process.env: What it is and why/when/how to use it effectively

Joseph Matthias Goh Follow
Dec 24, 2017 · 5 min read

So you've gotten past your first few tutorials in Node.js and you've probably seen the line `app.listen(process.env.PORT)` or something to that effect. Why not just specify the port as `3000` instead of typing sixteen characters?

What it is

The `process.env` global variable is injected by the Node at runtime for your application to use and it represents the state of the system environment your application is in when it starts. For example, if the system has a `PATH` variable set, this will be made accessible to you through `process.env.PATH` which you can use to check where binaries are located and make external calls to them if required.

Why environment is important

An application needs to be deployed in order for it to be useful, and this can be anything from code that creates a simple website to complex APIs that engage in intensive computations. Ultimately, if an application is not deployed, no one can use it and it serves no purpose.

When we write our code, we can never be sure where our application can be deployed. If we require a database in development, we spin up an instance of it and we link to it via a connection string — something like `127.0.0.1:3306`. However, when we deploy it to a server in production, we might possibly need to link it to a remote server, say `54.32.1.0:3306`.

Assuming the non-use of the environment, we'd need to either make sure a database is available on the same machine as the application so we can call `127.0.0.1:3306`, which results in tight coupling, low availability and no scalability (we can only deploy one instance of the application since we're depending on that one database).

Or, we'd have to modify our code into a series of branch conditions:

```
let connectionString;
if (runningLocally()) {
  connectionString = 'dev_user:dev_password@127.0.0.1:3306/schema';
} else if (...) {
  ...
} else if (inProduction()) {
  connectionString = 'prd_user:prd_password@54.32.1.0:3306/schema';
}
const connection = new Connection(connectionString);
```

This makes the code more tedious to test because more code execution branches exist, and it is also an eyesore. Now once again using the environment:

```
const connection = new Connection(process.env.DB_CONNECTION_STRING);
```

Specifying an external service dependency allows us to link to a remote load-balancer protected database cluster which can scale independently of the application, and allows us to have multiple instances of our application independently of the database service.

In general, it is considered good practice to always treat service dependencies as attached resources. Define these using your environment.

How to use it

The act of providing environment variables is referred to as *provisioning*. We have two levels to work with when dealing with server provisioning: infrastructure and application levels. We can either set the environment

through application level logic, or we can use a tool to provision an environment for us.

A common application level tool is `dotenv` which allows us to load environment variables from a file named `.env`. Install it via:

```
npm install dotenv --save
```

Loading your environment variables is a one-liner:

```
require('dotenv').config();
```

While this is convenient for development needs, it is considered bad practice to couple an environment with your application, so keep it out by adding `.env` to your `.gitignore` file.

At the infrastructure level, we can use deployment manager tools like PM2, Docker Compose and Kubernetes to specify the environment.

PM2 uses an `ecosystem.yaml` file where you can specify the environment using the `env` property:

```
apps:
  - script: ./app.js
    name: 'my_application'
    env:
      NODE_ENV: development
    env_production:
      NODE_ENV: production
    ...
```

Docker Compose likewise allows for an `environment` property to be specified in a service manifest:

```
version: "3"
services:
  my_application:
    image: node:8.9.4-alpine
    environment:
      NODE_ENV: production
      ...
    ...
```

Kubernetes has an equivalent `env` property in the pod template manifest which allows us to set the environment:

```
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: my_application
spec:
  ...
  template:
    spec:
      env:
        - name: NODE_ENV
          value: production
        ...
```

When to use it

Application configurations

Application configurations do not affect the logical behaviour of the application. For example, your application knows it needs to listen on a port to be accessible, but it doesn't need to know exactly which port. In such situations, we classify these values as environment variables so that we can leave the job to deployment orchestrators to deconflict ports and organise the necessary network map for our application to be accessible.

Linked services configuration

Use the environment to specify how your application should connect to a service dependency. This allows your code to be cleaner and improves testability by allowing the test environment to inject its own set of mock values so that failure can be injected and tested for. Your application needs to connect to a service, but it doesn't need to know exactly where as long as it can connect to it. Leave that to the deployment manager.

Development tools instrumentation

When developing an application locally, it's often useful to have some code instrumentation allowing for fast feedback loops and error isolation. Think live-reload or hot-reload. These features can either be in code blocks triggered by an if-else branch based on the state of `process.env.NODE_ENV`, or via a customised environment variable such as `process.env.HOT_RELOADING_ENABLED`.

Anti-patterns

Some common ways that environments are used wrongly are:

1. Overusing `NODE_ENV` — we are taught by many tutorials out there to use `process.env.NODE_ENV` but not much more, resulting in the tendency to

do if-else branches based on the value of `NODE_ENV`. This kinda kills the purpose of using environment variables.

2. Time sensitive information — if your application requires an SSL certificate/rotating password to communicate with another application deployed within the same server, it would be unwise to specify that as an environment variable. The injected environment represents the state of the environment at runtime and will remain static.

3. Configuring the timezone — Leon Bambrick said in 2010: "There are 2 hard problems in computer science: cache invalidation, naming things, and off-by-1 errors." I'll add another, timezones. When deploying with high availability, our application can be instantiated in multiple availability zones. One instance might be running in a fancy data centre in San Francisco and another in Singapore with our users coming from London. Transact in UTC and leave the timezone resolution to the client-side.

. . .

That's It Folks

Using `process.env.*` correctly results in applications that can be tested with ease and deployed/scaled elegantly. After all, if an application is never

deployed, was it really ever written? Till the next time~

P.S. If you liked this article/found it useful, please consider giving a few □□ so that this may appear on the newsfeed of others like yourself. If you're interesting in reading articles on life as a software engineer and technical musings on agile product development, devops and node.js, you can follow me too to keep updated on what I'm up to.

Nodejs JavaScript Software Development DevOps Web Development

ideas take center stage - with no ads in sight. Watch