- ⏱️ 11 minutes
- ✅ Completed

Acclimating to Environment Variables

As your Express applications increase in complexity, the need to have a convenient way to configure your applications will also increase.

For example, consider an application that uses a database for data persistence. To connect to the database, do you simply provide the username and password to use when making a connection to the database directly in your code? What if your teammate uses a different username or password? Do they modify the code to make it work for them? If they do that, how do you keep the application working on your system?

It's not just the differences between you and your teammates' systems. Your applications won't always run locally; eventually they'll need to run on external servers to facilitate testing or to ultimately serve your end users. In most cases, your application will need to be configured differently when it's running on an external server than when it's running locally.

You need a solution for managing your application's configuration! When you finish this article, you should be able to:

- Recall what environment variables are and how they're commonly used;
- Set and get an environment variable value;
- Store environment variable values in an `.env` file;
- Use a module to organize environment variables; and
- Understand how to run npm scripts in different environments.

What are environment variables?

To understand what an environment variable is, we need to start with understanding what an environment is.

An environment is the system that an application is deployed to and running in. Up to now your applications have been running on your local machine, which is typically referred to as the "local environment" or "local development environment".

For real life applications, there are usually several environments—aside from each developer's local environment—that the application will be deployed and ran within:

- Testing - An environment that's used to test the application to ensure that recent changes don't affect existing functionality and that new features meet the project's requirements.
- Staging - An environment that mirrors the production environment to ensure that nothing unexpected occurs before the application is deployed to production.
- Production - The environment that serves end users. For applications that need to support a large number of users, the production environment can contain multiple servers (sometimes dozens or even hundreds of servers).

Environment variables are application configuration related variables whose values change depending on the environment that the application is running in. Using environment variables allows you to change the behavior of your application by the environment that it's running in without having to hard code values in your code.

How are environment variables commonly used?

In an earlier lesson, you learned how to use the Sequelize ORM to connect to a PostgreSQL database to retrieve, create, update, and delete data. To connect to the database, you provided values for the following configuration variables within a module named `config/database.js`:

```
module.exports = {
  development: {
    username: "mydbuser",
    password: "mydbuserpassword",
    database: "mydbname",
    host: "127.0.0.1",
    dialect: "postgres",
  },
};
```

The database connection settings that you use in your local development environment—in particular the `username` and `password` values—won't be the same that the testing, staging, or production environments will need.

Sequelize allows you to define database connection settings per environment like this:

```
module.exports = {
  development: {
    username: "mydbuser",
    password: "mydbuserpassword",
    database: "mydbname",
    host: "127.0.0.1",
```

```
    dialect: "postgres",
  },
  test: {
    username: "testdbuser",
    password: "testdbuserpassword",
    database: "testdbname",
    host: "127.0.0.1",
    dialect: "postgres",
  },
  production: {
    username: "proddbuser",
    password: "proddbuserpassword",
    database: "proddbname",
    host: "127.0.0.1",
    dialect: "postgres",
  },
};
```

While you could hard code different settings for each environment this approach is inelegant, difficult to maintain, and insecure. Application configuration can unexpectedly need to change in test, staging, and production environments. Having to make a code change to change an application's configuration in a specific environment isn't ideal.

Using environment variables for the database connection settings separates the configuration from the application's code and allows the configuration to be updated without having to make a code change.

Where else should you use environment variables? Anywhere that the behavior of your code needs to change based upon the environment that it's running in. Environment variables are commonly used for:

- Database connection settings (as you've just seen)
- Server HTTP ports
- Static file locations
- API keys and secrets

Setting and getting environment variable values

Now that you know what environment variables are and how they're used, it's time to see how to set and get an environment variable value.

Setting an environment variable value

The simplest way to set an environment variable, is via the command line, by declaring and setting the environment variable before the `node` command:

```
PORT=8080 node app.js
```

You can even declare and set multiple environment variables:

```
PORT=8080 NODE_ENV=development node app.js
```

The `NODE_ENV` environment variable is a special variable that's used by many node programs to determine what environment the application is running in. For example, setting the `NODE_ENV` environment variable to `production` enables features in Express that help to improve the overall performance of your application. For more information, see this page in the Express documentation. Sequelize also uses the `NODE_ENV` variable to determine which section of the `config.json` file it will use for database configuration.

This approach also works within an npm `start` script:

```
{
  "scripts": {
    "start": "PORT=8080 NODE_ENV=development node app.js"
  }
}
```

Getting an environment variable value

To get an environment variable value, you simply use the `process.env` property:

```
const port = process.env.PORT;
```

The `process` object is a global Node object, so you can safely access the `process.env` property from anywhere within your Node application.

If the `PORT` environment variable isn't declared and set, it'll have a value of `undefined`.

You can use the logical `OR` (`||`) operator to provide a default value in code:

```
const port = process.env.PORT || 8080;
```

Storing environment variables in a `.env` file

Passing environment variables from the command line is not an ideal solution. Defining an npm `start` script keeps you from having to type the variables again and again, but it's still not a convenient way to maintain them.

Using the `dotenv` npm package, you can declare and set all of your environment variables in a `.env` file and the `dotenv` package will load your variables from that file and set them on the `process.env` property.

To start, install the `dotenv` npm package as a development dependency:

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

Remember that npm tracks two main types of dependencies in the `package.json` file: dependencies (`dependencies`) and development dependencies (`devDependencies`). Dependencies (`dependencies`) are the packages that your project needs in order to successfully run when in production (i.e. your application has been deployed or published to a server that can be accessed by your users). Development dependencies (`devDependencies`) are the packages that are needed locally when doing development work on the project. Passing the `--save-dev` flag when installing a dependency tells npm to install the dependency as a development dependency.

Then add an `.env` file to the root of your project that contains all of your environment variables:

```
PORT=8080
DB_USERNAME=mydbuser
DB_PASSWORD=mydbuserpassword
DB_DATABASE=mydbname
DB_HOST=localhost
```

Pro tip for VS Code users: *Install the DotENV extension to add syntax coloring in `.env` files.*

Loading environment variables on application startup

Using the `dotenv` npm package, it's easy to load your environment variables when your Express application starts up. Just run this code before you configure and start your Express application:

```
// app.js

const express = require('express');

// Load the environment variables from the .env file
require('dotenv').config();

// Create the Express app.
const app = express();

// Define routes.

app.get('/', (req, res) => {
  res.send('Hello from Express!');
});

// Define a port and start listening for connections.

const port = process.env.PORT || 8080;
```

```
app.listen(port, () => console.log(`Listening on port
${port}...`));
```

Another way to use dotenv

Another way to use the dotenv module is to load it before your app loads on the Node.JS command line by using Node.JS's `-r` option to require a module immediately. To use it this way you could change your `npm start` command in your package.json to look like this:

```
  {
  "scripts": {
    "start": "node -r dotenv/config app.js"
  }
}
```

Doing it this way makes sure that all of the environment variables are loaded before you execute any of the code of your app.

Whichever way you decide to go, the main point is to load the contents of your `.env` file as early as possible so those variables will be available to your code.

Keeping the `.env` file out of source control

It's important to keep your `.env` file out of your source control as it will often contain sensitive information like database connection settings or API keys and secrets. If you're using Git for your source control, make sure that your `.gitignore` file includes an entry for `.env` files.

If you're working on a team, you'll need a way to document what the contents of the `.env` should look like. One approach is to update the project's `README.md` file with instructions on what environment variables need to be defined in the `.env` file. Another option is to add an `.env.example` file to your project that mirrors the contents of the `.env` file but replaces any sensitive information with dummy values:

```
PORT=8080
DB_USERNAME=dbuser
DB_PASSWORD=dbuserpassword
DB_DATABASE=dbname
DB_HOST=localhost
```

In many companies, managing the environment variables or `.env` files will be handled by whatever process the company uses to get the code deployed and running on the actual servers. Often companies will have dedicated teams of System Administrators or "DevOps" personnel that handle these tasks. As a developer you may have to work with these teams to determine what the best strategy is for getting the environment variables set for your application.

Using a module to organize environment variables

Earlier we mentioned that the `process` object is a global Node object, which means that you can safely access the `process.env` property from anywhere within your Node application. While that's true, you might find it helpful to encapsulate all of your `process.env` property access into a single `config` module. The `config` module has a single purpose: to import all of your environment variables and export them to make them available to the rest of your application:

```
// config.js

module.exports = {
  environment: process.env.NODE_ENV || 'development',
  port: process.env.PORT || 8080,
  db: {
    username: process.env.DB_USERNAME,
    password: process.env.DB_PASSWORD,
    database: process.env.DB_DATABASE,
    host: process.env.DB_HOST,
  },
};
```

Creating a `config` module also gives you a convenient place to optionally provide default values and to alias environment variable names (notice how the `NODE_ENV` environment variable is being aliased to `environment`).

Any other module in your application that needs access to a configuration variable value just needs to require the `config` module:

Make sure this is done after the environment variables are loaded if you are using the `dotenv` module.

```
// app.js

const express = require('express');

// Load the environment variables from the .env file
require('dotenv').config();

// Get the port environment variable value.
const { port } = require('./config');

// Create the Express app.
const app = express();

// Define routes.

app.get('/', (req, res) => {
  res.send('Hello from Express!');
});
```

```
// Start listening for connections.
app.listen(port, () => console.log(`Listening on port
${port}...`));
```

Notice how destructuring is being used to get a specific configuration variable value when requiring the `config` module:

```
const { port } = require('./config');
```

Without using destructuring, you could get the `port` configuration variable value like this:

```
const config = require('./config');
const port = config.port;
```

Or like this:

```
const port = require('./config').port;
```

Any of these approaches work fine. Deciding which to use is one of the many stylistic choices you'll make as a developer.

Running npm binaries

Using npx to run an npm binary like the Sequelize CLI won't work if you've defined environment variables in an `.env` file for your database connection settings. You'll need to use a tool like the `dotenv-cli` npm package as an intermediary between npx and the Sequelize CLI to load your environment variables from the `.env` file and run the command that you pass into it.

To start, install the `dotenv-cli` package as a development dependency:

```
npm install dotenv-cli --save-dev
```

Then use npx to run the `dotenv` command passing in the command to invoke using the set of environment variables loaded from your `.env` file:

```
npx dotenv sequelize db:migrate
```

Defining environment specific npm scripts

Sometimes you may want to run different npm scripts for different `NODE_ENV` environments.

For instance you might have this in your package.json for local development.

```
{
  "scripts": {
    "start": "nodemon app.js"
```

```
  }
}
```

But in production we may not want to run nodemon, since our code won't be changing constantly. Perhaps, production needs a package.json like this instead:

```
{
  "scripts": {
    "start": "node app.js"
  }
}
```

To keep from having to manually change your npm `start` script before you deploy your application to the production environment, you can use a tool like the `per-env` npm package that allows you to define npm scripts for each of your application's environments.
To start, install the `per-env` package:

```
npm install per-env
```

Then update your npm scripts to this:

```
{
  "scripts": {
    "start": "per-env",
    "start:development": "nodemon app.js",
    "start:production": "node app.js",
  }
}
```

If the `NODE_ENV` environment variable is set to `production`, then running the `start` script will result in the execution of the `start:production` script. If the `NODE_ENV` variable isn't defined, then the `start:development` script will be executed by default.
Using this approach, you can conveniently define a `start` script (or any predefined or custom script) for each environment that your application will be deployed to.

What you learned

In this article, you learned how to

- set and get an environment variable value
- store environment variable values in an `.env` file
- use a module to organize environment variables
- handle running different npm scripts in different environments.