Development

A guide to modern software development. This is version 0.5. The cover photo is a view of the Queen Elizabeth II bridge that forms part of the Dartford River Crossing. It was taken from Erith Pier.

Christopher Eyre

Abstract

A personal set of practices for software development.

Contents

1 Development

`A beginning is a very delicate time - Frank Herbert, Dune.`

1.1 Introduction

This is a book about software development practices. My only claim is that these techniques have worked for me over a range of projects. Hopefully these will be useful to other people.

It will be technology agnostic, but may use specific platforms and tools. This book highlights the (sometimes extreme) positions that I take on things (my views on upgrading have been described as militant). Software Development is very personal.

My degree was in Mathematics where I focused on Mathematical Physics and Game Theory.

I have been working in software development for 25 years. In that time I have worked in:

- Defence
- Banking
- Futures Trading
- Insurance
- Sports Betting
- Digital Publishing.

Here is my work history:

- Defence Research Agency/Centre for Defence Analysis
- Admiral Computing

- Morgan Stanley Dean Whitter
- Apollo Magazines Ltd
- City Networks
- Munich Re/Watkins Syndicate
- MF Global
- Bet Genius
- Pottermore
- Codurance

Software development skills are highly portable. There are domain specific knowledge that is needed but the main processes are the same.

Currently I work for Codurance, a software consultancy that was formed from the Software Craftsmanship community.

They are almost always recruiting https://codurance.com/careers/#our_roles (mention Christopher Eyre)

Software Craftsmanship attempts to restore the Extreme Programming practices to Agile. Craftsmanship involves caring about what you are doing and trying to do better.

Here is the Software Craftsmanship manifesto:

```
* Not only working software, but also well-crafted software

* Not only responding to change, but also steadily adding value

* Not only individuals and interactions, but also a community of professionals

* Not only customer collaboration, but also productive partnerships
```

My blog https://devrants.blog has been updated (almost monthly) for over a decade and a half. I use it as an external memory.

Currently I am a team lead — the one that the client invites to meetings and acts the main day to day contact point for the clients management and from other teams. The team lead act as both developer and business analyst.

There are various styles of team lead. My preference is to be very hands on with the development and to lead by example — I don't expect the team to work on anything that I would not do myself (subject to having the correct skill set).

Over the last 6 years I have been working almost exclusively with cloud hosted software using Azure, Heroku, Google Cloud and AWS.

Most experienced developers frequently spend time trying to recreate the winning formulas from successful past projects. This is my attempt to collect and unify the best practices that I have encountered across my career.

The target audience are professional software developers or recently promoted team leads. You need to know what Extreme Programming is, follow some agile process (Scrum/Kanban) and practice Test Driven Development.

1.2 Thanks

There are several people that I need to thank:

- My wife Karen for putting up with me writing.
- Steve Lydford for giving me the idea to write a book.
- Sandro Mancuso and Mashooq Badar for forming the friendly community of professionals that is Codurance.
- All the teams (and managers) that I have worked with over the years.

1.3 The New Team Lead

At the start of 2019 I took over as team lead of a delivery team. A Delivery Team is responsible for the support and development of a software product.

Discussions with my new team triggered some notes that have expanded into this book. There were things that I had taken for granted that my team members considered novel.

1.4 You Can't Do Everything At Once

`The engines cannae take it Captain - Scotty`

It is important to note that you can't do everything at once.

A team lead has a lot to look after. Remember that you do have a team to delegate to.

Here is a short list of things that a team lead could focus on:

- No Broken Builds.
- Ensure all components are updated.
- Minimise Production Outages.
- Clean Logs.
- Monitor everything.
- No dead code.
- Automated everything.
- Keep the team happy.
- Does the team have the right skills?
- Is the management happy?
- Is the management well-informed?
- Usage monitors.
- Communication with other teams.
- Logging.
- [Monitor Your Bills]
- Engagement with customers.
- Experiments in progress.
- Retiring services.
- Performance Reviews.

This is not even a complete list! Pick the most important thing first and then move to the next once it is under control. It took three months to get to clean builds. Focus on what is important (and that will change from day to day)!

This book will cover the above topics.

1.5 Shattered Focus

Developer documentation frequently talks about flow and focus. This is why you should not interrupt developers too much.

A team lead will typically have to handle so many distractions that flow is no longer possible. It is a useful skill to be able to context switch across a wide range of distinct streams of work. Context switching and still being able to see the big picture are essential.

Currently my team has 10 threads of current and upcoming work. I am the point of contact for the distractions keeping my team free to focus. They will be updated sufficiently frequently to do their jobs but allow them to maintain flow.

1.6 Know Your Numbers

It is important to be familiar with key usage metrics of your system. You need to know how many users you have at a given time. These are important enough to have visible on a dashboard somewhere.

Without knowing this you won't know that a logged error listing 100 users affected is either a significant system outage or just background noise. These figures are best displayed over a larger time window (think days or weeks). Trying to do this over hours will mean that you lose all sense of proportion. Having multiple days in view provides something to compare against.

1.7 Know Your Clients Business Model

The system that you are working on will be valuable to someone. You need to know what the business model is. This will not always be obvious or indeed consistent across an organisation. Understanding this makes working with the client much easier. Decisions that a client makes will always align with this business model.

McDonald's business model is about renting property and providing supplies to franchises. Selling drinks and burgers is secondary.

If its an end user pay to use system, then the team needs to react differently to an annual subscription system. Pay to use needs continuous new features or content. An annual subscription needs to be stable and would be happy with a new feature each quarter. The business model of a high value subscription services is about allowing salesmen to have a conversation with the right person.

1.8 Know Your Team

It is important to know the various members of your team. You need to know their likes and dislikes, the skills they have and those that they would like to gain. Everyone has detailed specialist skills that can help in the right circumstances. Pair with them all as frequently as possible. If this is not an option, then arrange one-to-one sessions with them. People are your most valuable asset.

If you know your team you should be able to identify gaps in knowledge or over concentration in a limited subset. Ensure that your team are provided training for the missing skills. This is even more important if the team inherits a system that they have not built. This either requires time to investigate, additional training or an understanding that support could become difficult.

2 Development Practices

Most of the development practices that I use with my team are straight from `The Pragmatic Programmer`.

This chapter covers a range of techniques to assist the smooth running of a project.

2.1 Principles for Effective Delivery

`If it's a good idea, go ahead and do it. It is much easier to apologize than it is to get permission - Grace Hopper`

My team support and update a complex system. Our stakeholders would like us to add new features on a regular basis.

We are also responsible for keeping the system operational. Keeping the system alive is more important than new features.

In case of critical updates we can split into multiple streams. Keep one stream adding the new features and the rest fixing the problem.

We do not ask to fix broken builds — that is a priority. We don't ask to apply updates or upgrade infrastructure that is being decommissioned.

In a cloud environment you may need to move providers every few years. Prepare for this. With enough providers this can result in multiple shifts per year. How to migrate away should be a consideration of choosing a provider.

The team needs to be aware of upcoming work (but not all at the same level of detail). Try not to have the entire team attend every meeting. There will be meetings that the whole team should attend, but those are likely to be project kick-offs.

2.2 Be The Change You Want To See

```
If we could change ourselves, the tendencies in the world would also change.
As a man changes his own nature, so does the attitude of the world change towards
... We need not wait to see what others do. - Gandhi
```

If something is wrong don't ignore it or complain. Start looking for a way to improve things.

This can be:

- Missing Tests
- Broken Builds
- Missing Documentation
- Poorly factored code
- Inefficient Processes
- Missing or Poor Logging
- No useful dashboards

Start the improvement and ask for help to continue it. If large add specific work to the backlog to fix it.

I encountered this when trying to test some code in a Ruby static site generator. Ruby allows you to extend any class. The code that I was using had expanded another class and added methods that only it used. This made testing this class difficult as you needed to stand up the large and complex class just to test one small aspect. I started to complain about the poor design choices — before I realised that I had the ability to fix it.

A refactoring to reference the large class rather than extend it led to some far better test coverage. The difficult class could be mocked. The testing of which found a bug that had bloated the size of most blog posts on the site.

We now have 100% code coverage of this codebase. This is only meaningful if you are making assertions. Code coverage tools can be useful to find untested parts of the codebase. Chasing 100% coverage can be a time-wasting distraction or a means to more reliable code.

If you are missing some documentation, then start by writing it. It can be a short placeholder. It need not be detailed or perfect. Having something to improve is a big advantage over nothing.

Spotting an inefficient process is one thing. Suggesting an improvement is much better.

2.3 No Broken Builds

`Don't leave broken windows - Pragmatic Programmer`

When I arrived at my current project (as team lead) the first thing that I noticed was the quantity of [broken builds]{index}. The team uses Jenkins, and we had a build monitor on the wall displaying all the builds with the failed ones first in red followed by the successful ones in green. Each branch of each repository got a square on the board (a majority of the projects being old style pipelines had 5 or so stages displayed) The build board was displaying around 30 broken builds.

The team was clearly unable to see where to start. To be fair to them there had been an almost complete change of staff, the longest serving having been with the project for only three months, excluding the team lead that I was replacing. The team had also recently inherited a suite of neglected (yet production) code consisting of twelve repositories (the environment uses micro-services and uses a repository as the unit of deployment).

A majority of these failures were caused by test instability — these were true flickering tests (pass on one run, fail on another). Initially it seemed that these were caused by missing or out-of-date node packages. Fixing the missing package triggered another build which due to chance worked. These tests then failed again on another day. I started keeping a log of the broken builds found each morning. The build process triggers and automatic overnight build of everything, which as we are using a Continuous Delivery pipeline will result in everything being redeployed daily if the tests all pass. Some investigation lead to the discovery that the integration tests had failed to wait for the docker images for key infrastructure (RabbitMQ/MongoDB) to start before running the tests.

Now that the builds are almost all green the team now jump on a broken build as soon as it is noticed.

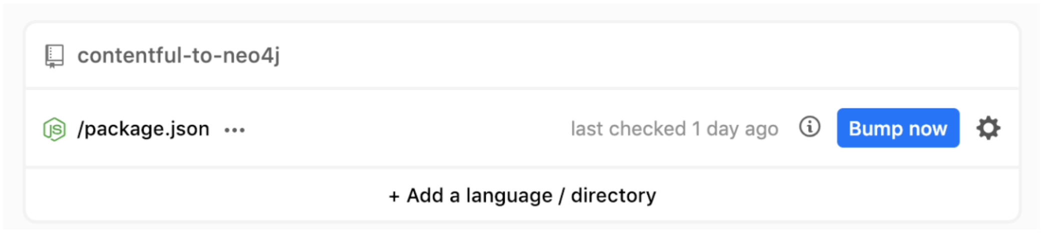
2.4 Taming Dependabot

When I started out developing in Delphi the number of dependencies that you took on was significantly lower and slower moving than modern JavaScript development. Back then a project would typically have one or two custom libraries added to the Visual Component Library (VCL) and everything else was custom-built. These libraries may have got an update every year but could typically be left alone for two or three years. It was fairly straightforward to keep these updated, provided that the supplier stayed in business.

Contrast this to a Node app built with the react-starter-kit. That has over 2000 node modules as dependencies. These are continually being updated and new versions released. Keeping on top of the security fixes and general updates is a major undertaking.

This is where Dependabot comes in. You grant Dependabot access to your git repository and allow it to raise pull requests. It will even merge them for you automatically if you have a suitable build server attached and the build passes.

For open source projects Dependabot is free. I have set this up on my https://github.com/chriseyre2000/contentful-to-neo4j project.


contentful-to-neo4j

/package.json ··· last checked 1 day ago

Figure 1: Dependabot control panel

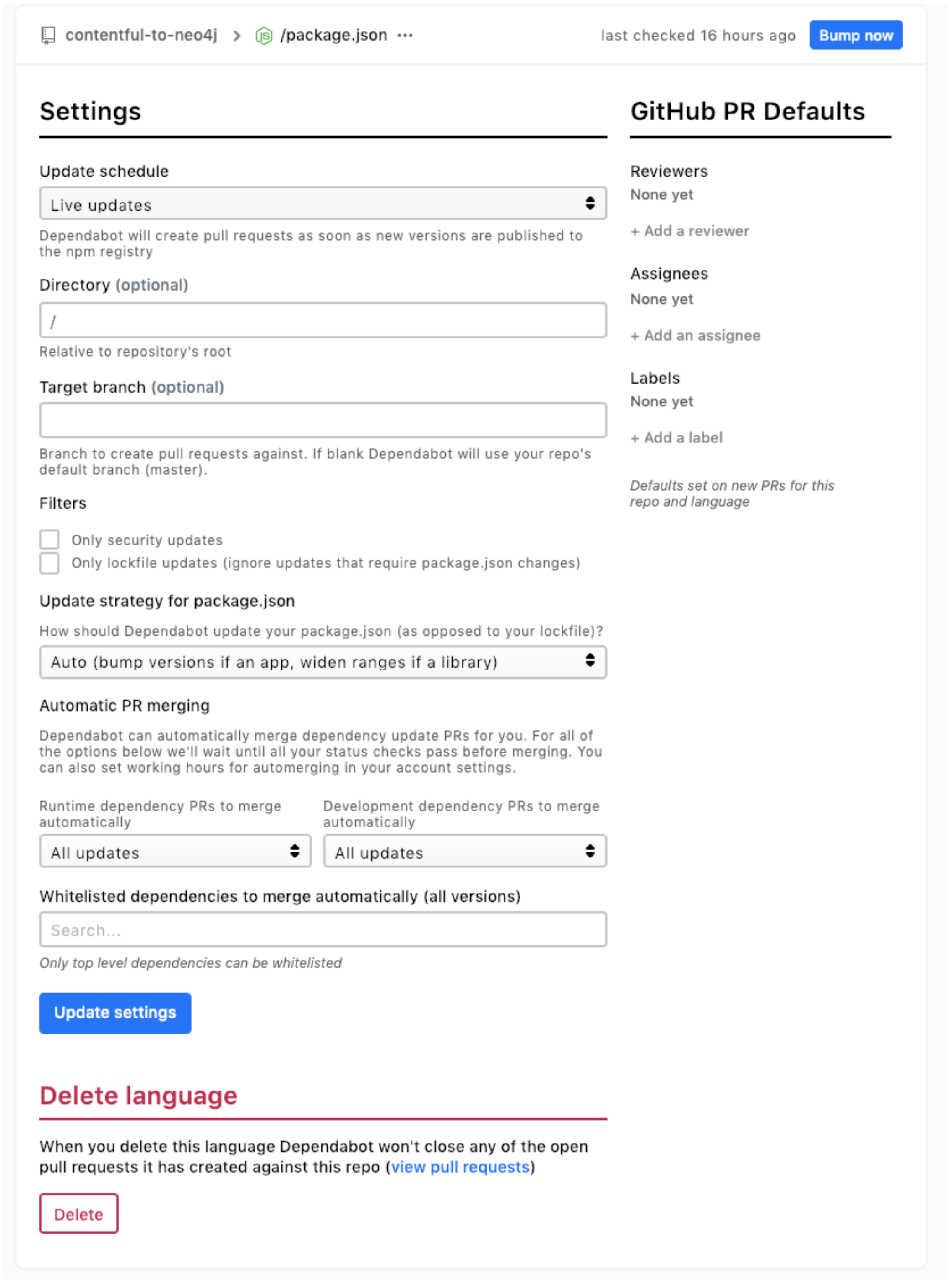
If you click on the gear icon to the side you get the detailed page:

Here I have chosen to take all updates as soon as they are known about and to accept automatic merges once the build server passes.

For my build server I use Circle CI. Which is also free for open source projects:

To get this added to Circle I needed to add a .circleci/config.yml file to my project:

```
# Javascript Node CircleCI 2.0 configuration file
#
# Check https://circleci.com/docs/2.0/language-javascript/ for more details
#
version: 2
jobs:
build:
docker:
# specify the version you desire here
- image: circleci/node:8.14

# Specify service dependencies here if necessary
# CircleCI maintains a library of pre-built images
# documented at https://circleci.com/docs/2.0/circleci-images/
# - image: circleci/mongo:3.4.4

working_directory: ~/repo

steps:
- checkout

# Download and cache dependencies
- restore_cache:
keys:
- v1-dependencies-{{ checksum "package.json" }}
# fallback to using the latest cache if no exact match is found
- v1-dependencies-

- run: yarn install
```


contentful-to-neo4j > /package.json ··· last checked 16 hours ago Bump now

Settings

Update schedule

Live updates

Dependabot will create pull requests as soon as new versions are published to the npm registry

Directory (optional)

/

Relative to repository's root

Target branch (optional)

Branch to create pull requests against. If blank Dependabot will use your repo's default branch (master).

Filters

- [ ] Only security updates
- [ ] Only lockfile updates (ignore updates that require package.json changes)

Update strategy for package.json

How should Dependabot update your package.json (as opposed to your lockfile)?

Auto (bump versions if an app, widen ranges if a library)

Automatic PR merging

Dependabot can automatically merge dependency update PRs for you. For all of the options below we'll wait until all your status checks pass before merging. You can also set working hours for automerging in your account settings.

Runtime dependency PRs to merge automatically: All updates

Development dependency PRs to merge automatically: All updates

Whitelisted dependencies to merge automatically (all versions)

Search...

Only top level dependencies can be whitelisted

Update settings

Delete language

When you delete this language Dependabot won't close any of the open pull requests it has created against this repo (view pull requests)

Delete

GitHub PR Defaults

Reviewers

None yet

+ Add a reviewer

Assignees

None yet

+ Add an assignee

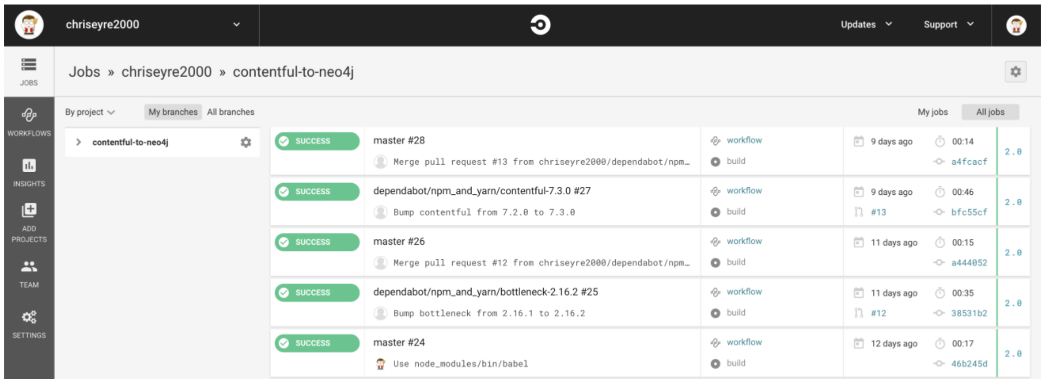
Labels

None yet

+ Add a label

Defaults set on new PRs for this repo and language

Figure 2: Dependabot details



Jobs » chriseyre2000 » contentful-to-neo4j

```
SUCCESS  master #28                                         workflow  9 days ago   00:14    2.0
         Merge pull request #13 from chriseyre2000/dependabot/npm…  build                 a4fcacf
SUCCESS  dependabot/npm_and_yarn/contentful-7.3.0 #27       workflow  9 days ago   00:46    2.0
         Bump contentful from 7.2.0 to 7.3.0                build     #13          bfc55cf
SUCCESS  master #26                                         workflow  11 days ago  00:15    2.0
         Merge pull request #12 from chriseyre2000/dependabot/npm…  build                 a444052
SUCCESS  dependabot/npm_and_yarn/bottleneck-2.16.2 #25      workflow  11 days ago  00:35    2.0
         Bump bottleneck from 2.16.1 to 2.16.2              build     #12          38531b2
SUCCESS  master #24                                         workflow  12 days ago  00:17    2.0
         Use node_modules/bin/babel                         build                  46b245d
```

Figure 3: Circle CI dashboard

```
- save_cache:
paths:
- node_modules
key: v1-dependencies-{{ checksum "package.json" }}

# run tests!
- run: yarn test
```

This is the minimal build that you need to permit Dependabot to upgrade your project automatically. This works well for an open source project.

You see this in GitHub:

This is what happens when everything just works.

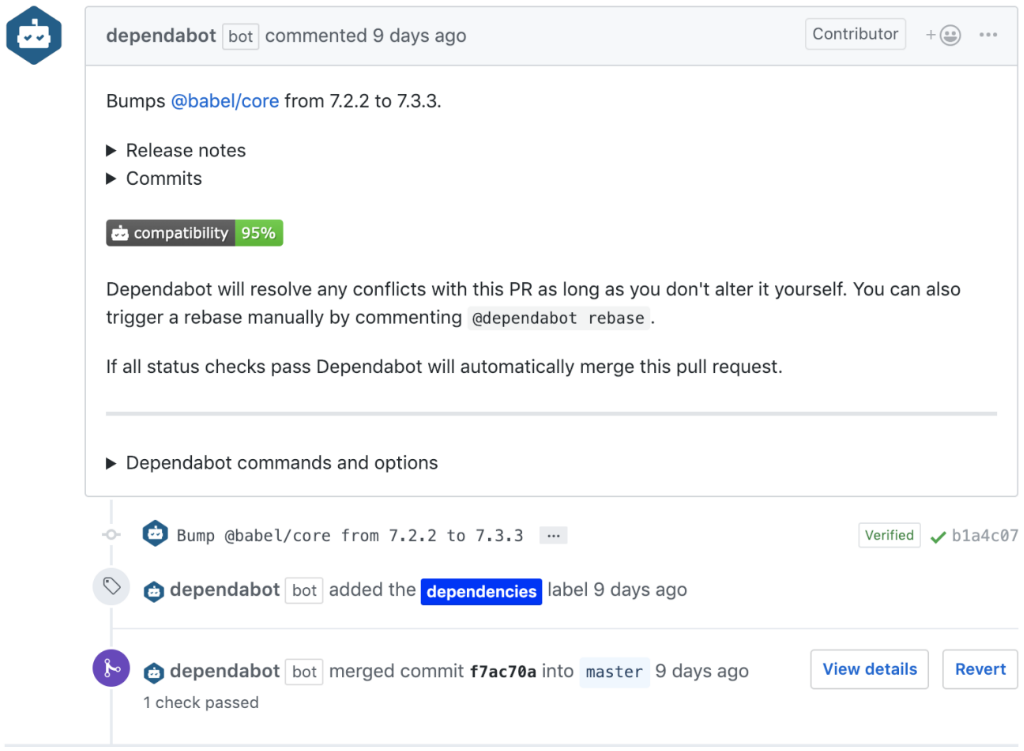
If you need to intervene you can fix the code on the branch and wait for the build.

Dependabot commands are sent by making comments on the PR mentioning `@depndabot`.

`@dependatbot merge` asks dependabot to merge and delete the branch.

`@dependabot rebase` asks dependabot to rebase the change – very useful if another dependabot change has updated the lock file.

`@dependabot recreate` asks dependabot to recreate the PR. This is similar to rebase but will retrigger a build even if there are no changes. Does anyone else have builds with network dependencies (jenkins, browserstack &c)?


dependabot bot commented 9 days ago

Contributor

Bumps @babel/core from 7.2.2 to 7.3.3.

▶ Release notes
▶ Commits

compatibility 95%

Dependabot will resolve any conflicts with this PR as long as you don't alter it yourself. You can also trigger a rebase manually by commenting `@dependabot rebase`.

If all status checks pass Dependabot will automatically merge this pull request.

▶ Dependabot commands and options

`Bump @babel/core from 7.2.2 to 7.3.3` ... Verified ✓ `b1a4c07`

dependabot bot added the **dependencies** label 9 days ago

dependabot bot merged commit **`f7ac70a`** into `master` 9 days ago

1 check passed

Figure 4: GitHub dashboard

Dependabot is also good at working with Jenkins — provided you are using Jenkinsfiles. I am currently working on a large project that has over 40 repositories and we don't have time to move all of our projects to use Jenkinsfiles.

The solution to this is to add a small Jenkinsfile to the project that just runs the unit tests (or as many tests as you can fit). There is a risk that your tests will diverge, but having some tests that run as part of the dependabot permits automatic merging.

The process to making dependabot work for you is:

- Ensure that your project has a package.json file (or equivalent for other supported build tool)
- Enable dependabot for that project on it's website (https://dependabot.com/).
- Enable updates and auto merge as you wish
- Configurate a CI server to build the code
- Archive github repos you are not using (remember to close any open PR's before archiving)
- Remove dependencies that you don't need
- Fix broken PR's

I would advise adding a few projects at a time to dependabot. Turning them all on at once left us with a backlog of 360 Pull Requests!

One of the major benefits of taking all updates as soon as they happen is that the team remains familiar with the projects in the maintained suite (especially useful if the team did not write them). Things will break but it's likely that the problem will be mentioned in the current release notes. This assumes that you have managed to keep everything updated. Crossing several major versions in an upgrade can be painful. Not upgrading can result in failed builds due to the tools being used falling out of support (currently we are on Node 10 Long Term Support, and a Node 4 docker image recently failed to build. Docker files will rot in time due to required dependencies being archived).

2.5 Merge Small Commits

My preference is for trunk based development. This is the only route to continuous delivery. It does require a trustworthy team, feature flags and a healthy suite of tests.

Code should always be developed in small steps backed by tests. If you find a case that can't be tested then I would suggest that you consult the index pages of `Working Effectively with Legacy Code` and read the appropriate chapter. Don't be afraid to write more test code than implementation.

2.6 Refactor Ruthlessly

Badly named code will only slow you down next time you are in a project. If you can make an api assumption explicit then it will help in the future. For example a recent elasticsearch upgrade uncovered an index loader that required pairs of calls. Adjusting the client code to make this explicit will save time the next time the code is read.

Consistent naming across microservices will help when you need to apply that crosscutting patch. Having the same service with a different name in each microservice is a huge productivity drain. Cleaning the code up as you go will help the next person to look at the code and it may be future you.

2.7 No Dead Code

Removing dead code from a system is a great way to improve productivity. This can include removing an unused project. The most extreme case I have seen was when we managed to eliminate two-thirds of a codebase. This was a system that had been built by cut-and-paste from a previous system.

Dead code can mask the true structure of an application. Some modern languages will provide warnings if a code path is unused.

2.8 Small Fast Feedback

We learn faster when the result is closer to the cause. Always aim to reduce the feedback loop. This means that all tests must be fast (seconds). Deployments need to be fast (minutes). If you break something roll it back as soon as you have the reason why.

2.9 Read The Error Message

Always be sure to read the error message. It's amazing how much time can be saved by this small piece of advice. If it's unclear correct it. Error messages need to be compact yet useful.

Detailed stack traces are useful in places (especially if that place is deep inside a docker container).

If the error messages are unclear, then improve them.

2.10 Leave A Broken Test

If you are going to be leaving some code for a while (such as a emergency or overnight) leave a test broken on the branch that you are parking the code on. This will speed up the recovery of your thought processes. This has an amazing ability to restore mental maps even days after you last worked on the problem.

2.11 All Bugs Are Missing Tests

When a bug happens in production, then there is a missing test. It may not be at the unit test level but it is missing.

Add a failing test before you try to fix anything. The test insures that you are fixing one version of the problem. There are times when there are lots of ways of generating the same symptom. Ensure that you include the reported one in the fix where possible.

2.12 Improve Something Every Day

Make a small improving change to the system each working day.

- Add to the documentation.
- Fix the most frequently found error message in the logs.
- Rename something to be clearer.
- Resolve a PR on the backlog.

Over time these will amount to big changes.

2.13 Preview

Thanks for reading the preview version of this book.

The full version is available from https://leanpub.com/development2019