MozMeao Consultant Guidelines

Release 1.0

Mozilla

Mar 15, 2018

Contents

These guidelines are meant to ensure consultants working for *Mozilla Marketing Engineering & Operations* (MozMeao) have clear instructions to follow when delivering code. Whenever possible these guidelines should be integrated into contracts and statements of work.

Contents:

CHAPTER 1

Getting Started

Prior to entering into a contract with a consulting company, please read through and familiarize yourself with this documentation.

There are 3 relationships MozMeao can have with consultants.

1. Fully Outsourced
2. Fully Outsourced With Handover
3. Partially Outsourced

It is important to define the type of relationship before creating a contract and SOW (Statement of Work). The type of relationship informs what the constultants and Mozilla are responsible for.

Fully Outsourced

1. All code written by consultant.
2. Site not hosted on Mozilla resources.
3. No expectations of MozMeao resources providing maintenance or feature devleopment.

Fully Outsourced With Handover

1. All code written by consultant.
2. Site may be hosted on Mozilla resources.
3. Expectations of MozMeao resources providing maintenance or feature devleopment post launch.

Partially Outsourced

1. Non production ready code is written by consultant.
2. Site may be hosted on Mozilla resources.
3. MozMeao resources finish development and are responsible for feature development post launch.

CHAPTER 2

Questions For RFPs

It is a challenge to vet agencies during the RFP (Request for Proposal) stage. You may be including the development of the 'what' in the RFP so it's entirely possible that very few technical details will be known at the RFP stage.

However there are a few questions you can include in the RFP to tease out potential risks. Any risks should be mitigated by addressing them specifically in the SOW/Contract.

Questions To Include In An RFP

- Have they launched projects similar to what you are asking for? If so, include examples and include links to source code if available.
- Do they work on projects similar in scale to what you are asking for? If so, include examples and include links to source code if available?
- What will be the composition of the team working on this project? What are each person's expertise and experience?
- Do they have engineers on staff who will be responsible for this work? If sub-contracted, who are the subcontractors?
- What is their preferred/proposed tech stack for this application?

CHAPTER 3

Communication

Establish and agree upon communication channels in your kickoff meeting.

- **Single point of contact:**
  - Establish a single point of contact at Mozilla responsible for communicating with the consultants for the duration of the project.
- **Define how the team will communicate including:**
  - Meeting cadence.
  - Vidyo, Slack, IRC, Email, or other appropriate communication methods.
  - Establish how progress on deliverables will be tracked and communicated.
  - Reporting & tracking bugs.
  - Establish documentation, what and where?

CHAPTER 4

Best Practices

Websites you design and build for Mozilla as a consultant should adhere to the same set of best practices used by Mozilla for its own websites. Here are some practices you should follow throughout a project:

4.1 Progressive Enhancement

By Default, Mozilla websites should practice progressive enhancement. Begin with simple, meaningful HTML. Add CSS and JavaScript as progressive layers in a way that still falls back to usable, accessible content in older browsers that don't support the latest features or in the event of JavaScript failing.

Don't rely exclusively on JavaScript to serve static content. This is less about catering to the few people who purposely disable JavaScript and more about ensuring some fault tolerance as a best practice. While single-page websites that render the entire screen using client-side JavaScript may have a place, the reality is that 99% of the time they don't need to be built in such a way. If a JavaScript error occurs or someone is using an older web browser, their experience should degrade gracefully. No one should be greeted by an empty, white screen.

4.2 Feature detection

Going hand-in-hand with progressive enhancement, Mozilla websites should be built using feature detection. Browser sniffing via the user agent string is generally considered a bad practise when it comes to determining website compatibility. Sites that rely on UA sniffing in order to deliver or block content to certain browsers do not meet our quality standards. MDN has a great article on how best to implement feature detection on websites.

4.3 Performance

Mozilla websites should strive to load quickly and be responsive on all types of devices, whether it be on desktop, tablet or mobile. Favor server-rendered content over client-rendered content. Develop a performance budget for your pages during the design phase, and keep a close eye on areas such as page weight, and the number of images, fonts, and

CSS/JS assets that you use. Use services such as Web Page Test or Lighthouse to monitor your website performance during development.

A few general guidelines:

- On a 3G connection, aim for document complete in under 4 seconds, and for the page to be fully loaded in under 6 seconds.
- Keep the number of custom web fonts (including bold and italic variants) to a minimum. 3 web fonts in total (including bold/italic variants), is a good target ceiling. Mozilla websites typically use a combination of Open Sans and Zilla Slab.
- Avoid large CSS frameworks such as Bootstrap or Foundation as they are by necessity generic and can be heavy-handed.
- Client side JavaScript frameworks should be used as an enhancement, not as the default backbone for delivering content. Large frameworks such as React or Angular should be used only when they are jointly identified as being the appropriate solution for a specific requirement.
- Images must be responsive through any combination of CSS, srcset, or SVG. Avoid particularly large images or a large number of images on a page (even if each one is relatively small). Production-ready images should also be optimized and/or compressed as appropriate.

A good way to avoid a poorly performing web page is to set a performance budget. An optimal goal is to deliver pages at or under a total of 600KB. A reasonable breakdown for potential assets would be:

- Fonts - 200KB
- Images - 200KB
- JS - 100KB
- CSS - 20KB

Before you enter the prototyping phase of your project, agree a performance budget for your pages and then monitor throughout the development stage.

4.4 Accessibility

Strive to make all Mozilla websites reasonably accessible to people with disabilities and those using assistive technologies. In many cases this isn't a matter of adding accessibility features into a site, but simply *not* adding obstacles that make the site harder to use. Accessible sites are better for everyone, not only those with disabilities.

A few guidelines:

- Include descriptive `alt` text for any meaningful images in HTML.
- Include `<label>` elements in forms. Don't use `placeholder` as a label.
- Navigation, forms, and interactive widgets like dropdown menus and modal windows should be keyboard accessible.
- Ensure sufficient color contrast so text is readable for most users, including those with color vision deficiencies.
- Text should be resizable for those who need larger type. Make reasonable allowances for text to wrap and reflow when its size changes.
- Include ARIA attributes in HTML where appropriate.

4.5 Localization

Many Mozilla websites are localized in a large number of locales and different languages. If your project has localization (l10n) considerations then there are a few guidelines to follow:

- One of the biggest design challenges is ensuring text is both live and expandable. Text included in images or videos is prohibitively time consuming (if not impossible) to localize. The amount of actual text in a given message can vary greatly across languages, meaning the height and width of the surrounding elements must be able to adapt. Precise typography – such as controlled alignment, fitting text in an unusual shape, or specifying where lines should break when text wraps – is often unrealistic. Expect the shape and size of text blocks to change.
- Choose web fonts with broad character sets. Many display fonts only support Roman alphabets and don't have all the glyphs to correctly render Russian, Swedish, or Polish (for example). Some languages have entirely different alphabets and no one font can serve them all, so expect some languages to be rendered in an entirely different font. Also avoid over-reliance on ALL CAPS or italics, as some scripts don't lend themselves to these treatments.
- Be considerate about how pages may be read in right-to-left languages, specifically Arabic and Hebrew. Those languages flip the direction of text but also tend to reverse the direction of other UI elements, where objects on the right are seen "first."
- Avoid symbolism or imagery that might be strongly identified with only one country or nationality (e.g. a bald eagle representing the concept of freedom). People in other countries may not understand such symbols, and at worst they may find them offensive.

CHAPTER 5

Browser Support

Mozilla websites should be tested in a broad range of browsers, though that doesn't mean a site must look or behave exactly the same in all browsers. Sites should look good and work well in any modern browser, even if it's not perfectly identical.

The current generation of browsers (versions released within the last few years) are all pretty even in their support of most modern web standards, but many people still use older browsers and deserve equal access to information. Consider following a system of graded browser support wherein the oldest or least capable browsers can still access all the content and essential functionality, just without all the bells and whistles better browsers can enjoy. If you use a progressive enhancement methodology, support for older/other browsers may already be a given.

Unless there are specific reasons not to, you should look to fully support (at the very least) the latest versions of all major browsers (Firefox, Chrome, Safari, Opera & Edge), as well as the last 2 versions of Internet Explorer (10 & 11). All other browsers can have degraded support.

Don't use proprietary features supported only by a single browser unless what you're making is a demo of that specific feature. Be mindful when you use emerging standards supported in some browsers but not others and include fallbacks for less capable browsers. Check MDN or CanIUse for compatibility info. If you use vendor prefixes in CSS, include the prefixes for all supporting browsers as well as the unprefixed standard.

Finally, don't rely on user agent sniffing to determine which browsers can and can't access a website or use its features. Instead, favor feature detection that allows older browsers to degrade gracefully in what they show.

CHAPTER 6

Coding Style

Code you deliver to Mozilla as a consultant should be written according to the same standards as code written by any other Mozilla developer. Plan for your code to be maintained by other coders and comment liberally.

Standardizing a common style for writing and formatting code helps a team or organization maintain a sensible code base. The goal is consistency and predictability so that any coder can view the work of another coder and quickly follow along. We have documented style guides elsewhere that go into greater depth, but we'll also summarize the main points here for some common languages.

Important: We urge you to read through the Mozilla Webdev Bootcamp docs for more detailed guidelines.

6.1 General

- Use spaces for indentation. Never use tabs – history has shown that we cannot handle them.
- Use four spaces to indent most languages, but two spaces in HTML. HTML lends itself to a lot of nested elements and indenting each level four spaces can lead to long lines and messy formatting.
- Wrap lines at 80 characters when possible. HTML is often the exception here as well.
- Eliminate trailing whitespace at the end of lines. Blank lines should have no spaces.
- Include a single blank line at the end of files.

Note: Using EditorConfig in your project can help enforce most of the above rules automatically. You can see an example `.editorconfig` file here.

6.2 Python

- Follow PEP8.
- Follow Pocoo's extensions to PEP8, although these are a little less strictly enforced across Mozilla projects.
- Check your code against a linting tool. We highly recommend Flake8 for this.
- Use single quotes unless double (or triple) quotes would be an improvement.
- To continue a new line use a `()` not `\`.
- Indent code with either a hanging indent or a 4 space indent on the next line.

6.3 HTML

- Use HTML5, with the appropriate doctype: `<!DOCTYPE html>`
- Omit XML-style trailing slashes: `<br />`
- Use lowercase for tags and attributes.
- Use double quotes for attribute values.
- Validate your markup.
- Avoid inline or embedded CSS or JavaScript.
- Indent nested elements two spaces; don't use tabs.
- Use the most semantically valuable element suited to the content.

6.4 CSS

- Use a linting tool such as Stylelint.
- Multi-line rules, not single line.
- Four space indentation; don't use tabs.
- Order declarations alphabetically (with some exceptions).
- Use the simplest, least specific selector possible.
- Make meaningful names, not presentational.
- All lowercase for classes and IDs, no camelCase.
- ID selectors are allowed but use them sparingly and appropriately.
- Don't use `!important`.
- Use unitless `line-height`.
- Group related rules into sections.
- Order sections and rules from general to specific.
- Validate!

6.5 JavaScript

- Use a linting tool such as ESLint.
- Assign each variable on a newline, not comma-separated.
- Use `[]` to assign a new array, not `new Array()`.
- Use `{}` for new objects, as well.
- Always use semicolons.
- Always use `===`.
- Always use single quotes.
  - Exception: `"Don't escape single quotes in strings. Use double quotes."`
- Cache regex into a constant.
- Try to avoid ternaries.
- Check for truthiness, not lack of falseness.

CHAPTER 7

Testing

All code provided by a third party that is intended to be run in a production environment should be adequately tested. Testing is generally split between two distinct styles:

1. Unit tests can be set to run automatically against pull requests in GitHub using CI.
2. Functional tests can be set to run automatically on deployment and run in multiple browsers.

7.1 Unit Testing

- Backend code should be unit tested. For Python we suggest using pytest.
- JavaScript code should also be unit tested. When run in the browser, we

suggest using Karma and Jasmine. For Node, either Mocha or Jest are good options.

7.2 Functional Testing

- For cross-browser functional testing we suggest using Selenium

with pytest-selenium.

CHAPTER 8

Quality Assurance

Who is doing QA and how should be determined prior to a signing a contract or SOW.

Generally there are 2 approaches for QA used with consultants, the first being the most common:

1. Combined QA, Mozilla + Consultant
2. Consultant QA Only

8.1 Combined QA

Consultants are expected to deliver working code, however in many cases they do not have the same standards for QA or expertise to fully deliver a project. In most circumstances we will want to have MozMeao provide additional QA reviews and testing.

In this scenario you should do the following.

- **Request a MozMeao QA Resource:**
  - They should review the contract and SOW and determine their level of involvement.
  - They should develop a QA plan that is reviewed and agreed upon by Mozilla and the consultant.

8.2 Consultant Only QA

If the consulting firm is large enough and the project is fully outsourced they may handle the entirety of the QA process. This must be defined in the SOW along with any project specific areas of concern. These areas will be specific to the project but some areas to think about are listed below.

- Browser support.

- Mobile support.
- Accessibility support.
- Verifying proper analytics behavior.
- Testing integration with any 3rd party services.
- Load testing.

Even in circumstances where the consulting firm is handling the entirety of the QA, a method for Mozilla submitting bugs should still be established at Kickoff.

CHAPTER 9

Design Assets

Design assets you deliver to Mozilla as a consultant should be provided in an agreed format and should be ready for a developer to turn into production ready code.

Things to consider include:

- The ideal design mockup should be an HTML prototype that can be viewed and resized in a web browser.
- If an HTML prototype is not provided, either a Sketch of Photoshop static mockup is acceptable.
- If a static mockup is provided, both a desktop and mobile representation of a page should be included.
- If image assets intersect with the edge of an artboard in a static mockup, make sure to include those assets uncropped in the document, so a developer can export and use them in context of a responsive web page that can be larger than the fixed sized artboard.
- Image assets should be included in an appropriate format. We prefer SVG whenever possible, but if an image includes many paths or gradients then sometimes a PNG or JPG makes more sense. If non-vector images are provided, high resolution versions should be included.

CHAPTER 10

Hosting

There are 2 common scenarios for hosting a project developed by consultants.

1. Consultant Hosted
2. Mozilla Hosted

10.1 Consultant Hosted

Consultant hosted means the 3rd party is taking responsibility for hosting the website. The project is not hosted on Mozilla hardware and Mozilla is not responsible for either performance or uptime.

The following responsibilities should be outlined in a SOW:

Consultant Responsible

- **Billing, either direct or pass through.**
  - Include duration of hosting.
  - Provide estimates of all costs (hosting/bandwidth/other) including any possible overages.
- **Ensuring hosting is performant.**
  - Ensure project can handle expected traffic.
  - Ensure uptime for duration of project.

Mozilla Responsible

- **Domain Configuration:**
  - Registration of any domains.
  - DNS & SSL for any domains.

10.2 Mozilla Hosted

Mozilla hosted means the project is hosted on Mozilla infrastructure. Mozilla is responsible for providing performant hosting.

The following responsibilities should be outlined in a SOW:

Consultant Responsible

- **Delivering Code:**
  - Deploying and configuring code that works properly on Mozilla's Infra.
  - Ensuring project works as intended on Mozilla Infra.

Mozilla Responsible

- **Providing Infrastructure & Documenting Environments & Deployment:**
  - Registration of any domains.
  - DNS & SSL for any domains.
  - Providing (if needed) dev, stage and production hosting.
  - Documentation on how code will be deployed to each environment and by whom.
  - Any specific implementation details specific to Mozilla infra.

CHAPTER 11

Version Control

All code provided by a third party must reside in version control and be made available to Mozilla.

The following details should be decided upon and outlined in the SOW.

- **Using Version Control:**
  - Project code must be in version control.
  - Code should reside on Github unless another platform has been specifically agreed to.
  - Github usage and workflow should follow WebDev guidelines .
  - Private repositories are allowed but must be mutually agreed upon.
  - Access to repository should be granted to Mozilla at kickoff.
  - At project completion Mozilla must have full admin to the repository or the repository must be transferred to Mozilla.

CHAPTER 12

Licensing

- All code provided in a project should be licensed under the MPL version 2.0.
- If a project contains any third-party libraries or frameworks, that third-party code should also contain an open source license that is compatible with the MPL version 2.0.
- If a project requires any proprietary code or licensing, those details should be discussed early, before development begins.