

Modern Development Handbook

*Using agile methods to build
quality web applications*

Table of Contents

Modern Development Handbook	1.1
Table of contents	2.1
Introduction	2.2
An ideal process	2.2.1
Agile development	2.2.2

Part I: Technical

Git workflow	3.1
Main branches	3.1.1
Feature branches	3.1.2
Pull requests	3.1.3
Reviewing pull requests	3.1.4
Building a release	3.1.5
Emergency fixes	3.1.6
Why we use branches	3.1.7
Summary	3.1.8
Automation	3.2
Continuous integration	3.2.1

Automated testing	3.2.2
Deployment automation	3.2.3
The staging system	3.2.4
Coverage reporting	3.2.5
Summary	3.2.6
Coding practices	3.3
Test-driven development	3.3.1
Types of tests	3.3.2
Why we test	3.3.3
Linting	3.3.4
Readme files	3.3.5
Inline documentation	3.3.6
Summary	3.3.7

Part II: Management

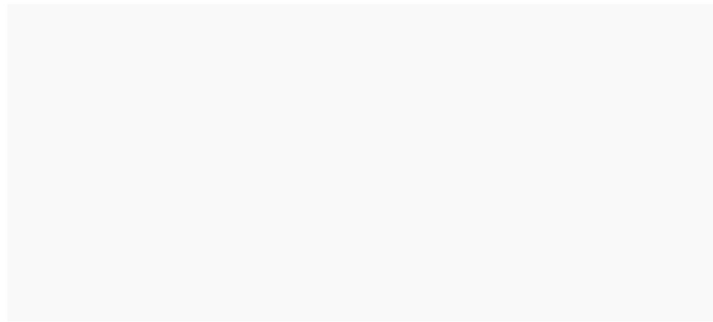
Project lifecycle	4.1
Phases of a project	4.1.1
The "launch" myth	4.1.2
Sprints and milestones	4.1.3
Versioning	4.1.4
Release notes	4.1.5

Summary	4.1.6
Collaboration tools	4.2
Project management tools	4.2.1
Writing stories	4.2.2
Submitting bug reports	4.2.3
Communication	4.3
Communicating effectively	4.3.1
Using Slack	4.3.2
Sprint kickoffs	4.3.3
Sprint retrospectives	4.3.4
Summary	4.3.5
Team management	4.4
Team composition	4.4.1
Subteams	4.4.2
The scrum-master	4.4.3
Summary	4.4.4
Miscellaneous	4.5
Documentation	4.5.1
Onboarding	4.5.2
Project management tools	4.5.3

RICO STA. CRUZ

Modern Development Handbook

*Using agile methods to build
quality web applications*



DRAFT · 2017

Modern Development Handbook

© 2017+, Rico Sta. Cruz. This work is licensed under [CC-BY-NC-SA-4.0](#).

Authored and maintained by Rico Sta. Cruz with help from contributors ([list](#)).

Table of contents

- **Introduction**

What is "Agile" software development, and how it can help you build software in a sustainable, frictionless, and quality-driven way.

Part I: Technical

- **Git workflow**

How to use Git to your advantage with branch-driven development.

- **Automation**

Using Continuous Integration tools to automate your testing and deployments.

- **Coding practices**

How to code without losing your soul.

Part II: Management

- **Project lifecycle**

Be wise and be smart.

- **Communication**

Collaborate with your team and stakeholders effectively.

- **Team management**
How to manage your team.

Introduction

This book is about processes and workflows in building web and mobile applications. You'll learn:

- Setting up a team-oriented web development workflow using Git, CI, and other tools
- Managing a project timeline using milestones and sprints
- How to manage a team of developers

What's not in this book:

- Technical details of how to use Git or how to code JavaScript

Playbook

This handbook is a culmination of lessons we've learned in working as a team in building modern web applications. It documents our practices in [Proudcloud](#) and [Mashupgarage](#), web development consultancy companies that I'm involved with. Our practices evolved organically from the [Agile software development](#) ideas we're rooted in.

Standard practices

There are no new ideas on this book. Many the points described here are standard practices in teams across the world. All this book does is compile these practices into one book.

An ideal process

The processes you'll learn in this book are all made to achieve a common goal. That goal is to make web development:

- **Sustainable** - A development team needs a process that it can sustain for months and years.
- **Frictionless** - Any barriers to getting work done should be minimized, if not removed.
- **Quality-oriented** - Our process tries to maximize the available talents in each team.
- **Resilient** - Changes are inevitable and an ideal process should account for that.

These ideas are not new or unique to our team. In fact, there are established principles that share these goals. One of such is [Agile development](#), which everything in this book is rooted on.

Agile development

The *Manifesto of Agile Software Development* (agilemanifesto.org) is a short list of principles written in 2001 by 17 developers to help others with their combined experience. It has 4 core values:

- **Individuals and interactions** *over processes and tools*
Self-organization and motivation are important, as are interactions like co-location and pair programming.
- **Working software** *over comprehensive documentation*
Working software is more useful and welcome than just presenting documents to clients in meetings.
- **Customer collaboration** *over contract negotiation*
Requirements cannot be fully collected at the beginning of the software development cycle, therefore continuous customer or stakeholder involvement is very important.
- **Responding to change** *over following a plan*
Agile methods are focused on quick responses to change and continuous development.

Git workflow

For team collaboration to be sustainable, we need to follow a few rules. Our workflow is based on these ideas:

- Everything you do has to be reviewed by someone else.
- It should be easy to review other people's work.
- There should always be a working version (`develop`).
- There should always be a release-ready version (`master`).

Main branches

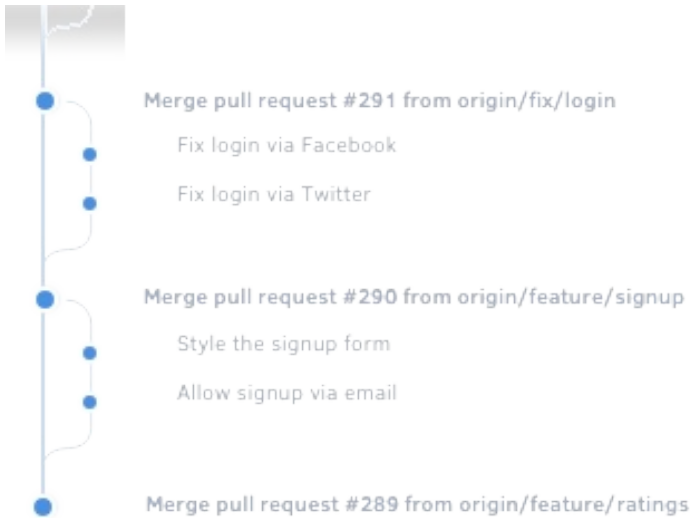
There are two main branches to be maintained: `master` and `develop`.

The develop branch

The `develop` branch is what we consider the main branch. Feature branches merge into this branch.



Nobody should push commits directly into this branch. Instead, work starts by creating feature branches from `develop`. They'll be merged back in when finished. In practice, the `develop` branch only has merge commits from feature branches.



The `develop` branch is always in a "ready to test" state. All automated tests should always be passing in this branch. You should also set up a staging server to sync with the latest `develop` version for testing (see [automated deployments](#)).

The master branch

The `master` is maintained to be in a *production-ready* state. While the `develop` branch is ready for developers and testers, the `master` branch is ready for users.



At the end of a sprint, the `develop` branch will be merged into `master`. This effectively promotes the current development version into a production version. From then, automated deployments will take care of deploying it to production. This means whatever's live on production at any moment is always what's the latest on `master`.

► See also...

Feature branches

To start working on anything, you first create a *feature branch*. These branches can either be a `feature`, a `fix`, or a `chore`.

```
# Start from the main branch, `develop`
git checkout develop

# Create your branch locally
git branch feature/improve_rating_system

# Work, work...
git commit -m "Show stars for user ratings"
git commit -m "Allow users to click on stars"

# Push your branch as often as possible
git push --set-upstream origin feature/improve_rating_system
```

All work start as a feature branch. Nobody should directly push commits to the main branches (*master* and *develop*). Working this way brings us a lot of benefits:

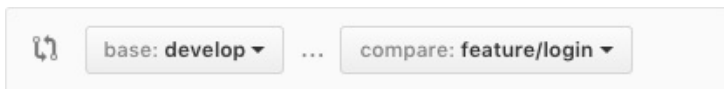
- Your work can easily be reviewed later through [pull requests](#).
- It can be reverted later on as one branch, if need be.

Pull requests

GitHub's Pull Request feature is used to ask your team to merge a branch in. This is perfect for our branch-based approach of working in branches and asking your teammates to review them.

Creating pull requests

Create a pull request from your feature branch (eg, `feature/login`) into the main branch (ie, `develop`). This is usually created before, or just around the time, a feature branch is finished.



Writing a pull request

Write a short title for your pull request. A good format to follow is:

`<Feature>: <changes>`

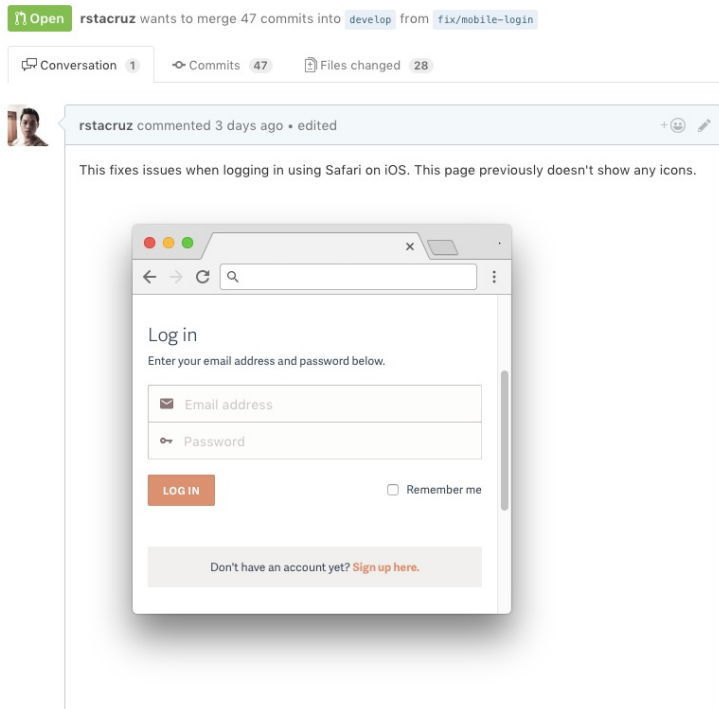
Write a short paragraph summarizing your changes. Include these additional details as well:

- Screenshots. This helps reviewers understand the context of your changes.

- Note if there's anything missing.
- Links to relevant issues (in Trello, Pivotal Tracker, and alike).

Here's an example of a simple pull request:

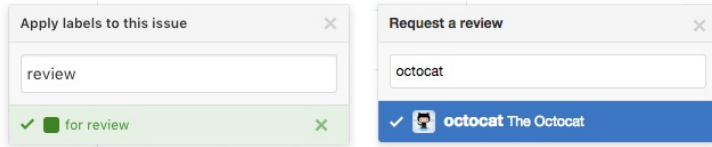
Login: fix mobile issues #313



A pull request example with a screenshot.

Requesting a review

When a feature branch is finished, tag it as *For review* and request for review. It's then your reviewer's job to understand your changes. This way, knowledge is always shared between at least 2 people; there'll be no part of your project that only one person knows about.



► See also...

Reviewing pull requests

A pull request reviewer has these responsibilities:

- Understand the original author's changes so that the knowledge is shared across the team.
- Find any issues the changes may have.

A reviewer should do one of these things:

- Approve it (by merging the pull request); or
- Reject it (by request for changes); or
- Ask others to review it as well (by saying "LGTM").

Who can review

Anyone on the team can review anyone else's code. Senior members typically do most the reviews in practice, but any member of the team can review pull requests as long as they can satisfy the responsibilities listed above.

The only person who can't review the code is the original author.

Understanding the pull request

As a pull request reviewer, you will be partly responsible for the pull request's code. If anyone has concerns about it in the future, it's going to be the author and the reviewer who should be able to answer these concerns.

A reviewer should also test the pull request, if possible.

Approving

Simple. Merge the pull request to approve it!

If you'd like to offer your review but don't want to merge it yourself, you can leave a comment saying "LGTM." This can either mean "looks good to me" or "let's get this merged." This is useful if you'd rather wait for the input of another colleague.

Rejecting

A reviewer may request for changes as a way of rejecting a pull request.

- Use GitHub's "request changes" facility to request for changes.
- Remove the `For review` tag in the pull request.

When the author's done the changes, it should be tagged back as `For review`.

► See also...

Building a release

A release is simply a merge commit in `master`. To build a release, you'll need to do these steps:

1. Merge `develop` into `master`.
2. Merge `master` back into `develop` (if needed).
3. Create a release.

Merging into a clean master

For clean merges of `develop` into `master`, you can use the GitHub interface. Simply create a pull request from `develop` into `master` and merge it using the GitHub website. This works when `master` is clean and up-to-date with the latest `develop` branch, and has no merge conflicts.

Merging into a dirty master

If `master` has changes that are out-of-sync with `develop`, you may run into merge conflicts. The only way to resolve this is by using the command line. This is the only time pushing directly into `master` is acceptable.


```
# Switch to the master branch
git checkout master

# Merge develop into master; do a fetch first
# to ensure we have the latest origin/develop
get fetch
git merge origin/develop

# Resolve any conflicts and commit the result
git mergetool
git commit

# Push into master
git push
```

Synchronizing develop

There are commits in `master` that are not in `develop`, it may be a good time to merge those into `develop`. Make a pull request of `master` back into `develop`. Doing this is not necessary, but it will clean up your git history.


Creating a release

Every deployment should have a version number tag. Simply create a release in GitHub. This will create a git tag for you (equivalent to `git tag v1.0.4 master && git push --tags`).

- Create a GitHub release for the latest `master`.
- Tag it in the format `vX.X.X` (eg, `v1.0.4`).

v1.0.4

@


 Target: master ▾

Excellent! This tag will be created from the target when you publish this release.

v1.0.4

Write

Preview

 Markdown supported

Login and signup flows have been optimized for user experience.

Attach files by dragging & dropping, [selecting them](#), or pasting from the clipboard.

Creating a new GitHub release.

Emergency fixes

There are times when bug fixes must be made outside the release cycle. This is called a *hotfix*.

The main branches (`develop` and `master`) are never to be pushed directly to, and hotfixes are no excuse to violate this rule. Instead, they should be made as hotfix branches.

Hotfix branches

Hotfixes branches are just like feature branches. They're worked on in the same way: pull requests, review, merge. The only difference is that they're branched from `master` instead of `develop`.

```
# Switch to the master branch
git checkout master

# Create your branch
git branch hotfix/security-fix

# Start working!
```

Merging hotfixes

Pull requests are made for hotfixes too, just like feature branches. A colleague will have to review and merge your pull request, ideally.

The `master` branch will be updated once a pull request is merged in. This should trigger your automated deployment process and deploy the hotfix to production servers.

Syncing the develop branch

Once the pull request is merged, `master` now has the fixes, but not `develop`. In these cases, you can do one of two things:

- Option A: Merge `master` back into `develop`.
- Option B: Merge the hotfix into `develop` as well ("backporting").

Option A is preferred in most cases, with option B only being used when merging `master` back may be too complicated.

Self-merging

There may be cases when fixes are time-sensitive and colleagues aren't available to review and merge your changes. If this happens:

- Create a pull request as usual.
- Merge the pull request yourself.
- Leave a comment in the pull request with the reason for urgency.
- Inform your teammates afterwards so they can still review your changes.

► See also...

Why we use branches

We've been describing a git workflow that relies heavily on branches. This isn't the only way to use Git. In fact, the most obvious way to use Git is how the Git manual first tells us to: push all commits to a single `master` branch.

This works for solo developers, but it breaks down once you have two or more on the team. Let's go through the problems that would come up in a single-branch model.

Problem: changes aren't contiguous

X - X - 0 - 0 - X - 0

Working on a single branch
ranches
(interleaved)

X - X - X - 0 - 0 - 0

Working using feature b
(contiguous)

If two people are working on 2 features, their commits may end up being interleaved. That is, for features `x` and `o`, the commits for either feature may not show up next to each other. This makes reviewing changes more difficult.

Problem: changes can't be reverted

To revert a feature that spans multiple commits, all those commits have to be reverted individually.

```
# Single-branch workflow: reverting each commit individually
git revert 1a84c2b9
git revert 9bf3cb70
git revert 6b9c5cb4
git revert 215c3820
```

Git solves this by grouping multiple commits together into a single *merge commit*. This is what happens when a pull request is merged. This merge commit can then be reverted, and doing so will revert all commits that it groups together.

```
# Feature branch workflow: only one commit to revert
# (Assuming 75a6c2a0 is a merge commit)
git revert 75a6c2a0 -m 1
```

Problem: reviewing is difficult



Hey buddy, can you review these commits? e433a7f1...fe860319 Oh and 8880576c and 8dec7864 too

vs.



Can you take a look at PR #283?

It becomes harder to review changes when they're not contiguous or grouped in any way. By grouping them together, you can create a GitHub pull request where your peers can browse through your changes easily.

The solution

The Git workflow described in this chapter is a logical evolution brought about by the problems above. By compartmentalizing each piece of work into branches, we make it easier to manage large amounts of work over long periods of time.

Summary

The branch-based Git workflow helps your team collaborate seamlessly. By making sure that every change is reviewed, you get a useful trail of changes in the form of pull requests and merge commits.

- Everything you do has to be reviewed by someone else.
- It should be easy to review other people's work.
- All work happens in feature branches and have pull requests.
- Nobody should ever commit directly into `develop` or `master`.
- Set up a staging system to test the latest version.

The main branches

- `develop` is always ready for developers and testers.
- `master` is always ready for users.

Making changes

- Create a feature branch.
- Send a pull request for this feature branch into `develop`.
- Your team will need to review and merge this pull request.

Releasing versions

- Merge `develop` into `master`.
- Create a GitHub release.

Deploying emergency hotfixes

- Create a branch from `master` .
- Send a pull request for this branch into `master` .
- Merge `master` back to `develop` if necessary.

Staging system

- Create a staging system that looks like your production system.
- Set up CI to auto-deploy to staging whenever tests pass.

See also

This git workflow is based on [git-flow](#).

- [git-flow](#) (nvie.com)
- [GitLab flow](#) (gitlab.com)
- [Git manual](#) (git-scm.com)

Automation

There are three things that need to be automated in every project:

- **Tests** - Automated testing streamlines your team's review process by making sure that changes don't break old behavior.
- **A staging server** - A server should always be available to test the latest development version.
- **Production updates** - Updating the production system should be as easy as possible.

The more you automate, the less you worry. The less you worry, the faster you can make progress. These automations can be set up using a Continuous Integration service.

Continuous integration

The screenshot displays the Travis CI web interface. On the left, a sidebar lists several repositories with their build status (green checkmark for success), commit count, and duration. The main area shows the build details for the 'green-eggs/ham' repository, including the commit message 'master adding in Oh the places you'll go!', the commit hash 'd019f29', and the build status '209 passed'. Below this, a build log is visible, showing the execution of commands such as 'git clone', 'git checkout', and 'npm install'.

Repository	Status	Commit	Duration	Finished
green-eggs/ham	Success	#22	30 sec	less than a minute ago
one-fish/two-fish	Success	#266	33 min 46 sec	30 minutes ago
hop-on/pop	Success	#701	22 min 54 sec	about an hour ago
horton-hears-awho	Success	#209	53 sec	about 2 hours ago
ohtheplacesyoull-go	Success	#778	6 min 55 sec	about 4 hours ago
thegrinch/whosolechristmas	Success	#35	15 min 50 sec	

A modern development pipeline will need a continuous integration (CI) service. A CI service lets you run tasks whenever your project gets updated. These are usually used to automatically run tests in your project. CI services are used to automate:

- Running unit and integration tests
- Deploying to websites
- Generating builds (eg, mobile and desktop apps)

Popular CI services

There are many CI services available today. Most of them offer free plans for open source projects, with paid plans available for private projects. Here's a selection of some of the more popular ones at time of writing:

- [Travis CI](#) is popular among open source projects.
- [Appveyor](#) is useful for Windows projects.
- [Codeship](#) specializes in Docker.
- [GitLab](#) offers a CI service with its platform.
- [Semaphore](#)
- [Circle](#)

It doesn't really matter which CI solution you pick. What's important is that you use a CI to automate your testing and deployments. What you choose typically depends on the features you need and your budget.

Automated testing

Having a CI test your code automatically is critical in a branch-based Git workflow. At the very least, you should have a CI service configured for your project to:

- Run tests in every `git push` to your repository.
- Display test results in GitHub and Slack.

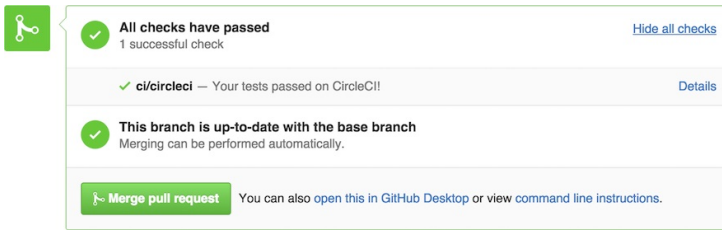
Setting up automated testing

Set up your CI to run tests on every Git push. The steps for this varies for every CI service, so consult your CI manual for details. Typically, this means listing a set of commands that your CI will run when it detects changes.

Here's a contrived example:

```
# Example CI script for Node.js and Ruby
npm install
bundle install --path=vendor/bundle
npm test
bundle exec rspec
```

GitHub integration



When reviewing pull requests, you'll need to ensure that the tests are adequate and passing. A CI service can automate running tests for you and display those results on GitHub.

Most CI services integrate with GitHub and other Git hosting platforms. This helps enforce good pull request hygiene: every code change needs tests, and they all need to pass before merging.

► See also...

Deployment automation

Before you can use a CI for continuous delivery, you'll need to set up a deploy automation tool.

A modern project needs to have a simple set of commands to magically update your servers. Contrast this to legacy projects, where deployment may involve manually transferring files and using a browser to invoke certain commands. These tasks need to be automated so they can be repeatable and protected from human errors.

Example

For websites hosted on [Heroku](#), their platform already takes care of deploy automation for you. To update a Heroku website, you simply need to do a

```
git push :
```

```
# Deploy your app into Heroku's hosting platform
git push heroku master
```

In other cases, you'd have to use other tools. For Ruby projects, [Capistrano](#) is a popular choice. A typical Ruby/Capistrano project would be deployed like so:

```
# Deploy your app using Capistrano
bundle exec cap production deploy
```

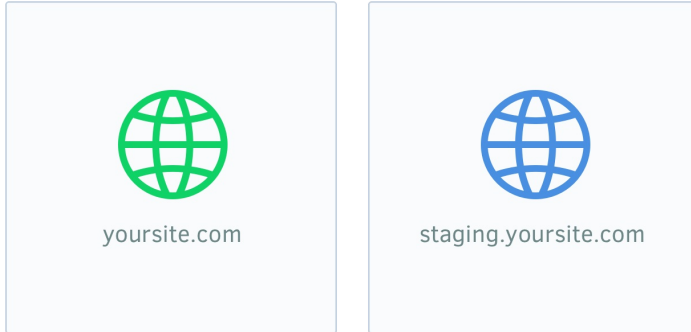
How to set up deployments

There are many different tools for this for many different programming languages. Some popular ones include:

- [Fabric](#) (Python)
- [Capistrano](#) (Ruby)
- [Shipit](#) (Node.js)
- [Edeliver](#) (Elixir and Erlang)

It doesn't matter which solution you pick, as long as your project's deployments are easy, automated, and reliable.

The staging system



In web development, a staging server is a separate site where the latest development version is always available. It's completely separate from your production system. This lets your developers, testers, and stakeholders try the latest changes without having to touch the production system.

Setting up a staging system

The most ideal staging setup is a system that's exactly like your production system: same type of server, same type of databases, but hosted separately. You can use a cheaper configuration, since less people will be using your staging.

Most projects start with using [Heroku](#) for their staging system. It's free, it supports almost every programming language, and setting a new website up takes 2 commands (see Heroku's [documentation](#)).

Automating staging updates

Deploy commands

```
1 git push --force heroku $BRANCH_NAME:master
```

```
2 heroku run rails db:migrate
```

Change deploy commands

Example deploy script in Semaphore.

A typical CI setup is have the following auto-deployments done:

- Auto deploy passing tests in branch `develop` to your staging server.
- Auto deploy passing tests in branch `master` to your production server.

When tests pass in a given branch, an auto deploy script will simply run certain commands.

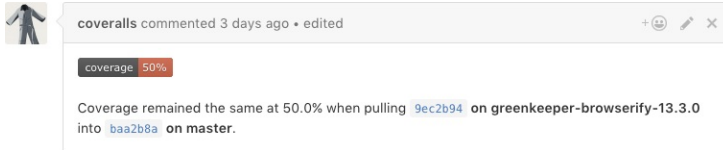
Seeding a staging system

The data in a staging system is completely different from your production system. It will probably start off as blank. You'll need some test data in your staging server to make it look like a production server. Making dummy data is a process called seeding. Seeding generates some dummy users, articles, and whatever your project needs to get it to a state where it "looks" like a production system.

To do this, you'll make a seed script in your project invoked by a certain command. In Rails projects, that's:

```
# Invokes db/seeds.rb  
bundle exec rails db:seed
```

Coverage reporting



Tests are often measured by how thorough they are using a metric called *code coverage*.

Code coverage is measured in percentage points. It approximates how much of the system is tested with a given test suite. A code coverage of 80% means that running the test suite ran 80% of the code in the project. The higher the number, the better.

Why track code coverage

Code coverage is often used to gauge the health of a project's test suite. Coverage will go down when changes are introduced without any tests. It's a good idea to track a project's code coverage to help reviewers gauge code quality of pull requests.

Code coverage services

CI services don't typically offer code coverage tracking. There are, however, other platforms that offer this. Like CI's, they're typically free for open source projects, with paid plans available for private projects. Here's a list of some popular services:

- [Coveralls.io](#) is popular with open source projects.
- [CodeClimate](#) offers code coverage tools as well as other "health" checks.

Are they worth it?

Code coverage reports offer valuable insight into a project's code quality. Most projects can probably do fine without it, though.

Summary

Using a Continuous Integration (CI) service to automate your testing and deployments lets your team move faster. It also enforces good testing practices by making sure that all tests pass before any code is merged into a project.

Using a CI

- Automate your tests.
- Automate your deployments to production and staging.

Writing code

- Every pull request needs tests.
- Tests need to pass before they're merged.

Automating tests

- Let your CI run tests for every pull request.
- Don't merge pull requests that don't pass CI tests.

Automating deployments

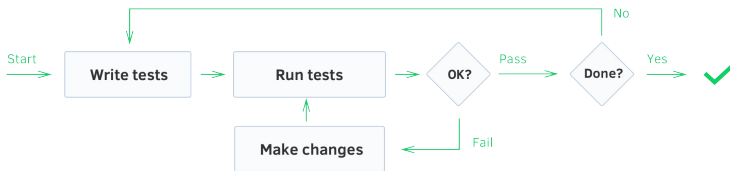
- Automate deployments to happen when certain branches pass tests.
- Make `develop` automatically deploy into a staging system.
- Make `production` automatically deploy into the production system.

Coding practices

This article is a stub. You can help by expanding it.

- Write tests.
- Set up linting.
- Make a README.

Test-driven development



Test-driven development, or TDD, means writing tests along with your code. It's not about writing tests after you're done, but making testing a habit as part of your development process.

- Tests should be written alongside your implementation.
- New features need to have new tests for it.
- Features can't be merged unless tests pass.

What to test

A healthy code base has tests written in this priority (most important to least):

1. Unit tests
2. Smoke tests
3. Integration tests

Units tests are easy to make and are a natural part of the process of writing models. Smoke tests are small and also easy to make. Integration tests are larger and harder to maintain, so it's often more economical to prioritize

writing smaller unit tests.

In the Git workflow

The main branches `master` and `develop` must always be in a working state. Pull requests can't be merged until their tests pass.

Types of tests

Tests will commonly be either a *unit test* that tests one thing, or an *integration test* that tests many things. Here's the same behavior tested in two different ways:

```
# Unit test example: tests the model
test 'Article publishing' do
  article.publish!
  expect(article.published_at).to eq Time.now
end
```

```
# Integration test example: tests your router,
# controller, view, and model all at once
test 'Article publishing' do
  post '/articles/my-draft-article/publish'
  expect(article.published_at).to eq Time.now
end
```

In web development, unit tests are often written for models and services, while integration tests are for controllers.

Smoke tests

A smoke test is a very shallow integration test. It usually just tries something and expects no errors to happen. The name comes from testing electronics to check if turning a device on produces smoke. In web development, it usually means just visiting a path and checking for errors.

```
test 'No fires happen' do
  visit '/article/my-article'
  expect(response.code).to eq 200
end
```

The simple test above will already check for a lot of things:

- Any errors in the view templates
- Any model or controller errors
- Middleware in the app server

► See also...

Why we test

Writing tests make you work faster. This seems like a backward thought at first, since writing tests take time. In practice, these tests are automating parts of your workflow, and automation always makes you faster in the long run.

F5-driven development

Let's consider the alternative to TDD. Development in the console usually involves a REPL shell that lets you invoke some code in your project. Here's an example in a typical Rails project:

```
$ bundle exec rails console
Loading development environment (Rails 5.0.0.1)

irb> User.new(first_name: 'John', last_name: 'Smith').full_name
"John Smith"
```

People often abuse REPL consoles as part of their coding process. A common anti-pattern is to use a REPL to test changes as you make them. A typical REPL workflow looks like this:

- Make some more changes.
- Press `Alt+Tab` to switch to a terminal.
- Press `Up` then `Enter` to retry the last command.

In the case of web development, switching to a browser and hitting `CtrlR` or `F5` to reload is also very common. Wouldn't it be easier to automate this process for you?

```
test 'Builds a full name do' do
  user = User.new(
    first_name: 'John',
    last_name: 'Smith')

  expect(user.full_name).to eq 'John Smith'
end
```

By writing those same REPL commands as a test, you can now re-run it using `ruby test/user_test.rb`. In fact, you can run all the tests you've ever written!

Tests as insurance

Tests let you update code mercilessly. Have you ever been in a project that uses very old versions of 3rd-party libraries such as Rails or jQuery? These projects are too afraid to upgrade knowing it may break things.

By writing tests, you'll be alerted of breakages when updating your project's dependencies. Changing the jQuery version may make some tests fail. You'd know then where to fix things.

Refactoring mercilessly

Tests aren't only for upgrading 3rd-party dependencies. Even updating one part in your project can result in breaking another part. A good refactoring process starts with writing tests. When changing code, it helps to know what it can break.

Tests as documentation

While tests are no substitute for formal documentation, it can provide a quick reference for other developers. Consider this test:

```
test('OrderCalculator calculating charges', t => {
  let order = {
    type: 'pickup', // or 'deliver'
    items: [
      { product: 'Glazed donut', price: 2.99, qty: 2 },
      { product: 'Coffee', price: 3.99, qty: 1 } ] }

  let charges = OrderCalculator.getCharges(order)
  t.deepEqual(charges, {
    subtotal: 9.97,
    tax: 1.19,
    delivery: 0,
    total: 11.16 })

  t.end()
})
```

Another developer would easily know at a glance what `OrderCalculator` does, what input it takes, and what results it yields.

Linting

```
function showUsername () {  
  var user_name = user.getFullName();  
  return user_name;  
}
```

✖ Error: Identifier 'user_name' is not in camel case.

Linters are tools that enforce source code writing style based on a set of rules. It helps impose coding styleguides in an automated way. A linter will check decisions such as:

- Syntax errors
- Tabs or spaces for indentation
- Naming of variables and functions
- Use of whitespace
- Patterns to use and to avoid

It's important to keep your coding style consistent. Even if it's mostly subjective, a consistent style helps maintain developer sanity and code readability.

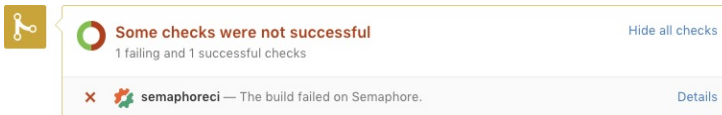
Automated linting

Linting should be part of your automated test suite. This lets you catch style violations before pull requests are merged. Here's an example using Travis CI:

```
# Example .travis.yml configuration to run CS
# and JavaScript linting
scripts:
  - npm test
  - npm run stylelint
  - npm run eslint
```

Why use a linter

Linting helps reviewers by catching style violations in automated tests so they don't have to. This means less effort needed for reviewing, and instant feedback for contributors.



Linting prevents merging when style issues aren't fixed.

Choose a coding style

There are many established styleguides and linter presets. Pick one and customize it if preferred. It doesn't matter which you choose, what's important is that your team has an authoritative set of rules that are enforced by a linter. This prevents petty "bikeshedding" arguments within your team.

- JavaScript: [Standard](#), [XO](#)
- CSS: [Config standard](#)

Linters examples

- [Stylelint](#) for CSS
- [Eslint](#) for JavaScript
- [Rubocop](#) for Ruby
- [Credo](#) for Elixir

Project lifecycle

This article is a stub. You can help by expanding it.

- Break your effort into milestones and sprints.
- Ideal sprint period is 2 weeks.
- Maintain good release hygiene.
- Use three-point versioning (semantic-style).
- Publish release notes.

Phases of a project

This article is a stub. You can help by expanding it.

There are two phases to a project:

- **Planning** - Research and planning
- **Iteration** - Where the actual work happens

Planning

This is where you do your research and design

- Market research
- Product design
- Visual design (UI/UX)
- Planning the next 2-4 sprints

Iteration

- Plan
- Build
- Test
- Release

The "launch" myth

Many system development lifecycle models ("SDLC") focus on a product "launch date" when the project is ready, and an "end date" when it's finished. These concepts don't apply to our web application model, which focuses on an iteration cycle.

No product launch

Every end of a milestone is a launch. At the end of every milestone, your team is expected to deliver a working product. You may have one big launch where your site is finally open to the public, but this is simply one of your project milestones.

Conversely, there's also no project "post-launch". Other lifecycle models tell us to measure metrics, fix bugs, and reflect after a launch. These should happen every sprint.

No project end date

Working on a product never ends. Apps and websites are continuously improved with enhancements and bug fixes. If it's finally feature-complete, then it's time to make new features, or plan new enhancements.

A project ends when the stakeholders decide it to end, and these shouldn't be any different from how sprints usually end.

Iteration is the key

To sum it up, our model focuses on an iteration cycle divided into milestones and sprints. Each sprint is essentially a mini-project that starts and ends.

Writing bug reports

Bug report example

Can't change passwords

When I'm logged into as a User, the "change email" form returns an error. Using Firefox on OSX. See `oursite.com/users/23/edit`.

300×100

A good bug report has a short title, a one-sentence short description, relevant details, and screenshots. Make sure these details are listed down, or are obvious enough:

- URL (*if applicable*)
- Environment (*OS, browser, website environment*)
- Screenshot
- Steps to reproduce
- Expected result
- Actual result

Writing titles

Keep the title short. Put any details in the body, not in the title. This makes it easier to digest a long list of issues.

- "When I try to change passwords, an error is raised." *...can be shorter*
- "Can't change passwords" *OK!*

Try to be specific. Titles that are too broad are not helpful.

- "Products error" *...too broad*
- "Can't buy products" *...still too broad*
- "Product page error" *...still too broad*
- "Product page: the buy button isn't clickable" *OK!*

Full example

This example also includes steps, actual result, and expected result details. While some reports don't need all these details, it's a good idea to include them anyway.

Bug report example

Can't send email

The "send email" button isn't clickable. Using Firefox/OSX on Production. See oursite.com/user/23/edit .

300×100

Steps

1. Log in as a user
2. Go to the Inbox page
3. Click "write new email"
4. Click "send"

Expected: The email will be sent.

Actual: A JavaScript error appears when "send" is clicked.

—

Team communication

This article is a stub. You can help by expanding it.

- Use Slack!
- Optimize for working asynchronously
- Use Slack notifications
- Maintain a team "Twitter"

Using Slack

A team should have these channels for every project:

- A channel where everyone is, including developers and stakeholders (usually `#general`).
- A developer-only channel (usually `#dev`).
- Notifications for project tracker updates (usually `#issues`).

General channel

#general

Stakeholder: Good morning guys, are we still on track for sprint 5?

Daphne: Yep! We should be about 80% there right now.

This is the default channel for most discussions. Collaborating with stakeholders happens here. Most conversations should happen here, unless they happen to be too technical, in which case they should be in `#dev` .

Developer channel

#dev

Fred: Can you guys take a look at PR #283? I need a review

Velma: Hold on, let me take a look at it.

Have a channel where only developers are present, without any stakeholders. This is where everyday tech-related discussions will happen. Banter in this channel is usually not relevant to stakeholders, so it's best to leave them out so not to overwhelm them.

Temporary channels

#_facebook-campaign

Marketer: We need a new landing page

Fred: We're on it. What do you need?

Create new channels for discussion threads, and archive them when you're done. This lets certain discussions happen separately from everyday chatter, keeping logs clean. It also prevents overwhelming people by leaving out those who may not be involved. A good practice is to these with an underscore, such as `#_feature`.

Project tracker channel

#issues

GitHub: Shaggy submitted pull request #283: *Homepage: fix responsive issues*

Trello: Scooby added a new card: *Facebook landing pages*

Integrate your project management tool (eg, Trello) and GitHub's pull requests into a channel such as `#issues`. Don't talk in this channel, take discussions to `#general` instead since conversations are hard to track among notifications.

Other notifications

Also consider adding notifications for other services. This largely depends on the nature of your project. Here are some ideas:

- `#analytics` for website analytics (eg, Google Analytics)
- `#support` for customer support tickets (eg, Intercom)
- `#code` for GitHub activity
- `#ding` for sales notifications

Team management

This article is a stub. You can help by expanding it.

- Have a shot-caller (aka scrum master)
- Work in sub-teams if possible
- Ideal team size is less than 10 people
- Break your product apart if needed