Extra Notes - Systems - Environments & Distributions

- Dr Nick Hayward

A brief introduction to environments in a systems approach to design, development, and management.

Contents

Intro We may consider an app's development from the perspective of various environments and build distributions. These commonly include

- development
- staging
- production

App environments As we build an application, we commonly work with various development environments to help progression.

For example, we may progress from *development* to *staging* in preparation for finalising each version of an app for deployment in the *production* environment.

development environment As we develop and build an app, the *development environment* will be used to allow debugging, checking stack traces, getting various diagnostics for app performance &c.

So, we might be working with a local test web server, a dummy data store, and so on.

This environment is also the closest to the developer's code, without minification and optimisation using tools such as Uglify. It is also the code that is used as the record for version history, and ongoing, continuous development.

The app itself will customarily be built with a *debug distribution*. This means we're able to turn on debugging flags, output, logging &c., which will not be part of the final production or release app.

staging environment This environment is used to ensure an app works as expected on a given device, hosted environment, server &c.

For example, we might configure a remote server to match our required production and release setup. This allows us to test the application and project on a remote hosted server, which closely resembles final production requirements and performance without releasing the app to users.

We may run test user scenarios, check connections, debug potential dependency issues, and generally ensure that the app runs as expected on a production grade setup ready for release.

If completed successfully, the app should then progress from staging to production and deployment.

The staging environment should provide a seamless deployment to the production and release environment, assuming each has been configured correctly.

production environment This environment should provide an opportunity to build the app for release and distribution.

A production environment will include optimisation and minification of the app's code, assets, and associated resources.

We may also add release specific encodings and tags, including auto-generated headers, filenames, image stylesheets, and so on to help improve performance, general maintenance, and app security.

However, the code itself is now frozen for release to the end user. As a developer, we do not modify this code directly. Instead, any changes will need to progress through each of the above environments prior to updating a release for the production environment.

User documentation may also be produced and available as part of this environment.

Build distributions As we build an app, we may consider it as a combination of various parts forming the whole.

As we consider various distributions, we should be developing with the same ingredients. As we add these ingredients, such as code and assets, to various distributions, we are then able to vary the concept and structure of the app.

As we develop an app using this mixture of code, assets &c., we may choose to change the assets, modify the distribution, but we will still be working in a given development environment.

So, we may work with either `debug` or `release` distributions, but within any number of different environments. These environments can then be configured to use either of these distributions to achieve the required end result.

However, there is no one-to-one relationship between a given distribution and environment.

choose the right distribution We may commonly consider a distribution relative to the underlying purpose of the target environment.

For example,

- *debugging* - aim for rapid development and debug the app
- *release* - aim for performance and uptime with app

These purposes will help determine the chosen *build distribution*.

Then, in the current development environment, the distribution will be tuned to meet the required development needs.

Common questions for choosing a distribution

- Q: is *idempotent* required for the build flow?
  - Debug: yes - crucial for rapid development and iterative development…
  - Release: yes
- Q: which type of testing?
  - Debug: linters, rapid unit tests…
  - Release: all the available, required tests

- Q: do we need ease of debugging?
  - Debug: yes
  - Release: no
- Q: Do we need optimised, minified code output?
  - Debug: No
  - Release: yes - tuned for performance
- Q: Do we need a fast build flow?
  - Debug: yes - helps with continuous development
  - Release - no, not essential…

build distribution for production As a counterpoint to development, we have a *production environment*.

We are aiming for high-end, quality applications, which are ready for users.

Production is, of course, the environment, which ultimately serves an app to our end users. Access and data is set ready for users, including appropriate security, and privileges.

We may see this as a useful contrast to development environments, which commonly use *dummy* data and broader access.

A production environment will also be built using a *release* distribution,which focuses upon performance with appropriate compression, minification, and obfuscated sources.

build distributions for staging For certain app development, commonly server-based or remote hosting, we may also have an intermediary environment between *development* and *production*.

A *staging environment* may be used to replicate, as close as possible, production configurations. However, a notable difference between staging and production is the nature of user data and remote services. Commonly, a staging environment will neither modify live user data nor interact with live production services.

The goal is to test a production release within a controlled and monitored environment. If an existing production app is live, a staging environment should not interact with this live release.

However, in place of dummy data for development, staging may use a curated, fixed version of current production data. In effect, we may use a snapshot of the required production data.

For build purposes, it is common practice to consider the app relative to a release. So, we may use a release build to ensure optimisation, minification &c. is working as expected ready for release and deployment.

In effect, we may consider a *staging environment* as a replication of *production* without end-user access. It allows us, as developers, to test a production ready app without compromising the production and release.

Build targets For the majority of app development, we may consider build targets relative to *debug* and *release* distributions. Naming may reflect such requirements.

However, general purpose tasks, such as linting, may be applicable to both targets.

So, task naming may often follow the convention of appending or prepending *debug* or *release*. e.g. `jade:debug` .

Such clear separation of naming conventions makes it easier to create aliases for building an app towards a given distribution.

e.g.

```
grunt.registerTask('build:release', ['rollup:release', 'uglify:release', 'clean:release'];
```

custom task We may also define a custom task, and specify relative to builds, e.g. debug and release

```
// custom task - build meta for default debug
grunt.registerTask('buildMeta', function() {
    console.log('debug build...');
    const options = this.options();
    const timestamp = +new Date();
    const contents = `developer = ${options.developer}\ntimestamp = ${timestamp.toString()}
                    \nbuild = ${options.build}`;
    // write details to file for project records...
    grunt.file.write(options.file, contents);
});
```

This will build a markdown file with default options for a debug build. e.g.

```
buildMeta: {
    options: {
        file: './meta.md',
        developer: 'debug tester',
        build: 'debug'
    }
},
```

This default build for debug may be contrasted with the updated custom task for release builds, e.g.

```
//custom task - build meta for release
grunt.registerTask('buildMeta:release', function() {
        console.log('release build...');
    // define task options - incl. defaults
    const options = this.options({
        file: 'build/release_meta.md',
        developer: "spire & signpost",
        build: "release"
    });
    const timestamp = +new Date();
    const contents = `developer = ${options.developer}\ntimestamp = ${timestamp.toString()}
                    \nbuild = ${options.build}`;
    // write details to file for project records...
    grunt.file.write(options.file, contents);
});
```

We may also abstract the creation of the metadata file, and its contents, to a separate function

```
function metaBuilder(options) {
    const timestamp = +new Date();
    const contents = `developer = ${options.developer}\ntimestamp = ${timestamp.toString()}
                    \nbuild = ${options.build}`;
    // write details to file for project records...
    grunt.file.write(options.file, contents);
}
```

and then call this function in each custom task for metadata.

So, relative to variant targets for debug and release, we may update each build as follows

```
// debug build tasks - default tasks during development...
grunt.registerTask('build:debug', ['jshint', 'buildMeta', 'clean']);
// build tasks with specific 'release' targets...
grunt.registerTask('build:release', ['jshint', 'rollup:release', 'uglify:release',
                   'sprite:release', 'buildMeta:release', 'clean']);
```

Environment configuration As noted above, we've already seen how there is a distinction between *environment* and *distribution*.

However, there is also a need for separate configuration for environments. This config will, of course, be separate from the available distributions.

Config for distributions affects the build process, and should have no impact on the application itself. An environment-level config, however, is specific to the final app and its defined context.

For example, the app's current implementation for development compared to staging.

config options Config options will be a reflection of the context of the app for a given environment. For example, we may consider the following

- API authentication
- database connection
- session secrets (usually encrypted)
- web server settings (e.g. a custom port…)

Each of these settings may differ from *development* to *staging* to *production*. Such settings are also something we should avoid bundling with an app, even if the app build includes minification and uglify options. This does not hide or encrypt the settings, and can be reversed.

A sample app developemnt structure is as follows

- Build flows
  - debug distribution - optimise for debugging, testing
  - release distribution - optimise for performance
- then deploy built app to target environment
- App server
  - each environment includes app host
  - host may be local machine, remote server, cloud service &c.
- App server hosted environments
  - development - accessible by individual developer
  - staging - remote access to app for developers, select testers, &c. (trusted group…)
  - production - publicly accessible app

Environment-level config will include secret credentials, and any other applicable config that changes across environments.

Configuring an environment We may now consider how to manage config from different sources, including files, database, app memory, &c.

We also need to review options for securing config data for each environment. We may use various API keys, which vary relative to the current environment.

Whilst apps are developed with text based configs, and developer specific configs, this can quickly become tedious and difficult to migrate across environents. Such options are also prone to errors, accidental bundling in distributions, and so on.

However, we may consider the following alternatives for environment configuration.

waterfall config Waterfall is a simple option for storing configurations. It is, in fact, as simple as choosing relative priorities, which helps determine an order of precedence as we merge configurations.

This pattern is useful because it helps division of config options across different places, yet remain part of the whole.

So, we may commonly define config options in the following places for an app's development

- plain text config files
  - use for settings that do not compromise security, app access &c.
  - n.b. use with discretion
- encrypted files

    - secure distribution of files, config settings &c.
    - n.b. encryption may be hacked or reversed - do not use with API, DB setttings &c.
- machine level config
    - set operating system environment variables
    - standard `.env` files stored at the OS level
    - include local and server OS development
- command line arguments
    - passed to an app at the process level

When we consider each of the above config options, we need to remember their usage relative to an application. For each app, we still need to consider a single point of access for the config variables and settings.

Therefore, we may consider a *root service* for this configuration, and its varied sources.

This service should determine an order of precedence as it provides a requested value. In effect, we may commonly work from low to high priority.

For example, a port number for a server specified as a terminal argument will take precedence over a setting defined in an app config file. So, we may define direct arguments as high priority to text file config.

There are, of course, other options for config settings. We might use cloud storage, authentication based domains, and so on.

So, a sample local config file, `default_config.json` might include the following

```
{
    "NODE_ENV": "development",
    "PORT": 3286
}
```

This is low level config, so we may store this in a plain text file without concern for usage.

We may also use an encrypted file for similar environment settings and config, which may then be shared with fellow developers. We may also push such encrypted config files to version control repos.

However, we need to ensure that different private keys are used for such encrypted files.

encryption options for env config Transmission of config options requires some obvious security measures and controls.

For example, decrypted config files should not be committed to version control. The same is obviously true for the underlying encryption keys. We may, however, store the encrypted files using version control and shared repos.

We may also share simple command line tools to decrypt or update encrypted counterparts for these config resources.

A sample encryption and decryption flow is as follows,

- encryption flow
    - sensitive document - e.g. API secrets, DB connection details…
        * encrypt using a cryptographic key
    - private key - one per document
        * NOT added to version control
        * private RSA keys
    - encrypted document is now generated
    - encrypted data may now be added to version control - e.g. git
- decryption flow
    - encrypted document
    - private key - one per document
        * NOT added to version control
        * private RSA keys
    - sensitive document is now created - decrypted on local machine &c.
    - decrypted config settings &c. may now be used in environment
        * use to configure and execute app for developers &c.

To help us work with this secure flow, we may start with specific directories for a given project's development.

For example, we might create the following

- `env/private`
  - store insecure, unencrypted data for local project use
- `env/secure`
  - store encrypted data, files, &c.

The `env/private` directory should not be committed to version control, e.g. git.

Instead, we may distribute an encryption key using a secure channel, local device, encrypted messaging &c.

To use this key, we may then share in a project's repository a copy of config for various Grunt tasks or other similar build tools. These build tools allow developers to encrypt and decrypt required project files using a corresponding RSA key.

For this process, we may use three common Grunt tasks for encryption,

- generate the private key
- encrypt config &c. using private key
- decrypt config &c. using appropriate private key

So, with RSA encryption, project config may be considered as follows

- create a private key - not shared
- use private key to encrypt config, files &c.
- transmit encrypted file/s with rest of project dev
- for secure project files, update the plain file and then encrypt again…
- encrypted project files will not available to copies without key

environment config at local OS level Another option for environment level config is to avoid the need for encryption by setting config at the OS level.

For release environments, including staging and production, it is often preferable to store sensitive config settings and values in the local environment.

This removes such config from the app's development, and helps speed config updates without the need to push or redeploy project material.

Another benefit of system-level environment variables is that an app's behaviour may be updated without the need to modify the code itself.

However, a common issue with this approach to config is the initial overhead for setup. Whilst a developer may clone a given project's code base, they will not have access to config settings. This will need to be setup on the host system, and then tested with the app's code in a given environment.

Non-sensitive data and config settings may still be shared, but secure date &c. will need to be set local to the environment.

An example config for a local development environment might be saved in a JSON file, which we may block from version control using a standard ignore file.

For example,

```
"NODE_ENV": "development",
"PORT": 3366,
"API_SECRET": "...",
"API_KEY": "..."
```

This config file may also be shared with other project developers, although it will require encryption, as noted above, before sharing.

environment config for staging and production Hosted solutions, including both staging and production, require a different development path for environment variables.

Some hosted services, such as *Heroku*, provide a command line tool, known as *toolbelt*, to configure and maintain such environment settings and variables.

For example, we might set a Heroku hosted environment to `staging` . With this environment, we may use logging or simply test general usage and DB connections.

```
heroku config:add NODE_ENV=staging
```

We may also override such settings using the command line,

```
NODE_ENV=production PORT=3030 node app.js-
```

merge config as waterfall We may, of course, merge these separate sources of config to create an overall project config.

A useful option is a module such as *nconf*, which we may add to a project using NPM.

This module may be used to merge various sources, including JSON files, JavaScript objects, environment variables, command line arguments, and so on.

However, `nconf` will prioritise configuration settings on a simple *first come, first served* basis. So, we need to be careful how we order these sources in the config for `nconf` .

So, we can see the way `nconf` sets up environments,

```
var fs    = require('fs'),
    nconf = require('nconf');
//
// Setup nconf to use (in-order):
//   1. Command-line arguments
//   2. Environment variables
//   3. A file located at 'path/to/config.json'
//
nconf.argv()
 .env()
 .file({ file: 'path/to/config.json' });
//
// Set a few variables on `nconf`.
//
nconf.set('database:host', '127.0.0.1');
nconf.set('database:port', 5984);
//
// Get the entire database object from nconf. This will output
// { host: '127.0.0.1', port: 5984 }
//
console.log('foo: ' + nconf.get('foo'));
console.log('NODE_ENV: ' + nconf.get('NODE_ENV'));
console.log('database: ' + nconf.get('database'));
//
// Save the configuration object to disk
//
nconf.save(function (err) {
  fs.readFile('path/to/your/config.json', function (err, data) {
    console.dir(JSON.parse(data.toString()))
  });
});
```

In the example from the `nconf` website, we require Node's filesystem module and the installed nconf module.

Then, we define the order of checking and loading environment variables. The defined order is

- command line arguments - `argv`
- OS level `env()` settings
- loading defined json file for config settings

We can also set some additional config variables specific to `nconf` execution. In this example, the host and port are customised for the database.

Details are then logged to the console, and the config is saved to a defined JSON file.

Adding continuous development Continuous development allows a developer to work on app code &c. without many of the customary interruptions, such as server reboots, code refreshes, debugging, linting &c.

Continuous development often reduces the repetitive tasks in a development flow, thereby automating processes and development.

A build process may be automated and run whenever a pertinent change is detected.

This may be configured and integrated with a local dev environment or with a remote service.

add a watch task We may add a *watch* task to a build flow to allow a rebuild each time a given file is edited and then saved.

This has become common practice for client-side development, but it may also be applied to Node &c. development flows.

For Grunt, we may add the plugin module `grunt-contrib-watch` . This plugin watches the file system for code changes in a tracked project, and runs the affected tasks. So, as a file change affects a build task, that task will be executed and re-run for the given project.

The build process will now run in an automated manner, as expected for continuous development.

A basic `watch` example might include the following

```
watch: {
    rebuild: {
        tasks: ['build:debug'],
        files: ['public/**/*']
    }
}
```

This continuously checks the `public` directory for a file change or addition, which will then run the `build:debug` task. This type of `watch` provides a broad approach to managing project changes.

However, we commonly update different files, directories, assets &c., which may require different build tasks. So, it is not the most efficient solution to run build tasks for each and every project change and update.

Instead, we may target rebuilds relative to how a developer is working on a given project. We may update `watch` to include specific targets, often one per build task affected by potential file changes.

```
watch: {
    lint_client_side: {
        tasks: ['jshint:client-side'],
        files: ['public/js/**/*.js']
    },
    lint_server_side: {
        tasks: ['jshint:server-side'],
        files: ['src/**/*.js']
    }
}
```

This approach may seem a tad verbose and tedious, but it prevents unnecessary rebuilds and may help improve performance.

watch with live reload We may also use `watch` to add support for *live reloads*. There is built-in support with the `grunt-contrib-watch` plugin.

The reload option uses *web sockets*, a technology originally designed for browser based real-time communication and synchronisation. The *LiveReload* option listens for changes to monitored files, directories &c. It may then reload and refresh the current active app.

Support for the LiveReload task may be added as follows,

```
livereload: {
    options: {
        livereload: true
    },
    files: ['build/**/*', './*.html'],
},
```

This will provide a live reload server, which usually runs at `localhost:35729` . This object includes a property to confirm `livereload` , and then defines the files to watch to initiate a reload. In this example, we're watching the `build` directory, and its children, and then the root directory for any HTML files. This will include, of course, any changes to the default `index.html` file.

However, this server does not actually reload the app for us. We need to use a server to host the app, which is then monitoring this `livereload` server.

To help with this monitoring setup, `livereload` also provides a setup script for the test app. We may either add a link to this script in our project's `index.html` file,

```
<script src="http://localhost:35729/livereload.js"></script>
```

or use a Grunt plugin, `grunt-contrib-connect` to automatically inject it in our app's code. This is the preferred option for ongoing development.

We may install this plugin as follows,

```
npm install grunt-contrib-connect --save-dev
```

and then update the `Gruntfile.js` config as follows,

```
connect: {
    server: {
        options: {
          port: 8080,
          base: '.',
          hostname: '*',
          protocol: 'http',
          livereload: true,
        }
    },
},
```

To use these plugins, we need to update the required build tasks, e.g. we'll add connect and livereload support to the `dev` build task

```
// dev tasks - combine debug with watch, live server, and live reload
grunt.registerTask('dev', ['build:debug', 'connect', 'watch']);
```

We may then run this build task,

```
grunt dev -v
```

The `-v` flag outputs verbose messages to help us initialy check everything is running as expected.

Node.js - nodemon For Node.js based app development, we may add the package `nodemon` to build an app for each code change.

```
npm install -g nodemon
```

In effect, `nodemon` plays the simple role of restarting a `node` based app to show the latest updates.

To use this package, we may issue the following command

```
nodemon app.js
```

So, by default, nodemon will monitor all `.js` files in the current working project directory. However, we may also wish to limit files and directories nodemon is monitoring.

We may define such exclusions in a `.nodemonignore` file, which is similar in nature and practice as a standard `.gitignore` file.

For example,

```
# package control
./node_modules/*

# client-side js
./src/client/*

# testing
./testing/*
```

The specific directory locations need to be customised to a given project, but the underlying pattern will remain the same.

combine watch and nodemon It is possible to combine Grunt `watch` with Node's `nodemon` , although additional modules will be required to overcome the *blocking* nature of these tasks.

In effect, we're providing a work-around for the continuous lifecycles of these apps.

For Grunt integration, we may add `grunt-concurrent` , which will create a new process for each provided task.

References

- Grunt - JavaScript Task Runner
- Node.js
- NPM - nodemon