Standalone JavaScript Packages

Abstract

This document describes new practices for the Drupal community to create, use, distribute and maintain standalone JavaScript packages.

How to read and contribute to this document

This document is intended to be a dynamic, consensus building tool; not a rigid proposal to be accepted in whole or in part. Please fix typos and grammatical errors when you encounter them.

To add or identify a missing recommendation, either:

- Add a new recommendation.
- State the problem that needs to be addressed and prefix that statement with the “TODO” key word.

To clarify a section or recommendation, either:

- Ask for clarity in a comment.
- State the ambiguity and prefix that statement with the “QUESTION” key word.

To refine or remove a specific recommendation:

1. state your concern in a comment,
2. consider what the purpose of the recommendation was;
3. propose an alternative which balances your concern with the intent of the original recommendation and prefix it with the “ALTERNATIVE” key word.

Notational Conventions

Statements which do not use the key words below are merely setting context or providing explanation. Therefore, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document indicate actionable recommendations. These key words are to be interpreted as described in RFC 2119.

The key words “TODO”, “QUESTION”, “ALTERNATIVE”, and “IGNORE” in this document indicate that a section or topic is incomplete.

Table of contents

1. Introduction

New JavaScript language features and tooling have made distributing and maintaining small, reusable, single-purpose JavaScript packages easier than ever before. However, Drupal's current tools and practices do not support the distribution of these JavaScript packages according to community best practices.

This document recommends a set of practice and tool changes that will enable the Drupal community to distribute and maintain standalone JavaScript packages as "core projects" (in contrast to "contrib projects").

By adopting these recommendations, Drupal will be able to provide JavaScript packages that are easy for non-Drupal developers to adopt while maintaining the high standards of quality associated with Drupal core.

A standalone package refers to a JavaScript package which does not include the CMS project in its list of dependencies. For example, a replacement for jQuery.once would not require any code from the CMS codebase. Additionally, a JavaScript package which requires a Drupal CMS backend would not require the CMS codebase either, since it merely relies on a specific JSON response format, but does not require any CMS project code.

1.1. Naming Conventions

JavaScript package - For the purposes of this document and clarity, all words pertaining to a single purpose and independent JavaScript repository will be called "JavaScript package". Other common phrases for JavaScript packages include

- JS libraries
- JavaScript modules / JS modules
- libraries

2. Origins

Origins refer to the servers which host the elements that constitute a Drupal project. There are three origin types: artifact, issue, and code.

For example, https://packagist.org is an artifact origin, https://www.drupal.org is an issue origin, and

https://git.drupal.org is a code origin. Collectively, they serve all necessary elements of a functional Drupal project.

JavaScript package origins MUST be controlled by the Drupal project lead or appointees of the project lead (see Appendix E). They include: drupal.org, github.com/drupal, gitlab.com/drupal, and the "@drupal" namespace on NPM.

Using an issue or code origin other than drupal.org is an ongoing community discussion; it is therefore beyond the scope of this document to recommend another origin for issues or code.

2.1. Artifact origin

An artifact origin refers to the origin server which distributes JavaScript files that are intended to be used as a dependency in other JavaScript packages. An artifact usually excludes dev dependencies and may contain generated code. The Drupal project does not currently distribute standalone JavaScript artifacts.

The wider JavaScript community uses https://www.npmjs.com ("NPM") as its artifact origin. This artifact origin is the default when a user runs "npm require {project}". Other origins require custom configuration.

Release artifacts MUST be distributed via NPM under the "@drupal" namespace.

NPM package maintainers SHOULD be the Drupal Security Team, per Section 4.

TODO: validate the statement above once we have contacted the NPM Security Team, perhaps there is a better way to ensure the Drupal Security Team gets notified rather than making them maintainers of all projects on NPM.

Since core maintainers (as defined in Section 4.2.2) might not be Security Team members, Section 3.3 requires an automated process for publishing a release artifact to NPM without directly authenticating with NPM (e.g. via a CI pipeline).

2.2. Issue origin

An issue origin refers to the origin server which tracks bug reports, feature requests, and other discussions about a project.

At this time, a JavaScript package SHOULD use https://www.drupal.org for its issue origin. Additionally, it is RECOMMENDED that each JavaScript package maintain a unique project page on Drupal.org. Drupal core JavaScript packages SHOULD have a dedicated project type on Drupal.org.

TODO: create an issue for creating the new project type on Drupal.org, then link to it.

2.3. Code origin

A code origin refers to the origin server which hosts a version control repository.

At this time, a JavaScript package SHOULD use https://www.drupal.org for its code origin. Additionally, it is RECOMMENDED that each JavaScript Package maintain a unique project page on Drupal.org with a unique version control repository.

3. Change controls

Change controls are the procedures and policies which are intended to lower the burden of ongoing maintenance, to make new features easier to incorporate through flexible design, and to increase the stability of Drupal's software.

There are three kinds of change controls: requirements, workflow, and release management.

3.1. Workflow

A workflow control is a procedure for incorporating new code into a codebase. The Drupal project currently uses both a patch-based and a merge request-based workflow.

A JavaScript package SHOULD use the merge request workflow if it uses https://git.drupal.org for its code origin.

A change SHOULD NOT be accepted unless it has been reviewed by at least two parties other than its proposer.

3.2. Requirements

Requirement controls are a set of documented criteria that a JavaScript package must meet. Requirements can apply to code changes, issue summaries, issue metadata, or the JavaScript package itself. For example, a code requirement may stipulate that the

change code follows a particular style guide; an issue summary requirement might be that it contains a text snippet to be used in future release notes.

3.2.1. Package requirements

A JavaScript package MUST use the ECMAScript module format in its source code.

A JavaScript package MUST distribute artifacts that are compatible with the latest stable LTS version of node.js, or all browsers that are required by the Drupal CMS project. Therefore, packages which are not meant to be imported by other modules SHOULD fulfill this requirement by transpiling[1] a compatible release artifact.

A JavaScript package MUST use the latest stable version of Yarn 2 for dependency management and script running when working on it's source code.

A JavaScript package MUST use ESLint for its source code and it MUST be configured to use the "eslint:recommended" default. If a package uses TypeScript, it MUST use "plugin:@typescript-eslint/recommended"

A JavaScript package MUST use Prettier for its source code and it MUST be configured to use the "plugin:prettier/recommended" default. If a package uses TypeScript, it MUST use "prettier/@typescript-eslint" configuration.

A JavaScript package MUST use the following Prettier configuration:

```
{
  "printWidth": 120,
  "semi": true,
  "singleQuote": true,
  "tabWidth": 2,
  "trailingComma": "all"
}
```

A JavaScript package MUST use the following order for ESLint extended configurations:

```
{
  "extends": [
```

[1] **Transpiling** is a specific term for taking source code written in one language and transforming into another language that has a similar level of abstraction.

```
      "eslint:recommended",
      "plugin:@typescript-eslint/recommended",
      "prettier/@typescript-eslint",
      "plugin:prettier/recommended"
    ]
  }
```

A JavaScript package MUST include a "CODE_OF_CONDUCT.md" file as described in Section 6.

A JavaScript package MUST include a "SECURITY.md" file as described in Section 5.

A JavaScript package MUST have automated test coverage.

A JavaScript package that has a dependency on another core project MUST have integration tests for that project. For example, if a JavaScript package relies on an HTTP API provided by the CMS project, it should have a test for that usage.

A JavaScript package MUST define the following script names in its "package.json" file:

- **build**: create an artifact suitable for release.
- **test**: runs all tests.
- **lint**: checks coding standards.

A JavaScript package SHOULD define the following script names in its "package.json" file:

- **docs**: generates Markdown or HTML documentation files suitable for publishing online. These files SHOULD be created within a child directory of the project root named "docs".
- **start**: sets up everything needed (file watcher, server, etc.) to be able to easily develop the JavaScript package.

The scripts specified above MUST have a non-zero exit status if they fail; zero if they succeed.

3.2.2. Change requirements

Changes are considered to be accepted when they are committed to a package's change history (see Section 8.1).

A change MUST NOT be accepted unless it updates all of the in-line documentation which it invalidates.

A change MUST NOT be accepted before the online documentation which it invalidates is updated appropriately.

A change MUST NOT be accepted until it has been successfully linted.

A change MUST pass all existing tests.

A change MUST NOT alter, rename, or remove any exported symbol unless that symbol was documented with the "@private" JSDOC tag when it was initially exported.

A behavior change MUST add a passing test that validates the change itself.

A change MUST NOT introduce a dependency without undergoing the dependency evaluation process.

A change MUST pass the Drupal Core Accessibility and Usability Gates if it affects the user interface of any core project.

A change MAY export new or previously un-exported symbols.

A change MAY alter string literals.

3.3. Release management

A release MUST NOT be published unless the package passes the "test" and "lint" scripts defined in Section 3.2.1.

Releases MUST use semantic versioning and follow Drupal's naming requirements for pre-release tags.

If there is a known security vulnerability, a security advisory MUST be released for all versions depended upon by other core projects as well as all major versions of a package which were released during the current major version cycle of the CMS project. For example, if a JavaScript package published two releases with two major versions during Drupal 9's active support window (e.g. 1.4.2 and later 2.0.0), this rule requires the issuance of a security advisory for both of those major versions.

Deprecated code MUST be removed in a major release.

Major releases SHOULD NOT include any additions or refactorings. In other words, a major release ought to be the same as the last release of the prior major version with the exception of the

removed, deprecated code as outlined in the Drupal core continuous upgrade path policy.

A major or minor release MUST have a scheduled end of active support at the time of release. The date MAY be extended if a newer release is not available. It is RECOMMENDED that releases observe the active support schedule and release windows of the CMS project.

If a package depends on the CMS project's HTTP API, a release MUST be available which supports each supported version of the CMS project, beginning with the latest supported minor of the CMS project at its initial release.

A package MUST have a release which supports the supported major and minor versions of every core project on which it depends[2]. For example, if the CMS project introduces a breaking change, its maintainers are expected to help release a new version of the other affected core projects and vice versa.

If a core project depends on the JavaScript package, a major release MUST NOT be published until an issue(s) (see Section 2.2: Issue origin) has been created to update the core project which depends on the package and that issue has a passing test. For example, an issue to increment the required version numbers in the CMS project's "package.json" and "yarn.lock" files ought to be created before a release is published.

If the CMS project depends on the JavaScript package, the core CMS SHOULD be updated to depend on the latest compatible patch and minor version of the package prior to the beta deadline for each CMS minor release.

An automated and remote process for packaging and publishing a release artifact MUST be established (see Section 2.1). This process SHOULD be started with a single, documented command. Authenticating with an NPM account MUST NOT be required. The command MAY prompt the user for input.

4. Roles and responsibilities

Certain individuals are appointed to roles that are responsible for performing privileged duties that ensure the quality and future of the Drupal project.

[2] Support for Drupal 7 is an exceptional case because of its extended end-of-life support; packages are not required to support Drupal 7.

It is RECOMMENDED that one or more new individuals be appointed as Drupal core committers (see Appendix E). Their role SHOULD be distinct from the existing front-end framework manager role because the responsibilities associated with maintaining JavaScript packages as defined by this document are different than those of a front-end framework manager. The nature of this new role SHOULD be defined collaboratively among the appointees, the BDFL, and the existing core committers.

5. Security

The Drupal project has procedures and policies in place for reporting security vulnerabilities and for fixing, announcing, and releasing mitigations for those vulnerabilities. Since JavaScript package artifacts will be published on NPM (see Section 2.1), some special considerations and procedures are required for a standalone core JavaScript package.

5.1. Security Advisory Process

A JavaScript package SHALL be covered by the Drupal core security advisory process, without regard for the origins used by said JavaScript package.

A JavaScript package MUST issue all security advisories during a Drupal core security release window (third Wednesday of the month).

QUESTION: should the statement about which versions are covered above be incorporated into the "SECURITY.md" file described below?

A JavaScript package MUST have a file named "SECURITY.md" in their root source code directory (see Section 2.3). This file MUST contain the following text:

Security Policy

> This project is covered by the Drupal security advisory process, which is facilitated by the Drupal Security Team.
>
> To report a security vulnerability, follow the [process for reporting a Drupal Security issue](https://www.drupal.org/docs/develop/issues/issue-procedures-and-etiquette/reporting-a-security-issue). You can [report a security vulnerability for PROJECT NAME here](https://security.drupal.org/node/add/project-issue/project_name).

Do not disclose the nature of the vulnerability in public. The Drupal Security Team will publish an advisory and may issue a CVE once a fix is available.

[Learn more about the Drupal Security Team](https://www.drupal.org/drupal-security-team).

5.2. Special considerations for NPM

NPM maintains a dedicated security team and robust reporting process. Their process states the NPM security team will qualify the report and then notify the registered NPM package maintainer(s).

To ensure that the Drupal security team is notified before the project maintainer (as is the current practice), Section 2.1 requires NPM package maintainers to be existing security team members. To minimize the additional burden of processing these notifications, a simple procedure for those maintainers is described below.

When an NPM maintainer receives a security notification email from the NPM security team:

- The maintainer SHOULD immediately forward the notification to security@drupal.org or create an issue on https://security.drupal.org, time permitting.
- The maintainer SHOULD immediately respond to the NPM security team with a pre-written response.

It is RECOMMENDED that this procedure, including the pre-written response, be documented as a common security team procedure.

Once a corresponding security issue has been created on https://security.drupal.org, the Drupal security team SHOULD contact the maintainer responsible for the JavaScript package via email.

Once the issue is resolved, the Drupal security team SHOULD coordinate with the NPM security team to publish a security advisory for the affected JavaScript package, and the corresponding release marked as a security release in NPM.

Since the NPM security team might be unused to collaborating with a vigorous open-source security team, it is RECOMMENDED that the Drupal security team establish a contact within the NPM security team.

TODO: update this section once Tim has contacted the NPM security team and done some research on how NPM organizations function.

6. Code of conduct

The Drupal Code of Conduct SHALL be in effect, without regard for the origin on which participation occurs.

A JavaScript package MUST have a file named "CODE_OF_CONDUCT.md" in their root source code directory (see Section 2.3). This file MUST contain the following text:

> Code of Conduct
>
> This project is governed by the [Drupal Code of Conduct](https://www.drupal.org/dcoc).
>
> A Conflict Resolution Policy and Incident Report Form can be found by following the link above.
>
> The Drupal Code of Conduct consists of six ideals, listed below.
>
> * Be considerate
> * Be respectful
> * Be collaborative
> * When we disagree, we consult others
> * When we are unsure, we ask for help
> * Step down considerately

7. Licensing

A JavaScript package MUST use the MIT license.

TODO: BDFL must explicitly approve this rule.

TODO: validate that this can be relicensed so that we preserve the right to change our minds.

8. Best practices

The following subsections describe a set of recommendations for the benefit JavaScript package maintainers. These recommendations ought to result in a JavaScript package which is easier to maintain over time.

8.1. Source control management

Code changes MUST be tracked with the Git source control management system.

Each JavaScript package SHOULD have a unique change history.

TODO: commit format rules
https://www.conventionalcommits.org/en/v1.0.0/

8.2. Design

Private symbols SHOULD NOT be exported. If a private symbol cannot remain internal, it MUST be documented with the "@private" JSDOC tag.

A JavaScript package SHOULD export named constants or dedicated extensions of the "Error" object to disambiguate errors thrown by the package. In other words, a JavaScript package SHOULD NOT use strings or codes for this purpose.

8.3. Testing

A JavaScript package SHOULD have tests defined for all exported functions.

A JavaScript package SHOULD run tests on a regularly scheduled basis. For example, tests could run once every 24 hours.

8.4. Dependencies

A JavaScript package SHOULD NOT require production dependencies. However, dependencies MAY be added on a case-by-case basis with the written approval of a Drupal core release manager.

A JavaScript package MAY require development dependencies. However, it is RECOMMENDED that these dependencies be minimized.

A JavaScript package MAY require peer dependencies if it is essential to its functionality. For example, a React component package ought to have a peer dependency on React itself.

QUESTION

Appendix A. Technical background

To ease the burden of writing and maintaining cross-functional JavaScript, the Drupal project adopted the jQuery and jQuery UI libraries. Since jQuery and jQuery UI's introduction, web browsers have made significant progress in standardizing their implementations. At the same time, JavaScript itself has evolved.

Today, jQuery and jQuery UI are obsolete. It is now possible to write maintainable and cross-functional JavaScript without the use of any JavaScript frameworks (i.e. with "vanilla" JavaScript).

ECMAScript 6 ("ES6"), the specification which defines the latest JavaScript language, now supports information hiding and encapsulation via modules. Those new features have changed how JavaScript is distributed and maintained. Small, reusable, single-purpose modules are the best, current practice.

jQuery's obsolescence and the new ability to define ECMAScript modules means that Drupal's existing JavaScript and its policies are no longer suitable. Drupal should adopt new, more modern procedures and policies that share the norms of the web development community at large.

Appendix B. Current workflows

The Drupal project currently uses issue pages on https://www.drupal.org to enforce a common workflow. There are two formats for sharing code changes: patch files and merge requests.

Appendix B.1. Patch workflow

To begin the process of changing code, a contributor creates a new issue. The issue is typically categorized as a "bug report", "task", or "feature request" and its state is set as "Active". After the issue is discussed, a file is uploaded to the issue that contains changes that will resolve the issue. This file is referred to as a patch. The patch may or not initially contain a complete set of changes and multiple patches may be uploaded.

When the uploader determines that the patch is complete, the uploader changes the issue status to "Needs review". At this point, another contributor must review the issue and the patch. The reviewer will ensure that the patch meets the requirements described in Appendix C.

If a patch does not meet requirements, the reviewer will set the issue status to "Needs work" and describe the changes that should be made to the patch. The reviewer may also leave the issue status unchanged but request that changes be made to the issue summary or issue metadata. For example, the issue summary may not completely describe the requested change or the issue might be inappropriately tagged.

When the issue and patch meet the requirements described above, a reviewer (often a subsystem maintainer) will set the issue status to "Reviewed & tested by the community". At this point, a core committer will review the issue for a final time. If the issue and patch meet requirements, the committer will download the patch file, apply it to the Drupal codebase, and commit the change. In most cases, a committer will set the issue status to "Fixed" (if the change needs to be committed to a different branch, the issue status may be different).

Appendix B.2. Merge request workflow

In terms of issue statuses and requirements, the merge request workflow follows the same procedures as the patch workflow described above. In the merge request workflow, patch files are not uploaded to the issue. Instead, a new version control repository is created when work begins. Each issue has a dedicated upstream repository. This repository is known as an issue fork.

A developer (usually the issue creator) will download the issue fork ("clone") and commit their changes to it. They can perform this work with their local command line tools, or entirely within the interface at https://git.drupalcode.org. When their changes are complete, the developer will push those commits to the issue fork. When the developer determines that they have no further changes to make, the same "Needs review" and "Needs work" issue status procedures are followed as in the patch workflow described above.

When a reviewer or committer requests a code change, the developer makes those changes and pushes them to the issue fork.

After the issue reaches the "Reviewed & tested by the community" status and the issue passes a committer's review, the committer will commit the differences between the issue fork and the Drupal core repository to the core repository. The committer can do this with their local command line tools or from within the interface on

https://www.drupal.org (i.e. via a "merge" button). After the changes are committed, the committer will typically set the issue status to "Fixed".

Appendix C. Current JavaScript package change requirements

Each change must be accompanied by a corresponding issue on http://www.drupal.org.

Any change which might affect developers to have an accompanying change record linked from that issue. This applies to both breaking changes and new additions. The existing requirements for updating JavaScript are under-specified.

Each change must have in-line documentation following a published style guide.

Each change which might impact performance must have a performance profile attached to its accompanying issue.

Each change which has an impact on user interfaces must meet these documented accessibility requirements and pass an accessibility review. Additional, user interface changes must pass a usability review.

Each change must pass all existing tests and new features or bug fixes must have a new test added. In some cases, an issue to add a test can be created instead of providing a test in the change issue itself.

Each change must adhere to the backwards compatibility ("BC") and API policy. Explicit rules for JavaScript packages are not defined.

If a change introduces an alternative API to replace an existing API, the change must adhere to the JavaScript deprecation policy. Note that each JavaScript package change must follow "the same general best practices [as are followed] for PHP, namely: public APIs should always provide BC and a deprecation."

TODO: file a core issue to move Drupal.deprecatedError into a standalone package or find a replacement

Appendix D. Current security practices

There are two ways to report a security vulnerability: via https://www.drupal.org project pages or via email.

The Drupal Security Team has a set of procedures in place to receive security reports, to qualify those reports, and to notify project maintainers by email as appropriate. They also have infrastructure for discussing and testing proposed mitigations as well as a process for announcing and releasing those mitigations (a.k.a. the security advisory process).

The Drupal Security Team also observes a set of community norms. E.g. security releases are rarely released during periods when developers are unlikely to be available (e.g. during a DrupalCon).

Appendix E. Current governance roles and responsibilities

The Benevolent Dictator for Life (BDFL): The founder and chief decision-maker for the project. The BDFL appoints and replaces those maintainers, and spends time thinking about how to make the team function well. The BDFL also holds final say in any disagreements, and can be escalated to when decisions cannot be reached through the normal process. Finally, the BDFL also preserves the philosophy, culture, and principles of the project.

Core Committers: The small subset of people who, along with the BDFL, can commit changes to Drupal core itself. Core committers are a group of peers, and are expected to work closely together to achieve alignment, to consult each other freely when making difficult decisions, and to bring in the BDFL as needed. Within the core committer team there are:

Product Managers: Focus on user-facing features and Drupal's user experience. Product managers are expected to talk extensively with different types of Drupal’s users, gather data about how Drupal is used in the “real world,” and present this information back to Drupal's various audiences. Significant changes to user-visible features, the user interface, and install profiles must be approved by the product managers.

Framework Managers: Focus on cohesiveness of Drupal's architecture, APIs, and developer experience. Significant developer features, changes to public APIs, external libraries, and overall architecture must be approved by the framework managers.

Release Managers: Focus on processes to increase release stability, and on facilitating effective collaboration among contributors. The release schedule is set by the BDFL in

consultation with the other core committers, and the release managers make sure that we manage towards it. This includes rolling the releases, tracking the release schedule, facilitating communication between all participants, measuring progress, and defining the processes that help contributors collaborate effectively and efficiently. Release managers also make the final call on issue "metadata", such as status and priority, unless overruled by the BDFL.

Subsystem Maintainers: Maintainers of one particular part of Drupal core, such as the Entity system or Comment module. Anyone can be a provisional maintainer to a subsystem by simply contributing to it. Most subsystem maintainers start out contributing to their component by triaging issues, supplying patches, and reviewing others' patches.

Topic Maintainer: Subject-matter experts in a topic that spans multiple subsystems—Accessibility, Documentation, Performance, Testing, and Usability. These correspond to the core gates which all proposed changes must pass.

Initiative Coordinator: Community members appointed by the BDFL to coordinate strategically important work. Initiatives are typically cross-functional in that they span different subsystems and different topics, and can affect both Drupal the product and Drupal the framework.

Contributor: Anyone with a Drupal.org account can be a core contributor, simply by participating in the Drupal core issue queue. Contributors are empowered to drive changes within Drupal's code base, and to work with other contributors to help get changes they care about, using a variety of both technical and non-technical skills. Proposed changes follow a peer review process.