Developer's Guide

Earthdata Search Development Guide

Getting Started

Improving the Guide

This guide is a constant work in progress. If something is missing or confusing, please update the guide with better information.

Obtaining Code

Earthdata Search has been open sourced and is available at https://github.com/nasa/earthdata-search.

If you are a developer on the EDSC dev team, you will need to see your team leader to request the appropriate accounts and permissions to be added to the NASA Github account. Once that is done you'll be added to the EDSC project and can clone the repo.

If you are an external contributor, please fork the EDSC repo from https://github.com/nasa/earthdata-search.

There are a few other related code repositories that may be of interest to you. Links to those repositories are below:

- cmr-graphql
- edsc-echoforms
- edsc-timeline
- edsc-smart-handoffs
- edsc-geo
- earthdata-download

Initial setup

The README at https://github.com/nasa/earthdata-search contains documentation detailing configuration, setup, and running Earthdata Search. After following the installation guidelines in the README, follow the rest of the guide for the commands needed to run an initial setup.

Resources

JIRA Agile Board: https://bugs.earthdata.nasa.gov/secure/RapidBoard.jspa?rapidView=209

JIRA Project: https://bugs.earthdata.nasa.gov/browse/EDSC

Github site: https://github.com/nasa/earthdata-search

Bitbucket site: https://git.earthdata.nasa.gov/projects/EDSC

Bamboo Build: https://ci.earthdata.nasa.gov/browse/EDSC-DBN2

Communication

- Stand-ups occur daily at 9:45 AM EST. (EOSDIS Slack Huddle)

- Sprint planning meetings occur as part of the EED-2 SAFe efforts. Typically on the first day of each sprint.
- Retrospectives typically take place on the last Thursday of the sprint.
- Developers should use EOSDIS Slack accounts for chat and join the #edsc-dev channel for team communication. See your team lead for an invitation to the channel.

Contact the project lead for up-to-date meeting invitations.

Project Overview

Development Process

Earthdata Search uses an agile development process with two-week sprints. At all times, our main branch maintains a potentially releasable product increment. Once a commit is merged into main, it will immediately trigger a build and deployment to our SIT environment.

Sprints give us periodic checkpoints for feedback but are orthogonal to our releases.

Meetings

We try very hard to minimize interruptions. As such, our meetings are few and short. We have a combined sprint review and sprint planning before the start of each sprint where stakeholders are invited to see what we've done and give input on where we are going.

We have short daily stand-ups near the start of the work day to give status and note any problems.

We have periodic retrospectives, but more importantly, we try to maintain direct communication and adjust for problems as they arise.

Issue Types

We use JIRA issue types as follows:

- **Story:** A user-visible requirement which ought to be achievable by a single developer in a couple of days. These are captured in a consistent format, "As a [user type] user, I should <feature description> so that <value to user>". For instance, "As a user, I should be able to log in so that I may access information saved with my account."
- **Epic:** Groups similar stories into less granular pieces of functionality.
- **Bug:** A user-visible implementation problem that causes clearly broken behavior. Examples: a button that does not work, severely degraded performance, a security hole.
- **Improvement:** A user-visible implementation problem that, while not clearly broken, poses usability challenges. These are almost exclusively reserved for interface design issues since we often have a backlog of features which have been implemented but need UI polish. Improvements are optional and we could ship without them. Examples: a button that looks out of place, text that needs styling.
- **Task:** Necessary work that is not user-visible. Examples: Setting up a CI environment, research to make a recommendation, documentation.
- **Vulnerability**: A security vulnerability reported by code scans (e.g. Snyk).
- **New Feature:** Not used.

Estimation

We estimate stories using the Fibonacci sequence, 1, 2, 3, 5, or 8 points. These units do not correspond directly to developer-hours.

- **1 Point:** Trivial. Typically a change in configuration only. Example: Sorting a list by a new field when the logic for sorting is already in place.
- **2 Points:** Easy. Changes are straightforward and do not involve complex logic or new components. Example: Allowing the user to delete resources already displayed in a list.
- **5 Points:** Normal. Changes may involve complex logic, a new component, or cross-cutting changes to the codebase. This is the ideal granularity for stories. Example: Creating a new page which presents a list of resources (including appropriate links to that page, and tests for those links and the contents of the page).
- **8 Points:** Hard. Changes involve very difficult logic, extensive testing, or sweeping changes to the codebase. Implementation details may be highly uncertain and risky. Cannot reasonably be broken up into smaller stories. Use this estimation sparingly (only 2% of our v1 stories fell into this category). Example: Drawing ECHO spherical polygons on a Leaflet map.

Note: The following may change as the project transitions to sustaining engineering

Our estimation may differ from typical agile projects in the following ways:

Bugs are tackled as soon as possible. We cannot have a huge backlog of bugs. If we've marked something done and it is defective, we fix it. Quality is non-negotiable.

We do not re-estimate stories. Re-estimating stories changes the inputs to the backlog and throws off the numbers. Consistency is far more important than accuracy. Velocity will sort it all out.

Development

No change may go into main until it has undergone peer review in Github and is successfully building in GitHub Actions. Main is to remain deployable at all times, so ensure it contains production-ready code with green builds.

Occasionally, due to intermittent problems with test timing or execution order, our main build will fail. Fixing the issue and ensuring it does not happen again becomes the highest priority when this happens.

Day-to-day development proceeds as follows:

1. The developer chooses a JIRA issue from among those in the active sprint, assigns it to himself, and moves it to the "In Progress" column of the agile board
2. The developer implements the change in a new git branch named after the issue number, e.g. EDSC-123. Commits are also prefixed with the issue number, e.g. "EDSC-123: Add user accounts page".
3. (Optional) When in need of collaboration or input, the developer pushes the branch and opens a new pull request *without assigning a reviewer*. Collaborators may pull the branch or view and comment on the diff in Github.
4. When the change is complete, the developer pushes the branch and opens a pull request (*without assigning a reviewer*), triggering a GitHub Actions build for that branch, and ensures the build is green.
5. Once the build is green, the developer assigns the pull request for the branch to each member of the development team and moves the issue into the "Pending Review" column in JIRA.
6. The reviewer looks at the code, ensuring passing tests, adequate test coverage, code quality, and correct/complete implementation. He also runs the code and manually tests it, paying close attention to usability and consistency of like-features or interactions throughout the rest of the application.
7. The original developer fixes items brought up during review until the reviewer is satisfied and has approved the pull request. In most cases, we like to have 2 approvals. In some cases that is not needed. The team lead will have a good idea as to what needs additional review. Use your best judgement.
8. Once sufficient approvals have been granted via Github, the original developer merges the branch. At this point the remote branch can be deleted.
9. Once main has built and deployed to SIT, the original developer verifies the new functionality/fix is working, moves the JIRA issue to the "Ready for SIT Testing" column, and creates a JIRA issue test session for QA and verification.
10. If the QA process reveals updates need to be made, the QA team member works with the original developer directly to resolve any issues, following the process outlined above, until QA approves the changes.
11. The QA team member moves the JIRA issue into the "Done" column, typically with a resolution of "Verified Internal".
12. Once deployed to the UAT environment, the QA team member executes a regression testing protocol ensuring new features work correctly and that the build/release is stable. They also reach out to a primary stakeholder for the issue (for instance, the person requesting the change) so that they may test the implementation. If satisfied, the Product Owner adds "Verified in UAT" as a comment in the issue or the issue is moved to "Verified External" (in the case of an external reviewer).

Sprints

Sprints run 2 weeks and are grouped together in Program Increments (referred to as a PI) following the Scaled Agile Framework (SAFe) methodology (https://www.scaledagileframework.com/). The EDSC Scrum Master will initiate all scrum ceremonies and ensure communication is flowing both within the team and between external parties.

End Of Sprint TODO

At the end of every sprint, in order to make patching a release easier, please tag the main branch at the latest commit.

1. `git tag` (shows all tags)
2. `git tag -a "vX.X.X" -m "Sprint X.X release"` (adds the tag at the current commit)
3. `git push --tag` (pushes the tag to the remote)

Also remember to update the Release Version on bamboo for all of the projects

These tags are referred to as releases in Github and can be viewed here: https://github.com/nasa/earthdata-search/tags

Deployment

There are 3 shared deployment environments for Earthdata projects, including Earthdata Search:

System Integration Testing (SIT)

```
An environment for performing integration tests on builds, ensuring that new code works correctly in a production environment before placing it in front of customers or users. This is roughly equivalent to ECHO's "testbed" environment.
```

User Acceptance Testing (UAT)

```
An environment for verifying that builds meet the customer and user expectations before promoting them to operations. We expect partners and advanced users to use this environment frequently for their own testing, so it will be public and must have a high uptime. This is roughly equivalent to ECHO's "partner-test" environment.
```

Operations (Ops)

```
The production environment, running the canonical public-facing site.
```

Deploying changes

Changes to shared environments must be deployed through Bamboo via the Earthdata Search deployment project. Any successful build of the main branch in GitHub Actions will result in the code being send to the *main* branch of the ECC Git repo (BitBucket). Once that branch receives the code, the branch is built and deployed to the EDSC SIT environment (https://search.sit.earthdata.nasa.gov).

Development Style Guide

Introduction and Concepts

Progressive Enhancement

Progressive enhancement involves starting with a baseline HTML document that is readable and usable without any styling or javascript. We accomplish this by using semantic markup. From there we enhance the markup by unobtrusively adding CSS styles and Javascript.

By starting with working basic HTML, we ensure we have a page that's minimally usable by:

- Visually impaired users relying on screen readers
- Web crawlers which do not use CSS to determine content's importance and possibly do not evaluate Javascript
- Users who experience an error that breaks scripts or the serving of assets
- Test suites which otherwise do not need Javascript or CSS to perform their tests
- Users of unsupported browsers
- Users with Javascript or CSS disabled or who are attempting to access data from scripts (common in the scientific community)

The key point here is that a missing browser feature or a single script or style error should not render the site unusable.

Mobile-first Development

This is similar to Progressive Enhancement described above. We target narrow screens (typically mobile screens) first and add additional styles as screens get wider, using media selectors.

The reason for this is mostly practical. Mobile renderings tend to have a much simpler set of styles than larger renderings. If we targetted large screens first, much of our mobile effort would be in undoing the styles aimed at larger screens.

Another key point here is that we will plan to be mobile friendly from the start. It is much easier to build this in from the beginning than to attempt to construct it later on.

General Principles

Treat frontend authoring as a development discipline.

HTML is just markup and easy to learn. CSS is turing complete, but not really a programming language. They're easy to dismiss.

The reality is, though, that HTML and CSS provide very few mechanisms for code reuse and organization. Their size and complexity has a direct and perceptible impact on page speed, and they are the most user-visible part of the codebase. It is exceedingly difficult to create clean, performant, reusable, and extensible frontend assets. It generally requires much more care than the corresponding backend code, since backend languages are designed with these aspects in mind.

Frontend authoring is a development discipline and requires a great deal of care and consideration, to the point that most of this guide focuses on frontend development.

Minimize complexity from the beginning.

It is typically very difficult to extract complexity from front-end code. All new components should be focused on reuse, versatility, and extensibility. When possible, do not add new components at all, but reuse or extend existing components.

Create components, not pages.

When developing frontend code, the unit of reuse should be the module, component, or behavior, not the page. Design components that can be used across multiple pages or that are versatile enough to be used multiple times on the same page, possibly for different purposes. Write, CSS, Javascript, and partial HTML for components, not for pages, in order to promote robustness and reuse and keep code size in check.

Document decisions.

There may be good exceptions to every rule in this guide. When in doubt, follow the guide, but make exceptions as necessary. Always document these decisions. Further, whenever you write code in a non-standard way, or you are faced with multiple competing options and make an important choice, document those decisions as well.

Testing

Test everything.

Every line of application code, including UI markup, should be exercised in a test.

Test at the appropriate level.

Exercise boundary conditions, error handling, and varying inputs at the unit or functional level.

Integration tests should demonstrate user-visible system behavior.

Remember that integration tests run much more slowly than unit tests, so prefer to test more thoroughly at the unit level.

Build meaningful sentences with Jest blocks.

The chain of Jest "describe" blocks leading up to the final "test" block should form a human-readable sentence. This is particularly true for integration specs where we are documenting system behavior spec names.

Consider an example where we don't use this style.

Bad Example:

```
describe('Account creation', () => {
      …
      describe('messages', () => {
        …
        test('should display success messages', () => { … })
        test('should display failure messages', () => { … })
      })
      test('recovers passwords', () => { … }
      test('should send emails to users', () => { … }
    })
```

Consider the sentences produced by the above:

1. Account creation messages should display success messages.
2. Account creation messages should display failure messages.
3. Account creation recovers passwords.
4. Account creation should send emails to users.

The test fails to describe the system. Reading the sentences, we don't know why a particular behavior might happen. Some of the sentences don't entirely make sense.

We fix the problem by using more descriptive contexts and paying attention to the sentences we're constructing with our tests.

Improved Example:

```
describe("Account creation" do
      …
      describe('for users providing valid information', () => {
        test('displays a success message', () => { … }
        test('sends an email to the user', () => { … }
      })
      describe('for users providing duplicate user names', () => {
        test('displays an informative error message', () => { … }
        test('prompts users to recover their passwords', () => { … }
      })
    })
```

Consider the sentences produced by the above:

1. Account creation for users providing valid information displays a success message.
2. Account creation for users providing valid information sends an email to the user.
3. Account creation for users providing duplicate user names displays an informative error message.
4. Account creation for users providing duplicate user names prompts users to recover their passwords.

The above sentences more adequately describe the behavior of the system given varying inputs.

Avoid the ugly mirror problem.

http://jasonrudolph.com/blog/2008/07/30/testing-anti-patterns-the-ugly- mirror/

Tests should describe how the system responds to certain inputs. They should not simply duplicate the code under test.

Minimize test suite execution time.

The test suite should provide developers with rapid feedback regarding the correctness of their code. To accomplish this, they should execute quickly. Keep performance in mind when writing tests. The following guidelines will help minimize execution time:

1. Test varying inputs and edge cases at the unit or functional level, rather than the integration level.
2. Avoid running integration tests with Javascript enabled unless Javascript is necessary for the feature under test.
3. Avoid calling external services, particularly ones which cannot be run in a local environment. Use mocks for these services.
4. Avoid loading external images, CSS, and Javascript in integration tests.
5. Avoid or disable interactions and interface elements that will cause Capybara to wait. For instance, disable animations or transitions.
6. Skip to the page under test in integration tests, there is no need to start at the home page for every test (though you should have a test which verifies you can start at the home page).
7. Avoid increasing timeouts to fix intermittent problems. Find other means.
8. Time your tests.
9. Follow this style guide for performant HTML, CSS, and Javascript.

If performance becomes a problem, we may segregate tests into "fast" and "full" runs, but ideally we will avoid this.

Fix sources of intermittent failures immediately.

If you see a failure and you suspect it was caused by some intermittent problem, e.g. a timeout that is too short or an external service being down, it is not enough to simply re-run the tests. Fix the problem. If a problem truly cannot be fixed, document why, catch the specific error that cannot be fixed, and throw a more meaningful one.

React testing library best practices:

- Avoid using test-ids where possible, this reduces our applications accessibility so use this sparingly
- Favor retrieving elements by `Role` such as buttons, checkboxes etc. If there are multiple elements of the same role on a component use a secondary filter to ensure you select the correct ones for your assertions example: getByRole('button, {name: 'button-name'}
- Don't use `await` statements for userEvent methods, userEvent is already wrapped in an await
- Using a container instead of `screen` to select elements on the virtual DOM
- Don't perform side effects within a `waitFor` block
- For more see(https://kentcdodds.com/blog/common-mistakes-with-react-testing-library)

HTML

Use semantically meaningful elements where possible.

`div` and `span` convey no meaning and are rarely the most appropriate element in a given context. When adding an element, look at other elements that HTML provides to see if another is more appropriate. This helps developer readability, keeps styles simpler, and may aid accessibility or search engine indexing. Here are some questions you may ask:

- Does it contain a paragraph? Use a `p`.
- Does it contain a heading or subheading? Use a `h[n]`.
- Does it contain an item in a list of items? Use a `li`.
- Does it constitute a section of the page appropriate for an outline? Use a `section`.
- Does it contain header or footer information for a page or section? Use a `header` or `header`.
- Does it contain describe a form field? Use a `label`.
- etc

Avoid presentational class names.

We should be able to dramatically alter the look and feel of the site without changing (much) HTML in order to match changing designs or to accommodate new user agents. This is particularly important for responsive design, where multiple browsers may need to render the same HTML in different ways.

To allow this, use class names that describe an element's purpose, contents, or behaviors, rather than ones that describe its presentation.

```
Overly-presentational class names.  We may not want this element
      to appear to the left on mobile browsers, for example.
    -->
    <section class="left bold bordered">

    <!-- This one uses more semantically appropriate class names -->
    <section class="primary emphasized summary">
```

Avoid unnecessary elements. Keep nesting shallow.

Minimize the overall depth of HTML to decrease page size, increase readability, and improve rendering speed.

CSS with Sass

Earthdata Search uses SCSS to generate its CSS. It follows the guidelines for scalable CSS outlined by SMACSS, with key items reproduced in this document. The CI build checks CSS style using CSS Lint. Developers are strongly encouraged to read the CSS Lint rules.

Do not use id selectors (#id)

ID selectors indicate that a style is one-off and can only appear in one place on a page. Prefer to build reusable styles that could appear in multiple places or on multiple pages. Further, id selectors are very specific in terms of the CSS cascade, making them hard to override.

There are two exceptions to this rule: #site-header and #site-footer. These two areas of the page are typically significantly different than the rest of the site to the point where they need to override all styles, and the drawbacks of using id selectors mostly do not apply.

Do not use style attributes or inline styles.

Place all styles in CSS files. Mixing styles with HTML makes tracking, testing, and reusing styles much more difficult. Further, style attributes are incredibly specific and difficult to override.

Javascript components should also prefer to apply classes to elements rather than alter their style attributes for the same reasons.

Do not use `!important`

`!important` makes styles impossible to override unless the overriding rule also uses `!important`. In almost every case, using `!important` is unnecessary. Prefer to use selectors with higher precedence or alter overridden selectors to have lower precedence.

Very very rarely it is necessary to use `!important` because of third-party code. This typically happens when a third-party Javascript library adds an undesirable style attribute to an element. When given the choice between altering third-party Javascript or using `!important`, the latter is usually preferable. In this circumstance, document the decision. This is one reason we avoid setting style attributes, even in Javascript components.

Prefer selectors that describe attributes and components, not domain

concepts.

Consider the following element:

```
<ul class="granules">
```

`ul.granules` selects a list with very specific elements, which may only be available on one page. Any styles applied to this list are unlikely to be reusable. Adding classes that describe attributes of the list makes CSS styles more modular and reusable.

```
<ul class="granules zebra scrollable selectable">
```

Here we may use different selectors to apply different attributes to the list. `ul.zebra` may add zebra striping to the rows, `ul.scrollable` may add a scroll widget, and `ul.selectable` may provide hover and selection mechanisms. Any of these attributes could be useful on lists that do not describe granules, and other lists could mix and match attributes to match their needs. Fine-grained attribute-centric class names provide for better reuse.

Follow SMACSS rule categories and naming conventions

Follow SMACSS guidelines for grouping and naming CSS classes. Prefix layout classes with "l-". Prefix state classes with "is-". Do not prefix module classes. Use double dashes to indicate module subclasses, e.g. "module--subclass"

Minimize selector depth

As described by SMACSS' Depth of Applicability chapter, minimize and simplify CSS selectors. Deeply nested selectors are easy to create in Sass, but they are hard to understand and create a strong coupling to the underlying HTML structure. Further, they tend to be overly-specific, causing duplication in CSS rules. Whenever possible, keep nesting 1-2 selectors deep, with 3 being the maximum.

```
// Deep nesting
    .module {
      div {
        h1 {
          span {
            font-weight:bold;
          }
        }
      }
    }
    // Shallower nesting
    .module {
      h1 > span {
        font-weight:bold;
      }
    }
    // Shallow nesting
    .module-header-text {
      font-weight:bold;
    }
```

Use fast CSS selectors where possible.

A large page with many CSS rules can suffer rendering performance problems that make pages feel sluggish even on modern browsers and computers. Understand selector performance and follow these rules to allow pages to render quickly:

1. Use child selectors 2. Avoid tag selectors for common elements (div, span, p) 3. Use class names as the right-most selector

Use variables for colors and numbers.

Use Sass variables to describe all colors except black and white and all numbers except 0 and 1px. This makes it easier to find usages of measurements and change them as necessary.

Use Sass helpers for CSS3 styles.

Sass and compass provide helpers for CSS3 styles that normalize experimental browser extensions, for instance

```
@include border-radius(4px, 4px);
```

generates

```
-webkit-border-radius: 4px 4px;
    -moz-border-radius: 4px / 4px;
    -khtml-border-radius: 4px / 4px;
    border-radius: 4px / 4px;
```

Use these helpers to avoid overly-verbose CSS.

Target browsers with classes not CSS hacks.

The base site contains HTML boilerplate libraries which add CSS classes to the html and body element that detect commonly misbehaving browsers (older IE versions) and browser capabilities. Use these classes to target browsers or capabilities rather than relying on CSS hacks.

Javascript

Build components and modules, not one-off elements

Try to build Javascript components and widgets that could apply throughout the site, rather than on a single page or in a single situation. Good questions to ask when writing code is "Can I make this into a widget?" or "Can I apply this behavior to all elements with this class?". If the answer is no, perhaps the element could be described as a composition of multiple components (scrollable, zebra-striped, selectable list rather than one-off granule list)

Allow users to bookmark content and use the back button

Dynamic interfaces are great, but users should be able to bookmark their current location to return later, especially for complex search UIs like Earthdata Search. Further, we should allow the user to back up to previous states or leave the site via the back button.

When building the interface, use the History API to ensure that history entries are pushed to the stack appropriately. Push entries to the stack when the user reaches points they would reasonably expect to bookmark. Avoid pushing entries so frequently that backing out of a state using the back button becomes tedious or impossible.

Deployments

Main

Merging to main will kick off a build on GitHub Actions. Bamboo will automatically kick off a deployment to SIT, on a successful build.

If the build on main fails and requires intervention a manual deployment to SIT will be required.

Manual Deployments

Prerequisites

- Must have NASA VPN Token
- Must have code 700 VPN Token to login to Bamboo
- Permissions to deploy to the desired environment (may need to contact OPS)

Once on the NASA VPN, visit https://ci.earthdata.nasa.gov/browse/EDSC-DBN2 which will prompt you for your code 700 token, further instruction for each environment are below.

Deploying to SIT, UAT or OPS

From main on GitHub

This is the most common deployment method. After working a ticket, you've issued a Pull Request, that has been reviewed and merged into **main** on GitHub. When GitHub Actions completes a build on main it runs *bin/ecc-sync* which syncs **main** from GitHub to **deploy** on Bitbucket. Bamboo has a deploy ment trigger configured to deploy automatically on a successful build to SIT.

From arbitrary branch on GitHub

At any time you can push your code to Bitbucket and create a release from that branch. Note that you should not push these branches to any environment except SIT, and that you should check with all necessary stakeholders before doing so. Once you are done testing that branch on SIT, return SIT back to the normal main branch deployment.

Backports

The following example creates a back port for version **1.69.2**, meaning that is the version that is currently deployed and we need to create **1.69.3** to push out a bug fix.

First we'll need to ensure we have the most recent tags, so fetch everything from git.

```
git fetch
```

We'll create a new branch off of the currently deployed tag.

```
git checkout tags/v1.69.2 -b 1.69.x
```

Now we're on a branch that contains the exact code deployed for the tag **1.69.2**. We'll find the commits that we need to get into this back port and cherry pick each one.

```
git cherry-pick COMMIT-HASH
```

Once our commits have been cherry picked, we can tag the most head of our branch as the new release

```
git tag v1.69.3
```

Tags are simply pointers to commits, so now that our tag is created, push it to GitHub for tracking and in case we have a future backport to create.

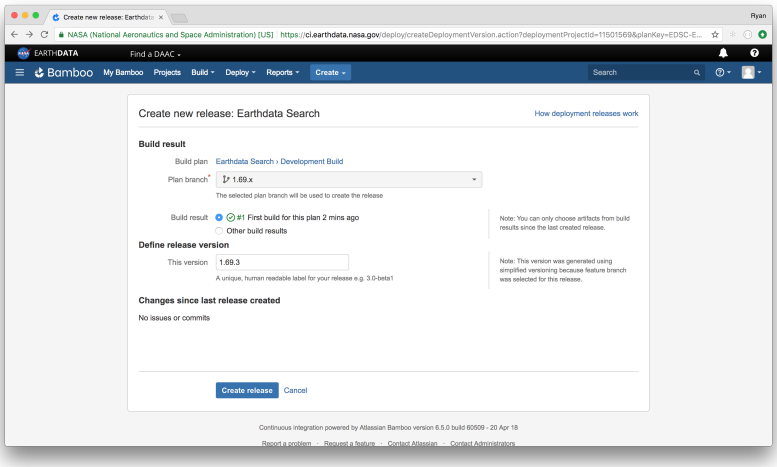
```
git push origin v1.69.3
```

In order to get a release created for OPS to deploy, we need to now push the branch to Bitbucket.

```
git push bitbucket 1.69.x
```

Creating the Release

Shortly after pushing to Bitbucket you should see a new build appear on Bamboo, this build will be your most recent commit. When the build is complete you'll see a button on the right hand side of the page title **Create release**, clicking that button will allow you to create a new release named **1.69.3**.



Select the **1.69.x** branch, name the release and click **Create release.** From here you'll be sent to the release, this is the link that should be provided to the OPS team to deploy.