SOFTWARE DEVELOPMENT LIFE CYCLE

Version Number: 0.1

Version Date: <3/15/2016>

TABLE OF CONTENTS

SEPARATION OF PRODUCTION AND TESTING DATA

DATA STORAGE

All production databases are hosted on a separate server and accessed with separate credentials from our development databases. This allows us to prevent cross-contamination and exposure of data between environments.

TEST DATA

Database seeders and model factories are used extensively in the testing of our applications to avoid the use of production data. These allow us to mock out normalized data as well as edge case data for use in the development cycle.

USE OF PRODUCTION DATA

In cases where the use of production data cannot be avoided, the following controls are in place to ensure that production data is secured when used outside of the production environment.

Business Justification

The Business justification for access to the data should must be documented detailing why production data, rather than data generated for test purposes, is required. This request is passed along to the Director of IT for approval before production data can be used outside of the production environment.

Documented Process

The process used for the production data to be transferred, handled and disposed of is documented below. This helps to ensure that a consistent approach is followed by everyone involved in the process.

1. Transfer of production data can only be done over an internal network connection.
2. Only the minimum amount of data that is required for testing should be downloaded. This reduces the data set that can be compromised.
3. The data should be retained outside of the production environment for the minimum time possible. The longer the data sits in the test environment (or outside the production environment), the higher the chances of compromise.
4. After the testing is completed, the data should be securely disposed of. If the data is in the file system, then a secure wipe tool should be used. If the data is in the database, the records should be deleted.

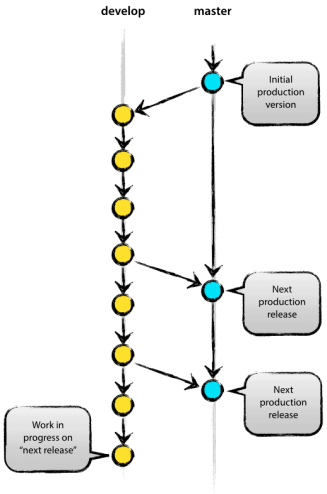
SEPARATION OF PRODUCTION AND TESTING CODE

VERSION CONTROL

All of our software is version controlled using GIT (https://git-scm.com/) and synced between contributors using GitHub (https://github.com).

Using a decentralized version control system allows multiple developers to work simultaneously on features, bug fixes, and new releases. This also gives each developer the ability to work on their own local branches of the code and in their own local environment. All development code is written, tested, saved locally before being synced to the origin repository. Writing code locally decouples the developer from the production version of our code base and insulates from accidental code changes that could affect our users. In addition, any changes involving the persistence layer (file system and database) are performed locally when developing new code.

BRANCHING MODEL



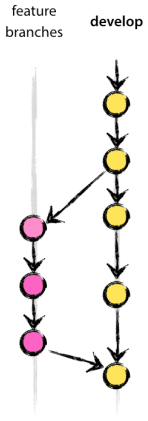
Production Branch

Depending on the repository, we consider the `origin/master` or `origin/production` branch to be the main branch where the source code always reflects the current state of the application in production.

Development Branch

We consider the `origin/dev` branch to be the main branch where the source code reflects a state with the latest delivered development changes for the next release. We may also refer to this as the "integration branch."

Feature Branches



May Branch off from:

`dev`

Must merge back into:

`dev`

Branch Naming Convention:

`feature-*`

Feature branches are used to develop new features for the upcoming or a distant future release. When starting development of a feature, the target release in which this feature will be incorporated may well be unknown at

that point. The essence of a feature branch is that it exists as long as the feature is in development, but will eventually be merged back into `dev` (to definitely add the new feature to the upcoming release) or discarded (in case of a disappointing experiment).

Feature branches typically exist in developer repos only, not in `origin`.

Finished features may be merged into the `dev` branch through a Pull requests.

Hotfixes Branches

May Branch off from:

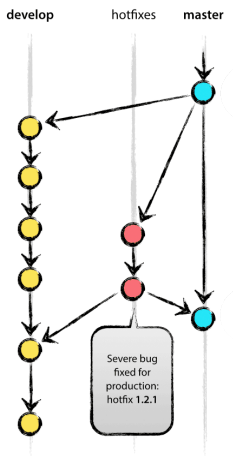
`master`

Must merge back into:

`dev and master`

Branch Naming Convention:

`hotfix-*`



Hotfix branches are meant to prepare for a new production release, albeit unplanned. They arise from the necessity to act immediately upon an undesired state of a live production version. When a critical bug in a production version must be resolved immediately, a hotfix branch may be branched off from the `master` branch that marks the production version.

The essence is that work of team members (on the `dev` branch) can continue, while another person is preparing a quick production fix.

When finished, the bugfix needs to be merged back into `master`, but also needs to be merged back into `dev`, in order to safeguard that the bugfix is included in the next release as well. The merge into both branches should be done through the use of a Pull requests.

CONTINUOUS INTEGRATION AND DEPLOYMENT STRATEGIES

PULL REQUESTS

When work has been completed on a hotfix or feature branch of a repository, those changes are merged into the `dev` and/or `master` branch through the use of a pull request. Pull requests let you tell others about changes you've pushed to a repository on GitHub. Once a pull request is sent, interested parties can review the set of changes, discuss potential modifications, and even push follow-up commits if necessary. Pull requests are also used to trigger automated testing and code-quality checks that must be completed and returned successfully before merging is allowed.

Within the details section of the pull request, the requesting developer should clearly note any related changes that need to take place before the commit is merged. These would include changes to the environment, database, or file-system.

AUTOMATED TESTING

When a pull request is initiated, our automated test suite is triggered to run against the new code. Our current test runners are listed below.

Travis CI

Travis CI is a hosted continuous integration platform that will build and test any changes to the codebase. The process is detailed below.

1. A new machine image is created.
2. The project and requested changes are downloaded to the new image.
3. All dependencies are attempted to be resolved and downloaded.
4. All tests for the codebase are run against the new project code.

If any of these steps fail, the build is rejected and a detailed error log is returned. Any branch that does not return a successful build cannot be merged.

Code Climate

Code Climate is a service that monitors the health of your code by checking for:

- Test-coverage
- Code Complexity
- Code Duplication
- Security Vulnerabilities
- Adherence to coding standards (PSR-2)

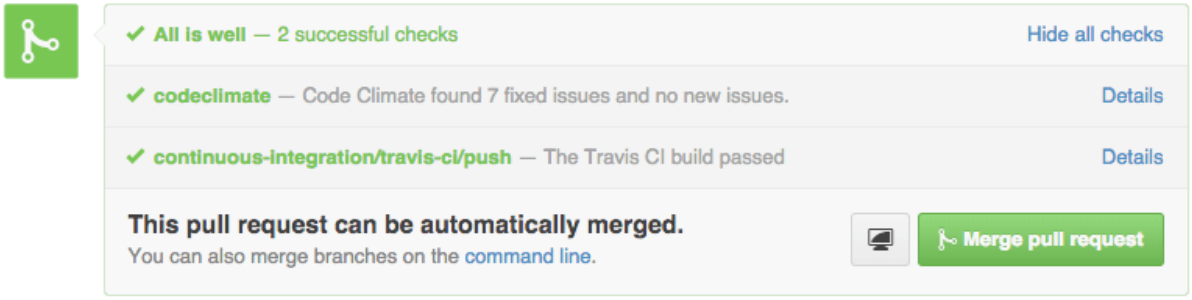
After running these automated checks against the proposed set of code changes, Code Climate will return suggestions and warnings as comments on the pull request, referencing specific locations where the issues stemmed from. The developer requesting the pull request can then review the suggestions and make changes to the code as necessary.

Security Advisories

To further insulate our code base, we include a package developed by some of the most recognized and accomplished experts in the industry to ensure that our application doesn't have installed dependencies with known security vulnerabilities. If any packages are marked as insecure, they will be removed or replaced with secure alternatives.

SUCCESSFUL PULL REQUEST EXAMPLE

Below you will find what a successful pull request might look like. Notice the automated tests which have passed successfully indicated by the green checkmarks.



CODE REVIEWS

Before any code can be merged, a code review must be performed. A code review is when another developer looks over the proposed changes and considers questions like:

- Are there any obvious logic errors in the code?
- Looking at the requirements, are all cases fully implemented?
- Are the new automated tests sufficient for the new code? Do existing automated tests need to be rewritten to account for changes in the code?
- Does the new code conform to existing style guidelines?

A code review should take place after all the code has been written and automated tests have been run and passed–but before the code is merged upstream. This ensures the code reviewer's time is spent checking for things machines miss, and prevents poor coding decisions from polluting the main line of development.

In rare instances (hotfixes), it may be necessary for a developer to merge code before there is time for the code to be reviewed by another developer. This requires express permission from the head of the IT Department and will be noted as such in the pull request.

MERGING A PULL REQUEST

Before merging a pull request, the merging developer ("developer") should check that all requirements listed in the details section of the pull request have been implemented. These changes would include environment, database, or file-system changes. After ensuring that all non-code changes have been implemented, the pull request can be merged through the

command line or through the GitHub UI. If the application is deployed through our zero downtime deployment process, the developer's job is complete. If not, it is the developer's responsibility to update the affected application by pulling down the most recent copy of the production code from GitHub.

If any of these changes necessitate system down-time, the merge must take place within a scheduled change window or at a time when the application is not in use.

ZERO DOWNTIME DEPLOYMENT

In any applications that require high availability, we have implemented zero downtime deployments. Zero downtime deployments allow us to make changes to our codebase without having to wait for a change window and also allows us to roll back to a previous working application state at the push of a button.

EXTERNAL SOFTWARE SECURITY AUDITS

To ensure that our applications are hardened to prevent against external or internal attacks, we take a proactive approach to look for any vulnerabilities that may be present in our software and patch them before they can be exploited. We do this through the use of a third party that conducts an annual manual penetration test as well as quarterly vulnerability scans.

These reports are used to assess any weak spots in our applications or our environment. Any areas of vulnerability are fixed in a timely manner.

APPLICATION FRAMEWORKS AND PLATFORM VERSIONS

APPLICATION FRAMEWORKS

In order to help protect against insecure coding practices and common vulnerabilities, we base our applications on well tested and community proven frameworks which have been released for long term support. All of our applications run on versions of their framework which are still being actively developed and receiving bug fixes. Some of the vulnerabilities these frameworks help to protect against are:

- Cross site request forgery (CSRF)
- Session hijacking

- Source code revelation
- Cross-site scripting (XSS)
- SQL Injection

PLATFORM VERSIONS

In addition to protecting our applications through the use of frameworks, we are also vigilant in keeping the platforms our applications are built on up to date. Updates to the server and network infrastructure are maintained on a routine basis. When major versions of software are released, we begin testing in our development environments and through our continuous integration platforms. Once proven to be stable, we begin the process of upgrading or replacing the underlying infrastructure to accommodate the new features. Critical security bulletins and patch releases are deployed quickly to ensure the security of our applications.

COMMITMENT TO DEVELOPER EDUCATION

One way that we attempt to improve and secure code in a less measurable way is through our commitment to developer education. This commitment is displayed in the opportunities and resources afforded to our development team on a continuing basis as well as through the bonus structures that are in place to encourage developer learning and participation.

DEVELOPER CONFERENCES

Developers are mandated to attend at least one conference a year related to their field of development. Conference fees, lodging, and associated expenses are paid by the company along with the required time off to attend. By exposing our team to the latest developments in their areas of expertise, we hope to encourage best practices and allow for new insights when refactoring current and legacy applications or building new ones.

EDUCATIONAL SUBSCRIPTIONS

Developer learning is supplemented by paid subscriptions to company approved or developer requested resources. These resources include access to web based learning platforms as well as monthly print publications. Subscriptions provide one of the best ways to ensure training materials are up to date and continually improved with new versions and development methodologies.

CERTIFICATIONS

The pursuit of industry recognized certifications are encouraged, financed, and incentivized by the company. Some of the encouraged programs include:

- Certified Secure Software Lifecycle Professional (CSSLP)
- AWS Certified Developer
- MySQL Developer