Squarespace Code Review Checklist

Courtesy of John Turner

Passes to complete every time

Sizing up. This is a preparatory pass intended to ease you into the right headspace for reviewing code, and to start setting expectations.

- ❑ **What is the general shape of this PR?** Is this a new feature? A bug fix? A refactor? A one-line change?
- ❑ **Is the PR the right size?** If someone has written a PR that's too large to review, this is the time to ask them to break it up.

Context. The goal of the context pass is to establish a rich mental model for the PR. Hopefully most of these questions will be answered in the PR description.

- ❑ **What is the change being made?** If you don't understand what the changeset is trying to do as part of the larger picture, it's nearly impossible to review it well.
- ❑ **Why is the change being made?** Asking why will allow you to gain broader context for your mental model.
- ❑ **At a glance, does the PR accomplish the intended change?** If you can spot something glaring or incomplete at first glance, comment on that first.

Relevance. The goal of the relevance pass is to connect your mental model of the changeset with its broader context in the engineering organization.

- ❑ **Is the change made by this PR necessary?** If you have deep knowledge in a particular subject area, you might be able to point out where changes are made that are unnecessary to accomplish the stated goal. All code has to be maintained, which can be costly. Sometimes the most effective thing that code review can accomplish is to prevent unnecessary code from being written.
- ❑ **Does this PR duplicate existing functionality?** If you see someone trying to duplicate existing functionality, point them to the pre-existing solution. If that solution doesn't do *exactly* what they need, see if there's a way to contribute to the pre-existing solution instead of rolling one's own.
- ❑ **Are there others that should be aware of this PR?** Since code review tends to happen in the context of teams, it's not uncommon for PRs that are important to other teams to go unnoticed. Take a second and tag your point of contact on other teams that might need to know about this PR.

Passes for more detailed review

Not all of these will be relevant in every case. Pick the ones that are.

Readability. The goal of the readability pass is to make sure that the person who reads the code in six months will be able to quickly build a coherent mental model of the code.

- ❑ **Is the change reasonably understandable by humans with little or no prior experience in the code base?** Pretend you know the language, but not the code base. Would everything read easily to you? If not, why?
- ❑ **Are any esoteric language features being used?** Esoteric language features, while occasionally useful, often hurt readability, even among language experts. If you see esoteric language features being used, ask if a simpler construct would work. If not, make sure that the feature is commented or otherwise documented to decrease cognitive overhead.

Production Readiness. The production readiness pass gives us assurance that the code is secure, documented, and reliable enough to be put into production without fear.

- ❑ **How will we know when this code breaks?** Normally the answer to this question is something like "If there is an unhandled exception, an alert will be fired that contains a stack trace." That's a perfectly reasonable answer. However it's answered, you need a way of knowing when the code breaks. If you don't need to know when it breaks, you can probably delete that code.
- ❑ **Is there new documentation required by this change?** Often times documentation becomes stale because it does not evolve along with the code that it documents. Take a second and ask whether new documentation needs to be created or if existing documentation needs to be updated.
- ❑ **Are there tests that prevent regression?** Most changes should come with a method for automatically verifying changes. If there aren't tests, there should be an explanation as to why.
- ❑ **Is this change secure?** Security is hard, and not everyone can be a security expert. However, most experienced programmers should know at least a few patterns of application programming that make your code less safe. If you write database code, you should know what a SQL injection vulnerability looks like. If you write frontend code, you should know what an XSS vulnerability looks like. If you're ever unsure, or if you're making potentially risky changes, tag someone from your company's security team on the review.

Naming. The goal of the naming pass is to make sure that the names of things reflect what they do.

- ❑ **Do the names communicate what things do?** Variable and function names should always strive to communicate what they do in a way that's unambiguous. A common

failure mode here is when a function communicates *most* of what it does, but leaves out an important detail.

- ❑ **Are the names of things idiomatic to the language that they're written in?** Every language has idioms for how variable and function names should be written. Violations of these rules cause cognitive overhead by defying the expectations of people reading the code. Worse, in languages like Go where case determines visibility, these violations can alter the design of the code in ways that weren't intended.
- ❑ **Do names leak implementation details?** Names should seek to communicate what they do, not how they do it. GetIPs is a relatively succinct and, depending on the context, decent name for a function. GetIPsByReadingFromAFile gives away way too much. A subtle version of this is when variable names contain their type, for instance "ipArray" instead of "ips." In statically typed languages, this amounts to line noise, as you're simply repeating the already-declared type in the name. In dynamically typed languages, you're pushing against the flexibility of dynamic typing by trying to add a layer of "pseudo" static typing.

Gotchas. The goal of the gotchas pass is to check for common programming oversights. Experienced programmers will have their own list, but I've included some of my favorites here.

- ❑ **What are the ways in which added or changed code can break?** A common way to get started with this question in dynamic languages is to look at variables and ask if they can be the language's version of "null."
- ❑ **Is this code subject to any common programming gotchas?** This includes things like off-by-one errors, transposition errors, memory leaks, null dereferences, etc.
- ❑ **Is spelling correct and consistent?** Incorrect or inconsistent spelling makes searching code much harder. One thing to check here is that misspelled names haven't been propagated by autocomplete.
- ❑ **Are there any dangerous defaults being set?** Defaults in code are a really convenient way to save the user of the code from unnecessary typing, but defaults need to be checked to make sure that they won't blow up unexpectedly when used.

Language-specific. The goal of the language-specific pass is to ensure that the code is up to the standards set by the community of experts in that language by your organization.

- ❑ **Is the code idiomatic to the language?** Non-idiomatic code increases cognitive overhead to read.
- ❑ **Are new patterns introduced?** Especially when a language is first being used at an organization, new patterns—reusable solutions to general problems—will be introduced. It's worth considering these new patterns, because they're bound to be copied by the next person who needs to solve that problem. If a new pattern is introduced that there's already a prescribed pattern for, call it out.
- ❑ **Does the code fall into common pitfalls for the language?** Every language has some common pitfalls that beginners to the language fall into. Examples of common

pitfalls are writing deeply nested list comprehensions in Python, or trying to write one language as though it were another.