Savas Labs Developer Docs Documentation

Release 0.2

Kosta Harlan, Dan Murphy, Lisa Ridley, Chris Russo, Tim Stallmar

Oct 17, 2018

Topics

CHAPTER 1

Drupal

This section is a work in progress!

1.1 Security updates

Our general policy is:

1. Only update contrib modules with security advisories (i.e. do not update all contrib modules to their latest stable release)
2. Submit a pull request with one commit per module updated, to make reviewing changes (and reverting, if need be) easier

When updating sites without test coverage, be especially cautious. When updating contrib modules, always:

1. Review the issue queue for the module and review bug reports that exist for the version being updated to
2. Review the code changes to the module and note if any backwards incompatible API changes have been made

Note that in some cases, the security advisory may not apply to the contributed module as used on the site (i.e. "Only users with `Administer flags` can exploit this vulnerability"), in which case using Update Advanced to ignore the update may be a good option.

1.2 Repository structure

1.3 Drupal root structure

e.g. contrib/custom/patched/features etc directories

settings.php set up, etc

1.4 Theming

1.4.1 Theme development

Drupal.org's theming guide is a vast resource for theming best practices and techniques. The guide is split between Drupal 6 & 7, and Drupal 8.

1.4.2 Best practices

Start by reviewing the theming guide's best practices section.

Keep logic out of template files

In Drupal 6 and 7, PHP logic should mostly exist in `template.php` and feed prepared variables to template files rather than executing the PHP within the templates. In Drupal 8 PHP can't be executed in template files at all and must live in `[theme name].theme`.

SMACSS

When creating a custom theme, use SMACSS or a simplified version for file structure.

BEM

Use the BEM methodology for naming CSS classes. CSS Wizardry has a great primer.

1.4.3 Workflow

If possible, it's best to compile CSS files as part of deployment and local development rather than committing compiled CSS. That said, in certain development workflows (e.g. Pantheon) it may be necessary to commit compiled CSS files.

CHAPTER 2

Git

2.1 Basic git workflow (git flow)

We use git to manage our work, and generally use the git-flow workflow. Read through the descriptions of feature branch workflow and git flow workflow for more background.

The essentials are:

- Each project has a base repository, either under the `savaslabs` GitHub account or under a client account.
- This repository generally has `master` and `develop` branches, although this can vary on a project-by-project basis. For chunks of work which involve a longer development time before being merged into master, we sometimes open a feature branch on the base repository.
- Individual developers fork the base repository to their own GitHub account, and add both the base repository (as `upstream`) and their fork (as `origin`) to their local git instance.
- Never push new code from your desktop to the base repository; instead push branches to your fork.
- Code changes on the base repository should always happen via pull requests that involves a peer review process.

General workflow:

1. Pull the most recent version of the `develop` (or other relevant base branch, depending on the project) branch from the base repo to your local environment.
2. Open a new branch for the new feature you're working on. Usually this corresponds to an issue number on redmine. Branches should be named `feature/issue#-description`, like `feature/824-refactor-module-code`. Following the git-flow model, this holds for all new work, even if the "feature" is actually a bugfix. The only exception is `hotfix` branches, which are not discussed here. We do this because it makes it easy at a glance to differentiate between feature branches in the list of branches on the repository.
3. Working locally, make commits for your work (see below).
4. When the feature branch is ready for review, rebase the feature branch on the most recent `develop` branch (if there have been changes). Developers may choose to merge the new changes into the feature branch instead.

5. If necessary, do an interactive rebase to cleanup redundant multiple commit messages.
6. Push the feature branch to your fork (the `origin` remote).
7. Open a pull request (see below) to merge the feature branch **from** your fork of the repo **to** the base repository.

2.1.1 GitHub flow

On simpler projects, rather than using the full git flow process (where features are merged into `develop` and then tagged releases are created separately on `master`), we use a streamlined workflow where feature branches are merged directly into the `master` branch. This workflow is described here.

2.1.2 Git flow extension

It is also possible to automate parts of the above workflow using the gitflow extension, which some developers at Savas Labs use. Project managers may use git-flow as well to create tagged releases.

2.2 Git commit messages

Good git commit messages are helpful for consistency with your collaborators. They also make your code workflow thoughtful and professional. There doesn't seem to be an industry-wide standard on *the* perfect message format, but for our purposes, the most helpful guide to reference is this one. It proposes this easy rule set to follow:

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Wrap the body at 72 characters
7. Use the body to explain what and why vs. how

Please read through and reference the post for more detail on each of the items above.

We also have a few in-house additions to this:

1. Reference the relevant Redmine/GitHub issue at the beginning of each commit, e.g. "Issue #123: Fix git commit message docs"
2. If you find yourself leaving important/lengthy notes in the commit body, consider whether that message should also be present as code comments for improved discoverability

2.3 Creating a good pull request

A pull request is an important piece of communication, both to other developers reviewing your code and to future developers who need to understand the history of the project.

Like commit messages, pull request titles should start with `Issue #___:` and be a one-line summary of what the PR does. Ideally, the branch used for the pull request also contains the issue number, e.g. `933-good-pr-docs`.

If your pull request is not ready for review, mark it as a work-in-progress by putting `[WIP]` in front of the title. Once your PR is ready for review, be sure to remove the `[WIP]` tag from the title and notify the developer responsible for reviewing it.

A good pull request description should:

- Give a concise summary of what changes are introduced to the codebase and/or functionality in the pull request. Often we do this using a bullet list.
- List any steps which will necessary to deploy the pull request, both locally and in production – this should have all the information necessary to deploy.
- Give any specific instructions that are necessary for testing. Include links to URLs for testing whenever you can!
- Specify any specific issues that you want developers doing peer review to be sure to investigate thoroughly.
- Give a timeframe for review.

After you've written your pull request, look it over and ensure that all the important details are also contained in code comments and commit messages.

Once the pull request is open, you should also assign a specific reviewer in GitHub, set the applicable Redmine task(s) status to `Developer Review`, assign those tasks to the reviewer, and paste a link to the pull request in the `GitHub PR` field. If the request is urgent, you should also communicate with the reviewing developer via some other channel (Slack, in person, etc) rather than relying on them to get GitHub or Redmine notifications.

CHAPTER 3

Coding standards

Writing code that adheres to standards is important for many reasons. Coding standards:

- Ensure that large projects are coded in a consistent manner
- Demonstrate our professionalism as developers
- Improve maintainability by making code easier to understand, review, and contribute to

3.1 Drupal coding standards

Each programming language has its own set of coding standards, and even within a language, different frameworks may have their own separate standards. That is the case for PHP and the Drupal framework.

When writing code for Drupal projects, we adhere to the Drupal coding standards. There is extensive documentation of the Drupal coding standards that our code should adhere to. Note, any custom PHP in a Drupal project (for example, a `RoboFile.php`, or Behat's `FeatureContext.php`) should also adhere to Drupal coding standards for consistency.

3.2 Good practices

In addition to adhering to Drupal coding standards, there are other good practices we abide by:

- Include code comments anywhere that it would help other developers review or understand your code. It's better to over comment than under comment. Use git commit messages to provide additional information, but ensure that all the context a developer needs to understand the code is available in the code comments.
- Always use descriptive, readily-understood variable names. For example use `$node` rather than `$n`.
- Write modular code — if you're copying a block of code from elsewhere in the codebase, ask yourself if you could convert that block of code into its own function.

3.3 PHP_CodeSniffer

To help us adhere to these standards, we use the PHP_CodeSniffer tool along with the Drupal Coder project which provides Drupal specific "sniffs" for PHP_CodeSniffer.

Drupal provides detailed instructions for installing PHP_CodeSniffer.

3.3.1 Command line usage

PHP_CodeSniffer can be run from the command line following these instructions. We recommend creating the `drupalcs` alias as explained in those instructions.

PHP_Codesniffer also includes the PHP Code Beautifier and Fixer command `phpcbf`. This command can be run from the command line to automatically fix your coding standards violations. More detail on the `phpcbf` command is provided in the the PHP_CodeSniffer wiki

3.3.2 PhpStorm integration

In addition to running PHP_CodeSniffer from the command line, it is useful to see coding standard violations highlighted in your editor, especially real-time as you develop.

If you use PhpStorm as your IDE, then you can use PhpStorm's built-in coding assistance to indicate violations of coding standards as you write or review code.

JetBrains (the team behind PHPStorm) provide documentation for Code Sniffer and Coder integration.

We also recommend updating PhpStorm's default syntax and formatting to conform with Drupal coding standards, further easing compliance with the standards. This guide documents steps for configuring PhpStorm for Drupal.

In addition, PhpStorm provides a Reformat Code command that automatically reformats selected code to your defaults settings.

3.4 Other tools

3.4.1 PHPMD

PHP Mess Detector (PHPMD) is another tool that helps you write better code. According to the documentation PHPMD "takes a given PHP source code base and look for several potential problems within that source... like: possible bugs, suboptimal code, overcomplicated expressions, and unused parameters, methods, [and] properties". The PHPMD documentation provides installation and command line usage instructions.

3.4.2 phpcs-security-audit

The phpcs-security-audit project provides additional secuity sniffs for PHP_CodeSniffer. Per the documentation "phpcs-security-audit is a set of PHP_CodeSniffer rules that finds flaws or weaknesses related to security in PHP and its popular CMS or frameworks".

3.5 Fixing legacy code

Often times, you'll find yourself updating legacy code. Unfortunately, not all developers adhere to coding standards as strictly as we do.

Our policy is that you should leave legacy code better off than the way you found it. If you find yourself updating pre-existing files, you should ensure that the entire file and not just your additions adheres to Drupal's coding standards before creating a pull request.

When updating legacy code to conform to Drupal standards the PHP_Codesniffer's `phpcbf` command and PHP Storm's "Reformat Code" command are very helpful.

To ease review of pull requests that affect legacy code:

- Coding standards fixes on legacy code should be in their own commits with no other additions - including bug fixes, feature additions, or refactoring.
- These commits should be the first commits in a pull request, with all other commits for bug fixes, feature additions, or refactoring coming after.

Abiding by these guidelines will make it much easier for your teammates to efficiently review your pull requests.

3.6 CSS and its preprocessors

CSS bloat (i.e. CSS that contains unneeded and/or repeated code) can cause performance issues and make future development more difficult and time-consuming.

To keep code as light as possible:

- Follow the DRY principle. If you find yourself repeating code, find a way to make that code modular. This is easy to achieve with Sass by using mixins and abstract classes.
- Learn when to use a mixin and when to use an abstract class (TL;DR: You should almost always use a mixin.)
- Always use as non-specific a selector as possible to avoid having to overwrite your own CSS. CSS Tricks explains specificity well.
- When writing SCSS, run `scss-lint` and follow the rules whenever possible. If you do need to make an exception (e.g. you need to use `!important` to override a default Drupal CSS rule that uses `!important`), leave a comment explaining why you're doing it.

To ensure that your code is maintainable by future developers (including Future You):

- Always include a docblock comment at the start of each file describing what the file does and anything non-obvious about the component.
- Include comments to describe the results of non-obvious rulesets, such as complex layouts or items that change drastically across screen sizes.
- Use the BEM class naming methodology. This makes the DOM more readable, and means you don't need to include a comment in the code indicating what a selector is targeting.

CHAPTER 4

Testing

4.1 Introduction

Testing is a critical part of the development process at Savas Labs:

- All projects should have automated tests
- New features added to a codebase should almost always come with tests to verify their functionality
- Bug fixes should almost always contain new tests or fixes to existing tests
- We can manage technical debt through good practices in automated testing

Why do we write tests?

- Define the feature you are building or the bug you are fixing
- Improve our code by checking against different assumptions
- Catch regressions
- Help future-you and or co-workers with refactoring
- A green :heavy_check_mark: on your GitHub pull request feels good

4.2 Types of tests

4.2.1 Linting

The simplest test is PHP linting. Running `php -l {some_directory}` will parse all PHP files and check for syntax errors.

All projects should include a lint check as part of the automated testing process.

4.2.2 Coding standards

All the custom code we produce must adhere to coding standards. For Drupal projects, this involves running the `phpcs` command with the `--standard=Drupal` option.

Example usage in a Makefile:

```
phpcs_config = --ignore=*.css,*.min.js,*features.*.inc,*.svg,*.jpg,*.json,*.woff*,*.
↪ttf,*.md \
--exclude=Drupal.InfoFiles.AutoAddedKeys

phpcs: ##@test Run code standards check.
  docker run --rm -v $$(pwd):/work skilldlabs/docker-phpcs-drupal phpcs --
↪standard=Drupal \
  tests drupal/sites/all/modules/custom $(phpcs_config)
```

All projects should include a coding standards check as part of the automated testing process.

4.2.3 Behat

All our Drupal projects should incorporate Behat tests, using the DrupalExtension plugin to provide access to the Drupal API from within the Behat testing environment.

We have examples of client projects to draw upon for example configuration of Behat.

Use Behat as a communication and discovery tool

Involve the client in defining the scenarios and features. This page contains a good overview of how you might do this. You should work with the client to define what the features/scenarios under development are before you write any code.

Write the feature/scenario definition before you build the functionality

Following from the above, do not write a Behat feature/scenario *after* you've coded the implementation. Work with the client to agree on how the feature should work. After you have defined the feature and how it should work in different scenarios, write the code that matches the specification so that the Behat test passes.

Prefer human-readable step definitions to reliance on selectors

Consider the following Behat step:

```
And I click on the element with XPath '//*[@id="edit-submit"]'
```

While we can read this and infer that `edit-submit` must be referring to the submit button on the page, it's much better to have something like this:

```
And I submit the form
```

Then, in your `FeatureContext.php` file, you could write the code which clicks on the element with the XPath.

The idea is to have a terse description of what the feature should do to provide for a shared understanding between the client and the developer. In this case, the client doesn't need to know about XPath or particular selectors; they will want to know what happens when someone submits the form.

Avoid adding more Behat tests than necessary

It's tempting to test everything with Behat. But it's important to counter-balance this with the fact that Behat tests are more difficult to maintain and slower to run than unit tests. Consider whether a unit test is more appropriate before adding a new Behat test.

4.2.4 Unit testing

We should almost always include unit tests on custom development for Drupal 8 projects.

Unit testing vanilla Drupal 7 code is not possible. But by making use of the XAutoload module, one can write object-oriented, unit-testable code.

We use PHPUnit for unit tests.

Consider using `phpspec` to design the code needed for a feature

The phpspec project assists developers in designing the specification for code. Code written using `phpspec` is going to be easier to unit test.

Use unit testing in conjunction with Behat tests

Using Behat to capture every permutation of a feature is difficult and costly to do. It's more efficient to use unit tests to handle testing the different inputs/outputs, and use Behat for a broader overview of the feature.

4.2.5 Manual testing

We should avoid manual testing as the primary means for verifying a feature's functionality, or the correctness of a bug fix. This is because the process is error prone, time consuming, and often tedious.

That aside, it's good practice in bug reports and pull requests to provide a summary of steps to test a feature or reproduce a bug. Manual verification can help developers and code reviewers confirm that no new, uncaught regressions have occurred. This is also a good opportunity for the code reviewer to step back and assess the feature or fix in the broader context of the project.

4.2.6 Not testing

Circumstances exist in which including tests in a feature or a bug fix is not possible. In these cases, the developer and project manager should confer with the CTO and/or the Principal Director.

4.3 Travis CI

We tie the above together using an automated testing tool called Travis CI.

Because we have standardized our development environments on a combination of Docker, AWS S3 for seed databases, and a Makefile for setup, configuring Travis CI is pretty straightforward, and does not differ in substance from local setup instructions.

All projects with tests should have those tests run on a per branch and/or per pull request basis.

4.4 Integration with project management

When creating issues in Redmine for completion of a feature/scenario, the best practice is to create a subtask for the test or tests required for that issue. That way, we can add more detail to the requirements for the test, as well as an estimate. Categorize the subtask as "Test" so that the issue queue is filterable by test related issues, and the total estimated/spent time on tests is visible.

CHAPTER 5

Redmine

Redmine is our project management tool at pm.savalabs.com. It is **the** authoritative location for project information. It is ideal to flow as much client communication through the tool as possible for the purposes of universally-shared documentation outside of personal mailboxes.

This document won't attempt to explain all of Redmine's features — if you haven't already done so, take a pause now and read through the Redmine User Guide which will answer many of your questions.

This document is focused on the Redmine workflow that we use in-house.

5.1 Configuring a project

5.1.1 Project organization

All projects should be organized with a project/sub-project hierarchy. For example:

Savas Labs (client/parent project)

- Savas Labs Maintenance (sub-project)
- Savas Labs Phase 1 (sub-project)
- Savas Labs Phase 2 (sub-project)

The top level project should be the client name, and *not* have an issue tracker. Example: `https://pm.savaslabs.com/projects/client`

Each sub-project should map to a specific scope of work with its own Harvest Project ID. For example, under the same top level client project, a "Phase 1" Redmine sub-project will be linked to it's specific "Phase 1" Harvest ID, while a "Maintenance" sub-project will be linked to a separate "Maintenance" Harvest ID.

Target Versions should be configured to be shared across sub-projects. This way software releases (e.g. 1.0.0) can track issues that are filed in "Maintenance" or "Phase 1."

The top level project should have a Wiki that contains all the information for the project and it's sub-projects. It's not easy to move a Wiki across projects, so existing projects that become sub-projects can retain their wiki pages. But as

of Feb 2017, all new sub-projects should *not* have the Wiki module enabled, and all Wiki content should live in the Top Level Project.

5.1.2 Set up tasks

- In the description field, document where important assets are like important git branches, dev/staging environments, etc
- Link Slack channel for updates to post to. If there is a lot of activity on the project, it's customary to create a `[project]-noise` channel to push to
- Set the Harvest project ID field for time-tracking purposes. You can configure multiple Harvest project IDs. Once a project ID has been set and time has been tracked to a project, please don't remove the project ID reference - just add a new project ID to the list.

5.2 The makings of the *perfect* task

Those who file detailed and informative issues deserve the greatest praise. A good issue will make life easier for developers, project managers, and clients. But good issues take a bit of work to create. Don't be tempted to simply add an issue title and move along. Take the extra couple of minutes to do the following:

- Be specific and detailed. A good issue is one that another developer can work on and complete without needing to ask the issue submitter any questions.
- Assign an estimate — even if it's a wild guess, it helps provide some context for the next developer who will look at the issue
- Assign a person
- Assign a due date and optionally a start date
- If applicable:
  - Include screenshots
  - Add a parent, sub, or related task (see *Linking issues*)
  - Set the milestone / target version (see *Target versions*)

Note that in almost all cases, clients are members of the project and can view and participate in discussion on issues. Even if the client isn't in Redmine, assume that everything you are writing might one day be seen by the client.

5.3 Linking issues

Redmine has helpful features for linking tasks. This is useful to developers and project managers in helping them understand how different issues are interrelated. You can specify helpful markers like `related to`, `blocks`, `blocked by`, `duplicates`, `duplicated by`.

5.4 Target versions

The `Roadmap` tab on the project page provides an interface to assign issues to a given "target version". The target version concept in Redmine is flexible: you can use it to plan for alpha/beta/final releases of a project, or to organize tasks into two week sprints.

5.5 Project documentation

We primarily use the wiki in each project for documentation, usually with sections for:

- Meeting notes
- Developer notes
- Environments
- Credentials

Add things like RFPs, proposals, mockups, and any other files to the `Documents` tab.

5.6 Keeping Redmine issues up to date with pull requests

After submitting a pull request, make sure to update the Redmine issue with a link to your pull request, and to set the issue assignee to the reviewer of your pull request.

In general, we prefer to do code review, including commentary on the pull request, in GitHub and not in Redmine.

CHAPTER 6

Time Entry

Accurate, informative and timely time entry is a crucial task for successful project management. (Puns always intended.)

Clients will review every time entry we record on their project, so it is essential to be thoughtful and professional in your language. Project managers will also delight in well-crafted, informative time entry messages.

6.1 Using Harvest

We use Harvest to record time.

Harvest is best used in real-time, as you switch from task to task. If you do this, at the end of the work day you have a (mostly) complete log of where you spent your time, and can spend a few minutes tidying up time entries and descriptions. In any case, **time entries for a given day must be entered by midnight of that same day**. This is important for project planning as well as for syncing entries with Redmine.

6.1.1 Tools

If you're on a Mac, you might find Harvest for Mac to be a useful tool. There are numerous other add-ons that the diligent time tracker might find helpful.

6.2 How to write the *perfect* time entry

A good time entry message is in some ways like a good commit message: done well, it tells a coherent story about what happened, when, why, and of course, the all-important "how long", which git doesn't do.

Some guidelines:

- Pay close attention to the *appropriate task* and log accordingly.
- Write an informative description. It may be concise but not empty.

  - Use professional language. Check your spelling.

- When applicable, reference the Redmine issue number in your Harvest time entry. We have a script that transfers time recorded in Harvest to time spent on issues in our project management tool which helps us monitor time estimated vs. time spent.

6.2.1 Anatomy of a good time entry

This references the issue, and outlines what specific aspects were worked on:

> **Issue #911**: Updates for scenarios where users register for online workshops on behalf of attendees other than themselves. This involves updating notification emails and reminder emails, adding a confirmation email, and modifying how online workshops are displayed on the user profile online workshop tab.

In order for our nightly sync from Harvest time into our PM system to work, **you must include in the time entry description the pattern of `#[issue-number]`**. That format is the hash symbol: # immediately followed by issue number with no space as shown above. If one includes multiple instances of this pattern, which one should avoid, the first one will be used for the sync.

This is much more informative than a time entry that simply reads "Issue #911" — while that provides some context, it then requires the project manager to look up the entry in Redmine, and still wouldn't tell them what specifically was worked on.

6.3 Tasks

Tasks fall into two primary categories:

- *Billable* tasks are those that are expected of us to invoice clients for based on our agreed upon scope of work.
- *Non-billable* tasks, naturally, we are not compensated for.

There are a set of "common" tasks that are added by default (*see exceptions*) to each client project and are the most commonly used. Note that "Admin" is the only non-billable task in the following list:

- **Admin** - *non-billable* catch-all for miscellaneous tasks; rarely used on client projects
- **Assessment/Estimating** - Estimating per a client's request.
- **Code Review** - PR review, other peer code analysis.
- **Design/UX** - Mockups, wireframes, UX tasks.
- **Development** - Writing code, configuring/building/writing software.
- **Project Management** - Client/internal calls/meetings, issue queue management, documentation, organizational tasks.
- **Research / Strategy / Consulting** - Researching a technical solution, advising a client, consulting on options and approaches. Discovery tasks.
- **Setup/Deployment** - Local dev environment, staging or production deployments.
- **Testing** - Writing automated tests, manual QA
- **Theme development** - Building templates, writing SASS/LESS/CSS.
- **Unestimated / Out of Scope** - Tasks that become part of a project that were not in the initial scope of work. These tasks can be driven clients with new requests or by developers taking a new approach. Generally speaking, if you are logging to this task, you should make sure your project manager is aware.

6.4 Internal projects and tasks

Internal projects help us track how we're investing our time internally, and although they have overlap in tasks with a typical client project, they have unique tasks attributed to them. Please ensure that you select the appropriate internal project for each internal task as there are multiple. As always, ask when you don't know.

- **Admin** - catch all, please be more specific when possible. Checking email, daily-standup.
- **Blog review** - Reading/reviewing providing feedback of others a teammate's blog.
- **Blog writing / research** - Blog writing and research for an article you're writing.
- **Business Development** - Reading a proposal, providing estimates for a proposal, conversing about a prospective project.
- **Meeting** - Internal calls or in-person meetings.
- **Mentoring** - Providing training to a teammate.
- **Skillshare** - Team skillshare, time researching / preparing for skillshare delivery.
- **Training** - Personal skill development.

6.4.1 Exceptions

Sometimes it will be necessary to tie projects to very specific tasks within a given scope of work to facilitate accurate reporting and ensure maintenance of budget. When this is the case, the project manager/lead will inform each team member and the appropriate tasks to select for the project will be clearly stated on the project wiki page in Redmine.

CHAPTER 7

Audits

When Savas Labs receives a work-in-progress project, we will review critical functionality, architecture, security, and code pre-development to ensure any risks are identified up front. Some of these items may be optional on certain projects but all are recommended when relevant.

7.1 Goals of site audits

- Identify risks so we can mitigate their impact
- Prepare ourselves for more accurate estimation
- Vet assumptions made by us and our clients
- Accurately quantify the amount of work remaining on the project
- Inform client of how much work previous developers have completed and any issues that have arisen
- Engender trust by showing clients we're competent, open, and transparent and guide them to a better, less risky, more informed place

7.2 Audit Checklist

7.2.1 General Security

- Run the Hacked module on the codebase to identify any patched code
- Run Security Review and Site Audit on site
- For Drupal 7 sites, run Drupalgeddon

7.2.2 Hosting

- Check server file permissions
- Find out who can access the server
- Find out how is code deployed
- Check file and directory permissions
- Check for server software patches/updates

7.2.3 Codebase

General

- Core and contrib modules are up to date with latest stable release
- Patches, if any, are documented and in a `/patches` directory
- Contrib/custom code is split into (sites/all)/modules/contrib and (sites/all)/modules/custom
- Credentials are not stored in version control
- Modules and themes live in proper place in Drupal

Custom code

- Format code for Drupal coding standards
- Run code through phpcs
- Ensure each custom module has a README explaining what it does
- Theme: review templates to ensure output is sanitized

Automated tests

- Review existing tests, if any
- If there are not any tests, ask the client to explain what the most important tasks are on the site

7.2.4 Database

- PHP error settings - Errors go to logs only, not to screen
- Permissions/roles (Security Review module will also help identify issues here)
- Use Schema module to identify any mismatched schemas, as well as custom tables that are not part of any module schema

7.2.5 Functionality of critical site features

Examples:

- eCommerce
- Mail setup

7.2.6 Useful Tests

- LinkChecker
- WAVE
- Mobile friendly test

CHAPTER 8

Securely storing credentials in 1Password

We use 1Password for Teams for securely storing and accessing credentials.

Our team's account is located at `https://savaslabs.1password.com/`.

8.1 Usage

1. Download the applications for your phone and computer.
2. Sign-in to the Savas Labs vault. You can use 1Password with your personal account, if you have one.
3. Place credentials in client-specific vaults. You may need to create a new Vault if it doesn't exist; ask if you have questions. Use tags as appropriate so it's easier for teammates to find credentials.
   (a) *Private*. This is your personal Savas Labs vault. You can store your personal account logins for websites (e.g. for `pm.savaslabs.com`, for client sites, etc)
   (b) *Savas Labs*. This is for shared credentials that don't apply to a single client, for example, Pingdom. (Note that some of our intrastructure credentials live in an Infrastructure vault that only admins have access to.)
   (c) *{Client name}*. Shared client credentials like server, API keys, etc. Your personal login credentials for the client go in your *private* vault, not here.
4. *Do not store sensitive information in Redmine*. It is okay to store things like development site URLs or non-critical information in the Redmine wiki. Use 1Password for server credentials, API keys, etc.
5. Update READMEs as needed so that teammates know where in 1Password they can find credentials.
6. If needed, we can add a client as a guest to a specific vault which eliminates the need for sharing credentials with a client via e-mail.

CHAPTER 9

Contributing to this repository

9.1 Building the project locally

9.1.1 Installing the requirements

Install the `sphinx` and `sphinx-autobuild` python utilities:

```
pip install sphinx sphinx-autobuild sphinx_rtd_theme recommonmark --user
```

Ensure that the `sphinx-build` and `sphinx-autobuild` commands are accessible to your terminal.

On Mac OS X, add the following entry to your `~/.bashrc`. Be sure to use the `$HOME` variable and not `~` as this may cause issues running `sphinx-build`:

```
export PATH=$HOME/Library/Python/2.7/bin:$PATH
```

On Linux, this would probably be:

```
export PATH=$HOME/.local/bin:$PATH
```

You may need to run `source ~/.bashrc` to ensure the access to the commands. You can add `source ~/.bashrc` to your `~/.bash_profile` (or possibly `~/.profile`) to automate this step.

9.1.2 Building the docs

You have two options for viewing the docs locally. You can build the docs using `make html` and then open the `_build/html/index.html` file in your browser.

Alternatively, run `make livehtml` and navigate to `http://localhost:8000` to view the docs. The HTML is updated as you edit the Markdown source files — no browser reload required!

9.2 Submitting your changes

To contribute to the docs at `developers.savaslabs.com`:

1. Fork this repository
2. Make your changes a. Edit an existing file b. If creating a new file, add a line entry to `index.rst` that corresponds to the new filename that you added
3. Run the tests with `./run-tests.sh`
4. Submit a pull request for the team to review