Extra Notes - Systems - Build First Development

- Dr Nick Hayward

A brief introduction to *build first* development.

Contents

Intro *Build first* development encourages a reusable build process, whilst similarly promoting abstracted, clean app design.

Benefits of this approach will often include the following

- automated processes with a reduction in human interaction - this should reduce errors in the build
- automated repetitive tasks - increases speed, productivity…
- scalable, modular app design
- improved testing and maintenance options by reducing overall complexity
- releases that conform to testing, best practices…
- code deployment following testing, debugging…

We commonly use three stages to an app's development with *build first*. These include

- *Build* process
  - compile and test an app using automated process
  - aimed at continuous development or tuned for specific performance requirements
- *Design*
  - design and development of code, logic, UI &c.
- *Deployment & Environment*
  - automation of release process and configuration of different hosted environments
    - deploy changes to a hosted environment, service…
    - environment config defines environment and services, e.g. database interaction…

So, we can already discern two primary components to *build first* development. For example,

- processes surrounding the project - e.g. building and deploying an app
- design and quality of the app's code, UI &c.

Each of the above relies on the other to achieve the required iterative design and development of an app.

Build processes A build *process* is commonly an automated approach to repetitive tasks for design, development, and management of an app.

For example, we might consider a process for

- dependency installation and management
- compiling of necessary app code
- testing, including standard *unit* tests
- …

Another common requirement is to ensure we are able to perform such processes in as few steps as possible, and commonly in a single step. This is often known simply as a *one-step build*.

A developer &c. should also be able to execute this *one-step build* as many times as necessary, and expect the same pattern of outcome. This is commonly known as *idempotence*. In effect, regardless of the frequency of exection, the result will be the same.

build phase Inherent to the *build* phase is a consideration of either development or release.

For example, we might build an app towards an initial release. This will require an ability to test and debug the app at regular intervals.

Then, we may build the app ready for release itself.

However, for many apps, we simply employ a process known as *continuous development*.

As such, this will not concern the *release* distribution of the app.

So, we may often consider *build* as follows

- *code* for the app
- debug & release distributions
- app built ready for deployment

The *debug* distribution may commonly include the following tasks,

- compilation, testing, watching

Whilst a *release* distribution may include the following tasks,

- compilation & testing
- optimisation & release notes

deployment phase After a suitable *release* distribution has been built, the deployment phase may then prepare and deploy this distribution.

The deployment should maintain fidelity with environment specific config used during the build phase. For example, secrets, keys, database connectors &c.

Tasks and workflows Tasks should follow a clearly defined set of steps, such as lint, optimise, build &c., which produce a specific goal.

i.e. action -> task -> goal

This series of steps is commonly referred to as a *workflow*.

However, a set of tasks may be interchangeable to produce various required workflows. So, a given task may be optional for a given workflow.

e.g. it may not be necessary to optimise images as part of a development workflow, but crucial for release.

This use of workflows creates a separation of concerns across build and deployment flows, for example.

development flow Productivity and monitoring changes are the key concerns for a development workflow.

This flow is commonly expected to produce a built app, although it may be focused on continuous development.

release flow In contrast to development, productivity and monitoring is not a key focus for release. It may, in fact, become a barrier to a quick, effective build for release if persisted for this workflow.

Therefore, this workflow is primarily concerned with optimising performance, and ensuring a well-tested app.

So, we may modify the development workflow to focus on optimisation, thereby reducing the byte size of the app, its code, and any required assets.

deployment flow The deployment flow does not build the app. It should reuse the build distribution, which is customarily produced in the *release* workflow. For example, the built app output from the expected *release* workflow.

This workflow will then deploy this build to a hosting environment.

So, deploying the app should be separate from the build and release workflows.

The inherent benefit to this separation of workflows is the simple ability to build and deploy an app quickly and efficiently.

App complexity and design Complexity in app design and structure may be considered relative to modular patterns, dependency management, async flow control, design patterns, and so on.

For many apps we develop, we may consider the following

- modularity and components
- design - e.g. MVC, async operations…
- testing options and practices

For the *build* process, defining a modular app architecture is important for maintenance of the underlying code. We may then augment this architecture by defining automated processes with continuous development, integration, and deployment.

modularity A modular approach to app design helps developers define components for required logic, functionality &c. Such components may then be structured as modules for re-use.

Each module should include concise functions, providing a single purpose to the module.

Modules may include external or local sources, including many third party options using package managers. Such package managers help abstact version control and dependencies for a selected module.

A simple benefit of modular design is reference to dependencies in the code, instead of the global namespace. This helps improve self-containment of the module.

design Design of an app is also helped by a clear identification of components and usage. Such *orthogonal* design helps with modular structures, and separation of concerns. It also has a complementary benefit for testing and further development.

For example, we might define a multi-tiered approach to an app's design. This may isolate the UI interface from the data and the business logic, each layer providing focused, specific support to the app's design.

testing Testing is commonly defined at multiple stages of the app's design and development.

For example, we may define continuous integration with regular tests for each push to version control and subsequent deployments. This may also be combined with continuous deplyments, perhaps as an end result of successful tests and builds.

We may also consider various options for fault tolerance, including standard logging, monitoring, and, perhaps, clustering.

For each stage of an app's development, it is common to consider appropriate tests, then adjust the build process accordingly, modify the code, and then repeat as necessary.

In effect, we define a *build first* approach to an app's design and development.

Sample tools As we design and build our apps, we're obviously concerned with code usage, abstraction, and the various phases of development noted above.

However, a key part of code quality and structure may include use of tools such as task runners, build tools, asset management, linters, and so on.

We may initially consider Linters, such as JSHint, task runners, and applicable bundlers.

Each of these tools provides support for various phases of the app's development, build, deployment, and ongoing maintenance.

linters A linter may be useful for determining syntax errors in an app's code. For example, it may return code issues such as

- undeclared variables
- missing semi-colons
- strict syntax errors
- …

It may also be useful for enforcing coding best practices, including common structural patterns and usage. Linting will also work with different code groups, from short snippets to a list of files.

For JavaScript, a Linter often completes the role of a traditional compiler, checking the code before build. In effect, it defines whether the JS code may be interpreted successfully by a JS engine.

Some Linters may also be configured to highlight overly complex code and structures. Perhaps, verbose functions, issues with code blocks such as conditional statements, scoping problems or mis-use, &c.

Many developers consider *linting* as the first desirable test for JS development. It becomes a worthwhile pre-cursor to unit testing and release.

basic lint usage For JavaScript based app development, a common option for linting is *JSHint*. This tool is written using Node.js, and is used for linting both files and code snippets. We may also combine its usage with build tools such as Grunt as part of a given build process.

To use the command line tool for JSHint, we need to install Node.js. For this installation, we may either use a direct install from the Node.js website, or a package manager such as Brew, NuGet, NPM &c.

Once Node.js is installed, we may then add JSHint's command line tool using NPM,

```
npm install -g jshint
```

We may then check the installed version using the standard command,

```
jshint -v
```

After successfully installing JSHint, we may then `cd` to the root of a project directory with JS files and code to check. To check all files at the CWD, we may issue the following initial command,

```
jshint .
```

Of course, we may also exclude certain directories and files. For example, for a Node.js based project it is common practice to exclude the `node_modules` directory for development linting of a local project

```
jshint . --exclude node_modules
```

If we execute this command, we might get the following type of return for files with errors

```
lib/spire/maths/circle.js: line 25, col 2, Missing semicolon.

lib/spire/maths/square.js: line 16, col 2, Missing semicolon.

lib/spire/spire.js: line 45, col 2, Missing semicolon.

3 errors
```

As we correct such errors, and then re-run the JSHint command, we will get no return output unless there are errors or syntactic issues &c. in the code. We're aiming for an empty return with no errors.