2 containers, you would use the following command:

```
Unset
aptible apps:scale --container-count [DESIRED_NUMBER]
```

d. Replace `[DESIRED_NUMBER]` with the number of containers you want.

4. **Scaling Databases:**
   a. Aptible users can restart a database to resize it. The command below restarts the database and resizes it to 2048 MB.

```
Unset
aptible db:restart "$DB_HANDLE" \
   --container-size 2048
```

5. **Regular Backups:**
   a. Aptible automatically backups your databases every 24 hours. You can also configure Aptible to retain additional daily backups, as well as monthly backups.
   b. For monthly backups, Aptible will retain the last automatic backup of each month. This means that you will always have at least one backup of your database that is no more than one month old.
   c. You can view your database backups in two ways::
      i. Using the `aptible backup:list` command.
      ii. Within the Aptible Dashboard, by navigating to the **Database > Backup** tab.

6. **Database Maintenance & Upgrades:**
   a. Aptible performs maintenance for databases since they're managed.

7. **Periodic Reviews with Aptible Metrics**
   a. Routinely monitor the metrics provided by Aptible. This helps in understanding when to scale up/down and in ensuring optimal performance. if you're using NodeJS, you could use a package like axios to make API calls to Aptible's metrics endpoint. Here is how it would look:

```
Python

import aptible
const axios = require('axios');
```

```
async function getMetrics() {
 try {
  const client = axios.create({
   baseURL: 'https://api.aptible.com', // Replace with Aptible's actual API endpoint
   headers: {'Authorization': 'Bearer YOUR_ACCESS_TOKEN'} // Replace with your access token
  });

  const response = await client.get('/metrics'); // Replace with the actual metrics API path
  return response.data;
 } catch (error) {
  console.error(error);
 }
}

async function main() {
 const metrics = await getMetrics();

 metrics.forEach(metric => {
  console.log(metric.name, metric.value);
 });
}

if (require.main === module) {
 main();
}
```

Note: Replace 'YOUR_ACCESS_TOKEN' and the API URLs with your specific details. The above is a simplified example; the actual implementation might require more configuration or error handling based on your needs.

8. **Handling Traffic Spikes:**
    a. Prepare for traffic spikes by pre-scaling your app and database temporarily.
    b. After the traffic normalizes, scale down to manage costs.

8. **Regular Security Reviews:**
    a. Regularly check for updates, make sure your NodeJS app and all dependencies are up to date.
    b. Run vulnerability scans using Aptible's in-built features. 7-.
9. **Cleanup and Optimization:**
    a. Remove services, endpoints, or databases you do not use to optimize costs and performance.

```
Unset
aptible apps:deprovision --app [APP_HANDLE]
```

    b. The code above needs to be specific to resource to deprovision
    c. Only run deprovision if you're sure you want to remove the app permanently
    d. Before running the command, take these precautions:
        i. **Be Specific:** Ensure the `[APP_HANDLE]` is correctly specified to avoid accidentally deleting the wrong resource.
        ii. **Finality:** This action is irreversible. Once you deprovision an app, it's gone forever. Double-check and be 100% sure before proceeding.
        iii. **Backup:** Always have a backup of your data, especially if you're making significant changes or deletions
10. **Tips:**
    a. Use metric drains to monitor your NodeJS app's performance and receive alerts when metrics go beyond your desired thresholds.
    b. Always test changes in a staging environment before applying to production.

H2: Troubleshooting and Debugging

Troubleshooting and debugging your Dockerized NodeJS app are inevitable. Doing so will help you maintain a reliable and smoothly functioning application. Follow these steps to identify and resolve issues quickly:

H3: Logging and Error Handling:

Use comprehensive logging in a NodeJS app to diagnose issues and understand the behavior of your application. Let's break down logging and error handling in a NodeJS app in Aptible:

1. **Initialize Comprehensive Logging:**

    a. Setup NodeJS Logging in your main server or a separate config file.

        - Use a package like Winston or Morgan for logging
        - Make sure log files, such as **`debug.log`,** have correct permissions and are accessible.

    b. Implement App-level Logging. In NodeJS, you might use Winston like this:

```
Unset
const winston = require('winston');

const logger = winston.createLogger({
  level: 'info',
  format: winston.format.json(),
  transports: [
    new winston.transports.File({ filename: 'error.log', level: 'error' }),
    new winston.transports.File({ filename: 'combined.log' }),
  ],
});

// For console output
logger.add(new winston.transports.Console());
```

    c. **Handle Exceptions** in your views or other components:

        i. In an Express.js route, you could have exceptions like this:

```
JavaScript
app.get('/some_route', (req, res) => {
  try {
    // some code here
  } catch (e) {
    logger.error(`Error encountered: ${e}`);
    res.status(500).send('Something broke!');
  }
});
```

2. **Capture System-Level Logs**

    a. **Database Queries**: Log database queries specific to your Node.js database client. For example, if you're using Sequelize, you can enable logging in its configuration.

    b. **Server Operations:** If you're using Express, you can capture logs from your Node.js server by using middleware like morgan to log HTTP requests.

    Here's a quick example with Sequelize and Morgan in an Express app:

```
JavaScript
// Enable Sequelize logging
const sequelize = new Sequelize('database', 'username', 'password', {
 logging: console.log,
});

// Enable Morgan for HTTP request logging
const morgan = require('morgan');
app.use(morgan('combined'));
```

3. **Integrate Error Reporting Tools:**

    a. Install Sentry's Node.js SDK:

        i. Run `npm install @sentry/node`

    b. Setup Sentry in your NodeJS project:

    Import Sentry and initialize it at the top of your main server file.

```
Unset
const Sentry = require("@sentry/node");

Sentry.init({
  dsn: "YOUR DSN HERE",
  tracesSampleRate: 1.0
});
```

Remember, you need to initialize Sentry at the beginning of your main server file, usually something like app.js or server.js.

4. **Review Configuration Settings**
    a. **Verify Debug Mode:** Set debug to `STDOUT` (local file isn't recommended). With `STDOUT` for errors, users can review log drain to check logs.
    b. **Database Settings:** Check your database connection settings. Usually found in a .env file or in your main server file..
    c. **Domain Verification:** Make sure your domain is whitelisted in your server's config. For Express.js, this might be handled by a CORS middleware.

5. **Review Configuration Settings**
    a. **Check server or container:** Use `echo $MY_ENV_VARIABLE` in the terminal to check environment variables.
    b. **Confirm NodeJS's accessibility to environment variables:** Use NodeJS's process.env.MY_ENV_VARIABLE to access environment variables within your app. For environment variable management, you can use packages like dotenv.

6. **Confirm Third-Party Apps & Middleware:**
    a. **Check package.json:** Review your dependencies and devDependencies in your package.json file to make sure you've got all the packages you need.
    b. **Review Middleware:** Go through your server setup, often in files like app.js or index.js, to make sure all middleware is properly configured.
    c. **Database checks:** Make sure your database models and migrations are set up correctly if you're using an ORM like Sequelize or Mongoose. Check for missing tables or errors in the console/logs.

7. **Validate External Service Configurations:**
    a. **Test connections:** Run tests or scripts to confirm connections to caching services, message brokers, or any third-party services are working.
    b. **Middleware and Database:** Make sure all middleware is correctly configured, and check for any database-related issues in logs or during startup.

8. **NodeJS Troubleshooting:**
    a. **Lint and test:** Run linting tools like ESLint and execute unit tests to catch any errors or issues.
9. **Verify Custom Configurations:**
    a. **Environment variables:** Make sure all your custom settings, like API keys or service URLs, are correctly loaded from your .env file or environment variables.
10. **Examine Logs on Aptible:**
    a. **Built-in Dashboard or CLI:** To check your NodeJS app's logs for any unusual behaviors or errors, you can use Aptible's built-in logging dashboard or CLI.
    b. **Log Drain:** You can also use a log drain to send your app's logs to a third-party service for analysis. This can be useful if you want to use a more sophisticated logging solution than Aptible's built-in logging.
    c. **How to Set up Log Drain:** Aptible doesn't store logs for you long term. Instead, you'll use a Log Drain to send logs to a third-party log storage and analysis platform.
        i. Navigate to the Aptible dashboard
        ii. Click on the “Log Drains” option on the sidebar
        iii. Click on “New Log Drain”
        iv. Type & Endpoint: Choose the log drain type like "Syslog TLS" or "HTTPS". Plug in the endpoint info you got from your log service. Could look like `logsN.papertrailapp.com:XXXXX`

    d. Monitor and Analyze Your Logs

    Once logs start flowing into your log management service

    - Set up alerts to be notified about critical errors or suspicious activities
    - Regularly review logs to ensure the application is running smoothly
    - Use the search and visualization tools of your log management service to dig deep into issues.

    f. Troubleshoot Issues

    When issues arise:

    - Use filters to narrow down the logs to a specific time frame or severity level

- Identify patterns — Are errors coming from a specific part of your app?
- Correlate with other events> Were there any deployments or changes around the same time?
- Contact Aptible Support if you're stuck