Drew Goodwin Follow
The audience is us.
Dec 1, 2016 · 8 min read

Create-react-app environments

How to create builds targeting multiple environments

Update: create-react-app 1.0.0 added built-in support for configuration files targeting the three pre-defined environments (`development` , `test` , and `production`), and 1.1.0 additionally adds support for variable expansion. I've added a section to the end of this article (see: Built-in support) summarizing the changes and limitations.

If you're using an older version of react-scripts, have not used environment configuration yet, or have more than three environments to build for, you may wish to read the entire article.

Environment variables

React imports environment variables that are defined in a *.env* file at the root of the project. Skip to **Environment Configuration** if you're already familiar with the details, but remember that that values in .env act as a defaults.

Overview

- Variables can be defined in your shell or in a *.env* file at the project root.
- Only variables starting with `REACT_APP_` are imported.
- Imported values are placed in `process.env` , for example `process.env.REACT_APP_SECRET_CODE` .
- The development server must be restarted to see new/updated variables.
- Variables from the shell take precedence those in a *.env* file.
- The value of `NODE_ENV` is set automatically to **development** (when using `npm start`), test (when using `npm test`) or **production** (when using `npm build`). Thus, from the point of view of create-react-app, there are only three environments.

Setting values

Variables can be defined prior to or while running `npm start` or `npm build` :

```
$ REACT_APP_SECRET_CODE=dev123 npm start
$ REACT_APP_SECRET_CODE=prod456 npm build
```

Values could be set in *package.json*, and therefore tracked in revision control:

```
"scripts": {
  "start": "REACT_APP_SECRET_CODE=123 react-scripts start",
  ...
}
```

Setting values in *package.json* is an easy way to share configuration amongst a team, but the approach doesn't scale particularly well. Think about what happens as the number of configuration variables or environments grows. Fortunately variable definitions can be put into a separate file named *.env*.

Values in *.env* are automatically imported by the development server and build process. The format is the same as in the shell (`name=value`), with variable per line:

```
$ cat .env
REACT_APP_SECRET_CODE=123
REACT_APP_HOST_ENV=staging
$
```

Advantages of putting definitions in *.env* include:

- Easy to read as configuration grows.
- Easier to automate (more on this below).
- *The values in the file act as a default*, and can be overridden by one of the methods above (defining in the shell). Thus others can override variables locally without changing *.env*. The multi-environment setup describe below will take advantage of this.

Create react app uses `dotenv` to process the configuration. See the dotenv documentation to learn more its syntax and semantics.

Using values

Imported values are placed in `process.env` automatically. They do not need to be imported manually. Imported values can be used in the same way as those defined in your JS files:

```
if (process.env.NODE_ENV === 'production') { ... }

app.listen(process.env.REACT_APP_PORT);
```

Remember that only those starting with `REACT_APP_` will be imported, and the development server or build must be restarted to pick up new values.

Environment configuration

Now we will cover using the environment variable support in `create-react-app` to build configuration for multiple environments. The aim is a simple way to standardize and automate the process of using environment specific values during the build, regardless of how you build your app.

Overview

- Define variables (with or without default values) in *.env*.
- Place environment specific values/overrides in *.env.local* file in the build location for the target environment.
- Optionally, create environment specific *.env* files (e.g. *.env.staging*) and consume these during the build process.

Define variables in .env

Variables could be set outside of create-react-app (e.g. in a shell startup script), but for the sake of consistency, replicability, and making things less mysterious, we should instead opt for a method that can be used universally in each environment—from local development to production. This helps ensure that a larger subset of the team is able to effectively lend a hand in supporting environments when the need arises. In other words, we want to use the method that best creates a process in which:

- The source of configuration values is known or can be easily deduced.
- Configuration is tracked and builds are reproducible.

At present the way this can achieved in create-react-app is with *.env* files.

Step 1, then, is to create a *.env* file for your project if you don't have one already, place your configuration in it, and update references to the configuration in your code. For now we'll assume these are your development values, but this will be revisited shortly.

It's also worth noting that by using .env files, variables are not global to the environment and will not persist beyond the running of the start command. In my opinion, this is a good thing. However, create-react-app does not yet support for dotenv-expand, so for some setups this may be a downside (*update:* as of 1.1.0 it does support expansion).

Environment specific values in .env.local

Dotenv (what react-scripts uses to import environment variables) also checks for a file named *.env.local* and, if present, imports variables defined within it too. In the case of the same value being defined in both files, the order of precedence is (highest first):

- Shell
- *.env.local*
- *.env*

Thus, values in *.env.local* override values in *.env*.

Assuming you use an intermediary server or folder from which to build for each environment, Step 2, then, is to create a *.env.local* next to the *.env* file with environment specific overrides.

If using default values in *.env* which are designed to be general purpose, this file may be empty for local development. For other environments (e.g. production) you'll probably want to set each value explicitly, as changes must be managed more carefully.

Lastly, an entry for *.env.local* should be added to the revision control ignore list (e.g. *.gitignore*) as it should not be checked in. If publishing a module with a *.npmignore* file, it should be added there as well.

Environment specific .env files

The *.env.local* method works well in the sense that, done right, every environment uses the same method to set and consume environment configuration. However, it still relies on files which are not tracked and therefore does not achieve the goal of reproducibility of the build.

One way to address this is to create *.env* files for each environment (e.g. *.env.staging* instead of *.env.local*) and to track these files in source control. In this case, dotenv will not automatically import the files, but the build script in *package.json* can be modified to do it instead (*update*: as of 1.0.0 this step is not necessary for the three pre-defined environments—the files are automatically imported):

```
"scripts": {
  "build": "sh -ac '. .env.build; react-scripts build'",
  ...
}
```

The `.` command is the same as `source`. It means to execute the file in the current context. In this case the environment variables within will thus be defined before running the next command: `react-scripts build`. This has the same order of precedence as when using *.env.local*.

For multiple build environments, the above method can be generalized:

```
"scripts": {
  "build": "sh -ac '. .env.${REACT_APP_ENV}; react-scripts
build'",
  ...
}
```

Now whomsoever has access to the pertinent configuration can build for that environment by specifying `REACT_APP_ENV` at time of build:

```
$ REACT_APP_ENV=staging npm run build
```

Additional shortcuts could be added for each environment:

```
"scripts": {
  "build": "sh -ac '. .env.${REACT_APP_ENV}; react-scripts
build'",
  "build:staging": "REACT_APP_ENV=staging npm run build",
  "build:production": "REACT_APP_ENV=production npm run
build",
  ...
}
```

Shortening the build command to:

```
$ npm run build:staging
```

Managing configuration in revision control

The advice given by dotenv is to ***not*** check these environment specific config files into revision control. I agree in principle—that checking production settings into your source repository is not a good idea—but I believe it is a very good idea to check your configuration in somewhere. Your builds should be reproducible, which means having the same configuration. Failing to account for this is asking for headaches down the road, when troubleshooting issues. So, unless you have another way to manage configuration and easily reproduce a given build, consider using a separate configuration repository, or a configuration management system, to store and retrieve environment configuration.

One advantage to committing a default *.env* file with local development configuration into revision control is that other team members will automatically receive updates and need only restart their development server when no additional setup is required.

NPM Module

As pointed out by Francis Rodrigues, the react-app-env is a module which automates much of what is covered here, and may be suitable for your needs. To use react-app-env:

Install it as a dev dependency:

```
$ yarn add --dev react-app-env (or npm install --save-dev)
```

And change the start and build scripts:

```
"scripts": {
  "start": "react-app-env start'",
  "build": "react-app-env build'",
  "test": "react-app-env test'",
  ...
}
```

And finally add at least two configuration files: *development.env* and *production.env*. Unlike the steps above, react-app-env automatically prepends `REACT_APP_` to variables. Thus the variable definitions in the example above would be changed to:

```
SECRET_CODE=123
HOST_ENV=staging
```

In the application, however, the full name including `REACT_APP_` must still be used to reference a variable. Given this I would personally prefer the library not automatically prepend, to prevent misunderstandings, but I see no option for this.

By default *development.env* is used imported on `npm start` and `npm test` , and *production.env* on `npm build` . The `--env-file` flag can be used to support more environments:

```
"scripts": {
  "build:staging": "react-app-env --env-file=staging.env",
  ...
}
```

The contents of *.env* can still be used as default values, but `REACT_APP_` is not automatically prepended to variables in this file.

Finally, the module expects the default environment files to be at the root of the project. To customize this (for example to support a separate configuration repository) and add more environments a setup similar to the manual one above could be used:

```
"scripts": {
    "build": "react-app-env --env-
file=config/${BUILD_ENV}.env build",
  "build:staging": "BUILD_ENV=staging npm run build",
  "build:production": "BUILD_ENV=production npm run build",
  ...
}
```

The main differences between this approach one describe above (which does not use react-app-env) is the automatic prepending of `REACT_APP_` , and possibly also better cross platform support.

Built-in support

Create-react-app 1.0.0 added support for automatically reading configuration files for the three pre-defined environments: `development` , `test` , and `production` .

As before, default (lowest priority) values may be defined in *.env*, and the value of `NODE_ENV` is set automatically based on the script used:

- start → development
- test → test
- build → production

The 1.0.0 release adds built-in support for a configuration file corresponding to each of these: *.env.development*, *.env.test*, and *.env.production*. The values from these files have higher priority than the values in *.env*. Each file (including the default file) supports another layer of overrides by adding *.local* to the end. The full order of precedence is (highest first):

1. shell
2. .env.*{environment}*.local
3. .env.*{environment}*
4. .env.local
5. .env

Support for dotenv-expand was added in 1.1.0, enabling variable expansion within the *.env* files (example from create-react-app docs):

```
DOMAIN=www.example.com
REACT_APP_FOO=$DOMAIN/foo
REACT_APP_BAR=$DOMAIN/bar
```

Variable expansion works with variables defined locally (in the same file), or on the shell, but unfortunately does not work across files. A value defined in *.env*, for example, is not able to be expanded within *.env.development*.