The GDS Way and its content is intended for internal use by the GDS community.

Accessibility

This accessibility statement applies to the GDS Way at https://gds-way.digital.cabinet-office.gov.uk/ (https://gds-way.digital.cabinet-office.gov.uk/).

This website is run by the GDS Way team at the Government Digital Service (GDS). We want as many people as possible to be able to use this website. For example, that means you should be able to:

- change colours, contrast levels and fonts
- zoom in up to 300% without problems
- navigate most of the website using just a keyboard
- navigate most of the website using speech recognition software
- listen to most of the website using a screen reader (including the most recent versions of JAWS, NVDA and VoiceOver)

We've also made the website text as simple as possible to understand.

AbilityNet (https://mcmw.abilitynet.org.uk/) has advice on making your device easier to use if you have a disability.

How accessible this website is

We're committed to making this website accessible, in accordance with the Public Sector Bodies (Websites and Mobile Applications) (No. 2) Accessibility Regulations 2018.

Compliance status

This website is fully compliant with the Web Content Accessibility Guidelines version 2.1 (https://www.w3.org/TR/WCAG21/) AA standard.

Feedback and contact information

If you need information on this website in a different format like accessible PDF, large print, easy read, audio recording or braille, contact gds-way-accessibility@digital.cabinet-office.gov.uk with details of your request.

We'll aim to reply within 3 working days.

Reporting accessibility problems with this website

We're always looking to improve the accessibility of this website. If you find any problems not listed on this page or think we're not meeting accessibility requirements, email gds-way-accessibility@digital.cabinet-office.gov.uk.

Enforcement procedure

The Equality and Human Rights Commission (EHRC) is responsible for enforcing the Public Sector Bodies (Websites and Mobile Applications) (No. 2) Accessibility Regulations 2018 (the 'accessibility regulations'). If you're not happy with how we respond to your complaint, contact the Equality Advisory and Support Service (EASS) (https://www.equalityadvisoryservice.com/).

Preparation of this accessibility statement

This statement was prepared on 1 September 2020. It was last updated on 6 October 2021.

We last tested this website for accessibility issues in March 2021.

This page was set to be reviewed before 24 March 2023 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way). This might mean the content is out of date.

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Building accessible services

Government services must be accessible to everyone. This includes anyone with a visual, hearing, speech, motor, learning or cognitive impairment. This also includes anyone with a temporary or situational disability, such as a person with a broken arm or working in a loud environment.

Building a service with accessibility in mind not only allows those with access needs to use your service, it also improves the service for everyone else. An accessibility problem with a website can be something that affects everyone, not just people who can only access the web with a keyboard or screen reader.

Government services are legally required to be accessible. This means services must comply with the international WCAG 2.2 AA accessibility standard. This requirement applies to all new and existing public sector websites and mobile applications.

Further reading:

- legal accessibility requirements for government services (https://www.gov.uk/guidance/accessibility-requirements-for-public-sector-websites-and-apps)
- the equality act (https://www.gov.uk/guidance/equality-act-2010-guidance)
- understanding WCAG 2.2 (https://www.gov.uk/service-manual/helping-people-to-use-your-service/understanding-wcag)

How to make your service accessible

When looking to make a service accessible, the GOV.UK Service Manual's general guidance on testing for accessibility (https://www.gov.uk/service-manual/helping-people-to-use-your-service/testing-for-accessibility) is a great introductory resource.

Consider accessibility from the start

You cannot achieve accessibility by adding some final touches - it must be considered at all stages of a project. You should review designs for possible issues, write and run tests throughout development, and test services with accessibility in mind.

Understand that not everyone reads content the same way

A sighted user might navigate a page from top to bottom, perhaps skim reading through headings and paragraphs to find the content they want.

Non sighted users can also skim read a page. Screen readers can announce content by element type, such as headings, paragraphs or links. For example, if a screen reader user expects a page to contain data in a table element, such as a train timetable, they might start by reading through all the tables on a page.

This is why semantic markup (https://html.com/semantic-markup/) and good heading structure are important when building accessible services.

Automated testing

Automated accessibility testing tools are useful for finding issues, but automated testing can only find around 30% of likely accessibility problems (https://alphagov.github.io/accessibility-tool-audit/). Automated testing shouldn't be the only accessibility testing carried out on a service, it should also be tested with real users.

Popular tools for automated accessibility testing include:

- Axe (https://www.deque.com/axe/)
- Lighthouse (https://developers.google.com/web/tools/lighthouse/) (which uses Axe for its accessibility tests, and can be run from Chrome Devtools without the need for additional installation)
- WAVE (https://wave.webaim.org/)
- ARC Toolkit (https://www.tpgi.com/arc-platform/arc-toolkit/)

You may find it useful to include automated accessibility testing as part of your continuous delivery (/standards/continuous-delivery.html) workflow.

Testing with assistive technologies

Test your service with assistive technologies throughout development, especially when you introduce a significant feature or make any major change. You should test in the assistive technology and browser combinations listed in the Service Manual (https://www.gov.uk/service-manual/technology/testing-with-assistive-technologies).

WebAIM have some useful articles that explain how to test with some screen readers:

- Testing with screen readers (https://webaim.org/articles/screenreader_testing/)
- Using JAWS (Windows) (https://webaim.org/articles/jaws/)
- Using NVDA (Windows) (https://webaim.org/articles/nvda/)
- Using VoiceOver (macOS) (https://webaim.org/articles/voiceover/)

- Using VoiceOver (iOS) (https://webaim.org/articles/voiceover/mobile)
- Using TalkBack (Android) (https://webaim.org/articles/talkback/)

You can run VoiceOver training on macOS by going to System Preferences > Accessibility > VoiceOver > Open VoiceOver Training…

Manual accessibility testing guide

The Government Digital Service's accessibility monitoring team has a publicly-available accessibility testing guide (https://github.com/alphagov/wcag-primer/wiki). The guide is currently labelled as a 'work in progress', and was published in 2022.

The guide outlines some approaches for testing websites and applications against the Web Content Accessibility Guidelines (WCAG) (https://www.w3.org/TR/WCAG/).

Using Assistiv Labs to test with assistive technologies

You can use Assistiv Labs (https://assistivlabs.com/) to test your service with JAWS and NVDA.

Contact the 🔒 Engineering Enablement team (https://sites.google.com/a/digital.cabinet-office.gov.uk/gds/directorates-and-groups/cto-and-ciso-office/engineering-enablement#h.f8vfqmbkqtgf) to request an account.

We have been assured by Assistiv Labs that the Virtual Machine (VM) is deleted when you exit, however in the interests of security and to make sure we do not leak any personal data or sensitive material:

- treat the VM you are using as if it was a shared device – like using a public computer in an internet cafe or library
- do not use it to test any confidential or secret service, or services containing content that is considered sensitive
- do not use any credentials from live systems with access to "real data" during your testing
- do not sign in to any accounts, unless those accounts are used solely for the purposes of testing (for example, a test account in an integration or staging environment)
- make configuration changes to the system through your own device rather than the VM

You should also make sure you understand how using Assistiv Labs affects any other requirements for your service, such as PCI (Payment Card Industry) compliance, and assess any additional risks. You can speak to your service's information assurance lead or the GDS Privacy Team for specific guidance.

Testing in an accessibility lab

You can test your service using an accessibility lab (https://gds.blog.gov.uk/2018/06/20/creating-the-uk-governments-accessibility-empathy-lab/), such as the one in the GDS London office. The lab contains devices with assistive technologies installed for testing, including Dragon, which is not available through Assistiv Labs. It also provides some simulations of using the web with accessibility issues, so developers can better understand what this is like.

Test content served by third party systems

If a service is built using an existing or third party system, such as a content management system or JavaScript framework, it should be tested for accessibility. Accessibility compliance cannot be guaranteed by any system, as even small changes can introduce accessibility barriers.

Testing with real users

Services should be tested by real users to evaluate how accessible they are. This can be with members of the public as well as specialist organisations who provide accessibility testing services.

Testing with real users can be a time consuming process. It is recommended that this is done only after other accessibility testing takes place.

Use the GOV.UK Design System

The GOV.UK Design System (https://design-system.service.gov.uk/) is a collection of website styles, components and patterns designed to be used to build government services. It provides pre-built website markup that has been developed specifically to be accessible. Using it will also save development time and provide a look and feel consistent with other government services.

While it may not have a full set of components for every service, it provides a solid foundation to work from that can be expanded to suit a service's individual needs.

You can also use the GOV.UK Prototype Kit (https://prototype-kit.service.gov.uk) to create working prototypes, share ideas with people and conduct user research.

Specific guidance

There is a common misconception that accessibility is difficult, time consuming and costly. This is not true, however the range of possible issues that could occur is too wide to detail here. Having said that, here are some common accessibility issues many websites have, why they're a problem and how to avoid them.

Non semantic HTML

Semantic HTML means using the appropriate element for the element being described. A common example of this is not using a button element for a button, but relying on JavaScript to provide the correct action when the element is used.

This is a problem because non-semantic HTML is difficult for technologies such as screen readers to understand. This means that some users will not be able to operate or interact with these elements, potentially rendering a service unusable.

Poor in-page navigation

As websites become increasingly large it is common to have navigation sections at the top of the page containing lots of links to other parts of the site. It is important to provide a mechanism for skipping past these elements to the main content for keyboard users, such as a skip link.

You should build pages using landmarks in order to allow users to navigate more easily between them. For example, an email application might have one landmark for the email folder pane (inbox, drafts and so on) and another for the email list pane.

Improper heading structure

Screen reader users can rely heavily on correct heading structure within a page in order to navigate and understand its content. The page title should be presented within an H1 element, and all other headings should follow a logical ordering, for example `H1`, `H2`, `H3`, not `H1`, `H3`, `H2`.

Form controls not associated with a label

All form controls should have a label associated with them that describes what the form control should be used for. Placeholder text (https://html.spec.whatwg.org/multipage/input.html#the-placeholder-attribute) is not an acceptable alternative.

Labels should provide a short but clear explanation of a form control. Further detail can be provided using a separate element. This should also be associated with the form control by using an attribute such as aria-describedby.

Meaningless or duplicated link text

The text of a link should describe what that link points to. It should not rely on the surrounding text for context. Users of assistive technology can navigate pages by element type, which means that link text such as ‘click here’ or ‘read more’ does not help a user understand where the link goes.

Poor colour contrast

Users can have a range of sight issues, including colour blindness or situational disabilities such as screen glare. Text on a web page should always be clear and readable and background images behind text should generally be avoided.

WCAG 2.2 level AA requires a contrast ratio of at least 4.5:1 for normal text and 3:1 for large text. Contrast checking tools such as WebAIM's contrast checker (https://webaim.org/resources/contrastchecker/) can provide a quick way to test if text is readable.

Poor keyboard navigation

Keyboard users rely on focus states to know where they are on a page. This means that elements such as links and form controls must have a clear focus state set in CSS to indicate when those elements have keyboard focus.

This issue relates back to the use of semantic HTML. The use of inappropriate elements can remove the ability for keyboard users to access controls on a page, making the service unusable.

Inflexibility

Services should have a responsive layout in order to work on any screen size. In addition, consideration and testing should be included for users who apply customisations to their web browsing, either through web browser controls or specific tools. Examples of these customisations include increasing the font size or applying a custom stylesheet, such as a high contrast theme.

Complex interactive pages

Pages that provide detailed interaction for the user must be built with accessibility in mind. If page elements are updated dynamically using JavaScript you should use attributes such as aria-controls and aria-live so screen reader users are informed when page content changes.

Image alt text and captions

Images should be described using words for people that cannot see them, either using the `alt` attribute or in the accompanying body text.

Captions using the `figcaption` element are sometimes not read aloud by screenreaders when the image has a blank `alt` text. Do not use the caption to describe your image - the description should be put in the body text.

The GOV.UK Design System provides guidance on using alt text (https://design-system.service.gov.uk/styles/images/#alt-text).

This page was set to be reviewed before 9 November 2024 by the page owner 🔒 #accessibility-community (https://gds.slack.com/messages/accessibility-community). This might mean the content is out of date.

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Tagging AWS resources

We use AWS for hosting (/standards/hosting.html). Most AWS resources support tagging.

This manual documents our efforts with tagging. In time, it may be upgraded to a standard.

Why tag?

The main reasons for tagging are:

- to be able to understand costs (by assisting queries in Cost Explorer)
- to understand the provenance of resources (by tagging with metadata about source code)
- security and assurance

Currently, we care most about understanding costs.

It's not always clear to a developer what impact their work has on AWS costs.

If resources are consistently tagged as part of a particular directorate, programme, product, component, team, and environment, it becomes much easier to understand how much money is being spent in each particular context.

AWS Cost Explorer supports using cost allocation tags to filter and group resources.

Note that using AWS Organizations to tag accounts does not help here, because account-level tags are not supported for querying in Cost Explorer.

Alerting and enforcement

Currently, we do not enforce tags.

In future, we may wish to consider mechanisms such as alerting on untagged resources, or automatically deleting untagged resources.

Mandatory

- `Product` : `GOV.UK One Login` / `GOV.UK` or `DSP`
- `System` : the name of the software system, for example `Authentication` or `Identity proofing and verification core` . Avoid abbreviations.
- `Environment` : should be one of `production` , `staging` , `integration` , or `development` .
- `Owner` : an email address for an owner for the resource. For dev environments, this will be an individual email address; elsewhere it will be a group address.

Optional

- `Service` : used to describe the function of a particular resource (for example: `account management` , `session storage` , `front end`)
- `Source` : the URL(s) for any source code repositories related to this resource, separated by spaces
- `Exposure` : should specify the level of exposure the resource has to determine its attack surface area. (for example `internal` or `external`)
- `Data Classification` : should specify the highest data classification level the resource handles. This will help internal security teams to know what level of controls to apply and help focus on the resources with greatest level of risk.
- `Cost Centre` : helps the organisation's accounting or financial management system to track and allocate expenses or costs to specific departments, teams, projects, or functions

References

This is based on:

- AWS documentation and best practices on tagging (https://docs.aws.amazon.com/general/latest/gr/aws_tagging.html)
- GDS's former tagging strategy (https://docs.google.com/presentation/d/1LHLKPclfrn5KVFrFd2WqyPOYpS6wXkIE4Lexb2rJNW0/edit#slide=id.g10d43e3636_2_51)
- MoJ digital tagging strategy (https://technical-guidance.service.justice.gov.uk/documentation/standards/documenting-infrastructure-owners.html)
- the duckbill group's guide to tagging best practices (https://www.duckbillgroup.com/blog/aws-cost-allocation-guide-tagging-best-practices/)

This page was set to be reviewed before 30 October 2024 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way). This might mean the content is out of date.

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Supporting different browsers

Test your service in the browsers listed in the Service Manual (https://www.gov.uk/service-manual/technology/designing-for-different-browsers-and-devices).

You can use BrowserStack (https://www.browserstack.com/) to test browsers that run on Windows, mobile phones and tablets. Contact the 🔒 Engineering Enablement team (https://sites.google.com/a/digital.cabinet-office.gov.uk/gds/directorates-and-groups/cto-and-ciso-office/engineering-enablement#h.f8vfqmbkqtgf) to request an account. You should also test with physical devices whenever possible.

You should also test your service with assistive technologies (/manuals/accessibility.html#testing-with-assistive-technologies).

This page was last reviewed on 7 November 2024. It needs to be reviewed again on 7 November 2025 by the page owner 🔒 #frontend (https://gds.slack.com/messages/frontend) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

How to review code

Comprehensive reviews help to keep code maintainable and reusable, and reveal gaps in documentation. Use the GitHub review (https://docs.github.com/en/github/collaborating-with-pull-requests/reviewing-changes-in-pull-requests/about-pull-request-reviews) feature with pull requests (PRs) (https://docs.github.com/en/github/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests) before merging code. The author is responsible for obtaining a code review, and merging the pull request.

Good review practice

You should consider whether PRs:

- have a clear purpose
- capture the simplest way to solve a problem
- suggest changes outside a project's scope
- need breaking into smaller PRs

You should also consider if a PR's suggested changes will contribute to technical debt (/standards/technical-debt.html), and if you can make suggestions to help solve existing technical debt.

When you suggest changes, you should explain your reasoning and refer to programming language style guides (/manuals/programming-languages.html) where appropriate. You may find it useful to provide short examples to explain your suggestions.

You should:

- only approve pull requests you fully understand
- give appropriate positive feedback
- flag up major issues quickly, and in person if necessary
- ask for clarification on anything that's not clear

You should not:

- rush a review, even if it's urgent
- repeatedly point out the same error pattern
- leave comments without context

Programming style

Good code should follow the principles of its language. See Programming language style guides (/manuals/programming-languages.html) for more information.

You should consider if the code in a PR has:

- an applicable edge case
- patterns consistent with similar code elsewhere in the codebase
- readable variable names, accurately representing their contents
- missing or additional elements following a merge or rebase
- capacity for reusability

You should always check code for linting issues. You could consider running automated linting before merging PRs, for example with GitHub Actions (/standards/source-code/use-github.html#using-github-actions-and-workflows).

Code libraries

If a PR involves changes to libraries, you should check the:

- changes are backwards compatible
- version number has been updated
- changelog has been updated, especially if there are major problems

Third-party dependencies

You should make sure that new or updated third-party dependencies are from reliable and stable sources, and that they do not break any existing code. You should also make sure they are needed, and are open source (/standards/publish-opensource-code.html) wherever possible.

You can read more about how to manage third party software dependencies here (/standards/tracking-dependencies.html).

Testing

Code changes should have appropriate test coverage. You should consider running tests as part of your review, and check that:

- all possible error cases are covered
- the tests pass in all appropriate environments
- test names describe what's happening in the test

You should consider if unit tests are enough, or if you need integration tests.

Deployment

You should consider whether code changes will impact the deployment process, and check that they:

- do not adversely affect other applications and systems
- do not have any potential security issues
- will deploy properly

The Service Manual contains guidance about deploying software (https://www.gov.uk/service-manual/technology/deploying-software-regularly).

Documentation

Before you approve a PR, you should consider if it affects any existing documentation. You should make sure the PR's documentation is clear and unambiguous, and in the right place.

If you want to learn more about writing clearly for technical audiences you can contact GDS technical writers using the 🔒 #gds-technical-writing Slack channel (https://gds.slack.com/archives/CAD0R2NQG).

Further reading

Find out more about writing and reviewing pull requests on the Pull requests page (/standards/pull-requests.html).

way) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Licensing

Any public repository must include a licence that details the terms under which the code or documentation is available to be used or modified.

Language

From The Guardian and Observer style guide (https://www.theguardian.com/guardian-observer-style-guide-l):

> In British English, licence is the noun and license the verb. So you need a licence to run a licensed bar, or you may need to visit the off-licence.

Some examples of this:

- Use the British English spelling of the noun *licence*, not the US English spelling of *license* (for example “you need a licence to drive a car.”)
- When talking about the permissions that a licence grants, or the act of using a licence, use *license* (for example “your pet shop needs to be licensed.”)
- When using a proper name, use the appropriate spelling for that thing (e.g. the MIT License (https://opensource.org/licenses/MIT).)

So you would *license* your software under a particular *licence*, such as the *MIT License*.

Guidelines for repositories containing code

Specifying the licence

Each repository should include a licence file. This should be called `LICENCE` or `LICENCE.md`.

GitHub's guidance for including an open source licence in your repository (https://docs.github.com/en/communities/setting-up-your-project-for-healthy-contributions/adding-a-license-to-a-repository#including-an-open-source-license-in-your-repository) uses the US English spelling of *license*, but will still show licence details for the British English spelling.

You should specify the licence and link to it in the repository's README (/manuals/readme-guidance.html). It's typical to include this information at the very end of a README under a 'Licence' heading.

Use MIT

At GDS we use the MIT License (https://opensource.org/licenses/MIT).

Make sure the licence content is included in full, including the title "The MIT License", so that readers are quickly able to see what licence is being used.

Copyright notice

The Copyright is Crown Copyright; you can put "Government Digital Service" in brackets.

For example, `Copyright (c) 2025 Crown Copyright (Government Digital Service)`.

The year should be the year the code was first published. Where the code is continually updated with significant changes, you can show the year as a period from first to most recent update, for example 2015-2019.

For more information on copyright notices, see the UK Copyright Service fact sheet (https://copyrightservice.co.uk/copyright/p03_copyright_notices).

Example

There is a good example of a licence in the pay-adminusers (https://github.com/alphagov/pay-adminusers/blob/master/LICENCE) repo.

Guidelines for repositories that are open documentation

Some repositories will produce websites serving documentation. The GDS Way is an example of this. In addition to the MIT License for the code in the repository, you should include the Open Government Licence (OGL) (https://www.nationalarchives.gov.uk/doc/open-government-licence/version/3/) for the documentation.

Example

The GDS Way (https://github.com/alphagov/gds-way) repo is a good example of licensing open documentation.

This page was last reviewed on 10 December 2024. It needs to be reviewed again on 10 June 2025 by the page owner #gds-way (https://gds.slack.com/messages/gds-way) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

CSS coding style

Contents

Linting

Your project should conform to GDS CSS/Sass linting rules. These are published as a stylelint (https://stylelint.io) configuration: stylelint-config-gds (https://github.com/alphagov/stylelint-config-gds).

Depending on your project needs, you may complement this initial set of rules with extra linting, for example:

- other stylelint plugins (https://stylelint.io/awesome-stylelint/#plugins)
- Prettier (https://prettier.io/docs/en/cli.html) to enforce further code formatting conventions

Why: Linting ensures consistency in the codebase and picks up on well-known issues. Using an shared set of rules across GDS ensures familiar conventions from one project to another.

Tools

Stylelint

Stylelint is a widely used linter for CSS and CSS-like languages, like Sass. It focuses on checking code quality rather than code formatting. Its system of shareable configurations

allows to share sets of rules to check your project against and plugins extends the rules it provides out of the box.

`stylelint-config-gds` is a shareable config for Stylelint, providing the rules your code should follow, both for CSS and Sass files (each as separate configuration).

In addition to this shared set of rules, you can use extra stylelint plugins or add rules as needs arise during the life of your project. In that area, automatically fixable rules are especially cheap to try out, as the tools will take care of updating your code for you.

Prettier

Prettier's only preoccupation is with code formatting, not code quality (https://prettier.io/docs/en/comparison). It can be used as a complement to Stylelint for further automated formatting, with much more advanced decisions in terms of indentation, spaces, or line breaks.

It runs as a separate command (`npx prettier`) and the should be no conflicts between the Stylelint rules and the formatting of Prettier.

When to run linting

On CI

Running linting in CI ensures that all pull requests meet our code conventions before getting merged on the `main` branch. You should have this configured as part of your project.

Through pre-commit Git hooks

Waiting for CI to know if the code follows the convention can take a bit of time. A pre-commit Git hook allows to get quicker feedback, directly on developers' machines. Errors that are automatically fixable can be fixed at that stage without human intervention, reducing the effort of linting for developers.

Tools like Husky (https://typicode.github.io/husky/) and lint-staged (https://www.npmjs.com/package/lint-staged) can help consistently run linting before commit by respectively:

- setting up the hooks when dependencies get installed
- running linting on the files staged for commit and adding any fixes to the current commit

In editors

To get even quicker feedback, editor plugins can highlight issues while editing files. They can correct automatically fixable errors on save, saving further development effort.

Each of the tools previously listed has plugins to help integrate with editors:

- Stylelint editor plugins (https://stylelint.io/awesome-stylelint/#editor-integrations)
- Prettier editor plugins (https://prettier.io/docs/en/editors)

Whitespace

Use soft tabs with a two space indent.

If you're using Prettier, this will be set up for you. Otherwise, you may want to configure a `.editorconfig` file (https://editorconfig.org/) accordingly.

Why: This follows the conventions used within our other projects.

Spacing

The GOV.UK Design System spacing scale (https://design-system.service.gov.uk/styles/spacing/) should be preferred before picking custom spacing values for your project.

If you do add custom values in your project, ensure the new values are consistent with the GOV.UK Design System spacing scale (ie. are pixel values in multiples of `5px`).

Use with deprecated libraries

If your project is not using the GOV.UK Design System, you should use pixel values in multiples of `5px`.

Use `rem` units strategically

As a general rule, per Josh W. Comeau's *The Surprising Truth About Pixels and Accessibility* (https://www.joshwcomeau.com/css/surprising-truth-about-pixels-and-accessibility/#strategic-unit-deployment-6):

> When picking between pixels and rems, here's the question you should be asking yourself: Should this value scale up as the user increases their browser's default font size?

Use `rem` units for `font-size` wherever possible.

You should generally avoid using `rem` for padding or border-width if these are likely to cause layout or readability issues when the font-size increases.

Vendor prefixing

When using CSS features which require vendor prefixes use autoprefixer (https://github.com/postcss/autoprefixer).

You should configure autoprefixer (https://github.com/postcss/autoprefixer#browsers) to target our supported browsers (https://www.gov.uk/service-manual/technology/designing-for-different-browsers-and-devices#browsers-to-test-in).

Sass nesting

Sass nesting should be avoided where possible, with the exception of pseudo selectors and classes introduced by JavaScript.

While nesting can help readability and reduce repetition, over use can make searching for selectors difficult and can hide complicated CSS that should be simplified.

```
.accordion {
  // styles when the accordion is not enhanced here
}
.js-enabled {
  .accordion {
    // styles when the accordion is enhanced here

    &:focus {
      // ...
    }
  }
}
```

This page was last reviewed on 27 February 2025. It needs to be reviewed again on 27 February 2026 by the page owner 🔒 #frontend (https://gds.slack.com/messages/frontend) .

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Dockerfile guidance

This style guide:

- provides some conventions for creating production-ready Dockerfiles at GDS
- supplements the official Dockerfile reference (https://docs.docker.com/engine/reference/builder/)

Why we use Dockerfile

The Open Container Initiative (https://opencontainers.org/) started in June 2015, and is an open governance structure for creating open standards around container formats. Docker were one of the first movers in this space. Their documentation defined the de facto standard and then delegated to the OCI.

The OCI's documentation on Containerfile (https://github.com/containers/common/blob/main/docs/Containerfile.5.md) is derived from Docker's Dockerfile documentation.

Using tags and digests in FROM instructions

The `FROM` instruction specifies the starting image for your Docker image build.

A tag is a short label you can use to reference an image, often referencing a version number.

As you cannot rely on the tag pointing to the exact same image over time, you should instead use a digest, which identifies the image by a hash of its contents. This makes sure that you are always referencing the image that you expect.

To get the digest, run `docker pull <tag>`. For example:

```
$ docker pull alpine:3.9
3.9: Pulling from library/alpine
```

```
Digest: sha256:769fddc7cc2f0a1c35abb2f91432e8beecf83916c421420e6a6da9f897546
Status: Image is up-to-date for alpine:3.9
```

As Dependabot (https://dependabot.com) has support for updating `FROM` lines which use digests (https://github.com/dependabot/dependabot-core/pull/100), you can still use Dependabot to keep your images up-to-date.

If you specify both the tag and the digest, then the digest takes precedence. The standard implies that you may only specify one or the other. Do not specify both as the behaviour may vary across different platforms.

Instead, the recommended way to keep track of the intended version is to add it in a line above FROM as a comment. For example:

```
# alpine:3.9
FROM alpine@sha256:769fddc7cc2f0a1c35abb2f91432e8beecf83916c421420e6a6da9f89
...
```

Please double-check that the digest and the commented version are consistent each time you upgrade, as Dependabot does not have perfect capability at either identifying the magnitude of the upgrade. It will also not update the comment automatically, so this will need modifying on any dependabot-raised pull requests.

Using multi-stage builds

Using multi-stage builds (https://docs.docker.com/develop/develop-images/multistage-build/) enables the drastic reduction of image sizes, which in turn decreases the time taken to launch the container. There can be many stages within a Docker□e. The result is a single layer image which discards the previous unrequired layers that were used in the compilation steps.

As an example;

```
# golang:bullseye
FROM golang@sha256:ecb3fe70e1fd6cef4c5c74246a7525c3b7d59c48ea0589bbb0e57b1b3
WORKDIR /go/src/github.com/alphagov/paas-aiven-broker/
RUN git clone https://github.com/alphagov/paas-aiven-broker.git .
RUN go mod download
RUN go build

# alpine:3.9
FROM alpine@sha256:769fddc7cc2f0a1c35abb2f91432e8beecf83916c421420e6a6da9f89
```

```
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /go/src/github.com/alphagov/paas-aiven-broker/paas-aiven
COPY --from=builder /go/src/github.com/alphagov/paas-aiven-broker/paas-aiven
CMD ["./paas-aiven-broker", "-config", "config.json"]
```

Building from this Dockerfile requires no changes to the existing build process e.g. `docker build -t myimage:latest .`

It is also possible to stop the build at a specific stage using a command such as `docker build --target builder -t myimage:development .` which then enables running the container locally to debug the image.

Running programs as process ID (PID) 1

The program running as PID 1 inside a container is responsible for:

- cleaning up orphaned child processes
- handling signals
- returning the exit status from the container

Most programs are unsuited to running as PID 1 inside a container. For example:

- `bash` will not pass signals through to its children; for example, `SIGTERM` will not lead to the container being shut down
- `java` exits with an exit status of 143 when sent a SIGTERM, even if the application shuts down cleanly
- `node` does not reap orphaned child processes whose parent has exited

Tini (https://github.com/krallin/tini) provides a program suitable for running as PID 1 inside the container. You can use Tini to avoid the problems highlighted above. Tini is included by default with the Docker runtime or Alpine Linux images.

You can use `tini` by passing the `--init` option to Docker when running your container or set Tini as the `ENTRYPOINT` for your container. For example:

```
ENTRYPOINT ["tini", "--"]
```

or for Java programs, to map an exit status of 143 to 0:

```
ENTRYPOINT ["tini", "-e", "143", "--"]
```

Subshells

The instructions `RUN`, `CMD` and `ENTRYPOINT` have 2 forms:

- freeform text (for example `CMD "run -x"`)
- an array-style (for example `CMD ["run", "-x"]`)

You should use the array-style syntax where possible.

A Linux syscall will directly execute all commands specified using the array-style syntax, without an enclosing subshell. This process is more efficient and removes any ambiguity over how the commands will be interpreted.

In the case of `ENTRYPOINT` or `CMD`, using the freeform text syntax means that a shell becomes PID 1 and most programs should not run as PID 1, as explained above.

For more information about the special role of PID 1:

- avoid running NodeJS as PID 1 under Docker images (https://www.elastic.io/nodejs-as-pid-1-under-docker-images/)
- docker-node best practices - Handling Kernel Signals (https://github.com/nodejs/docker-node/blob/main/docs/BestPractices.md#handling-kernel-signals)
- Docker and the PID 1 zombie reaping problem (https://blog.phusion.nl/2015/01/20/docker-and-the-pid-1-zombie-reaping-problem/)

Links

- A smarter Dockerfile linter (https://github.com/hadolint/hadolint) that helps you build best practice Docker images
- `conftest` policy to validate the label rules (https://github.com/deanwilson/opa-policies#gdsway-dockerfile)

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Go style guide

> Note: Go is no longer a core language (/standards/programming-languages.html#go) for backend development in GDS.

The purpose of this style guide is to provide some conventions for working on Go code within GDS. There are already good resources on writing Go code, which are worth reading first:

- Effective Go (https://golang.org/doc/effective_go)
- Code Review Comments (https://github.com/golang/go/wiki/CodeReviewComments) (documenting points that have been raised in Google code reviews)

Code formatting

Use `gofmt` (https://golang.org/cmd/gofmt/). This means all Go code reads in the same way which is important when looking at unfamiliar code (https://blog.golang.org/go-fmt-your-code).

You may also want to use `golint` (https://github.com/golang/lint) which assesses code style.

Code checking

`go vet` (https://golang.org/cmd/vet/), which checks correctness, should be used as part of your build process.

If you are writing concurrent code, use the race detector (https://blog.golang.org/race-detector) to detect race conditions.

External dependencies

As of Go 1.11 there is an officially developed module system (https://blog.golang.org/using-go-modules), which is finalized in Go 1.13: `go mod`.

A historical pain point of development in Go was the manipulation of `$GOPATH` which is no longer required when using `go mod`.

There are several Go depency management tools in use at GDS, other than `go mod`:

- dep (https://golang.github.io/dep/) (no longer in development)
- godep (https://github.com/tools/godep) (no longer in development)
- glide (https://github.com/Masterminds/glide) (no longer in development)
- gopkg.in (https://labix.org/gopkg.in), which provides a method of using versioned import paths
- vendoring (ie copying source code in some manner to a location you control)

All new projects should use the officially developed `go mod` unless they have specific requirements not met by `go mod`

Vendoring is still common practice in Go, but if you are using `go mod` with Go 1.13+ then the go command will download and authenticate modules from the highly available Go module mirror (https://pkg.go.dev/cmd/go#hdr-Configuration_for_downloading_non_public_code) and Go checksum database run by Google which alleviates the need to vendor your modules

Web frameworks

While it's difficult to provide any guidance that will be generally applicable, there are couple of useful things to consider when structuring your Go program.

The first is that Go's standard library is modern and powerful. If you just need simple HTTP routing and handling, the `net/http` package will probably meet your needs (https://golang.org/doc/articles/wiki/).

The second is that if `net/http` falls short, it's worth choosing something that integrates well with it.

The third is that Go is not a language we use at GDS for developing fully-fledged web applications, eg rendering HTML. If you find yourself rendering HTML using Go, consider using a different GDS supported language.

Channels

Signalling

Channels that are being used purely for signalling should use an empty struct rather than boolean or int types.

Using an empty struct declares that we're not interested in the value sent or received; only in its closed property.

See this talk (https://talks.golang.org/2012/10things.slide#11)

```
func worker(quit <-chan struct{}, result chan<- int) {
  for {
    select {
    case result <- rand.Intn(10000000):
    case <-quit:
      return
    }
  }
}

func main() {
  quit, result := make(chan struct{}), make(chan int)
  for i := 0; i < 100; i++ {
    go worker(quit, result)
  }
  // Wait for a worker to return a good result
  for {
    if <-result > 9999998 {
      break
    }
  }
  close(quit) // terminate all the workers
  fmt.Println("All done!")
}
```

Testing

There is some use across GDS of gomega (https://onsi.github.io/gomega/) for matching, and ginkgo (https://onsi.github.io/ginkgo/) for BDD testing.

If you find yourself writing too much boilerplate when testing Go, consider reaching for these libraries.

Gomega can help make your tests more readable:

```
err := doAThing() // this should fail
Expect(err).Should(HaveOccurred())
```

And Ginkgo can help set up a test suite:

```
var _ = Describe("Something", func() {
  It("should do a thing", func() {

    Expect("hi").To(HaveLen(2))

  })
})
```

which can be run with:

```
ginkgo
```

Tests can be focused (using regular expressions), to speed up development:

```
ginkgo -focus 'a thing'
```

Configuration parsing

There are a number of tools in use at GDS for parsing configuration and arguments:

- alecthomas/kingpin (https://github.com/alecthomas/kingpin) (not actively maintained) for parsing simple command line arguments and environment variables
- urfave/cli (https://github.com/urfave/cli) for building more advanced CLI tools
- spf13/viper (https://github.com/spf13/viper) for parsing configuration files

These tools can make it easier to accept configuration parameters and help to make your application self-documenting.

This page was last reviewed on 23 April 2024. It needs to be reviewed again on 23 April 2025 by the page owner 🔒 #golang (https://gds.slack.com/messages/golang) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

HTML coding style

Browser support

The GOV.UK Service Manual lists the minimum browsers that GOV.UK services are required to support (https://www.gov.uk/service-manual/technology/designing-for-different-browsers-and-devices#browsers-to-test-in). Tools such as GOV.UK Frontend, that are used to help build services, may have stricter standards of browser support (https://github.com/alphagov/govuk-frontend#browser-support). Therefore, you will need to test your code in different browsers depending on which part of GDS you are working in.

Your page should work with only HTML, before adding any CSS or JavaScript. Read building a resilient frontend using progressive enhancement (https://www.gov.uk/service-manual/technology/using-progressive-enhancement).

Document structure

Use HTML5 to structure your page semantically. Use ARIA roles (https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Roles) to help older assistive technologies map the sections correctly.

Léonie Watson explains how to use ARIA landmark roles to deliver the best experience for screen reader users (https://tink.uk/screen-readers-aria-html5-too-much-information/).

This document does not prescribe whether to use single or double quotes in attributes, whether to omit values from attributes that do not need them or whether to add trailing slashes to void elements. However, the approach you adopt should be consistent throughout your code.

Header

`<header role="banner">` should be used to contain your home link, branding, search, and any global navigation you may have.

Main content

`<main role="main">` should be used to identify the main content of your page.

A “Skip to main content” link should be added to help screen reader and keyboard users skip past your general site navigation and get straight to the content. You can use the skip link component from the GOV.UK Design System (https://design-system.service.gov.uk/components/skip-link/) for this.

Navigational elements

Use `<nav role="navigation">` to wrap groups of links that are not already in another context (for example, `<footer>`). Common examples of navigation sections are menus, tables of contents, and indexes.

If you use more than one `<nav>` block in your page it is advisable to give each one an accessible name (https://www.tpgi.com/what-is-an-accessible-name/) using aria-labelledby (https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Attributes/aria-labelledby) or aria-label (https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Attributes/aria-label), to make clear the type of navigation contained by each.

The GOV.UK blog has more information regarding the use of the `<nav>` tag (https://insidegovuk.blog.gov.uk/2013/07/03/rethinking-navigation/).

Aside

`<aside role="complementary">` should be used for content related to the primary content of the webpage, but which does not constitute the primary content of the page.

Examples include author information, related links and related content.

Footer

`<footer role="contentinfo">` should be used for the footer of the site.

Region landmarks

Use region landmarks to surface important areas of a page where it’s not appropriate to use any of the other types of landmarks.

You can create region landmarks by using the `<section>` tag. The `<section>` must have an accessible name (https://www.tpgi.com/what-is-an-accessible-name/), which can be provided using the aria-labelledby (https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Attributes/aria-labelledby) or aria-label (https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Attributes/aria-label) attributes. A

`<section>` tag without an accessible name is semantically the same as a `<div>` tag (https://w3c.github.io/html-aam/#el-section).

Region landmarks should only be used where users will likely want to be able to navigate to the grouping easily and to have it listed in a summary of the page.

You can already use headings and the other landmarks to identify sections of the page. Region landmarks should be used only when those are proven not to work well enough and doing so saves users more effort than it adds.

Individual element guidance

Headings

Headings should be in order - for example `<h1>` then `<h2>`, but not `<h1>` then `<h3>`.

There should only be one `<h1>` element in a page.

Text emphasis

Italics should be avoided according to the GOV.UK content style guide (https://www.gov.uk/guidance/style-guide).

Bold can be used sparingly, but can make large blocks of text difficult to read.

Semantically, words voiced with emphasis can be marked up with `<em>`, whereas words conveying a sense or urgency or warning should be marked up with `<strong>`. In theory screen readers could adopt a different tone of voice for this markup, though none of the major ones currently do.

Browsers will render `em` in italics and `strong` in bold by default, so you may need to override `em` to not use italics.

As a rule, avoid using `<i>` or `<b>`, which are only useful in rare cases. It is usually better to use the `font-style` and `font-weight` CSS properties.

Images

Read the Design System's Images section (https://design-system.service.gov.uk/styles/images/).

Buttons vs links

Links should be used for navigating to another page. Buttons should be used for submitting forms or interactions within the page (for example, expanding an element).

Empty links (`<a href="#">`) should be avoided.

Links should not open new tabs or windows by default, but if users choose to do so (through keyboard shortcuts or right-click context menu) then we should respect their choice.

Visually hidden elements

Sometimes it can be helpful to provide additional content for screen reader users that might not be needed for sighted users. For example, the 'Skip to main content' link covered earlier.

Do not use `display: none` on this text, as this will get ignored by screen readers.

Instead, use CSS which ensures the text gets 'conceptually' rendered, even if the result is not visible on the screen. You can use the Sass mixins from govuk-frontend (https://github.com/alphagov/govuk-frontend/blob/master/src/govuk/utilities/_visually-hidden.scss) or the classes `govuk-visually-hidden` and `govuk-visually-hidden-focusable` .

Visually hidden text is often a sign that something can be simplified or made visible to benefit sighted users too, so should be used sparingly.

This page was set to be reviewed before 2 May 2024 by the page owner 🔒 #frontend (https://gds.slack.com/messages/frontend). This might mean the content is out of date.

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Style guides

You should read this guide alongside the programming language recommendations (/standards/programming-languages.html).

Code style guides

Developers read code much more often than they write it. These guidelines are intended to improve the readability of code and make it consistent across GDS projects.

A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.

But most importantly: know when it's ok to be inconsistent. Sometimes the style guide just does not apply. When in doubt, use your best judgement. Look at other examples and decide what looks best. And do not hesitate to ask if you're unsure.

Some good reasons to ignore a particular guideline include:

- when applying the guideline would make the code less readable, even for someone who is used to reading code that follows this style guide
- to be consistent with surrounding code that also breaks it (maybe for historic reasons) – although this is also an opportunity to clean up the existing code
- when the code in question is older than the introduction of the guideline and there is no reason to modify that code
- when the code needs to remain compatible with older versions that do not support the feature recommended by the style guide

We've got a consistent style for:

- CSS/Sass (/manuals/programming-languages/css.html)
- Docker (/manuals/programming-languages/docker.html)
- Go (/manuals/programming-languages/go.html)
- HTML (/manuals/programming-languages/html.html)

- Java (/manuals/programming-languages/java.html)
- JavaScript (/manuals/programming-languages/js.html)
- Node.js
- Python (/manuals/programming-languages/python/python.html)
- Ruby (/manuals/programming-languages/ruby.html) (tested with RSpec (/standards/testing-with-rspec.html))

Some of the guidelines in the style guides are codified in a .editorconfig (/manuals/programming-languages/editorconfig) file. Place a copy of this file in your project's repository to have tooling that supports EditorConfig (https://editorconfig.org/) automatically meet the guidelines.

This page was last reviewed on 6 November 2024. It needs to be reviewed again on 6 November 2025 by the page owner #gds-way (https://gds.slack.com/messages/gds-way) .

View source Report problem GitHub Repo

Accessibility

OGL

All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Java style guide

The purpose of this style guide is to provide some conventions for working on Java code within GDS.

The Google Java style guide (https://google.github.io/styleguide/javaguide.html) is a good starting point. The old Sun/Oracle Java style guide (https://www.oracle.com/java/technologies/cc-java-programming-language.html) is still largely relevant but it has not been updated since 1999 and does not reflect recent changes to the language.

The more far-reaching *Java for Small Teams (https://ncrcoe.gitbooks.io/java-for-small-teams/content/)* ebook can be read online at no cost. While not free, *Effective Java (https://www.pearson.com/en-gb/subject-catalog/p/effective-java/P200000000138/9780134685991/)* by Joshua Bloch (Addison-Wesley, 2017) is also an excellent resource. The GDS Library has physical copies of the book or you can 🔒 buy a copy using the Learning and Development budget (https://sites.google.com/a/digital.cabinet-office.gov.uk/gds/learning-and-development/apply-to-use-the-learning-and-development-budget).

While the above resources are good guides, they may conflict with either each other or your team's established practices. We favour consistency across our code, so make sure that you have the agreement of your team when considering using a new or different method or paradigm to those currently in use.

We generally use IntelliJ IDEA (https://www.jetbrains.com/idea/) within GDS. Using it consistently helps when ensemble programming (pairing and mobbing). GDS have purchased licences for the commercial editions of IntelliJ IDEA for some teams. Some features of IntelliJ IDEA like Code With Me (https://www.jetbrains.com/code-with-me/) and JetBrains AI (https://www.jetbrains.com/ai/) have not passed an information assurance review and must not be used (Full Line code completion (https://www.jetbrains.com/help/idea/full-line-code-completion.html) can be used because it runs entirely on your device).

Code formatting

Variable and field names should match the names of their types (for example, an `InputStream` could be named `inputStream` or `is`), or have descriptive names reflecting

the context of their use (for example, a `Set<String>` where each `String` is a username could be named `usernames`).

Always use curly braces around the body of an `if`, `else`, `for`, `do` or `while` statement, even if it's only a single line. Follow the Kernighan and Ritchie (K&R) 'Egyptian brackets' style where there is no line break before the opening brace. See the braces section of the Google Java style guide (https://google.github.io/styleguide/javaguide.html#s4.1-braces) for more details.

Use the GDS Way EditorConfig file (/manuals/programming-languages/editorconfig), which has settings for things like code indentation. Place a copy of this file named `.editorconfig` in the root of your project to have IntelliJ IDEA and some other editors automatically apply the settings. If your editor does not support EditorConfig, manually configure its settings to match.

Some Java teams within GDS have had success using the Spotless (https://github.com/diffplug/spotless) auto-formatter. Some of the formatting styles supported by Spotless are quite opinionated and may want to make lots of changes if added to an existing project. The `ratchetFrom` option makes Spotless only format changed files (though this can require it to check out a lot of code in some cases, which may have performance implications if used as part of a build pipeline). Alternatively, you may wish to configure Spotless to match your existing conventions. For a new project, it's probably easiest to use a style's default settings.

Java EE and Jakarta EE

Java EE (https://www.oracle.com/java/technologies/java-ee-glance.html) (Enterprise Edition), a collection of APIs used by many server-side Java applications, was spun out from Oracle and handed over to the Eclipse Foundation (https://www.eclipse.org/), who renamed it Jakarta EE (https://jakarta.ee/) due to not having the rights to the Java trademark (Jakarta is the largest city on the island of Java).

Beginning with version 9, Jakarta EE switched from using the `javax` package namespace to the `jakarta` package namespace, instantly breaking all applications that referenced the old package names. (Be aware that other APIs — including some that are part of Java Standard Edition — also use the `javax` package namespace.)

Migrating from Java EE to Jakarta EE is hard. Many libraries and frameworks use Java EE or Jakarta EE so migrating completely may involve updating many of your dependencies to new versions that work with Jakarta EE (if they exist).

Some libraries and frameworks make this a little easier by having two parallel versions, one that works with Java EE and one that works with Jakarta EE, while being otherwise equivalent. This makes it possible to upgrade to the latest Java EE version of a dependency and then migrate across to its Jakarta EE equivalent in two separate steps.

If you are starting a new project, use Jakarta EE only from the outset unless you have a compelling reason not to.

Dependency injection (DI)

Consider whether a dependency injection framework is appropriate for your project before using it.

Some programmes use the Guice (https://github.com/google/guice) dependency injection framework. There are two current versions of Guice: Guice 6.0.0 (https://github.com/google/guice/wiki/Guice600) supports Java EE and the `javax` package namespace while Guice 7.0.0 (https://github.com/google/guice/wiki/Guice700) supports only Jakarta EE and the `jakarta` package namespace.

Imports

Wildcard imports should be avoided. They can cause existing code to break if a new type is added to a package with the same name as a type in another package.

This infamously happened with Java SE 1.2, which introduced `java.util.List` when there was already `java.awt.List`, breaking any classes that used `List` and imported both `java.util.*` and `java.awt.*`.

You should configure IntelliJ to explicitly import all classes and static methods in Settings → Editor → Code Style → Java → Imports with “Class count to use import with *” and “Names count to use static import with *” both set to a very high number, for example 1000.

Static imports should be avoided in most cases because the names of methods and constants often do not make sense without the context of the type they’re from.

Static imports are appropriate in some cases. Tests often read more fluently when assertion and matcher methods, which are well known and widely understood, are statically imported. For example, compare:

```
Assert.assertThat(actual, Is.is(expected));
```

… to:

```
assertThat(actual, is(expected));
```

Similarly, `Math.max(…)`, `Math.PI` and `ChronoUnit.DAYS` could be statically imported because their names make sense on their own. However, `LocalDate.of(…)` should not be statically imported because the type it comes from is crucial for comprehension.

The IntelliJ 'Optimize Imports' (Ctrl+Option+O) command (in the Code menu) sorts imports and removes any that are unused. Before committing, you can select all the changed classes (for example, in the changes view (https://www.jetbrains.com/help/idea/managing-changelists.html) of the commit tool window (https://www.jetbrains.com/help/idea/tool-windows.html) or the modified files pane of the commit dialogue (https://www.jetbrains.com/help/idea/commit-changes-dialog.html)) and then use this command to fix the imports for just the classes you modified.

Optionals

If a getter may return `null`, you should usually return an `Optional` instead.

`Optional` is intended to only to represent the absence of a result: do not use it for fields or method parameters. Java language architect Brian Goetz posted a StackOverflow answer explaining the intent of `Optional` (https://stackoverflow.com/questions/26327957/should-java-8-getters-return-optional-type/26328555#26328555) with further reasoning.

You almost never need to use the `isPresent()` or `isEmpty()` methods on an `Optional`. Use `ifPresent(…)`, `ifPresentOrElse(…)`, `map(…)` or `flatMap(…)` instead. See DZone Java Zone's article *Optional isPresent() Is Bad for You (https://dzone.com/articles/optional-ispresent-is-bad-for-you)* for more details.

Optionals work best when used in a functional style, which can take time to learn. The DZone Java Zone article *26 Reasons Why Using Optional Correctly Is Not Optional (https://dzone.com/articles/using-optional-correctly-is-not-optional)* has some guidance.

Local variable type inference (the var keyword)

You can use `var` in Java 10 onwards with local variables to have the compiler infer the type. This is especially good when invoking a constructor:

```
// Java 9
ByteArrayOutputStream outputStream = new ByteArrayOutputStream();

// Java 10+
var outputStream = new ByteArrayOutputStream();
```

It can also remove duplication where the return type and the variable name are the same:

```
// Java 9
CheckResponse checkResponse = service.getCheckResponse();

// Java 10+
var checkResponse = service.getCheckResponse();
```

Be mindful that `var` hides the type of the variable. If the variable name makes the type obvious, this is usually not a problem. But if it is not clear from either the variable name or the right-hand side of the assignment, it might be better to explicitly write the type.

If you are using the diamond operator in an assignment, you will usually find that updating it to use `var` also requires you to replace the diamond operator with the appropriate generic type parameter:

```
// usernames is Set<String>
Set<String> usernames = new HashSet<>();

// usernames is HashSet<Object> (probably not what you want)
var usernames = new HashSet<>();

// usernames is HashSet<String>
var usernames = new HashSet<String>();
```

The OpenJDK project has some style guidelines for local variable type inference (https://openjdk.org/projects/amber/guides/lvti-style-guide). Oracle's guide to local variable type inference (https://docs.oracle.com/en/java/javase/21/language/local-variable-type-inference.html) also contains some recommendations.

Records

Java 16 introduced record classes, which provide an excellent way to model immutable data with a concise syntax that eliminates the need to write tedious and error-prone code for basic functionality. Records also support pattern matching. Dev.java has a good introduction to records. (https://dev.java/learn/records/)

Prefer functionality in the Java standard library

Where possible, use functionality from the Java standard library rather than external libraries.

Keep in mind that improvements to the Java standard library mean that some external libraries which were popular in the past now add less value. For example, while Joda-Time was significantly better than the date and time libraries built into older Java versions, the `java.time` package introduced in Java 8 renders it redundant. Similarly, Google's Guava (https://github.com/google/guava) is very useful (and recommended) but the unmodifiable collections built in to Java 9 largely remove the need for Guava's immutable collections. The HttpClient (https://docs.oracle.com/en/java/javase/11/docs/api/java.net.http/java/net/http/HttpClient.html) introduced in Java 11 is good enough that you might not need a third party one.

When using external libraries, favour those which complement the Java standard library. For example, consider a library that introduces an object that behaves like a list. If this type does not implement `java.util.List`, it will not automatically benefit from the standard library's support for features like streams.

Make sure any external library you use is appropriate for your purposes and avoid relying on internal implementation details of external libraries. If your IDE's code completion suggests a method from an external library, make sure it's a supported part of the library's defined API.

Comments

Agree with your team to what extent you permit comments in your code. Some teams look more favourably on method-level and class-level Javadoc describing the context and responsibility of code, rather than inline comments.

It's often possible to make code more readable so it does not need comments (https://ncrcoe.gitbooks.io/java-for-small-teams/content/style/20_prefer_readable_code_to_comments.html). The book *Clean Code (https://www.pearson.com/en-gb/subject-catalog/p/clean-code-a-handbook-of-agile-software-craftsmanship/P200000009044/9780132350884)* by Robert C Martin (Pearson, 2008) has some advice on how to do this.

Comments rarely need to describe *what* some code is doing or *how* it's doing it. Comments explaining *why* the code is doing something, particularly if it's non-obvious or requires external contextual knowledge, may be helpful.

When deciding whether or not something is non-obvious enough to require a comment, bear in mind that you are probably more familiar with the code you're writing than most people who read it will be. Questions and misunderstandings from code reviewers can act as a guide (but keep in mind that other remedies such as renaming variables or methods may be better than adding comments).

Outdated comments can be worse than no comments at all because they may be misleading. Consider this before adding a comment and always keep comments up to date when changing code.

As an alternative to comments, commit messages can be a good place for describing why a particular piece of code was added, removed or modified. Tools like `git-blame` are built into most IDEs, allowing a developer to easily pull up the commit messages for revisions to a particular line of code. The age of each commit provides an indication of whether the commit message may be outdated.

Testing

Use JUnit for unit tests. It can also be used for many integration tests.

The latest JUnit 5 (https://junit.org/junit5/) is not a drop-in replacement for JUnit 4 (https://junit.org/junit4/). The JUnit Vintage test engine (https://junit.org/junit5/docs/current/user-guide/#migrating-from-junit4) makes it possible to run JUnit 4 tests within JUnit 5, though not all JUnit 4 features are supported. If you use lots of JUnit 4 rules and alternative runners, or rely on other testing libraries that integrate with JUnit 4, you will have to make more changes. Teams within GDS have found it typically takes a few days to upgrade a project from JUnit 4 to JUnit 5 (keeping the test classes as JUnit 4 where possible). JUnit 4 continues to be maintained.

When you upgrade from JUnit 4 to JUnit 5, have a strategy for gradually making more and more of your test classes use JUnit 5. A good rule is to write all new test classes in JUnit 5 and to migrate over existing ones when you need to make major changes to them (generally it's best to do the migration in a separate pull request first before making other changes).

For new projects, use JUnit 5 unless you have a reason not to.

Use Mockito (https://site.mockito.org/) for mocking. If you're still using an older version, upgrading should be fairly straightforward: Mockito 3 contains no breaking changes, Mockito 4 only removes deprecated APIs and Mockito 5 mainly changes internals.

Code checking

We encourage the use of static analysis. Static analysis tools for Java include CodeQL (https://codeql.github.com/), SonarQube Server (https://www.sonarsource.com/products/sonarqube/), Codacy (https://www.codacy.com/) and CheckStyle (https://checkstyle.sourceforge.io/). However, be aware that such tools can detect an overwhelming number of problems if applied to an existing project, which tends to result in their checks being ignored.

If possible, configure your static analysis tools using configuration files and keep these in your project repositories. This makes your settings more portable and makes it easier to perform the same checks in different places (for example, in a cloud service and on your own computer).

Some cloud-based static analysis tools, including Codacy, can work directly with GitHub repositories. Only grant a tool access to your repositories if it has been approved by an information assurance review. The Digital Identity programme has approval to use SonarQube Cloud (https://www.sonarsource.com/products/sonarcloud/) on public repos.

Try to minimise compiler warnings. If you cannot remove a warning, use an appropriate annotation to suppress it, preferably with a comment explaining why. For example:

```
public void frobulateFoos() {
    LOGGER.info("Frobulating foos");

    // LibFoo returns a raw list but every element is always a Foo
    @SuppressWarnings("unchecked")
    List<Foo> foos = (List<Foo>) fooService.getFoos();

    foos.forEach(Foo::frobulate);
}
```

Dependencies

Teams should agree their own rules for how to approve new dependencies and where dependencies can retrieved from (for example, Maven Central (https://central.sonatype.com/)). When deciding whether or not to add a new dependency, consider its trustworthiness, longevity, licence and whether it appears to be actively maintained.

Try to keep up to date with the latest versions of your external dependencies. Older dependencies often contain security vulnerabilities. If you wait to upgrade your dependencies, you may find you have to make large version jumps to lots of dependencies at once, which can be painful. Frequent, smaller updates are almost always preferable.

Dependabot (https://docs.github.com/en/code-security/dependabot) (which is part of GitHub) can automatically open pull requests to upgrade your libraries and other dependencies. Be aware that not all dependency upgrades are backwards compatible. Major version upgrades are more likely to cause problems than minor upgrades. Dependabot provides compatibility scores and links to release notes, which can help you make an informed decision. Do not merge a dependency upgrade unless it passes your automated tests. If it's a major upgrade, monitor for any problems after it's deployed.

Dependabot makes assumptions about version numbers that not all dependencies follow. This can cause it to open pull requests that update to beta versions, release candidates or

even alternative variants of the dependency. Always have a human sense-check each Dependabot pull request before merging.

If you are not ready to upgrade to a particular version of a dependency, tell Dependabot to ignore that version by using a dependabot.yml (https://docs.github.com/en/code-security/dependabot/dependabot-version-updates/configuration-options-for-the-dependabot.yml-file) file. A comment in the file explaining why it has been ignored can be useful. Using a dependabot.yml file is preferable to using `@dependabot ignore` commands because it's much more visible and it's also easier to tell Dependabot to stop ignoring a version upgrade.

There have been cases of bad actors raising malicious pull requests on repositories that appear to come from Dependabot. Make sure a pull request really comes from Dependabot before merging.

Dependabot only deals with direct dependencies, not transitive dependencies. Even if Dependabot thinks your project's dependencies are fully up to date, you may still have outdated transitive dependencies (which may contain security vulnerabilities).

When viewing a Maven POM file, IntelliJ IDEA can warn you if your transitive dependencies have reported security vulnerabilities.

Build tools

You should use either Gradle (https://gradle.org/) or Maven (https://maven.apache.org/) as the build tool. Use recent versions if you can. Maven now supports the Bill of Materials (BOM) (https://www.jvt.me/posts/2021/08/28/java-bom/) concept, which can simplify dependency management (Gradle also supports Maven BOMs).

Web frameworks

The Dropwizard (https://www.dropwizard.io/) web framework is used widely within GDS.

Dropwizard 3.0.0 and Dropwizard 4.0.0 were released simutaneously in March 2023. They have equivalent functionality but Dropwizard 3 continues to use Java EE and the `javax` package namespace while Dropwizard 4 migrates to Jakarta EE (https://jakarta.ee/) and the `jakarta` package namespace (Jakarta EE is the successor to Java EE).

Dropwizard 2 is no longer supported as of 31 January 2024. (https://github.com/dropwizard/dropwizard/discussions/7880) If you are still using Dropwizard 2, upgrade to Dropwizard 3 (https://www.dropwizard.io/en/release-3.0.x/manual/upgrade-notes/upgrade-notes-3_0_x.html) rather than trying to upgrade directly to Dropwizard 4.

If you are using Dropwizard 3, you can consider trying to upgrade to Dropwizard 4 (https://www.dropwizard.io/en/release-4.0.x/manual/upgrade-notes/upgrade-notes-4_0_x.html). You

may find you have to upgrade some (but not all) of your other dependencies from versions that use Java EE to versions that use Jakarta EE.

Dropwizard has built-in support for validating requests with Hibernate Validator. Use Dropwizard's validation (https://www.dropwizard.io/en/stable/manual/validation.html) in preference to rolling your own except in cases where Dropwizard's built-in functionality cannot meet your validation requirements.

JDK

Use long-term support (LTS) releases

New major versions of the JDK are released twice a year (in March and September). Every two years, Oracle release a long-term support (LTS) version, which is maintained and receives updates for several years. Most other Java vendors follow Oracle's lead, though their exact support lifecycles vary. Non-LTS releases usually stop receiving updates as soon as the next major version comes out. Within the Java ecosystem, many libraries and tools only really support and test against LTS releases.

For this reason, it is highly recommended to use an LTS release of Java. If you choose to use non-LTS releases, be aware that you will have to perform major version upgrades every six months and risk being stranded if libraries and tools you rely on are not prompt in adding support for the latest Java release.

If you are starting a new Java project, do not use anything older than the latest LTS release unless you have a good reason (for example, compatibility issues).

If you are currently using an older LTS release, you should be planning to upgrade to a later LTS version. There is usually an overlap of several years when both the current and some previous LTS releases are supported. Use this time wisely and balance the risk of upgrading before your tools and dependencies fully support a new LTS release against the risk of leaving the upgrade to the last minute. Bear in mind that your dependencies may stop supporting an LTS release before your JDK vendor does.

In general, upgrading between recent Java versions is a lot easier than it was a few years ago, due to better encapsulation within the JDK itself (making it harder for libraries to depend on implementation details that change from version to version) and because the ecosystem has gotten used to the rapid release schedule.

JDKs from different vendors

Recent versions of the Oracle JDK can be used free of charge (https://www.oracle.com/downloads/licenses/no-fee-license.html) for commercial and production purposes under the terms of a bespoke licence. OpenJDK is open source under the GPLv2 with the Classpath Exception (https://openjdk.org/legal/gplv2+ce.html) but Oracle only provide general-availability OpenJDK builds (https://jdk.java.net/) for the latest release.

The Adoptium (https://adoptium.net/) (formerly AdoptOpenJDK) project (part of the Eclipse Foundation (https://www.eclipse.org/)) provides fully open-source TCK-certified pre-built OpenJDK binaries under the name Eclipse Temurin (https://projects.eclipse.org/projects/adoptium.temurin). For LTS releases (such as Java 8, 11, 17 and 21), Adoptium have committed to releasing free updates for several years.

In addition, Temurin has benefits such as being available in package repositories, having friendly installers for desktop use, and offering ready-made Docker images containing OpenJDK (https://hub.docker.com/_/eclipse-temurin). It's easy to install Temurin on your computer (https://adoptium.net/en-GB/installation/) for development purposes using Homebrew (https://brew.sh/) or similar.

Amazon Corretto (https://aws.amazon.com/corretto/) is a free OpenJDK distribution. The AWS Lamdba runtimes for Java (https://docs.aws.amazon.com/lambda/latest/dg/lambda-java.html) use Corretto. Newer Java versions have significantly faster cold start times, which can be beneficial for Lambda.

Publishing artifacts

We recommend publishing artifacts (for which the source is already public) to Maven Central (https://central.sonatype.com/).

You should *not* use Maven Central to publish artifacts for which the source is closed or contains other proprietary assets, as it is a public repository with anonymous access. If you need to publish private artifacts for internal use, consider running your own Sonatype Nexus Repository (https://www.sonatype.com/products/sonatype-nexus-repository).

Claiming group IDs

You will need to claim one or more group IDs (https://maven.apache.org/guides/mini/guide-naming-conventions.html) in order to publish an artifact to Maven Central. The central repository is run and operated by Sonatype, and you will need to register for an account on their Jira instance to request control of a group ID.

The credentials used to create this account will be used during the publishing process, and should be stored safely and in accordance with any programme-specific guidance.

You should follow Sonatype's guidance on registering for and claiming a group ID (https://central.sonatype.org/publish/requirements/coordinates/) such as `uk.gov.example`. It is likely that you will be asked to prove your identity as a government actor, to prevent malicious parties publishing artifacts and claiming that they have been issued by the UK government.

Signing artifacts

Sonatype requires that artifacts published to their repository have been signed with a published GPG key. Each programme that wants to publish artifacts should create their own GPG key, publish it to a public keyserver (https://central.sonatype.org/publish/requirements/gpg/#distributing-your-public-key) (keys.openpgp.org (https://keys.openpgp.org/) and keyserver.ubuntu.com (https://keyserver.ubuntu.com/) are recommended), then store the private key safely and in accordance with any programme-specific guidance.

The key and its associated passphrase will be used during the publishing process. This will require making it available safely to the task runner, for example Concourse, that is performing the deployment.

We recommend additionally publishing the public keys elsewhere, for example as a public GitHub repository, so that a third-party user knows an artifact is signed by a key that we have declared we use for signing.

Sonatype publishes build-tool-specific guidance for publishing and releasing artifacts on Maven Central (https://central.sonatype.org/publish/publish-guide/#deployment).

This page was last reviewed on 13 January 2025. It needs to be reviewed again on 13 July 2025 by the page owner #java (https://gds.slack.com/messages/java) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

JavaScript coding style

Contents

Linting

As a base, we follow conventions established by the standardjs (https://standardjs.com/) project. They allow us to be consistent about how we write code while reducing the time spend debating which linting rules to pick.

Depending on their needs, projects may complement this initial set of rules with extra linting, for example:

- other ESLint (https://eslint.org/) plugins, like es-x (https://github.com/eslint-community/eslint-plugin-es-x) to check API compatibility with browsers your project supports
- Prettier (https://prettier.io/docs/en/cli.html) to enforce further code formatting conventions

You should be cautious to only amend the initial set of rules to resolve compatibility issues, and not as a means to adjust rules to individual preferences.

Why: Linting ensures consistency in the codebase and picks up on well-known issues. Using an opinionated set of rules allows us to limit time spend picking rules, focusing instead on getting consistency, which is more important.

Tools

StandardJS's command line interface

If you're not looking to make any amends to the StandardJS conventions, you can use StandardJS' `standard` command line interface (https://standardjs.com/#usage) to lint files in your repository without extra set up.

```
npx standard
```

standardx

If the StandardJS rule set conflicts with the browsers your project supports, you can use standardx (https://github.com/standard/standardx) to amend which rules are running.

Once installed you can then override standard rules with an `.eslintrc` (https://eslint.org/docs/v8.x/use/configure/configuration-files) file or an `eslintConfig` entry in package.json (example (https://github.com/alphagov/govuk_publishing_components/commit/ea7f0becc76f73780b6cb33701bea9e58f15f91a)).

ESLint

ESLint is the most widely used JavaScript linter, and actually what StandardJS uses under the hood. Using it directly allows you to benefit from other plugins in the ESLint ecosystem to complement standard conventions, and keep up to date with newer rules, for example related to newer language features.

Standard can be integrated by adding the `eslint-config-standard` to your ESLint configuration.

When adding extra ESLint plugins, most come with a `recommended` configuration that's worth using as a starter, rather than deciding on each rule individually. You can then add or remove rules as needs arise during the life of your project. In that area, automatically fixable rules are especially cheap to try out, as the tools will take care of updating your code for you.

Prettier

Prettier's only preoccupation is with code formatting, not code quality (https://prettier.io/docs/en/comparison). It can be used as a complement to ESLint for further automated formatting, with much more advanced decisions in terms of indentation, spaces, or line breaks. It runs as a separate command (`npx prettier`) and the `eslint-config-prettier` (https://prettier.io/docs/en/integrating-with-linters) ensures there'll be no conflicts between the rules of ESLint and the formatting of Prettier.

When to run linting

On CI

Running linting in CI ensures that all pull requests meet our code conventions before getting merged on the `main` branch. You should have this configured as part of your project.

Through pre-commit Git hooks

Waiting for CI to know if the code follows the convention can take a bit of time. A pre-commit Git hook allows to get quicker feedback, directly on developers' machines. Errors that are automatically fixable can be fixed at that stage without human intervention, reducing the effort of linting for developers.

Tools like Husky (https://typicode.github.io/husky/) and lint-staged (https://www.npmjs.com/package/lint-staged) can help consistently run linting before commit by respectively:

- setting up the hooks when dependencies get installed
- running linting on the files staged for commit and adding any fixes to the current commit

In editors

To get even quicker feedback, editor plugins can highlight issues while editing files. They can correct automatically fixable errors on save, saving further development effort.

Each of the tools previously listed has plugins to help integrate with editors:

- StandardJS editor plugins (https://standardjs.com/#are-there-text-editor-plugins)
- ESLint editor plugins (https://eslint.org/docs/latest/use/integrations#editors)
- Prettier editor plugins (https://prettier.io/docs/en/editors)

Whitespace

Use soft tabs with a two space indent.

If you're using Prettier, this will be set up for you. Otherwise, you may want to configure a `.editorconfig` file (https://editorconfig.org/) accordingly.

Why: This follows the conventions used within our other projects.

Naming conventions

Follow the following conventions when naming symbols in your JavaScript code.

Why: The naming of objects in the code helps developers know how to interact with them. It also follows the conventions of the standard library.

Variables, functions and parameters

Use camelCase (http://eslint.org/docs/rules/camelcase) when naming variables, functions and parameters.

```
// Bad
const this_is_my_object = {}
const THISIsMyVariable = 'thing'
function ThisIsMyFunction(this_is_a_param) { ... }

// Good
const thisIsMyObject = {}
const thisIsMyVariable = 'thing'
function thisIsMyFunction(thisIsAParam) { ... }
```

Classes and constructors

Use PascalCase when naming classes or constructors.

```
// Bad
class user {
  constructor(options) {
  this.name = options.name
}
function profile(options) {
  this.user = options.user
}
const Bob = new user({
```

```
  name: 'Bob Parr'
})
const BobProfile = new profile({
  user: Bob
})

// Good
class User {
  constructor(options) {
    this.name = options.name
  }
}
function Profile(options) {
  this.user = options.user
}
const bob = new User({
  name: 'Bob Parr'
})
const bobProfile = new Profile({
  user: bob
})
```

HTML class hooks

When attaching JavaScript to the DOM use a `.js-` prefix for the HTML classes.

Eg `js-hidden` or `js-tab`.

Why: This makes it completely transparent what the class is used for within the HTML. It also makes it much easier to search in a project to remove old behaviour.

Styling elements

Do not apply styles directly inside JavaScript. You should only ever apply CSS classes and style from there.

Why: This reduces the risk of clobbering user stylesheets and mixing concerns across different code bases. Also see HTML class hooks.

Strict mode

You should add the `'use strict'` statement to the top of your module functions.

Why: This enables strict mode (https://developer.mo illa.org/en-US/docs/Web/JavaScript/Reference/Functions_and_function_scope/Strict_mode).

Strict mode converts many mistakes, such as undefined variables, into errors which makes it easier to determine why things are not working. It also forces scope so you do not accidentally export globals.

Modules

Avoid assigning modules to the `window` global scope.

Bundlers such as Rollup (https://rollupjs.org/) avoid the need to do this manually.

To do this manually, wrap your code in a closure, then attach the module to the global scope with a namespace:

```
;(function (global) {
  'use strict'

  var GOVUK = global.GOVUK || {}

  ...

  GOVUK.myModule = ...

  ...

  global.GOVUK = GOVUK
})(window); // eslint-disable-line semi
```

Why: attaching to the `GOVUK` object keeps us from polluting the global namespace. Checking for or creating the `GOVUK` object means the module can be reused on any project (internal or external) without having to modify it. You get the benefits of strict mode which include stopping your module from leaking variables into the global scope. The IIFE (https://en.wikipedia.org/wiki/Immediately-invoked_function_expression) should be wrapped with semicolons to ensure no issues with concatenation can happen.

Module structure

Module logic should be broken down into small testable functions. The functions should be exposed as methods on the module rather than hidden inside a closure.

```
// Bad
function myModule ($element) {
  function showThing () { ... }
  function hideThing () { ... }
  function submitThing () { ... }
  function getArgumentsForThing () { ... }

  $element.addEventListener('click', submitThing)
}

// Good
function MyModule ($element) {
  $element.addEventListener('click', this.submitThing.bind(this))
}
MyModule.prototype.showThing = function () { ... }
MyModule.prototype.hideThing = function () { ... }
MyModule.prototype.submitThing = function () { ... }
MyModule.prototype.getArgumentsForThing = function () { ... }

// Good
GOVUK.myModule = {
  showThing: function () { ... },
  hideThing: function () { ... },
  submitThing: function () { ... },
  getArgumentsForThing: function () { ... },
  init: function ($element) {
    $element.addEventListener('click', this.submitThing.bind(this))
  }
}
```

Why: Having small well named functions lets developers who are unfamiliar with the code understand what is going on faster. Having logic in small functions makes it easier to unit test each of those functions to prove they performs as expected. Having those functions exposed as methods on the module makes it possible to test those functions in isolation.

jQuery

Avoid jQuery in new projects.

In older projects put together a plan to migrate away from jQuery.

Why: jQuery had been used to provide browser support for older browsers. However, browser support for ES5 JavaScript is now widespread enough that a library like jQuery is unnecessary. The older versions of jQuery that we use have security vulnerabilities and are no longer maintained by the jQuery team.

Supporting older browsers

Use native web APIs where possible.

Use feature detection (https://developer.mo illa.org/en-US/docs/Learn/Tools_and_testing/Cross_browser_testing/Feature_detection) before polyfilling (https://developer.mo illa.org/en-US/docs/Glossary/Poly l), to support older browsers.

Method arguments

Favour named arguments in a object over sequential arguments.

```
// Bad
function addAutoSubmitToInput (input, action, timeout, debug) { ... }

// Good
function addAutoSubmitToInput (input, options) {
  var action = options.action,
      timeout = options.timeout,
      debug = options.debug
  ...
}
```

Why: by using named options you do not necessarily have to read the internals of the method being called to work out what the arguments mean.

Given a call to `addAutoSubmitToInput($input, './search', 20, false)` you would have to go to that method to find out what `20` or `false` mean.

A call to `addAutoSubmitToInput($input, { action: './search', timeout: 20, debug: false })` gives you context as to what the arguments mean. It also makes it easier to refactor arguments without having to change all method calls.

Connascence of naming is a weaker form of connascence than connascence of position (https://en.wikipedia.org/wiki/Connascence_%28computer_programming%29#Types_of_connascence).

This page was last reviewed on 13 September 2024. It needs to be reviewed again on 13 September 2025 by the page owner 🔒 #frontend (https://gds.slack.com/messages/frontend) .

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Using Node.js at GDS

This document describes how we write Node.js code at GDS. It is a list of guidelines that developers should follow in order to make code more consistent across all Node.js projects, making it easy for developers to change projects or start new ones.

Introduction

The guidance set out here should be followed in conjunction with the advice on the main programming languages manual page (/manuals/programming-languages.html).

The advice here is specifically about Node.js. Generic guidelines on writing code are out of scope.

Most guidelines listed here are recommendations, and the authors acknowledge that there will sometimes be valid exceptions.

Node versions

> Only use Long Term Support (LTS) versions of Node.js.

These are even-numbered versions (for example, Node.js 18.x or 20.x). However, it is important to keep an eye on the Node.js LTS Schedule (https://github.com/nodejs/Release#release-schedule) as to when versions move in and out of LTS.

> Do not use the `--harmony`, `--experimental-vm-modules` or other in-progress feature flags

Staged or *in-progress* features are not considered stable by the Node implementors.

> Specify the supported version of Node.js in your `package.json`.

Use the engines (https://docs.npmjs.com/files/package.json#engines) key.

Source formatting and linting

> Lint your JavaScript code with StandardJS (https://standardjs.com)

See the general JavaScript style guide for linting guidance (https://github.com/alphagov/styleguides/blob/master/js.md#linting).

Project directory structure

> Organise files around features, not roles

The following structure means you don't require lots of context switching to find related functions and your `require` statements don't need overly complicated paths.

```
├── product
|   ├── index.js
|   ├── product.js
|   ├── product.spec.js
|   └── product.njk
├── user
|   ├── index.js
|   ├── user.js
|   ├── user.spec.js
|   └── user.njk
```

> Don’t put logic in index.js files

Use these files to `require` all the modules functions.

```
// product/index.js
const product = require('./product')

module.exports = {
  create: product.create
}
```

> Store test files within the implementation

Keeping tests in the same directory makes them easier to find and more obvious when a test is missing. Keep the project’s global test config and setup scripts in a separate `test` directory.

```
├── test
|   └── setup.spec.js
├── product
|   ├── index.js
|   ├── product.js
|   ├── product.spec.js
|   └── product.njk
```

See also:

- Node.js Project Structure Tutorial (https://blog.risingstack.com/node-hero-node-js-project-structure-tutorial/) on Rising Stack (https://blog.risingstack.com/)

Language constructs

Declarations

> Use `const` and `let`, avoid `var`

Embrace immutability (more below) and do not let hoisting (https://www.adequatelygood.com/JavaScript-Scoping-and-Hoisting.html#declarations_names_and_hoisting) confuse you.

Functions

> Prefer `function` for top-level function declarations and `=>` for function literals

```
// Use:
const foo = function (x) {
  // ...
}

// Avoid:
const foo = x => {
  // ...
}

// Use:
map(item => Math.sqrt(item), array)

// Avoid:
map(function (item) { return Math.sqrt(item) }, array)
```

Acceptable exceptions include single expression functions, such as

```
const collatz = n => (n % 2) ? (n / 2) : ((3 * n) + 1)
```

or curryied functions, which are more readable using the arrow notation:

```
const foo = x => y => z => (x * y) + z
```

Be aware that because anonymous functions do not have a name, a stack trace will be harder to read when debugging. Also remember that arrow functions keep their context's `this`.

Classes

> Use the `class` keyword to define classes

There are many different ways to define classes in JavaScript. `class` has been added to the standard specifically to resolve this. In short, use `class` for classes and `function` for functions.

```
class Rectangle {
  constructor (height, width) {
    this.height = height
    this.width = width
  }

  area () {
    return this.width * this.height
  }
}
```

Asynchronous code

> Use asynchronous versions of the Node.js API functions whenever possible.

For instance, avoid `readFileSync` but instead use `readFile`. Even if your code is less readable as a result and that particular piece of code does not need to be asynchronous (because you cannot proceed until you've read that file anyway), it will not block other server threads. However if your program does not use concurrency, synchronous versions are sometimes preferable as they are more readable and less heavy on the operating system.

> Avoid inline callbacks

```
// Prefer:
const pwdReadCallback = function (err, data) {
  // ...
}
fs.readFile('/etc/passwd', pwdReadCallback)

// over:
fs.readFile('/etc/passwd', (err, data) => {
  // ...
}

// Another example:

const done = function (resolve, reject) {
  return () => {
    try {
      // Do something that might fail
      resolve('Success - all went well')
    } catch (err) {
      reject(err)
    }
  }
}

const waitAndSee = function (resolve, reject) {
  setTimeout(done(resolve, reject), 2500)
}

const log = status => message => console.log(status + ': ' + message)

new Promise(waitAndSee)
  .then(log('Success'))
  .catch(log('Failure'))
```

This will avoid “callback hell” and encourage organising callback functions linearly, in an event-driven fashion. It also makes it easier to write unit tests for those functions.

```
const pwdReadCallback = function (err, data) {
  // ...
```

```
}
const userLoggedInCallback = function (authToken) {
  // ...
}
const dbRequestResultCallback = function (req, res) {
  // ...
}
```

Separating out a callback may pose a problem when it needs its original scope. This can be handled by currying the callback:

```
fs.readFile(filename, callback(filename))

const callback = filename => (err, data) => {
  if (err) {
    console.log(`Failed reading ${filename}.`)
    throw err
  }
  // ...
}
```

Use of `async/await`

You should use async functions, and the await keyword, to organise your asynchronous code. This will make your code easier to read and write.

```
// Without async/await:
const getUserFromCreds = function (username, password) {
  return getUserIdFromAuth(username, password)
    .then(userId => {
      return getUserFromApi(userId)
    })
    .then(user => {
      return {
        id: user.id,
        name: user.name,
        age: user.ageInYears
      }
    })
}
```

```
// With async/await:
const getUserFromCreds = async function (username, password) {
  const userId = await getUserIdFromAuth(username, password)
  const user = await getUserFromApi(userId)

  return {
    id: user.id,
    name: user.name,
    age: user.ageInYears
  }
}
```

The `await` keyword will pause execution until its promise fulfills. This means a sequence of awaited promises can only execute in the order they're declared, each being forced to wait for the previous one.

The promises API provides several methods for managing sequences of promises (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#static_methods), which can be used to solve this problem, for example:

```
const getUserFromCreds = async function (username, password) {
  // Use await to make this promise blocking:
  const userId = await getUserIdFromAuth(username, password)

  const [avatarImgUrl, user] = await Promise.all([getAvatarForUser(userId),

  return {
    id: userId,
    name: user.name,
    age: user.ageInYears,
    avatar: avatarImgUrl
  }
}
```

If none of those offer the control needed, you can also instantiate promises separately from when you await them (https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Async_await#handling_asyncawait_slowdown), for example:

```
const getUserFromCreds = async function (username, password) {
  // Use await to make this promise blocking:
  const userId = await getUserIdFromAuth(username, password)

  // Start other promises running before awaiting:
  const avatarPromise = getAvatarForUser(userId)
  const userPromise = getUserFromApi(userId)

  const avatarImgUrl = await avatarPromise
  const user = await userPromise

  return {
    id: userId,
    name: user.name,
    age: user.ageInYears,
    avatar: avatarImgUrl
  }
}
```

Functional programming

> Use JavaScript's functional programming features

JavaScript has the advantage that it offers functional programming concepts natively, like functions as first-class objects, higher-order functions (like `map`, `reduce` or `apply`) or pattern matching.

Following functional programming principles, such as immutable data structures and pure functions, produces code that is easier to test, less prone to runtime errors and is often more performant. Write functions as expressions that return values of a single type.

Side effects in a function can have bad consequences, especially within `map` or `reduce`. `this` is better avoided, as it can refer to different objects depending on the context in which a function is executed and lead to unexpected side-effects.

Remain aware that JavaScript is not just a functional language and you will most probably have to mix functional and object-oriented concepts, which can be tricky to get right.

More guidance:

- An Introduction to Functional JavaScript (https://www.sitepoint.com/introduction-functional-javascript/)
- JavaScript Allongé (https://leanpub.com/javascriptallongesix/read)

Errors

> Make sure you handle all errors

Envisage all error scenarios, in particular in asynchronous callback functions or Promises, and have a fallback for any situation. Consider programmer errors (bugs) as well as operational errors (arising from external circumstances, like a missing file). Bugs that are caught by exceptions should be logged and the execution stopped, and the supervisor will restart the process. Use the built-in `Error` object instead of custom types. It makes logging easier.

Your application should not trust any input, for example from a file or an API. So the application should handle all operational errors and recover from them.

Security

> Maintain NodeJS Security best practices (https://nodejs.org/en/docs/guides/security/).

Use the LTS version.

If using the core HTTP Server, make sure all routes have error handling. Without error handling a DoS attack is possible. Using a framework should mitigate this as it should set default values for timeouts on idle requests.

Use `npm ci` over `npm install` as this enforces the use of the lockfile so that inconsistencies between the lockfile and package.json are represented as errors and not ignored.

Don't Monkey Patch (https://en.wikipedia.org/wiki/Monkey_patch) existing properties in runtime to change expected behaviour. This can cause side effects in core NodeJS modules and the libraries you are using. For example:

```
// Dont do this.
Array.prototype.push = function (item) {
```

```
  // overriding the global Array[].push
};
```

An extensive list of security mitigations can be found on the NodeJS Security best practices (https://nodejs.org/en/docs/guides/security/)

Node.js's HTTP server

> Offload Node's server as much as possible

Do not expose Node's HTTP server to the public. Use a reverse proxy to serve static assets, and cache content as much as possible. Implement adequate supervising: pm2 (https://pm2.keymetrics.io/) is recommended for Node.js-specific servers.

Transpiling

> Stick to JavaScript

Avoid anything that compiles to JavaScript (except for static type checking, see below). Examples to avoid include CoffeeScript, PureScript and many others (https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS).

If you would like to use static typing consider using JavaScript extensions like TypeScript (https://www.typescriptlang.org/), or Flow (https://flow.org/). This has the advantage of making it easier to prevent runtime errors, by adding type information to variables or parameters and transpiling the source to JavaScript, reporting errors when wrong types are used.

However be aware that there are disadvantages too: augmented source code will no longer be executable uncompiled. You may also need some extra type definitions if you want to use external libraries.

Generally, anything that departs from the standard JavaScript syntax in a way that a Node developer would have trouble reading your code is advised against.

Frameworks

It is important to remember that we use the govuk-frontend (https://github.com/alphagov/govuk-frontend) to build pages on websites at GDS. This restricts

the choice of webservers to ones that easily support rendering content using Nunjucks (https://mozilla.github.io/nunjucks/). This rules out many currently popular webservers that use React or other templating approaches.

> Use Koa or Express for web applications, avoid lesser-known frameworks

Express is widely used at GDS. It is the oldest Node.js webserver framework and there is a wide range of examples and helper libraries available on the wider internet. However is has less support for more modern JavaScript features, most notable support for `await/async`. These issues require extra workarounds.

Koa can be used as an alternative if it better fits your requirements.

Koa vs Express (https://github.com/koajs/koa/blob/master/docs/koa-vs-express.md) is a good comparison guide to help inform that decision.

If you find yourself looking for an MVC framework or if you need an ORM to manage records in a database, you probably should not be using Node.js in the first place (/standards/programming-languages.html#frontend-development).

Libraries

> Avoid libraries that produce esoteric code, or that have a steep learning curve

Using advanced libraries can make code very hard to read for developers not familiar with them, as useful as they may be. For instance, Ramda (https://ramdajs.com/) lets you write:

```
R.cond([
 [R.is(Number), R.identity],
 [R.is(String), parseInt],
 [R.T, R.always(NaN)]
])
```

This is compact and useful, but not easily understood up by a developer not familiar with Ramda.

Generally, readable code is better than compact or advanced code. Optimisation can lead to very arcane code, so should only be used when necessary.

```
// Prefer:
for (let i = 0; i < 10; i += 1) {
  for (let j = 0; j < 10; j += 1) {
    console.log(i, j)
  }
}

// Over:
for (let i=0,j=0;i<10 && j<10;j++,i=(j==10)?i+1:i,j=(j==10)?j=0:j,console.lo
```

Node Package Manager (NPM)

> Do not reinvent the wheel, but do not over-use the amount of external code you import.

Using other people's code is a risk. NPM is no exception and is arguably worse than other package managers as the small granularity of packages leads to a very large number of dependencies. Your application may easily end up relying on software written by hundreds of different people, most of whom you do not know.

However, given that it's impossible not to depend on foreign code, you should do your best to reduce the risks involved:

- avoid relying on packages for functionality you can easily implement yourself, especially if a package has a lot of functionality you do not need.
- empirically check the trustworthiness of a package you wish to use: check its popularity, author reputation and so on.
- Use tools like GitHub security alerts (https://help.github.com/en/github/managing-security-vulnerabilities/about-security-alerts-for-vulnerable-dependencies) or Snyk (https://snyk.io/) to check packages for vulnerabilities

> Use `npm init`, `npm start`, `npm ci`, NPM scripts and so on

Making full use of NPM's features will simplify your continuous integration and deployment.

> Lock down dependencies

Avoid incompatible upgrades that may break your application. NPM 5 does it by default, but be careful when upgrading or adding a new package.

> Regularly check for unused dependencies, and make sure you do not have dev dependencies in production

Do not slow down your deployments with unused code.

> If you build modules you think could be useful for others, publish them on npm

Because Open Source is a Good Thing (https://technology.blog.gov.uk/category/open-source/).

> Published modules should be tested and supported on the last two current LTS versions of Node.js.

See also:

- Controlling the Node.js security risk of npm dependencies (https://blog.risingstack.com/controlling-node-js-security-risk-npm-dependencies/)
- I'm harvesting credit card numbers and passwords from your site. Here's how. (https://hackernoon.com/im-harvesting-credit-card-numbers-and-passwords-from-your-site-here-s-how-9a8cb347c5b5)
- you might not need gulp or grunt (https://www.freecodecamp.org/news/why-i-left-gulp-and-grunt-for-npm-scripts-3d6853dd22b8/).

Further reading or information

In the GDS context, the guidelines provided in the links below are advisory only. The guidance provided here takes precedence in case of conflicts.

- **NodeJS Security best practices (https://nodejs.org/en/docs/guides/security/)** - A guide to security best practices by the Node working group.

- **Node best practices (https://github.com/goldbergyoni/nodebestpractices)** — A comprehensive guide to Node.js best practices
- **node.cool (https://github.com/sindresorhus/awesome-nodejs)** — A curated list of Node.js packages and resources by Sindre Sorhus
- **JS documentation (https://developer.mozilla.org/en-US/docs/Web/JavaScript)** — MDN's definitive JS docs
- 🔒 **#Nodejs (https://gds.slack.com/messages/nodejs)** – community's Slack channel

This page was last reviewed on 24 April 2024. It needs to be reviewed again on 24 April 2025 by the page owner 🔒 #nodejs (https://gds.slack.com/messages/nodejs) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Python style guide

This manual is designed to aid developers in writing Python code that is clear and consistent, within, and across, projects at GDS.

Code formatting

We use Ruff (https://docs.astral.sh/ruff) to format our code. Ruff aims for cross compatibility with the popular opionionated formatter Black. Where Ruff and PEP 8 do not express a view (for example, on the usage of language features such as metaclasses) we defer to the Google Python style guide (https://google.github.io/styleguide/pyguide.html). Use these as references unless something is explicitly mentioned here. These rules should be followed in conjunction with the advice on consistency on the main programming languages manual page (/manuals/programming-languages.html).

If you want to add a new rule or exception please create a pull request against this repo.

Maximum line length of 120 characters

PEP 8 (https://www.python.org/dev/peps/pep-0008/) dictates a preferred maximum line length of <= 79. This is is a hangover from developing in a Unix terminal window. The vast majority of developers are now using an IDE which can handle a greater line length.

Couple this with the fact that much of the time GDS developers are coding web apps and have to deal with nested `JSON` objects, ORM model definitions/ queries, and error/ url strings and this convention begins to show its age.

Linting

Ruff

This manual advises the use of the Ruff (https://docs.astral.sh/ruff) command line checker as an all in one formatter, linter, codestyle and complexity checker.

How to use Ruff

First you should add the Ruff module (available from PyPI (https://pypi.org/)) to your 'dev' or 'test' requirements/dependencies.

You'll then likely want to run it alongside your unit tests.

Ruff ignores

Ruff can ignore particular lines or files.

A particularly useful feature of ruff is the ability to specify rule exemptions per directory or file.

Commonly it's used for ignoring unused imports in module level `__init__.py` files or imports not being at the top of a file in settings files or scripts.

The feature is documented in the Ruff documentation, under per-file-ignores (https://docs.astral.sh/ruff/settings/#lint_per-file-ignores). You can also see an example in the Notifications API repo (https://github.com/alphagov/notifications-api/blob/main/ruff.toml).

Common Con□guration

Notify is already running the latest verions of Ruff (https://docs.astral.sh/ruff) on all of its repos. You can find an example of their configuration in the root of any repo in the `ruff.toml` file (https://github.com/alphagov/notifications-api/blob/main/ruff.toml).

```
line-length = 120

target-version = "py311"

select = [
    "E",  # pycodestyle
    "W",  # pycodestyle
    "F",  # pyflakes
    "I",  # isort
    "B",  # flake8-bugbear
    "C90",  # mccabe cyclomatic complexity
    "G",  # flake8-logging-format
```

```
]
ignore = []
exclude = [
    "migrations/versions/", ...
]
```

In the above file we exclude directories we want the checker to ignore completely, include optional linting such as applying isort rules to ensure imports are sorted for minimal git diffs, set the maximum line length and set the target python version.

Note: you can also ignore rules on particular lines of code or files by adding a `# noqa` comment - see ruff's noqa syntax (https://docs.astral.sh/ruff/linter/#error-suppression).

Additional linting resources

- Notify Config (https://github.com/alphagov/notifications-api/blob/main/ruff.toml): A production config to base off
- Ruff error codes list (https://docs.astral.sh/ruff/rules/)

Environments

This manual advises the use of uv to manage different versions of Python you have installed.

To create virtual environments call `uv venv -p python3.13` from your project root directory. This will create a virtual environment with that specific python version in a folder called `.venv`. This folder should be excluded in your `.gitignore` file. For more information see Python virtual environment primer (https://realpython.com/python-virtual-environments-a-primer/)

direnv (https://direnv.net/) is used to manage environment variables. This ensures project specific variables do not clutter your main environment or the environment of other projects. direnv uses `.envrc` for general project specific variables and you can use a non version controlled `.secrets` to store sensitive information. We recommend you put `source .venv/bin/activate` in your `.envrc` file to ensure your virtual environment is always activated.

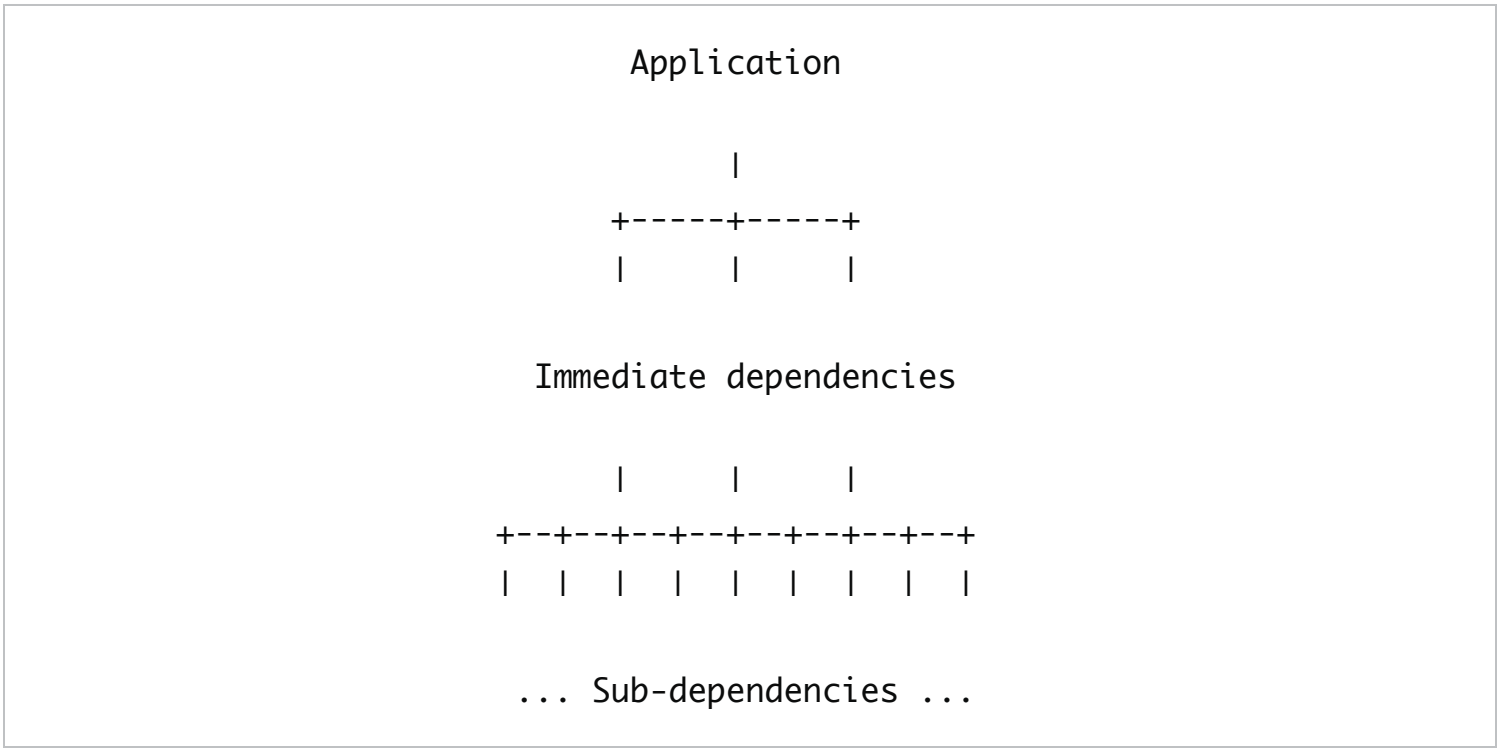
Dependencies

A Python application project typically brings together Python packages from PyPI, with others written in-house (or otherwise not distributed through PyPI).

These packages are the applications immediate dependencies. Additionally, any package can have its own immediate dependencies, where they draw on other packages. From an

application's point of view, the dependencies of the packages it requires are its sub-dependencies.

The below diagram shows a simplified view of the resulting pyramid of dependencies; however it is important to note that the hierarchy can repeat itself infinitely.



There are two ways of specifying dependencies in Python world: as a specific version or as an allowable range.

Different considerations apply to dependency management depending whether you are packaging a library, or creating an end-product such as an application or a bundle of scripts. In general, you should only specify specific versions when creating a Python system that sits at the top of the dependency pyramid; otherwise there is a danger of creating version conflicts.

Applications

These recommendations apply wherever you need a reproducible set of dependencies, such as a complete web application, perhaps with many dependencies and sub-dependencies. It also applies to a collection of scripts that are deployed into the cloud and run automatically (for example, batch jobs).

A good strategy for specifying your application's Python dependencies has two desirable characteristics - they should be:

1. Reproducible (predictable)

   Pin your application's full dependencies – specific versions, rather than ranges – or you'll get unpredictability between your dev environment and other environments. You want a new starter to avoid small hard-to-spot problems. And you want parity between

what you test locally, what is tested by CI, and what you deploy, or you risk new issues appearing on a live server. Additionally these things can be hard to diagnose.

2. Kept up-to-date

   Security issues are found in libraries, so it is important to choose libraries that are maintained and to ensure your team has a strategy to ensure security updates are installed without significant delay. The how to manage third party software dependencies (/standards/tracking-dependencies.html) section gives further context and discusses tools that can help, such as snyk.io (https://snyk.io/).

Your README should document an easy-to-follow process by which all your Python dependencies can be upgraded to get bug fixes and security fixes, without introducing breaking changes to your build.

Your pinned dependencies should be fully specified in a file called `requirements.txt`, and checked into your version control system (VCS). For projects with only a small number of dependencies, maintaining this manually (for example, installing with `uv pip install`, then using `pip freeze`) may be adequate.

GDS recommends using uv (https://docs.astral.sh/uv/) for managing dependencies.

Put your top-level requirements in a `requirements.in` file, and then use `uv pip compile requirements_for_test.in -o requirements_for_test.txt` to generate a `requirements.txt` file. Both the .in and .txt files should be commited to your repository.

List dependencies only needed for development or testing into a separate `requirements-dev.txt` file.

Libraries

This recommendation applies to any Python repository that intended to be installable (into a virtual environment, a container, or onto bare metal) as a dependency of some larger system or application. It may be applicable to repositories that provide scripts to be run by developers or other end-users, but is not recommended for code that's intended to be deployed on its own into the cloud.

Use a `pyproject.toml` (https://packaging.python.org/en/latest/guides/writing-pyproject-toml/) to specify your library's configuration. When specifying the dependencies of your library and the version ranges with which it can be reasonably expected to work:

- The range you choose will depend on the guarantees each dependency makes about backward-compatibility. For example, if you're currently using version 1.3.1 of a semantically-versioned library, it would be reasonable to specify a range such as `<2.0,>=1.3.1`. However, for a library that does not make that guarantee, you might specify a more restricted range, such as `<1.4,>=1.3.1`.

- Update this file whenever you are ready to test and validate a new version that falls outside the existing range.

> If you have dependencies that are not available on PyPI (for example, because you've fixed a bug by forking the code), then you can use a PEP 440 (https://www.python.org/dev/peps/pep-0440/#direct-references) git reference in your `install_requires` list.
>
> - In the past, we've documented such dependencies in a `requirements.txt` file (example (https://github.com/alphagov/digitalmarketplace-utils/commit/dc16012af6b55d9eda4e8dd7fee514103682a5c7#diff-4d7c51b1efe9043e44439a949dfd92e5827321b34082903477fd04876edb7552)), but any application wanting to depend on your library then needs to manually copy those sub-dependencies into its own list (example (https://github.com/alphagov/digitalmarketplace-briefs-frontend/commit/5fb3df85bf9fa109ba3eaf1750a4fba4e92ef2a8#diff-4d7c51b1efe9043e44439a949dfd92e5827321b34082903477fd04876edb7552)).

Specify dependencies needed only for testing your library in `tox.ini` if you are using Tox (https://tox.readthedocs.io/en/latest/) (example (https://github.com/alphagov/notifications-python-client/blob/f27b67a53371c68c36583f985c29b0526e2294b9/tox.ini)), or in a `requirements-dev.txt` (example (https://github.com/alphagov/digitalmarketplace-utils/blob/222e50c022eae4d9b2569b148cdfc642b08733cf/requirements-dev.txt)) file otherwise.

This page was last reviewed on 14 February 2025. It needs to be reviewed again on 14 February 2026 by the page owner 🔒 #python (https://gds.slack.com/messages/python) .

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Ruby style guide

Code formatting

The GOV.UK programme publishes and maintains a gem, rubocop-govuk (https://github.com/alphagov/rubocop-govuk), which provides linting rules for Ruby projects. These rules extend those described in rubystyle.guide (https://rubystyle.guide). The rubocop-govuk project has overriden rules to either match GDS style precedence or to handle scenarios where rules were problematic to adopt.

rubocop-govuk (https://github.com/alphagov/rubocop-govuk) also includes rules for Rake (https://github.com/ruby/rake), RSpec (https://rspec.info/) and Ruby on Rails (https://rubyonrails.org/). The RSpec and Rails rules extend the respective styleguides, rspec.rubystyle.guide (https://rspec.rubystyle.guide) and rails.rubystyle.guide (https://rails.rubystyle.guide), where only rules that have proved problematic have been overriden.

Conventional tooling

GDS uses the MRI (https://en.wikipedia.org/wiki/Ruby_MRI) implementation of Ruby in projects. You should not adopt a different one (such as jRuby (https://www.jruby.org/)) for projects without clear justification. The recommended tool for managing multiple Ruby installations is rbenv (https://github.com/rbenv/rbenv). For dependency management bundler (https://bundler.io/) should be used.

For web applications, we advise using Ruby on Rails (https://rubyonrails.org/) due to its stability, broad adoption and large community. For smaller web applications we have chosen Sinatra (https://sinatrarb.com/) before - however this should be approached with caution as these can easily become large applications over time that are less conventionally organised than Rails applications.

Testing with RSpec

For testing, the RSpec (https://rspec.info/) framework is the conventional and preferred choice. RSpec provides an expressive syntax that lends itself to producing readable tests. For testing functionality from a user's perspective it is preferred to implement tests in RSpec rather than adopt an abstraction such as Cucumber (https://github.com/cucumber/cucumber-ruby), this is due to us finding it easier to maintain tests where the definition and implementation are in the same file.

Further reading

- GOV.UK documentation on publishing a Ruby gem (https://docs.publishing.service.gov.uk/manual/publishing-a-ruby-gem.html)
- GOV.UK conventions for Rails applications (https://docs.publishing.service.gov.uk/manual/conventions-for-rails-applications.html)

This page was last reviewed on 23 April 2024. It needs to be reviewed again on 23 April 2025 by the page owner 🔒 #ruby (https://gds.slack.com/messages/ruby) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Writing READMEs

There should be a README for every GDS GitHub repository. A user expects a README to help them:

- understand what the project is
- evaluate whether the project is useful for them
- learn how to use the project
- understand how to contribute to the project

You should write READMEs in Plain English (https://www.gov.uk/guidance/content-design/writing-for-gov-uk#plain-english) and define any technical terms that may be unfamiliar to someone new to the project.

Avoid using terms like ‘just’ or ‘simply’ when writing your README. You should not assume your user has any prior knowledge of your project.

If you want help writing a README, you can ask a technical writer in #tech-writers on Slack.

Length of your README

A README should be an overview and list of instructions to help someone get started with your project.

If you find yourself writing a lot of content for your README, consider moving some of the more detailed documentation, such as API reference information or configuration advice, into a separate document.

Some project teams at GDS like to include these in a `docs` folder within the same repository and link to them from the README.

Structuring your README

Use the alphagov template (https://github.com/alphagov/readme-template/blob/main/README.md) for GDS READMEs.

Test your documentation

Test your README instructions before you publish to make sure users can follow your documentation. You can also ask a:

- member of your team to try the instructions and make sure they work
- technical writer to review the content

This page was last reviewed on 7 November 2024. It needs to be reviewed again on 7 May 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Writing release notes

When you make changes to code that will affect third parties, you should also publish a release note that clearly explains what's changed about your service's functionality, semantics or syntax.

You do not need to publish a release note if your service is used only internally, or used by people rather than through an API, but even in those cases it can be a good way to track changes without reading the commits.

You can publish a release note as either a:

- GitHub release note (https://docs.github.com/en/github/administering-a-repository/releasing-projects-on-github/managing-releases-in-a-repository)
- markdown file in the root directory of your release

Structure your release note so it has headings for:

- breaking changes - where users must change their code to avoid it breaking after they update
- new features
- deprecated features - where users should change their code ahead of a breaking change in a future release
- bug fixes

Write release notes in the GOV.UK style (https://www.gov.uk/guidance/content-design/writing-for-gov-uk#writing-to-govuk-style). See GOV.UK Frontend's v3.0.0 release note (https://github.com/alphagov/govuk-frontend/releases/tag/v3.0.0) as an example. Be clear and concise, use the active voice and address your users directly. You may want to get help from a technical writer.

Use an active verb-based heading for each change, and to group related changes. For example 'Update file paths'.

For each change, start by telling your users directly how the change affects them. Start sentences with:

- “You can now…”
- “You can no longer…”
- “You no longer need to…”
- “The API now…” or “The API no longer… ” - if you need to talk about an element directly, for example when you’re describing a bug fix

For breaking changes, start sentences with:

- “You should…”
- “You can change…” - if it’s one of several options
- “Change…” - if it’s the only option
- “You must change…” - if something will go wrong otherwise

You should avoid:

- writing from your team’s perspective - avoid “We’ve fixed…”, “We’ve added” or “Fixed a bug where…”
- links to ‘further guidance’ - your release note should contain all the guidance that’s essential to updating
- in-depth reasons for changes - add a link to a pull request, a blog post or your documentation instead
- the passive voice
- images

You should also write a README for GDS repositories (/manuals/readme-guidance.html).

Using code examples

You should only include ‘after’ code, not ‘before’ code. This means it’s easier for users to know which code to copy and paste, and to check if they’ve made the changes correctly.

Lead into a code example with either:

- a colon - if it’s the exact code a user should use:

  In your `assets` path, add `afolder/` :

  ```
  /node_modules/afolder/assets
  ```

- ‘For example’ - if the user’s code may be different to the example:

  Add an `assets-` prefix to data-module attribute values. For example:

```
<div data-module="assets-section">
```

Guidance outside GDS

You can find more information from:

- learning to love release notes (https://www.writethedocs.org/videos/prague/2018/learning-to-love-release-notes-anne-edwards/) - a talk at the 2018 Write the Docs conference
- the how to write release notes (https://ffeathers.wordpress.com/2017/08/19/how-to-write-release-notes/) blog - a post by a technical writer at Google

This page was last reviewed on 19 November 2024. It needs to be reviewed again on 19 May 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Security overview for websites

These 15 steps are primarily for running independent websites that are not on the gov.uk domain; however, the principles here apply to all Cabinet Office websites and applications.

1. Access Control

Must Do

Use named accounts to log in for all systems (from registrars to content management systems).

Use Single Sign-On (SSO) wherever possible to link accounts to Google using OAuth or OIDC. If the service does not support SSO, make sure you use a gov.uk email and enable multi-factor authentication (MFA), ideally using a phishing-resistant mechanism like WebAuthn (hardware keys).

If applicable, use more secure forms of MFA for higher risk accounts. For example, hardware tokens such as Yubikeys for privileged accounts, app-based authentication for standard accounts, and SMS MFA for low risk accounts.

Where SSO is not available, or as an additional measure where possible, consider using context-aware access controls (see Google for example (https://workspaceupdates.googleblog.com/2022/04/context-aware-access-admin-console.html)).

Consider what monitoring and alerting should be implemented for privileged account login/use.

Have a process to review who has access regularly.

If available, enable different roles for different people's needs.

See how to manage access to your third-party service accounts (/standards/accounts-with-third-parties.html) for more details.

Why?

Using named accounts makes incident response and non-repudiability (making it harder for someone to deny they were responsible) easier; if an incident occurs, named accounts make it easier to work out the cause or vector and to then restrict access.

Using SSO makes Joiners, Movers and Leavers (JML) processes simpler. As soon as someone leaves and their Google/IT account is removed, their access to services will be removed.

Using MFA prevents many common forms of attack that would otherwise give attackers access to accounts.

Context-aware access further limits unauthorised access where credentials have been compromised.

See the NCSC's guidance on identity and access management (https://www.ncsc.gov.uk/guidance/introduction-identity-and-access-management).

2. Domain Name Registrar

Must Do

Make it more challenging for adversaries to take over your registrar, which is where your domain is configured.

Contact your registrar and ask if they offer any additional safeguards to prevent takeover.

Ask your registrar for contact details that you can use in the event of an incident and note these in your incident response plan.

Enable DNSSEC if practicable (Cloudflare has a great explanation on DNSSEC (https://www.cloudflare.com/en-gb/dns/dnssec/how-dnssec-works/)).

Ensure your WHOIS details are correct and contain the organisation or department name, the main building address, and a generic shared email address (*service*-webadmin@organisation[.]gov.uk) and do not enable privacy guard.

Why?

If an adversary successfully gets access, they could redirect your domain to a malicious or damaging website, and it might be difficult to revert.

Having correct and available WHOIS records helps malware and security researchers understand the domain's ownership and reach out if there's an issue. It can also help prevent search engines from delisting or flagging the domain as spam.

3. Domain Name System

Must Do

Domain Name System (DNS) is where your registrar points to, using Name Server (NS) records, and contains the records for web servers, email and other applications. DNS may be the same service as your registrar; if it's not, follow those steps and apply them to the DNS service.

Do you need email to or from the domain name?

- No - then make sure you disallow sending emails using the domain name; this involves adding four records (an MX and three TXT records) - see NCSC's protecting parked domains guidance (https://www.ncsc.gov.uk/blog-post/protecting-parked-domains).
- Yes - follow best practices (see NCSC guidance on email security (https://www.ncsc.gov.uk/collection/email-security-and-anti-spoofing)) like setting up appropriate:
  - Sender Policy Framework (SPF) TXT record
  - Domain Keys Identified Mail (DKIM) TXT record(s)
  - Domain-based Message Authentication, Reporting, and Conformance (DMARC) TXT record, and use the NCSC's Mail Check (https://www.ncsc.gov.uk/information/mailcheck)

Have a process for reviewing and removing DNS records that are no longer needed.

Consider using a short Time To Live (TTL) for your main website (A or AAAA) but not too short as that can introduce vulnerabilities (5 minutes or 300 seconds is the recommended value).

Consider using longer TTLs for other records (such as email, MX or TXT records) - the recommended value is 24 hours or 86,400 seconds.

If availability is critical, consider a backup DNS; either use infrastructure as code to deploy to both or use automation to export and import the zone. Also, consider enabling the secondary DNS as another active name server.

See how to manage DNS records for your service (/standards/dns-hosting.html#how-to-manage-dns-records-for-your-service) for more details.

Why?

As with registrar, inadequate DNS controls could lead to an adversary pointing your domain to a malicious site or version of your site. One such attack is called DNS hijacking (https://www.ncsc.gov.uk/news/alert-dns-hijacking-activity).

Without implementing appropriate email DNS records, you could be at risk of enabling email spam from your domain, which would impact your search results and ranking.

4. Patching

Must Do

If your service uses a Content Management System (CMS), like WordPress or Joomla, or uses a non-managed infrastructure, then you - or your supplier - must have processes in place for keeping the system and software up-to-date.

Consider setting up a patching schedule with the development team or supplier to manage the patching process. You may want to:

- introduce this as a formal item in any governance meetings
- consider including any periods where you may want to pause or defer patches if you are concerned about availability during key events

If you are using Infrastructure-as-a-Service (IaaS) or Platform-as-a-Service (PaaS), you may want to set up a regular pipeline that gets the latest updates, tests your website or application with those, and then pushes the tested updates to your live environment.

See NCSC's guidance on patching (https://www.ncsc.gov.uk/blog-post/the-problems-with-patching) for more information about patching.

Why?

If you neglect to patch and keep the systems up-to-date, adversaries may use known vulnerabilities to impact your service.

5. Development, Test or Staging Environments

Must Do

Do not overlook the security of these environments, as a compromise or incident may cause the same disruption to your development team or supplier as in the live environment.

Consider setting up an IP allowlist or additional access controls to only allow authorised people to access.

If you're using different domains, follow the same principles listed throughout this guidance, including setting appropriate and public WHOIS information.

Do not have actual user data (logins or personal information) in these environments.

Consider removing notification ability or restricting it to designated recipients to stop accidental notifications.

6. Security Headers

Must Do

Configure appropriate security headers to help your users' browsers understand what your service should be doing:

HTTP Header	Notes
Strict-Transport-Security	Also known as HTTP Strict Transport Security (HSTS), tells browsers to enforce HTTPS
Content-Security-Policy	see below for details on CSP
X-Content-Type-Options	When set to "nosniff" it tells browsers not to assume a content type
X-Frame-Options	Can defend against clickjacking
Referrer-Policy	Tells browsers how much information to give to third party services when a user clicks their links on your site
Permissions-Policy	Allows control of the browser features, such as denying access to the webcam if your website or service does not need to access it
Cross-Origin-Embedder-Policy	Sets the permission for how assets are loaded
Cross-Origin-Opener-Policy	^
Cross-Origin-Resource-Policy	^
Feature-Policy	*Partially deprecated* (see OWASP guidance on Feature-Policy (https://owasp.org/www-project-secure-headers/#feature-policy)).

	Previously used to specify features the browser could access, primarily replaced with `Permissions-Policy`
Expect-CT	*Deprecated* (see OWASP guidance on Expect-CT (https://owasp.org/www-project-secure-headers/#expect-ct)). Previously used by browsers to check the certificate transparency logs, which helps determine the legitimacy of a HTTPS connection. It is no longer needed as a response header
X-XSS-Protection	*Deprecated* (see OWASP guidance on X-XSS-Protection (https://owasp.org/www-project-secure-headers/#x-xss-protection)). Previously used to mitigate XSS (cross-site scripting), this has been deprecated and should be set to `0` to disable the functionality in older browsers

Remove/disable the “Server” header; this is slightly contradictory to the NCSC’s Secure by Design principle (https://www.ncsc.gov.uk/information/secure-default) of “security through obscurity should be avoided” but it does make it more challenging to target your service.

You can find an example of implementing the headers using Lambda@Edge in the alphagov/aws-lambda-at-edge-examples (https://github.com/alphagov/aws-lambda-at-edge-examples) repo.

Use https://securityheaders.com (https://securityheaders.com) to check your security headers - tick the “Hide results” checkbox before clicking “Scan”.

Why?

Security headers help users’ browsers determine the intended functionality of your website or service and make it more difficult for an adversary to cause an incident.

7. Content Delivery Network

Should Do

Along with Access Control for the provider, make it harder to cause a denial-of-service attack against your service by using a Content Delivery Network (CDN), such as AWS Cloudfront.

Configure appropriate caching so that the CDN can serve most, if not all, traffic.

Consider a backup CDN, and this could be as simple as a static version; see Backup Static Origin for more details.

Implement origin access controls so that the origin cannot be impacted directly. The recommended approach in AWS is to add a 'secret' header from the CDN to the origin and have the origin check that header in an AWS WAF rule.

If you are using CloudFront:

1. Consider limiting the edge locations if you only expect traffic originating from the UK.
2. Set up AWS Shield Advanced (it's paid for and already available in Cabinet Office's AWS environment), see AWS Shield Response Team
3. Contact the 🔒 CDIO Cyber Security (https://gds.slack.com/messages/CCMPJKFDK/) team to send logs and get access to Splunk - see the logging (/standards/logging.html) page for more details.
4. Consider setting up AWS Origin Shield if you expect international traffic.

Why?

Good caching reduces the risk of a Distributed Denial-of-Service (DDoS) impacting your website or service.

A backup or secondary origin can mitigate a complete outage of your primary site for extended periods.

The origin access control prevents the CDN from being bypassed and traffic going directly to the origin servers, where they could be quickly overwhelmed by a DDoS attack. By implementing the control as an AWS WAF rule, traffic does not go to your origin servers; AWS scalable infrastructure handles it instead.

8. Third-Party Assets

Should Do

Unless you have a valid use case, do not use web assets, particularly JavaScript, hosted by a third party, such as the jQuery CDN. Using these introduces a vulnerability to users of your service that you cannot control; it also slows down your site.

9. Content Security Policy

Should Do

Content Security Policy (CSP) is one security header that tells browsers where a website should be loading resources and which resources it should be loading.

Follow these steps to configure:

1. Enable CSP in “Report-Only” mode by setting the Content-Security-Policy-Report-Only header and enable the reporting mechanism.
2. See the alphagov/browser-listener (https://github.com/alphagov/browser-listener) GitHub repo for a central reporting tool.
3. Review the browser developer console and Splunk logs and use those to tune the CSP configuration.
4. Change the header to Content-Security-Policy so that browsers can prevent potentially malicious activity.

Why?

Suppose an adversary discovers a vulnerability, such as cross-site scripting (XSS). In that case, they could include content or links to their malicious website - CSP helps prevent this by reducing the successful options available to an attacker.

10. Implement security.txt

Should Do

See vulnerability disclosure (/standards/vulnerability-disclosure.html) for information on how to configure.

You can find an example of implementing a redirect using Lambda@Edge in the alphagov/aws-lambda-at-edge-examples (https://github.com/alphagov/aws-lambda-at-edge-examples) GitHub repo.

You can check the configuration using https://findsecuritycontacts.com/query (https://findsecuritycontacts.com/query).

Why?

A security.txt file helps malware or security researchers contact the Cyber Security team if there’s an issue found on the site.

11. Web Application Firewall

Should Do

Make it more difficult to exploit a weakness in your service which could cause you an incident by enabling a Web Application Firewall (WAF).

If you have dynamic content that takes user input, you should enable AWS WAF and apply appropriate rules. For example, if you are running WordPress, you may want to enable the

managed PHP and “AWSManagedRulesWordPressRuleSet” rulesets, but “AWSManagedRulesWindowsRuleSet” is likely inappropriate as WordPress does not typically run on Windows.

Even if you do not have dynamic content, a WAF - even one with no enabled rules - can be helpful if you have ever have an incident in future by allowing for quick configuration.

Make sure you test after enabling WAF rules so they do not impact usability.

See more information on the Use a web application firewall (WAF) (/standards/web-application-firewall.html) page.

12. AWS Shield Response Team

Should Do

If you use the Cabinet Office AWS environment:

Enable the AWS Shield Response Team (SRT), previously known as DDoS Response Team (DRT), against applicable resources like CloudFront and Application Load Balancers (ALBs) - see the Amazon documentation on Shield Advanced and the SRT (https://docs.aws.amazon.com/waf/latest/developerguide/authorize-DRT.html).

You likely need to configure the following:

- DRT IAM (Identity and Access Management) role
- Route53 DNS health check(s)
- blank WAF(s)
- Proactive Engagement - ask the 🔒 CDIO Cyber Security (https://gds.slack.com/messages/CCMPJKFDK/) team for contact information to input here

Why?

The Cabinet Office pays for an enhanced service (Shield Advanced), enabling AWS to support us when certain attacks impact our web services. This allows us to utilise AWS’ expertise and take some of the pressure off when there’s an incident. Without this, your website or service may be unavailable for longer while mitigating malicious activity.

13. Vulnerability Scanning

Should Do

Find out about potential weaknesses early.

Sign up to NCSC's Web Check service with your gov.uk email address to have NCSC scan and report common findings to you: https://www.ncsc.gov.uk/information/web-check (https://www.ncsc.gov.uk/information/web-check).

Contact the 🔒 CDIO Cyber Security (https://gds.slack.com/messages/CCMPJKFDK/) team to have regular vulnerability scanning enabled.

Use https://www.ssllabs.com/ssltest (https://www.ssllabs.com/ssltest) to check your Transport Layer Security (TLS) settings (formerly known as SSL or Secure Sockets Layer) - tick the "Do not show the results on the boards".

Why?

Vulnerability scanning gives you an insight into what common weaknesses are present in your website or service; this may allow you to patch or mitigate the weakness before it's exploited and becomes an incident.

14. Backup Static Origin

Could Do

Have a simple static version of your site with reduced functionality; this could be used in the event of a CDN outage or if your dynamic origin becomes unavailable.

You could implement a scraper in AWS Lambda to regularly take a copy of your dynamic site. The scraper could filter only your domain or relative URLs, disable any forms (remove the form action and disable the submit button), and then upload the contents to a public S3 bucket.

15. Usage Analytics

Could Do

Consider using server analytics instead of remote analytics.

If you use remote analytics and handle sensitive data, such as user login, you should not include the remote JavaScript (as is the case with Google Analytics) on those paths.

Why?

Remote analytics, such as Google Analytics, introduces a vulnerability to your service and requires relaxing the content security policy. It can also slow down the site for users with slow devices or where they have low bandwidth.

This page was set to be reviewed before 27 December 2024 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way). This might mean the content is out of date.

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

How to manage access to your third-party service accounts

Your GDS team probably uses third-party tools to develop a service, such as GitHub and logit. Follow this guidance to manage access to any associated accounts.

When managing third-party service accounts ensure:

- only the correct people in your team have access
- team members have account access revoked when they leave
- you identify who in your team is doing what and when (this is more important for some tools, like Amazon CloudTrail (https://aws.amazon.com/cloudtrail/), than others like BrowserStack (https://www.browserstack.com/)).

Securely managing account credentials

You're likely to need a set of credentials to gain access to your third-party user account, for example passwords or secret access keys.

When managing these credentials, you must:

- keep the number of secrets you have to a minimum - this reduces the number of credentials to revoke when someone leaves GDS (passwords are particularly problematic secrets, other secrets such as secret access keys or OAuth tokens are more easily rotated)
- never use credentials associated with another individual
- encourage the rest of your team to avoid sharing credentials unnecessarily (technical limitations may mean this is sometimes unavoidable)

Managing accounts with organisation-level access

Often third-party services natively support organisation or team level access, for example the use of alphagov on GitHub (https://github.com/alphagov). In these cases you should make sure only the appropriate individuals on your team have access.

With team or organisation-level accounts, where possible you should:

- create separate accounts for each individual, rather than share credentials
- use a single-sign-on provider rather than create an account with a separate username and password - preferably use Google Workspace as it's a core part of the GDS leaver process, which means that access to linked third-party services will be revoked automatically when someone leaves
- make sure that you have a documented process for removing individuals's access when they leave GDS

Managing accounts with individual-level access

When services do not support organisation-level access, you may have to share individuals' sign-on credentials.

Sharing individuals' sign-on credentials can create risks as:

- you do not always know who can access a service (particularly if users only access a service rarely because you can forget who has access)
- the last person with account credentials may leave GDS, making it impossible for others to access a service in future

To reduce these risks, you should:

- create an account using a Google Group email address rather than an individual's email address
- set posting permissions to 'public' so your group can receive emails from outside GDS
- set viewing topic permissions to 'all members of the group' so only users with access to a service are able to view posts
- store passwords in your team's shared credential store (/standards/storing-credentials.html)
- use two-factor authentication
- make sure you have a documented process for removing an individual's access to the Google Group, credential store and for rotating credentials including any API keys

Managing GitHub accounts

When you join GDS, you will need a GitHub account. You can either use your personal account or create a new account. Either way, please add your @digital.cabinet-office.gov.uk email address to your GitHub account (https://github.com/settings/emails). You do not need to make it the primary address or public.

Each GDS team should have someone (usually a technical lead) who is an owner and can grant/revoke access to their repositories.

Managing Logit accounts

Use Google Apps Marketplace when you sign into Logit (this is the only available option).

Managing Amazon Web Services (AWS) accounts

The GDS Way provides guidance on how to access an AWS account, create a new account and account management (/manuals/working-with-aws-accounts.html).

This page was set to be reviewed before 23 January 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way). This might mean the content is out of date.

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

How to manage alerts

Your service should have a system in place to send automated alerts if its monitoring system(s) detects a problem. Sending alerts help services meet service level agreements (SLAs), and provide awareness of suspicious activity to enable incident response.

Sending alerts

Your service should send an alert when your service monitoring (/standards/monitoring.html) detects an issue that:

- affects service users
- requires action to fix
- lasts for a sustained period of time
- indicates compromise or suspicious activity (such as multiple failed login attempts or unrecognised escalation of privilege)

You should only send an alert for things that need action. Alert text should be specific and include actionable information (https://response.pagerduty.com/oncall/alerting_principles/#alert-content). You should not include sensitive material.

Create alerts based on events affecting users, such as slow service requests or unavailable webpages. You should also track events like disks being full or almost full, or databases being down. These are both events that can lead to more serious alerts that do affect users.

You must set up your alerts to distinguish between when something is wrong for a sustained period of time, and when it's a brief issue caused by temporary network conditions. Prometheus (https://prometheus.io/) supports this by using the `for` parameter in its alerting rule, which indicates the condition has to be true for a set period of time before it triggers an alert.

When you should not send an alert

You do not need to send an alert if no action is needed. For example, an alert showing a system's status is really a monitoring tool and you should use a dashboard to display this information instead. The GDS Way has more guidance on how to monitor your service (/standards/monitoring.html).

Specific examples of issues that should not trigger an alert include:

- an individual container instance dying
- a single task invocation failing

This is because the systems running the architecture like Amazon Elastic Container Service (Amazon ECS) (https://aws.amazon.com/ecs/) or Kubernetes (https://kubernetes.io/) will bring the instance back up, and the task will retry.

Situations where there's a long period of fewer than expected instances or repeated task failures should trigger alerts, as they show underlying problems.

Prioritising alerts

You must prioritise alerts based on whether they need an immediate fix. It can help to class issues as:

- interrupting - need immediate investigation and resolution
- non-interrupting - do not need immediate resolution
- security-related - may indicate compromise of the system

The Google Site Reliability Engineering (SRE) (https://sre.google/sre-book/table-of-contents/) handbook classifies "interrupting" issues as "pages", and "non-interrupting" issues as "tickets". Put non-interrupting alerts into a ticket queue for your support team to solve. Keep the ticket queue and team backlog separate to avoid confusion. You should specify an SLA for how long both types of alert take to resolve.

You should manage alerts using a dedicated tool which will allow you to:

- manage alerts in a queue
- triage alerts for review
- attach statuses to alerts, for example, in progress or resolved
- gather data about alerts to help improve processes

Recommended tools are:

- PagerDuty (https://www.pagerduty.com) to send high-priority / interrupting alerts
- Zendesk (https://www.zendesk.com) to manage non-interrupting alerts as tickets
- Splunk (https://splunk.com) to manage security-related alerts

You can also configure these tools to send alert notifications using email or Slack. However, you should only use email and Slack as additions to your primary alerting tool. If alerts only go to email or Slack, people may ignore, overlook, filter them out, or treat them like spam.

We recommend using dashboards and information radiators, also known as Big Visible Charts (BVC), like Smashing (https://github.com/Smashing/smashing) or BlinkenJS (https://github.com/alphagov/blinkenjs) in combination with alert management tools.

Further reading

For more information refer to the:

- Service Manual for more information on how to write alerts (https://www.gov.uk/service-manual/technology/monitoring-the-status-of-your-service)
- GDS Way for information about monitoring (/standards/monitoring.html)
- Google SRE handbook to find out more about site reliability engineering (https://sre.google/sre-book/table-of-contents/)

This page was last reviewed on 14 February 2025. It needs to be reviewed again on 14 August 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Documenting architecture decisions

You should record decisions that affect the architecture of your service, in order to preserve the context of your choices.

As agile projects age, it is sometimes hard to keep track of the reasoning behind the decisions made. This is especially true as new people join the projects when those involved in the early stages are no longer around.

It is important to preserve the reasoning so the current team can include it as context when making their own decisions about changes they need to make. For example, understanding whether a particular choice was made for the sake of expediency and can therefore be changed with little impact, or whether there were external reasons behind that decision that need to be factored in.

How to document decisions

Architecture decisions should be stored in version control so there is a record of what was changed, who by, and when. Decisions that affect a specific application should be in that application's code repository. You may also want to store larger-scale decisions in a central documentation repository.

A suggested format is the Architecture Decision Record (ADR), proposed by Michael Nygard in a blog post (https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions) and since adopted widely. That post describes the format in full, but as a summary it consists of the following sections:

- **Title**: a description of the decision (not the problem)
- **Status**: see lifecycle
- **Context**: the facts behind the need to make the decision
- **Decision**: what the team has decided to do
- **Consequences**: both positive and negative consequences of the decision

Amazon Web Services (AWS) also provide useful advice (https://docs.aws.amazon.com/prescriptive-guidance/latest/architectural-decision-records/welcome.html) about how and why the ADR process should be adopted.

Lifecycle of an ADR

In a new project or one with a lot of decisions to be made, consider setting up regular ADR discussion and review meetings. This will give the team time to work through these decisions.

Each decision goes through the following phases.

“Proposed” status

Assuming you’re using GitHub for version control, there is no need to record a ‘status’ for an ADR when it is first proposed. The status of your ADR’s pull request (PR) reflects the status of the decision until it has been accepted.

Consider using GitHub’s “draft PR” feature especially if you have lots of decisions at varying levels of readiness.

It’s important to find the right group of reviewers for your ADR. Ensure that you have input from all relevant technical disciplines (such as site reliability engineers, developers, security architects, etc) and from any teams that might be affected.

Some programmes use technical writers to ensure their ADRs meet our quality standards and are as accessible to the widest audience possible. Future team members that lack the full context you have for a decision will appreciate your efforts!

“Accepted” status

Once an ADR has been accepted then teams may need to know about its existence long into the future. Think about how to make your ADRs discoverable, especially if they require more than just a one-off set of code or infrastructure changes.

Again, the “accepted” status doesn’t need to be explicit in your document - it can be assumed that an ADR in the `main` branch of your GitHub repository is accepted. Otherwise, teams can get confused about the real status of a decision, as the ADR goes through the pull request process!

Implementation

In some cases there will be an implementation phase during which you may discover that the decision was wrong, or needs to be amended.

If there are clarifications, then updating your existing ADR may be appropriate.

You might find additional consequences to your decision, so consider updating the Consequences section to reflect your evolving understanding of the problem space.

If a decision was never implemented at all, then as long as your reviewers and other stakeholders agree then it may be appropriate to update your existing ADR with a new decision.

However, if some implementation of the original decision has occurred, you should write a new ADR to explain the new decision.

Any ADRs that have not been fully implemented across all the relevant teams should be a topic for your regular discussion and review meeting.

Consider using your issue tracker to make sure that decisions you've made are followed through and put into effect. The work isn't done until the decision is implemented!

"Superseded" status

If a decision has been superseded by another decision, the old ADR must be clearly marked as "superseded".

A "Status: superseded" line should appear directly under the heading. Add a link to the new ADR as soon as it's been accepted.

What an ADR is not

An ADR isn't a complete description of a software system or architecture.

Some teams use requests for comment (RFCs), wikis, team manuals, or other collaboration tools to document their architecture, and these should be updated whenever a new ADR is accepted.

This page was last reviewed on 20 November 2024. It needs to be reviewed again on 20 May 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Breaking down work

Working sustainably in small batches and performing regular releases is considered industry best practice (https://en.wikipedia.org/wiki/DevOps_Research_and_Assessment#DORA_Four_Key_Metrics). It reduces the risk of errors being introduced, makes delivery of value more predictable, and enables us to learn faster.

Our perception of value is not just putting working software and new features into the hands of users. It□s also:

- Learning about user needs.
- Improving operational resilience.
- Minimising toil (https://sre.google/sre-book/eliminating-toil/).
- Reducing cost of change.

We should aim to break down our work into independently releaseable pieces that each deliver their own value.

Why break down work?

There are many reasons why working in smaller batches provides an advantage in software development.

- It articulates value. If we are able to clearly state where the value is in a piece of work then we are more able to break it down into valuable milestones.
- It reduces waste. Code that is written but not integrated and being used is wasted effort, and we want to minimise that waste.
- It promotes predictable delivery. We often need to know when a feature is likely to be complete and having clear regular milestones facilitates those conversations with evidence rather than estimates.
- It enables fast feedback. We can learn a lot from regularly releasing our code to production environments even if the effects are not visible to users. We also get feedback on our code design because well-written code will be easier to iterate on.

- It boosts team morale. The unit of delivery is the team, and delivering value regularly is much better for morale than working on big-bang releases for months at a time.

How do we break down work?

- Example mapping (https://cucumber.io/blog/bdd/example-mapping-introduction/). Splitting up a large feature into individual capabilities will demonstrate where the dependencies are, which are the most valuable, and the subsequent ideal ordering.
- Get work live as soon as you can. Deploy changes safely behind feature toggles or limited to lower (non-production) environments to get fast feedback on the thing you've changed. For example, you might integrate your code with a third-party service but discard the result rather than take action; Martin Fowler calls this dark launching (https://martinfowler.com/bliki/DarkLaunching.html).
- Finish one thing at a time. Having too much work in progress (WIP) reduces the team's ability to react to change and feedback, and it's much better to avoid context switching. If your team has lots of work in flight because there are lots of blockers, then work with your delivery colleagues to establish why this is happening.
- Loosely coupled components and systems. Systems that need to be changed together will block each other, so try to build systems that have well-defined contracts but also operate under the robustness principle (be conservative in what you send, liberal in what you accept). For example, do not rely on ordering for JSON objects and ignore unexpected fields rather than cause an error.
- Failing early and safely. Working in small batches reduces risk of bugs, but when those bugs do happen it's much easier to roll back a small change than a large one. Your team should be aspiring to some form of automated rollback in case of errors in production environments, such as blue-green deployments or canary releases.

Further reading

Find out more about breaking down work:

- Why breaking down work is important (https://blog.probablyfine.co.uk/2024/09/27/why-breaking-down-work-is-important.html) - an expanded version of this page
- Software has diseconomies of scale – not economies of scale (https://www.allankelly.net/archives/472/software-has-diseconomies-of-scale-not/)

This page was last reviewed on 10 October 2024. It needs to be reviewed again on 10 April 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Use configuration management

Use configuration management (https://www.digitalocean.com/community/tutorials/an-introduction-to-con□guration-management) to manage, automate and standardise your infrastructure. When using configuration management you store your infrastructure as code (https://www.gov.uk/service-manual/technology/manage-your-software-con□guration#use-infrastructure-as-code) in a version control system such as Git.

Puppet

The use of Puppet (https://puppet.com/) at GDS is diminishing as we move more of our infrastructure to containers and higher level services. It's mainly still in use on GOV.UK (https://github.com/alphagov/govuk-puppet) but this will decline as more services are moved over to AWS EKS.

If your environment consists of a simple deployment artefact like an Amazon Machine Image (AMI) (https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html), Puppet may not be necessary, but the process for building that artefact must still be codified and version controlled.

Terraform

Use Terraform (https://www.terraform.io/) to configure third party cloud infrastructure like Amazon Web Services (AWS) or Fastly (https://www.fastly.com/).

Terraform supports a large number of providers (https://www.terraform.io/docs/language/providers/index.html), and you can configure it to support multiple environments with different parameters. See the govuk-aws (https://github.com/alphagov/govuk-aws) repository as an example.

Versioning

Due to the high rate of change in many cloud provider offerings we recommend you keep your Terraform versions and codebases up to date. A version manager such as tfenv

(https://github.com/tfutils/tfenv), already used by a number of GDS teams, can help you with supporting multiple versions.

Code anal sis

There are a number of Terraform focused static analysis tools in use at GDS. While none of them are yet ubiquitous they can help ensure your code is more idiomatic, consistent and secure and you should consider the benefits they could bring to your build pipelines.

- checkov (https://github.com/bridgecrewio/checkov) - “detects security and compliance misconfigurations”
- tfsec (https://github.com/aquasecurity/tfsec) - “spots potential security issues”
- tflint (https://github.com/terraform-linters/t int) “linter focused on possible errors, best practices and so on.”

Further reading

Find out more about configuration management in the Service Manual (https://www.gov.uk/service-manual/technology/manage-your-software-con guration#using-con guration-management-tools).

This page was last reviewed on 18 February 2025. It needs to be reviewed again on 18 August 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Use continuous delivery

Continuous delivery (https://www.continuousdelivery.com) helps you implement changes like new features into production quickly and safely.

Benefits

Using continuous delivery has multiple benefits.

Iterate code frequently

It's cheaper and easier to deliver small changes to production, meaning you can get feedback from users quickly.

Low-risk releases

Frequent small changes make it easier to diagnose problems, and the cost of fixing them is much lower than with large releases.

Quality software builds

Automated test suites allow you to quickly identify regressions in your software. This means you can focus on exploratory, usability, performance and security testing.

Maintainable code

Building and releasing software in small pieces helps you focus on writing small composable bits of code. This means your code is easier to maintain and update.

Essential concepts

To achieve continuous delivery you need:

- frequent integration to the main branch
- automatic build promotion
- production monitoring

Frequent integration to the main branch

We generally use Pull Requests (/standards/pull-requests.html) for our development processes. Long-lived branches mean a longer wait time to see changes in production, and a greater risk of merge conflicts. You should aim to have small Pull Requests, so that they are easier to review and can be merged more quickly. You should aim for your feature branches to live for only a day or two before getting merged. If a branch lasts longer than this, you could ask for help, or consider a different approach to the problem.

You can use approaches like feature-flagging (https://martinfowler.com/articles/feature-toggles.html) or modular architectures (https://continuousdelivery.com/implementing/architecture/) to deploy partially complete features. This reduces the size of changes going to production and encourages your team to build modular, configurable systems.

You should break down work (/standards/breaking-down-work.html) into chunks that make continuous delivery easier to achieve as opposed to large chunks and big-bang releases.

Automatic build promotion

You can quickly distinguish between good and bad builds with automatic build promotion. By deploying builds that have to pass multiple test jobs downstream of the initial build process you can get quicker feedback if the build fails.

Use production monitoring and alerting

You can understand the effect of your changes on production using production monitoring and alerting (/standards/monitoring.html). Monitoring essential parts of your system allows you to see if changes have any unintended impacts and to respond quickly in case of problems.

Further reading

Find out more about continuous delivery from:

- Architecting for Continuous Delivery (https://www.youtube.com/watch?v=Lx9ssegE6FA) - Jez Humble at Agile India 2016 (video)
- 2024 State of DevOps Report (https://dora.dev/research/2024/dora-report/) - see where you are and how to get to the next stage
- Accelerate: The Science of Lean Software and Devops: Building and Scaling High Performing Technology Organizations (https://wordery.com/accelerate-nicole-forsgren-phd-

9781942788331?cTrk=OTc2NDYwN□ Z8NWI2ZDg5NGJkY□ AyZjoxOjE6NWI2ZDg5NDQwM2ZhODguND□□TgxMTU6ODJlODM3ODY%3D) - by Nicole Forsgren Jez Humble Gene Kim

- Trunk Based Development (https://trunkbaseddevelopment.com/) - a source control branching method

This page was last reviewed on 19 November 2024. It needs to be reviewed again on 19 November 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Diagrams as code

Code can be used to de□ne the content, structure and appearance of a diagram which can then be rendered, instead of using a drawing tool directly. This approach is known as □ diagrams as code□ .

What are diagrams for?

> A good diagram enables you to more easily convey your design to others, and it provides a focus for your creative, as well as your technical thinking.

□ Drawing good architecture diagrams (https://www.ncsc.gov.uk/blog-post/drawing-good-architecture-diagrams) tips from NCSC

There are many ways to create a diagram, from pen and paper or whiteboard through to specialist diagramming software. The right tool will depend on the circumstances, including what the diagram is for and who needs to be able to contribute to it, but it□s always worth considering a diagram generated from code.

Diagrams are a form of documentation, and some teams already use a □docs as code (https://technology.blog.gov.uk/2017/08/25/why-we-use-a-docs-as-code-approach-for-technical-documentation/)□ approach. Diagrams as code builds on the rationale and ben□ts of docs as code, including

- keeping documentation close to source code
- ease of change
- a work□ow to review changes
- version history
- consistency

Accessibility

Refer to guidance on making images or charts accessible (https://www.gov.uk/guidance/publishing-accessible-documents).

Since the source of the diagram provides a text based representation of the generated image, this probably makes it more accessible than a diagram drawn through a graphic editor. It may be preferable to signpost users to get to the source of the diagram rather than duplicating it alongside the diagram itself.

Diagrams as code tools

Mermaid

Assuming you already use GitHub (https://gds-way.digital.cabinet-office.gov.uk/standards/source-code/use-github.html) then you can use Mermaid in Markdown files (https://github.blog/developer-skills/github/include-diagrams-markdown-files-mermaid/).

The Mermaid Live Editor (https://mermaid.live/) is a good way of trying out Mermaid, and seeing diagrams rendered from code in real time.

> Mermaid will automatically insert the aria-roledescription and, if provided in the diagram text by the diagram author, the accessible title and description.

- [ ] Mermaid Accessibility Options (https://mermaid.js.org/config/accessibility.html)

This works in a similar way to the `alt` attribute of an `<img>` element.

```
graph LR
  accTitle: Accessible Title
  accDescr: Accessible Description
```

We should consider contributing to Mermaid (https://mermaid.js.org/community/contributing.html) if there are aspects of its accessibility we are able to improve.

Open source diagrams as code

Consider making the code for your diagrams open in the same way as your other source code (https://gds-way.digital.cabinet-office.gov.uk/standards/source-code/index.html#publish-open-source-code).

Sequence diagrams as code

A sequence diagram (https://en.wikipedia.org/wiki/Sequence_diagram) is used to show how different entities in a system interact with each other and can be a useful way of designing new functionality or documenting existing behaviour.

Mermaid can render sequence diagrams (https://mermaid.js.org/syntax/sequenceDiagram.html). See the GOV.UK Forms GitHub repo (https://github.com/alphagov/forms/tree/main/diagrams/sequence-diagrams) for some examples.

This page was last reviewed on 19 December 2024. It needs to be reviewed again on 19 June 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Disaster Recovery

The GOV.UK Service Manual (https://www.gov.uk/service-manual) describes how you should design your service to maximise uptime (https://www.gov.uk/service-manual/technology/uptime-and-availability-keeping-your-service-online). However, it is inevitable that your service will experience unscheduled downtime at some point, which could have a serious impact on users. You must plan your response for when this happens.

What is disaster recovery?

In digital services, a disaster is a major, unplanned event that completely prevents you from providing your service to your users.

A disaster may also be an incident. However, whereas an incident will be handled as part of your incident management process (/standards/incident-management.html), some events may impact your service to such an extent that they need to be planned for in advance to ensure you can continue to provide your service if they occurred.

Examples might be:

- a cloud provider suffering a serious regional outage
- accidental or malicious data loss
- a security compromise resulting in the disclosure of sensitive information

Disaster recovery planning is the process of identifying the kinds of events that might make your service completely unavailable, understanding the likelihood and the impact of them occurring, planning what we would do if they occurred, and testing that our recovery plans work.

Understand risks and threats to your service

You should work with the 🔒 Information Security (https://sites.google.com/a/digital.cabinet-office.gov.uk/gds/directorates-and-groups/cto-and-ciso-office/information-security) and 🔒 Cyber Security (https://intranet.cabinetoffice.gov.uk/it-data-and-security/cyber-and-information-security-

services/) teams to understand the risks to your service. This will help you build a more resilient and secure digital service.

You should also work with risk and service owners to plan for the worst-case scenarios. This is particularly important for your data, as loss or theft of data is disastrous for most services.

Make sure you consider how disruption to or unavailability of your dependencies (https://www.gov.uk/service-manual/technology/uptime-and-availability-keeping-your-service-online#issues-that-can-affect-uptime) can affect your service. These dependencies can include obvious systems, such as your hosting provider, but also less obvious systems such as your:

- Domain Name System (DNS) provider
- Content Delivery Network (CDN)
- any Virtual Private Networks (VPNs) your development team might use to access production systems build pipelines
- external library and container registries (for example, RubyGems, PyPI or Docker Hub)
- secrets management providers

Disaster planning workshops

You might find it useful to run a disaster planning workshop to help identify the risks to your service. This should involve important service stakeholders, Information Security and/or Cyber Security colleagues, as well as product and technical members of your delivery team. The more involvement you have from different disciplines the more likely you will be to identify all the risks and understand them.

You should use a whiteboard or an online alternative for the workshop. Start with a technical architecture diagram of your service to help visualise dependencies and important assets, such as data stores. Then, either individually or in small groups, identify as many disaster scenarios as possible, raising sticky notes on the diagram for each one.

Once you have identified the risks to your service, the workshop members should go through each scenario and score each one in terms of the likelihood of them happening and the potential impact on the service if they occurred. A 0-5 scoring system is usually adequate for this. Again, the scoring will be more accurate the more disciplines are involved. An overall score can then be calculated for each scenario by multiplying likelihood and impact.

When all the scenarios have been scored, you will be left with a set of disaster scenarios that can be placed in priority order. You will need to decide what to do about each risk. You may need to draw up a plan for how you would recover if the scenario occurred. Alternatively, you may decide to do some work ahead of time to reduce the likelihood or impact of the event occurring. If the likelihood and impact are very low, you may decide to accept the risk and do nothing.

Back up your data

You should back up your data. How often you back up will depend on your Recovery Point Objective (RPO) (https://cloud.google.com/architecture/dr-scenarios-planning-guide#basics_of_dr_planning). An RPO determines what you consider an acceptable loss of data between the last backup and the point of data loss. For example, if you determine that up to 24 hours of data loss is an acceptable risk, you will only need to backup your data once daily. For lower RPOs (minutes or seconds), you should use continuous backups and point-in-time recovery (https://docs.aws.amazon.com/aws-backup/latest/devguide/point-in-time-recovery.html) services that most modern cloud providers offer.

Have cold or offline backups

A cold or offline backup is a copy of your backup, isolated from the rest of your production system. Offline backups help protect you from some disaster scenarios. If an attacker gains access to your live database service and your backups, you could experience a ransomware attack, or risk having your entire data estate deleted. You should implement offline backups so that even users with administrative access cannot alter or delete data.

Offline backups also help protect your data from accidental deletion. For example, copy your backups to a different account with limited access rights. Make sure:

- the backups are encrypted and versioned
- the account you are copying the backups from only has write permissions

Decide where to store your backups

Where you are storing your backups is also important for disaster recovery. Most cloud storage solutions have durability guarantees. However, an outage could occur in the region containing your live service and backups. If your service has very high availability requirements, you may need copies of your data in a different region, or with a different service provider. Your conversations with Information Security and service owners can help you determine your risk tolerance for these hopefully rare scenarios.

The National Cyber Security Centre (NCSC) article Of□ine backups in an online world (https://www.ncsc.gov.uk/blog-post/of□ine-backups-in-an-online-world) further explains why offline backups are important and suggests some approaches to use in a cloud environment.

Decide on manual or automatic recovery in the event of a disaster

Setting a Recovery Time Objective (RTO) (https://cloud.google.com/architecture/dr-scenarios-planning-guide#basics_of_dr_planning) for your service can help you decide your approach to service recovery in the event of a disaster.

For very low RTOs (for example, under one hour), you will need to architect your service to support automatic recovery. For example when hosting services on Amazon Web Services (AWS), ensure workloads are deployed across multiple availability zones (https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html#concepts-availability-zones) for resilience and automatic failover.

You should also consider different RTOs for different types of failure. For example, a service may be designed to withstand a failure in a single availability zone but not a major outage impacting multiple availability zones or an entire AWS region. In such cases, the service could be recreated in another hosting provider or in an alternative region by manually redeploying applications using infrastructure as code (https://www.gov.uk/service-manual/technology/manage-your-software-con□guration#use-infrastructure-as-code) and restoring databases from backups. This is likely to take longer than an hour and hence a higher RTO needs to be set and agreed. If the higher RTO is unacceptable, even in this hopefully rare scenario, the service will need to be rearchitected to ensure it can achieve a lower RTO.

Your disaster recovery strategy will depend on your specific service requirements. You may need different strategies for different parts of your service. The more immediate and more automatic the strategy, the greater the complexity and the cost of running your service is likely to be. The Plan for Disaster Recovery article from AWS (https://docs.aws.amazon.com/wellarchitected/latest/reliability-pillar/plan-for-disaster-recovery-dr.html) covers some common recovery patterns.

Agree your RPOs and RTOs with risk and service owners

Make sure you discuss and agree your RPOs and RTOs with risk and service owners. Not only is their expertise key to decision making, but it's vital they understand what will happen and why in the event of a disaster.

Test your disaster recovery plans

Disasters rarely happen, meaning that your disaster recovery plans are likely to become ineffective unless you test them regularly.

You must regularly test that you can restore data from your live and offline backups. You must test the procedure for restoring an offline backup so team members are familiar with the procedure for accessing and using the backup. It also serves as useful reassurance that backups are functioning correctly.

You must regularly test your manual or automatic recovery processes so that you are confident they will work in an emergency situation. This also helps team members build familiarity with the relevant procedures.

You should organise game days (https://technology.blog.gov.uk/2015/02/06/running-a-game-day-for-gov-uk/) to practise incident management (/standards/incident-management.html) and test disaster recovery procedures. Game days enable you to play out scenarios in a safe

environment so you can test and learn from how you respond. These scenarios will help build familiarity and increase confidence that your emergency procedures will work when you most need them to.

You can also practice incidents or recovery from disaster scenarios in a more low-pressure way. The GOV.UK PaaS team regularly spin the Wheel of Misfortune (https://technology.blog.gov.uk/2021/03/04/wheel-of-misfortune-incident-rehearsal-for-gov-uk-paas/) to help new and existing team members gain experience and confidence working with the platform.

This page was last reviewed on 14 February 2025. It needs to be reviewed again on 14 August 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

View source Report problem GitHub Repo

Accessibility

OGL

All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

How to manage DNS records for your service

To ensure users can connect to your service, you'll need accurate Domain Name System (DNS) records and a DNS provider to supply your DNS nameservers.

You can find information on choosing where to host your DNS and how to request nameserver delegation at GDS in the Service Manual (https://www.gov.uk/service-manual/technology/get-a-domain-name#choose-where-youll-host-your-dns).

As part of your DNS strategy and planning you need to ensure:

- your DNS record configuration is reproducible (for example, the DNS records are version controlled or autogenerated)
- you consider DNS availability as part of overall service availability.

For example, `publishing.service.gov.uk` is published to both Amazon Route 53 and Google Cloud DNS to remove vendor as a single point of failure. This means that even if AWS has a total outage, users can access parts of GOV.UK by using Google Cloud DNS to resolve addresses, and viewing content that is available in the CDN cache.

Note that if your service availability relies on a single vendor, there is less benefit to deploying DNS to multiple vendors.

When you implement your DNS strategy you should consider using:

- Amazon Route 53 (https://aws.amazon.com/route53/) as a cloud DNS web service
- Google Cloud DNS (https://cloud.google.com/dns/) for high nameserver availability

You can read more about service domains and DNS (https://www.gov.uk/service-manual/technology/get-a-domain-name) in the Service Manual.

(https://gds.slack.com/messages/gds-way) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

How to host a service

In GDS, we use Amazon Web Services (AWS) (https://aws.amazon.com) to host our services.

We follow the Government Cloud First policy (https://www.gov.uk/guidance/government-cloud-first-policy) and use AWS managed cloud services and Infrastructure as a Service (IaaS) solutions to host our services rather than using our own hardware.

See also

- Working with AWS accounts (/manuals/working-with-aws-accounts.html)
- Tagging AWS resources (/manuals/aws-tagging.html)
- Use configuration management (/standards/configuration-management.html)

This page was last reviewed on 6 November 2024. It needs to be reviewed again on 6 May 2025 by the page owner #gds-way (https://gds.slack.com/messages/gds-way) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

How to arrange and manage penetration tests

You should aim to run penetration tests (https://www.gov.uk/service-manual/technology/vulnerability-and-penetration-testing) on your service at least every 12 months. You must discuss all signi□cant changes with the GDS 🔒 Information Security (https://sites.google.com/a/digital.cabinet-office.gov.uk/gds/directorates-and-groups/cto-and-ciso-office/information-security) team. You must agree with the 🔒 Information Security (https://sites.google.com/a/digital.cabinet-office.gov.uk/gds/directorates-and-groups/cto-and-ciso-office/information-security) team when you will test and the scope of the tests. They will also assist with the procurement of external tests through an approved third party through the National Cyber Security Centre (NCSC) CHECK scheme (https://www.ncsc.gov.uk/information/using-check-provider). Alternatively, with the agreement of the Info Sec team, a member of the [COD Cyber] Team can carry them out internally, depending on the requirements.

Information Security have a GDS-level contract for ITHC services, which should make obtaining an ITHC for your service a more streamlined process.

You might need to schedule additional testing if you make signi□cant changes to your service. You should meet with the Information Security team regularly to discuss ongoing changes.

A signi□cant change could be when you:

- change a cloud service provider
- change stored data, for example if you introduce new data which can be classi□ed as personal data under GDPR (https://commission.europa.eu/law/law-topic/data-protection/reform/what-personal-data_en)
- implement signi□cant application changes or new features

You might need to use CSPs to assess the addition of a third-party partner, for example, a database processor or email provider (especially if the third-party partner is processing personal data)

Scope your test

An IT Health Check or security review can include:

- application penetration tests
- external network penetration tests
- firewall rulebase/ruleset reviews
- server build reviews
- networking and networking device Access Control List (ACL) (/standards/secrets-acl.html) reviews
- code reviews
- infrastructure-as-code reviews
- AWS configuration reviews
- red team engagements
- vulnerability scans

Before arranging a test you should consult with the Information Security team on:

- the beginning and end test dates. This will be an agreement between the team and the tester(s) based on the size of the project, rather than dictated to them
- the areas you want the tester to target, for example, bypassing authentication
- what you should exclude, for example, third-party managed infrastructure
- exploits that are out of scope, such as DoS attacks
- any specific technical capabilities to allow third-party testers to complete testing, for example, experience working with AWS security groups
- the specific technical scope of the test including IP addresses, URLs and GitHub repositories
- what technical documentation and tools are needed to facilitate testing and understanding of the application. For example, design documents, network architecture diagrams and technical configurations e.g. Swagger/Postman documentation for API tests

Schedule a test

To schedule a test, 🔒 Information Security (https://sites.google.com/a/digital.cabinet-office.gov.uk/gds/directorates-and-groups/cto-and-ciso-office/information-security) team.

If you plan to test any application, you must contact the Info Sec team at least 3 months in advance so they can organise the procurement (or call-off against the existing framework) for you.

If you are planning to ask the 🔒 Cabinet Office Cyber (https://intranet.cabinetoffice.gov.uk/it-data-and-security/cyber-and-information-security-services/) team to perform a test, you will need to enter the information listed in the scope your test section and the prepare for your test section into a Rules of Engagement document, where a scope can be agreed and signed off by both parties. As with an external company, you should give at least 3 months' notice to make sure you can schedule the test at a time that suits project timelines.

Prepare for your test

Before the test, you will be expected to share documentation with the testers, for example, up-to-date architecture diagrams. The documentation could also include information about the individual components of each device and application being tested.

You should run the tests on a separate test environment which replicates the behaviour of your live service.

To prepare your test environment you should:

- give the tester all the credentials, certificates and authentication they need to start immediately
- provide a technical person to contact in case the tester has any queries and to assist with any technical issues (e.g. provisioning accounts)
- note down the IP addresses of the testers and if necessary, add those IP addresses to any allow lists, making sure to remove them when testing has finished
- create temporary credentials for testers (testers should provide their own SSH public keys)
- give the tester the privileges required for the test, such as sudo access where appropriate
- notify your service providers in advance, for example by emailing GOV.UK PaaS Support - note that in most cases AWS do not require advance permission for penetration tests on your applications
- give the tester a distribution list of approved report recipients

Prior to the test, it may be beneficial to meet the lead tester and the Information Security team to discuss the test and confirm that all the prerequisites and necessary access are in place

During the Test

The lead tester should draw your attention and that of the 🔒 Information Security (https://sites.google.com/a/digital.cabinet-office.gov.uk/gds/directorates-and-groups/cto-and-ciso-office/information-security) team to any critical vulnerabilities immediately identified

It is advisable to meet at the end of each day with the lead tester and the Information Security team to discuss findings and the progress of the test.

What to do after testing

After your test, you should meet with the Information Security team to discuss and triage (risk assess) the test results. You can then prioritise work to mitigate any issues identified in the test and schedule a retest if needed.

Teams should work with the [COD Cyber] team, who can give advice, consult on fixing any issues and take appropriate further action when required.

This page was last reviewed on 27 February 2025. It needs to be reviewed again on 27 August 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

How to manage technical incidents

GDS incident management focuses on restoring normal operations quickly with minimal impact on users.

Technical incidents might also be cyber security or data loss incidents. You must report all suspected or actual cyber security incidents to the 🔒 CO:D Cyber Security team (https://intranet.cabinetoffice.gov.uk/it-data-and-security/cyber-and-information-security-services/) and to the 🔒 GDS Information Security team (https://sites.google.com/a/digital.cabinet-office.gov.uk/gds/directorates-and-groups/cto-and-ciso-office/information-security) using the 🔒 Incident response process (https://sites.google.com/a/digital.cabinet-office.gov.uk/gds/directorates-and-groups/cto-and-ciso-office/information-security/incident-response-process). You must report all actual or suspected data breach incidents to the 🔒 GDS Information Management team (https://sites.google.com/a/digital.cabinet-office.gov.uk/gds/information-management). These requirements should be included in your service manual/guides/processes.

Define incident priority

Define technical incident priority levels for your service's applications. For example potential incidents include:

- system access problems
- wider technical failures with possible reputational impact to GDS
- denial of service (DoS)
- data breach or leak
- defacement
- unauthorised use of systems
- suspicious activity, such as traffic from an unknown source

Assign a priority level to incidents based on their complexity, urgency and resolution time. Incident severity also determines response times and support level.

Example incident priority table

Classification	Type	Example	Response time	Update frequency
P1	Critical	Complete outage, or ongoing unauthorised access	20 minutes (office and out of hours)	1 hour
P2	Major	Substantial degradation of service	60 minutes (office and out of hours)	2 hours
P3	Significant	Users experiencing intermittent or degraded service due to platform issue	2 hours (office hours only)	Once after 2 business days
P4	Minor	Component failure that does not immediately impact a service, or an unsuccessful DoS attempt	1 business day (office hours only)	Once after 5 business days

Develop an incident workflow

Your team must understand what to do during an incident. Develop and document your incident workflow to reflect your service needs and team size.

Example workflow

Follow a prepared workflow to manage an incident to minimise its impact on your team and service users. Make sure that every step of the way is documented in writing using the incident report template (https://docs.google.com/document/d/1YDA13RU6wicXoKgDv5VucJe3o_Z0k_Qhug9EJC_XdSE/) [^1].

1. Establish an incident lead.
2. Inform your team.
3. Prioritise the incident.
4. Form an incident response team.
5. Investigate.
6. Contain.
7. Eradicate.

8. Recover.
9. Communicate to a wider audience.
10. Resolve the incident.

1. Establish an incident lead

Establish who your incident lead is. Find out who noticed the problem and if anyone else is investigating and fixing it. If that person is you, assume the role of incident lead.

2. Inform your team

Inform your team using your chosen tool, like Slack (https://gds.slack.com). If the incident involves a data or security breach, you must also notify:

- The 🔒 CO:D Cyber Security team (https://intranet.cabinetoffice.gov.uk/it-data-and-security/cyber-and-information-security-services/). You can also use the 🔒 #cyber-security-help Slack channel (https://gds.slack.com/messages/CCMPJKFDK/) to contact CO:D Cyber within normal working hours.
- The 🔒 GDS Information Security team (https://sites.google.com/a/digital.cabinet-office.gov.uk/gds/directorates-and-groups/cto-and-ciso-office/information-security).
- The 🔒 GDS Information Management team (https://sites.google.com/a/digital.cabinet-office.gov.uk/gds/information-management).

3. Prioritise the incident

Prioritise the incident and start tracking actions, updates and communications. Teams like GOV.UK PaaS (https://www.cloud.service.gov.uk/) and Notify (https://www.notifications.service.gov.uk/) do this by creating a new incident report - copied from the incident report template (https://docs.google.com/document/d/1YDA13RU6wicXoKgDv5VucJe3o_Z0k_Qhug9EJC_XdSE/) - and use it to track updates and progress.

4. Form an incident response team

Form a team with both an incident lead and a communications lead. The communications lead will make sure relevant parties are updated according to the incident priority table.

5. Investigate

Make sure you keep your incident report up to date. If the incident involves a data breach follow your team's GDPR documentation.

If the incident is a data or security breach you should follow steps 6, 7 and 8. If the incident is not cyber security-related, skip to step 9.

6. Contain

You should determine the right containment procedures. In some cases, you may require a forensic clone.

6.1 Short-term containment

You should start short-term containment measures as soon as you detect an incident. This could help minimise impact and maintain availability. Make sure that all affected systems are isolated from the non-affected systems.

6.2 Long-term containment

You'll need to make sure long-term containment is in place.

You should take the system offline if possible. Once the system is offline, you can proceed to step 7.

If the system has to remain in production, remove all malware and other artifacts from the affected systems, and harden the affected systems from further attacks. You should reimage the affected systems, or restore from the last known good backup.

6.3 Forensic clone

As well as gathering evidence to help resolve an incident, you should collect evidence to support any potential follow-on disciplinary or legal proceedings.

To maintain the forensic integrity of the environment you should:

- document all commands used during the investigation and keep the documentation up to date - include how the evidence has been preserved
- store any forensic images taken during the investigation in a secure location, to prevent accidental damage or tampering

7. Eradicate

Eradication may be necessary to remove components of the incident that remain on your systems, such as traces of malware. To help with eradication you should:

- identify all affected hosts
- remove all malware and other artifacts left behind by the attackers
- reimage and patch the affected system
- check backups, code, images and the affected systems are protected against further attacks

8. Recover

Recovery is necessary to reduce the impact on user confidence and to reduce the likelihood of further successful attacks.

You should:

- confirm the affected systems are patched and hardened against the recent attack, and possible future attacks
- decide what day and time to restore the affected systems back into production (if they were taken offline)
- check the systems you're restoring to production are not compromised in the same way as the original incident
- consider how long to monitor (/standards/monitoring.html) the restored systems for, and what to look out for

9. Communicate to a wider audience

If the incident is serious (P1 or P2) you'll need to contact a wider GDS audience and potentially your service users.

Your communications lead must manage:

- external and internal communications
- incident escalations

External and internal communications

Make sure internal and external parties, like Information Security or your service users are fully informed at every stage of your incident management process.

For example, teams including GOV.UK Platform as a Service (PaaS) (https://status.cloud.service.gov.uk/), GOV.UK Notify (https://www.notifications.service.gov.uk/) and GOV.UK Pay (https://www.payments.service.gov.uk/) use the StatusPage service (https://www.atlassian.com/software/statuspage) to trigger notifications to subscribed users.

Post regular updates to the status of an incident in the 🔒 #incident Slack channel (https://gds.slack.com/messages/CAD6S2B9Q). This helps people across GDS without having to find and follow multiple notification mechanisms for the different programmes.

Incident escalations

Notify internal escalation contacts of all high priority incidents (P1/P2). Contact the 🔒 GDS Information Security team (https://sites.google.com/a/digital.cabinet-office.gov.uk/gds/directorates-and-groups/cto-and-ciso-office/information-security) if you need help defining the escalation route for your service.

Report cyber security incidents

The incident lead, guided by the Information Security team, must inform the National Cyber Security Centre (NCSC) of any category 1, 2 or 3 incidents. The NCSC defines security incidents in its categorisation system prioritisation framework (https://www.ncsc.gov.uk/news/new-cyber-attack-categorisation-system-improve-uk-response-incidents).

Depending on the incident, the NCSC may be able to provide technical support.

10. Resolve the incident

Hold an incident and lesson learned review following a blameless post mortem culture (https://codeascraft.com/2012/05/22/blameless-postmortems/) so your service can improve. Add a row to the central GDS incidents summary spreadsheet (https://docs.google.com/spreadsheets/d/1TmKiIAUr6EH1XZa5MJquSyHGZnQBjrORUJjs6l4TwHU) linking to your incident report document.

Example incident management

- GOV.UK PaaS incident management process (https://team-manual.cloud.service.gov.uk/incident_management/incident_process/)

Further reading

Read the GDS Technical Incident Management Framework and Process (https://docs.google.com/document/d/1jfdS6cmxYe7AmM50BSDYU0i6MdaVlOomuacwer70_rc/edit) document for more information. For example, you can read more about:

- classifying incidents
- routes to support
- incident workflows - from request to resolution
- roles in the Incident Team for P1 and P2

[^1]: Note that the incident report template document can only be accessed by people within GDS.

This page was set to be reviewed before 29 January 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way). This might mean the content is out of date.

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

How to store and query logs

You should store logs to detect any potential errors within infrastructure and to respond to security incidents.

Use Splunk (https://gds.splunkcloud.com) to store and query infrastructure, application and audit logs.

Splunk is a cloud-based SaaS tool for short and long-term storage, visualisation, alerting, and reporting of cyber security log information.

Your product should have a proportionate design for short and long term storage of logs and ensuring the Confidentiality, Integrity, and Availability of logs.

The NCSC Cyber Assessment Framework, which GDS must comply with, has an entire category dedicated to Security Monitoring (https://www.ncsc.gov.uk/collection/caf/cyber-assessment-framework/caf-objective-c-detecting-cyber-security-events).

Logit deprecation notice

The shared GDS Logit (https://logit.io/) account can still be used for existing environments; however, you should use Splunk for new logging requirements.

Logit offers cloud-based on-demand, shared ELK (Elasticsearch, Logstash and Kibana) (https://www.elastic.co/what-is/elk-stack) stacks as a service. They are suitable for short-term storage of logs, and are accessible and queryable.

Short-term storage

You should not need to store logs spanning long periods in your short-term queryable store. Practical retention periods for short-term queryable logs are:

- no more than 7 days for non-production environments
- no more than 30 days production environments

You should consider storing security and audit events for up to a year, this is because the average MTTD (Mean Time to Detect) is 204 days (over 6 months) to identify a breach, according to a 2023 IBM data breach study (https://www.ibm.com/account/reg/us-en/signup?formid=urx-52258).

Your product may have legal or other requirements determining how long you should store logs. For example, the Payment Card Industry Data Security Standard (PCI-DSS) (https://www.pcisecuritystandards.org/pci_security/) requires 3 months of easily accessible logs and 12 months of archived logs.

Long-term storage

When storing logs, you should focus on long-term durability for compliance reasons or to identify long-term performance trends.

Splunk is appropriate for up to a year of storage. If you require more, Splunk can be configured to archive data to your own S3 bucket (https://docs.splunk.com/Documentation/SplunkCloud/latest/Admin/DataSelfStorage?ref=hk#Con□gure_self_storage_locations) - this saves files in a proprietary format so consider archiving to S3 in human-readable files using your own process.

Log shipping

If you have set up log shipping, you should consider:

- how your logs will get to short and long-term stores - we recommend using Amazon CloudWatch (https://aws.ama□ on.com/cloudwatch/) to collect logs and subscribing to the Cyber Security team's Centralised Security Logging Service (CSLS) (https://github.com/alphagov/centralised-security-logging-service).
- what happens if one of these stores is unavailable
- whether the store will be overloaded when it comes back online

You could have one process to ship logs to your long-term archive, and a second process to fill a short-term query tool from the long-term archive, for example you could combine Splunk with Amazon Cloudwatch and also archive to S3.

CSLS utilises Amazon Kinesis and can persist logs in the event that Splunk is unavailable and will resume sending data when Splunk is available again.

Splunk Cloud's recommended input is the HTTP Event Collector (HEC) - CSLS uses this.

Note: Splunk Cloud does not support syslog, you may need a forwarder if syslog is your only output option:

- Logstash HTTP output (https://www.elastic.co/guide/en/logstash/current/plugins-outputs-http.html) and HEC (https://docs.splunk.com/Documentation/Splunk/latest/Data/HECExamples)
- Fluentd to HEC (https://github.com/splunk/fluent-plugin-splunk-hec)

Filtering out sensitive information

You should ensure that sensitive information, such as query parameters containing personally identifiable information (PII) (https://en.wikipedia.org/wiki/Personal_data), is filtered out before logs are shipped. Make use of log filtering mechanisms which may be present in the application framework you are using, such as in Ruby on Rails (https://guides.rubyonrails.org/action_controller_overview.html#log-filtering).

Structured logging

In order to allow for rich querying of log data you should ensure that your logs are in a structured format.

Splunk (https://gds.splunkcloud.com) can automatically parse many common types of structured data such as CSV, JSON, and XML. A range of add-ons can be found on Splunkbase (https://splunkbase.splunk.com/) to parse data from commonly used technologies such as AWS, Azure, Palo Alto Firewalls, and more. Other formats may need field extractions (https://docs.splunk.com/Documentation/Splunk/latest/Data/Extractfieldsfromfileswithstructureddata) to be configured in Splunk.

For interoperability with pre-built apps and alerting, it is beneficial to align your logs to the Splunk CIM (Common Information Model) (https://docs.splunk.com/Documentation/CIM/latest/User/Overview).

The `Web` CIM (https://docs.splunk.com/Documentation/CIM/latest/User/Web) is most commonly used at GDS - it specifies particular field names for data, for example:

- `http_user_agent` for the client's user agent
- `src` for the source IP address
- `dest` for the origin server IP address

Access to Splunk

Access control for GDS users is managed by the IT Service Desk, use the helpdesk (https://gdshelpdesk.digital.cabinet-office.gov.uk) to request access. If you're unsure what role you should be requesting, ask in the `#splunk` Slack channel.

Advice for particular frameworks or platforms

Dropwizard

The dropwizard-logstash (https://github.com/alphagov/dropwi ard-logstash)library will set things up for you using the standard names. Splunk will be able to ingest this format.

GOV.UK PaaS (Cloud Foundry)

There is broker documentation (https://github.com/alphagov/tech-ops/blob/master/cyber-security/components/csls-splunk-broker/docs/user-guide.md) describing how drain logs to Splunk via Centralised Security Logging Service (CSLS) (https://github.com/alphagov/centralised-security-logging-service).

The GOV.UK PaaS Logging (https://docs.cloud.service.gov.uk/monitoring_apps.html#set-up-the-logit-log-management-service) documentation will help you configure Logit and drain logs into it from your app.

Contact

Any questions regarding storing and querying logs should be directed to the `#splunk` Slack channel in the first instance.

This page was last reviewed on 21 October 2024. It needs to be reviewed again on 21 April 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

How to monitor your service

In GDS, we follow the Service Manual guidance on how to monitor the status of services (https://www.gov.uk/service-manual/technology/monitoring-the-status-of-your-service) and set performance metrics (https://www.gov.uk/service-manual/measuring-success/how-to-set-performance-metrics-for-your-service).

We recommend using Pingdom (https://www.pingdom.com/) to monitor your service's availability from outside our network. To further make sure your service is working, you should:

- run regular smoke tests (https://www.gov.uk/service-manual/technology/deploying-software-regularly#using-smoke-tests-after-you-deploy) using a browser automation app such as Selenium
- implement a tool to ensure user journeys are working as you expect
- monitor applications for errors using an error tracking app such as Sentry (https://sentry.io/welcome/)
- implement con□guration management to set up repeatable monitoring

Using metrics-based monitoring

Collecting metrics on the performance of your service is useful for capacity planning (https://www.gov.uk/service-manual/technology/test-your-services-performance#start-with-capacity-planning) and autoscaling. You should apply metrics-based monitoring to measure aggregated numerical data about your service and create Grafana (https://grafana.com/) dashboards to view metrics from your datasource, for example related to your infrastructure or application.

We recommend using Prometheus (https://prometheus.io/) to gather metrics and monitor your service, as this is what many GDS teams are using. Amazon provides a Managed Prometheus service (https://aws.amazon.com/prometheus/), which you may wish to use instead of hosting Prometheus yourself. You may also consider using CloudWatch Metrics (https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/working_with_metrics.html) instead for simpler use-cases where the complexity of Prometheus is not required.

This page was last reviewed on 20 November 2024. It needs to be reviewed again on 20 May 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

How to name software products

Read this guide when you need to name components for your software product, for example applications, software libraries, plugins or frameworks. Your users should understand what something does from its name.

Your product name should be self-descriptive

Avoid using puns or branding for names as this makes it difficult for others to understand what it does.

Make sure you use the same name consistently whenever you're referring to the same product. For example the name of this website's GitHub repository is alphagov/gds-way (https://github.com/alphagov/gds-way).

These GDS product names clearly communicate their purpose:

- Travel Advice Publisher (https://github.com/alphagov/travel-advice-publisher)
- Manuals Publisher (https://github.com/alphagov/manuals-publisher)
- Smart Answers (https://github.com/alphagov/smart-answers)

These GDS product names are ambiguous and possibly confusing:

- Panopticon (https://github.com/alphagov/panopticon)
- Whitehall (https://github.com/alphagov/whitehall)
- Maslow (https://github.com/alphagov/maslow)
- Magna Charta (https://github.com/alphagov/magna-charta)

Further reading

The Service Manual has guidance on naming services (https://www.gov.uk/service-manual/design/naming-your-service), and this will be relevant when naming applications or packages.

This page was last reviewed on 7 November 2024. It needs to be reviewed again on 7 May 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Operating s□ stems for virtual machines

Ama□ on Web Ser□ ices □ irtual machines

If you're using AWS EC2 (https://aws.amazon.com/ec2/) and you do not require a Debian-based system, you should use Amazon Linux 2 LTS (https://aws.amazon.com/amazon-linux-2/release-notes/) as your operating system (OS).

Amazon Linux 2 was forked from `RHEL` / `CentOS` and the primary package manager is □ □ (http://yum.baseurl.org/wiki/Guides.html). It's mantained by Amazon and includes a Kernel tuned for enhanced performance on EC2. The OS is available as a virtual machine image (https://cdn.amazonlinux.com/os-images/latest/) and Docker image (https://hub.docker.com/_/amazonlinux/) for testing.

Non-AWS and Debian □ irtual machines

If you're not using AWS, or you need to use a Debian-based OS, you should use Ubuntu LTS (https://ubuntu.com/download/server) as an OS for your virtual machine (VM).

GDS recommends using Long Term Support (LTS) versions as the default choice for your VM OS. If you're running the latest LTS release and there are newer features not available in that release, such as newer kernels, consider using the official backports or trustworthy PPA sources rather than upgrading the whole OS to a standard (non-LTS) release.

You must make sure these backports and other sources receive regular and timely security updates. This should be less of a burden than having to update the OS on VMs every 6 to 9 months.

If you do use standard Ubuntu, you must migrate to the latest release within 3 months to ensure you stay within the support period and receive security updates. You must consider this carefully when proposing adopting standard Ubuntu releases as it can affect the stability of your running service.

Standard releases only receive security updates for 9 months after their initial release. LTS receives support for 5 years, or longer with paid-for extended support.

Hardening

You must apply hardening to your operating system. The National Cyber Security Centre (NCSC) Cyber Assessment Framework (https://www.ncsc.gov.uk/collection/caf/cyber-assessment-framework/caf-objective-b-protecting-against-cyber-attack) has a specific objective for this (B4.b).

For example, you should disable root SSH access and deny SSH access using a password. You must also make sure your OS is:

- automatically updated for security issues
- up to date with the latest OS releases
- able to run on a local VM

The NCSC has principles to follow for end user device (EUD) hardening (https://www.ncsc.gov.uk/collection/device-security-guidance/security-principles).

The NCSC also has guidance on hardening that's specific to the latest Ubuntu LTS version (https://www.ncsc.gov.uk/collection/device-security-guidance/platform-guides/ubuntu-lts).

The Center for Internet Security also publishes the CIS Benchmarks (https://www.cisecurity.org/cis-benchmarks), which provide a full breakdown of hardening options for specific operating systems, as well as other firmware.

This page was set to be reviewed before 15 November 2024 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way). This might mean the content is out of date.

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

How to optimise frontend performance

You should focus on frontend performance (https://www.gov.uk/service-manual/technology/how-to-test-frontend-performance) when developing your service's website. This will improve the user experience of your service by making your website respond faster and work better on all devices.

Prioritise performance tasks

You can optimise your site's frontend performance by prioritising tasks that will improve your site speed. Prioritise things you must do (high) over nice to have (medium or low priority) tasks.

For example:

Priority	Example	Action
High	Position assets correctly	Set styles at the top of the page (https://csswizardry.com/2013/01/front-end-perfo designers-and-front-end-developers/#section:styl scripts-at-the-bottom) and `defer` scripts (https://calendar.perfplanet.com/2016/prefer-defe
	Compress static resources	Minify (https://web.dev/reduce-network-payloads compression/) CSS and JavaScript and use a algorithm like Gzip (https://web.dev/optimizing-efficiency-optimize-encoding-and-transfer/#text_compression_with_gzip) and Brot (https://github.com/google/brotli) on assets.
	Set correct Headers	Set correct Cache-Control (https://developer.m US/docs/Web/HTTP/Headers/Cache-Control) an (https://developer.mozilla.org/en-

		US/docs/Web/HTTP/Headers/ETag) headers or optimal caching.
	Minimise unused code	Avoid including CSS and JavaScript that is r your site.
Medium	Include `width` and `height` attributes on images to minimise layout thrashing	Make sure to include these attributes to imp stability and the Cumulative Layout Shift (CL (https://web.dev/cls/) metric.
	Minimise TCP connections	Use fewer third-party domains to reduce the DNS + TCP + SSL negotiations per page.
	Investigate lazy loading (https://web.dev/fast/#lazy-load-images-and-video)	For pages with many images, only load imag immediate browser viewport.
	Investigate the impact of loading @font-face (https://developer.mozilla.org/en-US/docs/Web/CSS/@font-face) assets on perceived performance	Use the CSS `font-display` property or oth strategies (https://www.zachleat.com/web/comp webfonts/) to manage common issues like FO FOFT (https://css-tricks.com/fout-foit-foft/).
	Minimise HTTP requests	Minimise the number of CSS and JavaScript reduce the number of round-trips to the serv splitting' below.
Low	Reduce cookie size	Because every cookie is sent with each HTT enable HTTP/2 to enable HPACK header co HTTP/3 for QPACK.
	Investigate using a Content Delivery Network (CDN)	A CDN will improve site performance by usir of servers to deliver resources to users. The delivered resources from the server that is lo to the user. A CDN is well-suited to handling and traffic spikes.
	Keep JSON payloads small	Avoid adding too much data to JSON object them can be slow.
	Investigate using WebSockets (https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API)	Consider using WebSockets rather than XM (https://xhr.spec.whatwg.org/)for tasks involving from the server - an HTTP request packet ha of overhead, compared to 8 bytes for a Web packet.

Investigate using a service worker	Consider using a service worker (https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_S for fine grain cache control of critical assets machines instead of transferring them over t

Automate optimisation

You can automate performance optimisation using tools such as:

- Webpack (https://webpack.js.org/)
- Parcel (https://parceljs.org/)
- Rollup (https://rollupjs.org/guide/en/)
- Gulp (https://gulpjs.com/)

You should integrate these tools into your Continuous Delivery (CD) and Continuous Integration (CI) workflow (https://www.gov.uk/service-manual/technology/deploying-software-regularly) so they automatically run before deployment.

Consider automating common tasks like:

- CSS and JavaScript linting and optimisation
- CSS and JavaScript minification
- image optimisation
- sprite and icon generation
- SVG optimisation

Automate testing

You can automate frontend performance testing using third-party services such as:

- WebPageTest (https://www.webpagetest.org/)
- Google PageSpeed Insights (https://pagespeed.web.dev/)
- SpeedCurve (https://www.speedcurve.com/)
- Sitespeed.io (https://www.sitespeed.io/)
- Calibre (https://calibreapp.com/)

Using Speedcurve to automate frontend performance testing

You can use Speedcurve (https://www.speedcurve.com/) to test the performance of your service.

Speedcurve provides an extensive set of tools to test your service such as the following:

- Test from different locations and different browsers.
- Schedule regular tests using tools like Google Lighthouse and Webpage Test.
- Real user monitoring (https://insidegovuk.blog.gov.uk/2021/06/16/how-real-user-monitoring-will-improve-gov-uk-for-everyone/), which allows you to collect data that shows you how your real users experience the speed of your site.

Contact a Lead Frontend Developer to request an account.

Performance budget

You should set a performance budget (https://www.gov.uk/service-manual/technology/how-to-test-frontend-performance#set-a-performance-budget) for your website's pages. Once you've set a performance budget, test to check your website's pages stay within your budget. There are many tools available to do this, such as:

- Webpack Performance (https://webpack.js.org/configuration/performance/)
- performance-budget (https://www.npmjs.com/package/performance-budget)
- psi (PageSpeed Insights Reporting for Node) (https://github.com/GoogleChromeLabs/psi)
- Google Lighthouse (https://github.com/GoogleChrome/lighthouse)

Code splitting

You should concatenate your Javascript and CSS files (https://csswizardry.com/2023/10/the-three-c-concatenate-compress-cache/) by default, as this minimises the number of requests and makes compression more effective. However, you may benefit from splitting your code into multiple bundles in some cases:

- if a specific feature is only used sporadically, it may make sense to split it out and only load it on pages where it is needed.
- if some of your code is updated more regularly than others, it may make sense to split it out so that less frequently updated code is still cached.

Use HTTP/2 or above

HTTP/2 is supported in all of the Service Manual's browsers to test in (https://www.gov.uk/service-manual/technology/designing-for-different-browsers-and-devices). It improves speed for most users by allowing browsers to:

- Download multiple resources at once
- Compress cookies in request headers
- Prioritise specific assets for download.

You should also consider enabling HTTP/3. While this isn't supported as widely as HTTP/2, it is supported by most of the Service Manual browsers and it solves an issue with HTTP/2 that affects users with slow connections.

Further reading

You can find out more about improving your website's frontend performance by reading:

- web.dev - Fast load times (https://web.dev/fast/)
- Setting a performance budget (https://timkadlec.com/2013/01/setting-a-performance-budget/)
- My performance audit workflow (https://aerotwist.com/blog/my-performance-audit-workflow/)
- Front-end performance for web designers and front-end developers (https://csswizardry.com/2013/01/front-end-performance-for-web-designers-and-front-end-developers/)
- Improving web app performance with the Chrome DevTools Timeline and Profiles (https://addyosmani.com/blog/performance-optimisation-with-timeline-profiles/)
- Google Web Fundamentals: Optimizing Content Efficiency (https://web.dev/performance-optimizing-content-efficiency/)

The Service Manual has more suggestions about how you can test frontend performance (https://www.gov.uk/service-manual/technology/how-to-test-frontend-performance).

This page was set to be reviewed before 10 November 2024 by the page owner 🔒 #frontend (https://gds.slack.com/messages/frontend). This might mean the content is out of date.

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Pair Programming

GDS advocates for agile software development practices because we believe that flexibility and responsiveness-to-change are key to building software that meets the needs of our users.

At the core of agile software development is the understanding that fast feedback loops enhance speed and quality, and one way for developers in the team to get quick feedback on the quality of their software design is the practice of pair programming.

What is pair programming?

Pair programming is a collaborative method of developing software where two developers work together, as opposed to two developers working in isolation. There are many ways a pair might collaborate, such as driver (the person doing the implementation, thinking tactically) and navigator (the person thinking about the bigger picture, thinking strategically).

Bene□ts

Studies show that the reduction in defect rates (https://www.sciencedirect.com/science/article/abs/pii/S0950584909000123) for high-complexity pieces of work when pair programming outweighs the perceived decrease in capacity from having two people work on the same thing. The continuous discussion and review that pairing introduces, compared to a non-collaborative approach, results in better designs with fewer mistakes. Another study (https://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF) found that pair programming reduces development and staffing risk, enhances technical skills and improves team communications.

Pair programming is a practical way of building collective code ownership between members of the team. Distributing the understanding of the service being built means that teams are more resilient to people being ill, on leave, or leaving GDS, and as such are less likely to have delivery hiccups from sudden loss of expertise.

Pairing people of different levels of expertise (https://www.youtube.com/watch?v=lvs7VEsQzKY&t=317s) is an effective way of mentoring and fostering career development. For example, pairing someone new to GDS with an experienced developer within the team will benefit both; upskilling the new joiner, and reinforcing the knowledge of the experienced person by teaching them.

Working effectivel as a pair

Pair programming is a development skill in the same way that test-driven development is a skill. It's also a social and collaborative way of working and the way a pair works can vary depending on who you are pairing with. We provide some of the following advice to help pairs work effectively together.

Rotate pairs regularly

Pairs should rotate regularly at a cadence that makes sense for the team. This might be after a single piece of work or after a fixed period of time. Rotating pairs spreads knowledge, increases team resilience, and builds ownership of the codebase.

Discuss how to pair together

Pairs should agree on the way that they will pair before starting the work itself. Your pair might favour ping-pong programming (where one person writes a test, the other person makes it pass and then writes the next test) or strong-style pair programming (your ideas need to enter the codebase through the other person's input). This also includes agreeing when to take breaks.

Take breaks

Working together and staying focused for extended periods of time can be tiring. Pairs should make sure that they take regular breaks and where possible work in small iterative batches rather than large changes. You might find the Pomodoro technique useful or one of the many pair/mob programming timers available online. Have a debrief at the end of the day if it would be helpful, and raise any recurring issues at your team's retrospectives or with your tech lead.

Pair where appropriate

Pairing is not suitable for every type of work, so use your discretion within the team. You might choose not to pair on automated refactorings or work that would be described as Clear within the Cynefin model (https://en.wikipedia.org/wiki/Cynefin_framework). Remember that developers are people and as such might be unable to pair on any given day for their own personal reasons. Be respectful and understanding if someone is unable to pair.

Communicate effectively

Pair programming is both a social skill and a technological skill. It works most effectively in an environment where the pair respect each other's skills and needs, and communicate with openness, empathy, and honesty. People in teams that have embraced concepts like psychological safety (https://en.wikipedia.org/wiki/Psychological_safety) and non-violent communication (https://www.cnvc.org/) are likely to be well-equipped for pair programming.

Reviewing code as a pair

The standard approach to code review in GDS is that teams should use a pull-request (PR) process to ensure that all code is reviewed by another person. A pair can approve their own PRs on codebases where changes require an additional review, which is the vast majority of codebases at GDS.

This has several benefits, such as:

- speeding up the flow of work.
- enabling the pair to move on to another task without context switching (https://blog.ninlabs.com/2013/01/programmer-interrupted/).
- avoiding a third person having to context switch to review the code.

However, teams should adjust their policy on this approach depending on the risk tolerance of the codebase, team, or programme.

Pair programming and version control

Almost all version control systems, including Git, are oriented around a single-author-per-change model. This doesn't completely align with collaborative development practices.

A convention has emerged, and is supported by GitHub, to use the `Co-authored-by` (https://docs.github.com/en/pull-requests/committing-changes-to-your-project/creating-and-editing-commits/creating-a-commit-with-multiple-authors) trailer on commit messages to flag that multiple people have worked on a change.

For example, if you had two or more contributing as part of a pair or ensemble, you would structure your commit message like:

```
TICKET-123: Add a new feature
```

```
Co-authored-by: Person One <person.one@example.com>
Co-authored-by: Person Two <person.two@example.com>
```

This is visible within the normal Git log and within GitHub too; you will see multiple people attached to a change in the UI.

You should use your work email address in the `Co-authored-by` trailer.

Further reading

- Pairing and code review guidelines for GOV.UK Pay (https://manual.payments.service.gov.uk/manual/development-processes/pairing-principles.html)
- Code review guidelines in the GDS Way (https://gds-way.digital.cabinet-office.gov.uk/standards/pull-requests.html#reviewing-a-request)

This page was last reviewed on 2 December 2024. It needs to be reviewed again on 2 September 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Performance Testing

Make sure your service performs the way users need it to. If your service does not perform as expected under extra load it will impact your users.

You should run performance tests to check how your service performs in certain situations. For example, during a denial-of-service attack (DoS attack) (https://www.ncsc.gov.uk/collection/denial-service-dos-guidance-collection).

The Service Manual has more information about why you need to test your service's performance (https://www.gov.uk/service-manual/technology/test-your-services-performance) and how to start capacity planning and running performance tests.

Plan your tests

Plan and scope your test. Are you testing the entire service or specific user journeys? Which downstream services will the test impact?

Estimate costs

Estimate any additional infrastructure costs your tests will incur and share them with your service owner or product manager for approval.

Plan user journe□ s

Test complete journeys that are representative of how your service is used in production. For example, if a user has to sign in or register, plan tests where some users sign in and others register.

Plan test scenarios

A scenario is a hypothetical event when your service will be under heavy load. For example, every year the G-Cloud framework on the Digital Marketplace (https://www.gov.uk/guidance/buying-and-selling-on-the-digital-marketplace#buy-services-through-the-digital-marketplace) opens for applications from suppliers for several weeks. This activity

increases load on the Digital Marketplace (https://www.digitalmarketplace.service.gov.uk/) up to the deadline.

When most of its application infrastructure moved to the GOV.UK PaaS (https://www.cloud.service.gov.uk/) the Digital Marketplace service needed to retest the service. Its developers and performance analysts queried historical logs and Google Analytics for popular journeys, modelling normal and peak capacity needs for the previous year's G-Cloud framework application period.

This helped the team calculate figures for the largest number of concurrent user sessions and Requests Per Second (RPS) Digital Marketplace was able to support. This made sure the new platform could still cope with its expected network traffic.

Inform people

Inform people affected by your test. For example, tell developers or content specialists planning to use the test environment of your test's date, time and duration. Make sure everyone involved in the test agrees to its conditions.

Build our test

You do not need to fully hand code your tests because most load testing tools like Gatling (https://gatling.io/), include scenario recorders. Some test scenarios may still need manual preparation, for example, providing lists of test users or specific search terms.

Gatling is a popular load testing tool used by many teams in GDS and CDIO including GOV.UK (https://www.gov.uk/), GOV.UK Pay (https://www.payments.service.gov.uk/), and GOV.UK Notify (https://www.noti□cations.service.gov.uk/).

The SPOT Team on Digital Identity have had success using K6 (https://www.k6.io), a performance testing framework built in Go and scripted in Javascript.

When testing APIs or very simple web applications, `□ egeta` (https://github.com/tsenart/vegeta) is a tool used by GOV.UK Notify (https:///www.noti□cations.service.gov.uk), and GOV.UK PaaS (https://www.cloud.service.gov.uk), for automated performance testing. It can also be used interactively with `angle-grinder` (https://github.com/rcoh/angle-grinder), or `jaggr` (https://github.com/rs/jaggr) and `jplot` (https://github.com/rs/jplot).

Con□gure your test environment

Test environments must resemble your production environment as far as possible so tests are accurate. Isolate test environments to stop them affecting your production environment.

Consider how to monitor service performance (/standards/monitoring.html) during the load test. Your service will likely already have dashboards and graphs set up for common metrics such as CPU and memory utilization. Decide if your current metrics are sufficient or if you need to turn on additional logging or configure profiling traces.

Perform practice runs of your tests to make sure they work as expected. You may need to disable Cross-Site Request Forgery (CSRF) (https://owasp.org/www-community/attacks/csrf) tokens and rate limiting in your test environment.

Run your tests

Start with a low level of concurrent users and increase them to your anticipated maximum. If there are no problems, increase the amount of traffic until you reach a breaking point in your system.

Running □ our tests locall□

While designing and scripting your performance tests, you may be tempted to run them locally in order to develop them faster. This is fine for development and small-scale testing, however, it is generally not a good idea to do this for larger-scale testing as you won't get accurate results.

Any and all of these things could negatively impact the results of your performance test:

- The speed, quality or capacity of your local network connection (Wi-Fi, VPN, etc.)
- The technical specification of your computer and resources available
- The round trip time (RTT) between your computer and the environment being tested

For these reasons, you should run your tests on a more dedicated platform, such as on an EC2 Instance or ECS Container.

Running □ our tests on dedicated infrastructure

Running your performance tests on dedicated infrastructure is usually the better option. If you are a public sector organisation (such as GDS), you will likely already have access to a cloud compute supplier such as AWS, Azure or Google Cloud and can run your tests on a compute instance, or as a container on an orchestration platform like ECS or Kubernetes.

Running □ our tests on Software-as-a-Service (SaaS)

There are some providers out there offering performance testing as a paid-for service. Depending on the scale of your testing, this may be a consideration for you. However, you will have additional considerations, including:

- Security implications using an external provider
- Ingress/Egress to/from your testing environment
- Costs associated with paying a third party to run your tests

If you are working for a public sector organisation, you may also be required to undertake a procurement process before you are able to select and use an external supplier to run performance tests on your behalf.

Document and share your results

When testing your service, record:

- the load when the service broke
- how the service broke and what went wrong
- the changes introduced to improve performance

If your service meets your capacity requirements, document its upper limits and communicate them to the service owner.

Analyse your results

Testing tools should record the number of concurrent users, RPS, HTTP response codes and timings. These reports help determine your service's maximum supported capacity. If your service cannot meet your requirements or you find problems, you'll need to diagnose them.

You can fix common problems like inefficient database queries by introducing lazy loading and adding or altering indexes. If your web server stops or is slow to respond, it may be running out of available workers. This is a common issue when you're sending lots of requests to APIs and the workers become blocked awaiting a response. Adjusting web server configuration can help depending on your specific application and technology.

Problems with your infrastructure can be harder to resolve. For example, if your service is read-heavy you can introduce additional caching to improve performance.

You can scale infrastructure vertically by adding more power, for example upgrading CPUs, memory or storage. You can also scale horizontally by adding additional instances of your service's components. You can only do this if your service supports this type of scaling.

The Twelve-Factor App (https://12factor.net/) provides more information on how to scale your infrastructure and build SaaS.

Make sure you rerun your performance tests after any changes to confirm everything works as expected and you have not introduced any new problems to your service.

Performance testing for auto-scaling environments

Test auto-scaling infrastructures to make sure they scale up and down as expected and are cost-efficient. For example, if infrastructure scales at the correct thresholds, check if servers become stressed and response times slow down before scaling takes effect.

Performance testing helps determine if you're using the correct machine instance types for your workload. For example, more expensive larger specification machines may be capable of handling more traffic cheaper than adding additional smaller instances to your auto-scaling group. Depending on your service requirements the reverse may be true.

Performance testing for serverless environments

Testing can help you tune serverless components' performance. For example, Amazon prices AWS Lambda (https://aws.amazon.com/lambda/) on the number of function executions, allocated memory and estimated execution time. The AWS Lambda Power Tuning tool (https://github.com/alexcasalboni/aws-lambda-power-tuning) provides performance data at different Lambda configurations and can help you right-size your functions.

Performance testing will help determine if a function is taking longer to execute because it does not have enough memory allocated, or is over-provisioned with memory and costing more than necessary.

Measuring and getting results from serverless architecture

Serverless architecture can sometimes mean that a journey is asynchronous and does not result in a round journey that can be measured or recorded by a testing tool or framework.

HTTP and API Gateway
If your serverless application completes its journey in one or a small series of HTTP requests, you may be able to measure the results of your performance using only the performance framework.

SQS Queues, Step Functions and EventBridge
Some serverless applications do not use HTTP Requests to start or trigger a serverless journey, but rather, rely on SQS Queues, Step Functions or an Amazon EventBridge event to invoke one or more Lambda Functions.

If your serverless architecture uses one of these, you may have a harder time capturing the full journey within a testing framework - therefore, you might use a framework to simulate the traffic for your performance test and collect the results using another tool such as Splunk or CloudWatch.

If you can, you should try and annotate your requests with a testing identifier to make it easier to collate your results.

Testing strategies

Some testing frameworks support the execution of different kinds of testing strategies, enabling you to perform different types of testing such as:

- Smoke tests (To test a journey from start to finish)
- Soak tests (To test whether your application can handle a sustained load)
- Spike tests (To test whether your application can handle an unexpected surge or “spike” in load)
- Stress tests (To find breaking points or bottlenecks in your application or infrastructure)

Some testing frameworks may also allow you to simulate different patterns and stepping of traffic, such as choosing between a constant or ramping (exponential) arrival rate.

Further reading

To find our more about performance testing read:

- the Service Manual about testing your service’s performance (https://www.gov.uk/service-manual/technology/test-your-services-performance)
- The Twelve-Factor App (https://12factor.net/) a methodology for building software-as-a-service apps
- Gatling product documentation (https://gatling.io/docs/gatling/reference/current/) about maintainable and high performance load testing

This page was set to be reviewed before 23 October 2023 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way). This might mean the content is out of date.

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Principle of Least Privilege

The principle of least privilege (https://csf.tools/reference/nist-sp-800-53/r5/ac/ac-6/) is succinctly defined as “Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job.” It requires you to consider what permissions actors in your system need. For robotic or computer actors, like your deployment pipeline, this can be fairly easy to scope. Likewise, your end-users will have a clearly defined set of permissions they need to use your service.

All access provisioned for use within GDS must be provided on a least privilege basis.

Examples of privileged or higher security access are:

- root access
- updating operating systems
- performing “sudo” operations
- accessing secret credentials
- committing to repositories

You should use the principle of least privilege if you:

- need to grant required access rights to different users according to the nature of their job
- want to use different permissions for day-to-day tasks from those required for security-critical (privileged) tasks
- want to limit collateral damage when a normal user account gets compromised

Your team should:

- make users aware of this policy and be required to confirm their understanding of their access privileges and related conditions of use
- have a process for them to request changes to access as needed
- define roles for users and grant required privileges, or access rights, for those roles
- create the roles or credentials with the least possible privilege, with only necessary permissions required for normal users to perform their day-to-day jobs
- use the role or credentials with the least possible privilege as the default option

- use just-in-time (JIT) access provisioning to grant users an on-demand, time-limited privileged role or security token to access the privileged resources
- make sure session time of the privileged access (in non-development environments) is set to no more than 30 minutes, and/or terminates when the user logs out of their laptop
- establish an audit trail for the use of privileged access
- ensure approval and use of privileged accounts is kept to the absolute minimum necessary for a user to perform their job role
- in cases where JIT access is not implemented for users with privileged access that have critical business impact, implement a documeneted periodic review (cadence to be defined) of the need to continually have these privileged access granted to confirmed users
- have a Joiners, Movers and Leavers process, where line managers (or equivalent) arrange for privileged access to be removed (SLA to be defined) where it is not required. See this NCSC guide on identity management (https://www.ncsc.gov.uk/guidance/introduction-identity-and-access-management) for more information.

It's important to recognise opportunities for privilege creep / accumulation and to design in suitable processes for preventing it.

Examples

For human-readable secrets, such as a username and password, you should set up a separate secret or password manager (https://www.ncsc.gov.uk/collection/passwords/password-manager-buyers-guide).

If you're using the gds-users account to log into your AWS accounts, you should assume a read-only role by default. You should only assume an admin role when absolutely necessary for a specific task. You should set up the admin account so that the session timeout is less than 30 minutes. You should send the audit trail of admin access to the Cyber Security team. Use the Cyber Security Slack channel (🔒 #cyber-security-help (https://gds.slack.com/archives/CCMPJKFDK)) to set up the audit trail.

Further Guidance

- NCSC - 10 steps to cyber security (https://www.ncsc.gov.uk/collection/10-steps)
- NCSC - privileged user management (https://www.ncsc.gov.uk/guidance/introduction-identity-and-access-management#section_6)
- NIST Special Publication 800-53 - AC-6 least privilege (https://csf.tools/reference/nist-sp-800-53/r5/ac/ac-6/)
- Government cyber policy - B2 identity and access control (https://www.security.gov.uk/policy-and-guidance/government-cyber-security-policy-handbook/principle-b2-identity-and-access-control/)

This page was last reviewed on 6 February 2025. It needs to be reviewed again on 6 August 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Programming languages

We use a range of programming languages at GDS because we think using the right tool for the job gives us the best chance of building services that best meet users needs. This document does not apply to choosing off the shelf software (open source or not).

We focus on using a small number of programming languages for core software development tasks.

This should make it easier for developers to:

- move around the organisation
- develop shared components
- improve their personal development
- master how they operate applications

Frontend development

The Service Manual has information on using client-side JavaScript (https://www.gov.uk/service-manual/technology/using-progressive-enhancement). For server-side JavaScript we use Node.js (https://nodejs.org/).

GDS projects using Node.js include:

- GOV.UK Frontend (https://github.com/alphagov/govuk-frontend)
- GOV.UK Pay (https://www.payments.service.gov.uk/)
- GOV.UK PaaS (https://www.cloud.service.gov.uk/) (TypeScript)
- GOV.UK Prototype Kit (https://govuk-prototype-kit.herokuapp.com/docs)
- GOV.UK One Login (https://www.sign-in.service.gov.uk/) - see repositories in TypeScript (https://github.com/search?l=TypeScript&q=user%3Agovuk-one-login+topic%3Adigital-identity&type=Repositories) and in JavaScript (https://github.com/search?l=JavaScript&q=user%3Agovuk-one-login+topic%3Adigital-identity&type=Repositories)

You can use Node.js to render a web interface for your service. For example, GOV.UK Pay has created thin, client-facing applications that do not store data (although they may

retrieve data from an API).

You can use TypeScript (https://www.typescriptlang.org/) when teams think it's appropriate. For example, the GOV.UK PaaS team uses TypeScript because they are used to working with a statically typed, compiled language, and they think the compilation and static-analysis tooling is better for their workflow. There's more information about TypeScript on the Node.js page (/manuals/programming-languages/nodejs/).

Backend development

Our core languages for backend development are:

- Java (/manuals/programming-languages/java.html)
- Python (/manuals/programming-languages/python/python.html)
- Ruby (/manuals/programming-languages/ruby.html)
- JavaScript / Node.js (/manuals/programming-languages/nodejs/)

We've chosen these languages because they are successfully used by teams at the moment, and we are confident in how to host and operate applications written in them. You should carry out new development in one of these languages.

Python

You should write new Python projects in Python 3. Python 2 reached end of life in 2020 (https://www.python.org/dev/peps/pep-0373/). Python 3 is now well-supported by application frameworks and libraries, and is commonly used in production.

Go

Go is no longer a core backend development language in GDS.

The only Go service currently in production operation is the GOV.UK router (https://github.com/alphagov/router), and it is the core language for Cloud Foundry (https://www.cloudfoundry.org/), which GOV.UK PaaS uses, although GOV.UK PaaS is being decommissioned (https://gds.blog.gov.uk/2022/07/12/why-weve-decided-to-decommission-gov-uk-paas-platform-as-a-service/). As such, the knowledge and experience of building and running services in Go is small and decreasing.

Go *may* be an appropriate language for instances of systems programming, like proxying, routing, and transforming HTTP requests. However you should only write these sorts of components if there is no alternative maintained open source tool available.

Languages we do not use for new projects

We used Scala in the early days of GDS. GOV.UK Licensing is the only remaining application written in Scala but we've found it hard to support because of a lack of skills in GDS. Do not use Scala for new projects.

Mobile Development

For developing mobile apps, we use: - Swift (https://www.swift.org/) for iOS (https://developer.apple.com/) - Kotlin (https://kotlinlang.org/) for Android (https://developer.android.com/).

To give users the expected experience on their respective platform, we prefer to use the native languages over cross-platform solutions.

Swift

When starting a new app project, you'll likely want to use the most recent version of Swift which will be installed with Xcode.

Kotlin

We prefer to develop Android apps using the latest stable version of Kotlin.

Using other languages

There will be sensible reasons to not follow the above guidance on languages. For example when you're:

- extending an existing codebase or ecosystem
- scripting in a particular environment
- experimenting during an alpha (with an expectation that it's replaced by something we have more confidence in for beta)
- working in a very specific or unusual problem domain, like heavy use of WebSockets

The set of languages we're comfortable supporting will change over time.

If you want to use a new language, talk to your Head of Technology and then create a prototype. If it goes well you can open a pull request (/standards/pull-requests.html) to change this document.

If you're having problems using one of the languages we support, open a pull request to document the issues.

This page was last reviewed on 25 February 2025. It needs to be reviewed again on 25 February 2026 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Pull requests

Pull requests (PRs) let you tell others about changes you've pushed to a branch in a repository on GitHub.

Why should you use pull requests?

We use PRs (https://docs.github.com/en/github/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests) at GDS to:

- enable code review (/manuals/code-review-guidelines.html) from colleagues
- provide a lightweight change management process
- share responsibility for the code going live
- spread knowledge and make others aware of changes
- notify people outside the immediate group of reviewers

Cautionary notes

You should not use PRs for architectural review, as you should do the architectural review before raising the PR.

It is your responsibility to make sure you get reviews for your PRs and merge them. You should not rely on someone else noticing, reviewing, and merging your PR for you.

You do not need to merge all changes for a piece of work in a single PR. You should break down the changes into several PRs to get feedback earlier.

Guidance for each step

Feel free to ask for help on Slack (for example, 🔒 #tech-community (https://gds.slack.com/archives/CACV4GHCL)) if you are unsure how to do any of these things.

Opening a pull request

You should make sure that:

- your local repository has the most recent changes on the target branch
- the title and description of the PR are clear and accurately represent your changes
- the title and description reference the source of the change, which could be a support ticket (Zendesk, or Service Now) or a Jira card
- the description contains links to any related PRs
- any screenshots have text descriptions for accessibility
- any contentious changes or side effects have clear descriptions of the pros and cons

PRs are an implementation of change review and depend on the continued support of a repository hosting provider such as GitHub. This means that we would lose data contained in PRs if we switched providers.

As such, the canonical description of changes should be in the commit messages. You can refer to the commit message style guide (/standards/source-code/working-with-git.html).

Reviewing a request

Taking time to properly review a PR is as important as making the change itself, and it is crucial that we show compassion when critiquing someone else's work.

April Wensel has a talk about Compassionate-Yet-Candid Code Reviews (https://www.slideshare.net/AprilWensel/compassionate-yet-candid-code-reviews).

It suggests 3 key questions to ask ourselves when reviewing or commenting on someone else's work:

- Is it true?
- Is it necessary?
- Is it kind?

Guidelines for review

Ask the PR raiser if you do not know how they want their change reviewed; they may prefer you to deliver feedback in person or while pairing.

Call out and celebrate good code, content, and design choices. This will also highlight good practice for others.

State explicitly if there are blocking issues.

Communicate with others who may consider reviewing the PR

Comment in the PR when discussions about the change are happening “offline” (outside the PR itself). Summarise any “offline” conversations as a comment in the PR for the benefit of others.

Comment in the PR if you have not finished reviewing the change. You might want to directly inform the PR raiser if they are expecting you to finish your review by a specific point.

Helpful things to consider while reviewing

Pull the branch to your local repository and try running the code; even if the tests pass, it might have bugs or unexpected side effects.

Consider the security and privacy implications of the change, paying special attention to where there is unsanitised user input or logging.

Make sure that documentation, including the README, is consistent with the code changes.

Addressing comments

You should address any comments flagged as blocking. This includes spelling and grammar mistakes in documentation.

You should amend minor changes into a previous commit, such as renaming a variable or adding a missing test. You should address major changes in a separate commit.

Make sure you follow existing commit guidelines when addressing changes. “Addressed feedback” is not a helpful commit message for future contributors.

Add a comment to show you have addressed all relevant comments.

A request for refactoring, while encouraged, should not block a merge.

Reviewing PRs from members of the public

We sometimes receive pull requests from members of the public, and we should treat them with the same compassion we would a colleague.

It is important that we also follow specific guidelines when dealing with people we do not know, some of whom will be doing this work in their own time.

Tone

Be positive; thank them for contributing in your first comment, even if you are going to reject their contribution.

Call out and celebrate good code, content, and design choices. A list of things to change can be dispiriting and discourage them from further contributions.

Make requests for improvement rather than telling them what to do. For example, “We think it might be better the other way round, what do you think?” rather than “Swap the order of the logic”.

Be explicit and take care with what you write. People do not always understand your intentions online and cannot get in touch using Slack as a colleague might.

Handling the PR

You should communicate in your first comment whether we are able to merge the change.

If we do not want to merge it because it does not fit with our plans, thank them and close the PR.

If it fits, but we cannot merge it due to quality or style issues, then tell them that we are able to merge the PR if they make some changes.

We can tell the requester what improvements we’d like to see, but we should not require the contributor to make them all. For example, we might add missing tests ourselves or collaborate with them to add tests.

We should close PRs due to lack of activity but invite people to reopen them if they pick things up again.

Practical advice

Try to comment on changed files rather than by commit, as the notifications are easier to follow.

Do not comment on PRs outside working hours, even to acknowledge them.

We do not want to set an expectation that we support making changes at night or during the weekend.

If the codebase has a changelog and the contributor has not added a line describing their change, raise a PR for this yourself after they have merged their PR. Regardless of who has written the changelog entry, add a “Thank you `@githubusername`”.

You should add a documentation credit thanking the contributor if you do not add documentation changes to the changelog.

Further reading

- Anna Shipman (https://github.com/annashipman) has written a useful blog post about how to raise a good pull request (https://www.annashipman.co.uk/jfdi/good-pull-requests.html).
- A great example of a good pull request (https://github.com/alphagov/frontend/pull/784) raised by Alice Bartlett (https://github.com/alicebartlett).
- A useful post about using automatic style enforcement to make pull request review more effective (https://technology.blog.gov.uk/2016/09/30/easing-the-process-of-pull-request-reviews/) by Paul Bowsher (https://mastodon.me.uk/@boffbowsh).

This page was last reviewed on 5 November 2024. It needs to be reviewed again on 5 November 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Tracking Access Control

You should track the list of users who have access to secrets by logging the permissions, such as accounts and credentials, associated with a security resource in a single, centralised Access Control List (ACL).

The ACL specifies who or what is allowed to access the resource containing secrets and the operations which are allowed to be performed on the resource. See Principle of Least Privilege (/standards/principle-least-access.html) for guidance on minimising access rights.

User access rights must be reviewed when people change roles or teams as part of your joiners, leavers and movers process. ACLs should also be regularly reviewed on a predefined cadence (e.g. monthly, quarterly).

Identified exceptions should be raised with the colleague responsible for risk management in the directorate for escalation.

Teams ACL review should be documented to re□ect:

- who (colleague) completed the review
- date review is undertaken
- next review date
- any changes to user status granted access, and the reason for change (if any)

Further guidance

- How to manage access to your third-party service accounts (/standards/accounts-with-third-parties.html)
- NCSC Cyber Assessment Framework - Principle B2 Identity and Access Control (https://www.ncsc.gov.uk/collection/cyber-assessment-framework/caf-objective-b/principle-b2-identity-and-access-control)

This page was set to be reviewed before 26 January 2025 by the page owner #gds-way (https://gds.slack.com/messages/gds-way). This might mean the content is out of date.

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Secret Auditing

Secrets are digital keys, certificates or credentials such as passwords, SSH keys and tokens. They're used for managing access permissions, encryption and decryption.

Ideally, each secret is:

- created for a particular purpose
- created for a particular entity (a person, machine or task)
- only valid for a set period of time

These parameters may not be practical in all the use cases, especially when the secrets management is a manual process, so we need auditing to know who accessed which secrets, when, and for what purpose.

When to carry out a secrets audit

Secrets auditing can be a continuous or sporadic process (i.e. automated or manual).

You may wish to carry out secrets auditing if you want to:

- understand who has access to which secrets, and evaluate whether this is appropriate for your security model
- be able to provide evidence someone has gained access to some secrets to understand the impact of a security breach
- confirm whether your onboarding/offboarding processes are working as intended (particularly in evolving privilege for movers)
- have confidence in rotating secrets
- see who has encrypted or decrypted a secret
- create alerts when sensitive secrets are being accessed

How to carry out a secrets audit

Your team should:

- log any changes to your access control list (/standards/secrets-acl.html) (ACL) and capture any deviations from your default, or expected access, ideally in Splunk. If you have documentation of secrets, you should make sure this is up to date as it can be used to underpin the audit process.
- automate the logging of secrets during the secrets life cycle, from generating secrets to rotating and destroying the secrets, if possible
- create alerts for use cases of access which may indicate a security breach
    - this should include use of high risk secrets, such as break-glass accounts or root
- look for any opportunities to replace secrets with ephemeral tokens, such as moving from hard coded AWS secrets in GitHub Actions to using OpenID Connect.
- consider whether your use of secrets can be reduced by relying on e.g. automated CI/CD processes (such as Terraform etc)

Your team must:

- use Github's Secret Scanning (https://docs.github.com/en/code-security/secret-scanning/about-secret-scanning), and consider using their Partner Program (https://docs.github.com/en/code-security/secret-scanning/secret-scanning-partner-program) features
- review the alerts on a regular basis, for example as part of the normal planning and prioritisation process
- ensure that the outcome of each alert is recorded, even if it is closed as a false positive

You should be conscious of the impacts of automated secrets rotation on service users. There may also be legal/contractual obligations to consider if you are changing how you manage user secrets.

This page was last reviewed on 19 February 2025. It needs to be reviewed again on 19 August 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

How to send email notifications

At GDS you should use the following standards for sending email notifications to service users and engineers.

How to send notifications to users

Use GOV.UK Notify (https://www.notifications.service.gov.uk/) when sending notifications to users from a GDS service, for example password resets and confirmation emails.

There's a complete feature list (https://www.notifications.service.gov.uk/features) available of what you can do with GOV.UK Notify, and you can also register for a Notify account (https://www.notifications.service.gov.uk/register) here.

The Service Manual has further information related to sending emails from your service domain (https://www.gov.uk/service-manual/technology/how-to-email-your-users).

How to send notifications to engineers

Use an alerting (/standards/alerting.html), monitoring (/standards/monitoring.html) or logging (/standards/logging.html) tool when you need to send automatic notifications from systems like Cron (https://en.wikipedia.org/wiki/Cron) to engineers.

To reduce inbox noise which can result in emails being ignored or overlooked, you should only use automatic email notifications from systems when the issue can't be picked up through monitoring or logging. In these situations send email using Amazon Simple Email Service (Amazon SES) (https://aws.amazon.com/ses/).

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Run a Ser□ ice Le□ el Indicator (SLI) □ orkshop

Setting SLIs helps you set realistic objectives for your service and avoid over-committing resources on Site Reliability Engineering (SRE) (https://sre.google/). SLIs bene□t your service by:

- de□ning Service Level Objectives (SLO) (https://sre.google/sre-book/service-level-objectives/) for your service□s user journeys
- helping prioritise your work and improve your infrastructure
- creating metrics to help classify incidents (P1-P4) (/standards/incident-management.html)
- measuring how your system performs in the medium to long term

The SLIs you de□ne must be speci□c to your service users□ experience. Your SLIs should prompt your SLOs and inform your Service Level Agreement (SLA).

Read the Google Site Reliability Engineering (SRE) manual (https://sre.google/sre-book/table-of-contents/) for more information about service level terminology.

Run □ our □ orkshop

Run your workshop with your team including your service□s:

- product manager
- delivery manager
- technical representative(s)

Your □rst workshop should not last longer than one and half hours and should focus on creating some initial results. Iterate these □rst SLIs over time and adjust your team□s practices as needed.

Run the workshop as a whiteboard exercise to capture and focus your team□s view on your service. Doing this will generate a discussion about what□s important to your users.

When you run the workshop:

1. Prioritise your most important user journeys.
2. Map your user journeys.
3. Define what good means to users.
4. Map out high-level system components.
5. Define your SLIs.
6. Create implementation tasks.
7. Observe and iterate your SLIs.

1. Prioritise your most important user journeys

People use your service to complete user journeys to achieve specific outcomes. Define your most important user journeys.

For example, in Digital MarketPlace (https://www.digitalmarketplace.service.gov.uk/) the most important user journey is where suppliers submit their bids. Your team will surface many user outcomes so prioritise 2 to 3 items to start with.

2. Map your user journeys

Your product manager should lead your team in mapping your service's user journeys, starting with the most important.

3. Define what good means to users

Define what "good" looks like for your service from your users perspective.

For example, if you're hosting a web service, "good" means your web service is "available" and "fast". If your service provides a type of publishing platform, "good" can mean how fast your service publishes data to live (data freshness).

4. Map out high-level system components

A technical person in your team, like a developer or site reliability engineer (SRE), should draw a high-level system diagram for your service.

The diagram should show the major system components for each user journey. This could include only 2 or 3 components, or multiple system-to-system interactions including 3rd party software providers.

5. Define your SLIs

De□ne potential SLIs and identify points in your service where you can measure them. These SLIs must re□ect your user□s de□nition of good.

Technical members of your team should contribute to where, how and what metrics they collect. These metrics will form your SLIs. Mark out your SLIs over a period of time, for example, a moving hourly window where your SLIs show system performance for the previous hour.

6. Create implementation tasks

Your team□s product or technical lead(s) seperate down your SLIs into tasks, for example using Trello, Pivotal Tracker (https://www.pivotaltracker.com/) or Jira Software (https://www.atlassian.com/software/jira). Some teams create an Agile epic (https://www.atlassian.com/agile/project-management/epics) to cover every task needed to carry out their SLIs and SLOs.

7. Observe and iterate your SLIs

After creating your SLIs, observe them over a period of time (for example 1 week). After this time, iterate your SLIs to better understand your service□s performance and how the SLIs help your team make decisions.

Case stud□ : Reliabilit□ Engineering Observe team

The Observe team organised a workshop in the form of a whiteboard exercise with their product manager, tech lead and technical architect to identify their □rst set of SLIs.

Prioritising user journeys

The Observe team identi□ed user journeys for their products. Observe users want to:

- know how their service is doing - by viewing a dashboard
- know if their service degrades
- update an alert
- be paged (alerted) when an issue affects users
- add new metrics
- debug live issues
- receive a ticket when there□s a hazard

The team prioritised the 3 most important user journeys:

- know how their service is doing - by viewing a dashboard

- be paged (alerted) when an issue affects users
- debug live issues

The team developed SLIs for the most important user journey: □ Knowing how their service performs (by viewing a dashboard)□ .

Mapping user journeys

The team mapped the user journey for □ choose a Grafana dashboard□ . Users look at a Grafana dashboard (https://grafana.com/dashboards) to get a general understanding of system performance and then focus on individual graphs using the time axis to debug live issues.

User journey for choosing a Grafana dashboard:

1. The user chooses a dashboard from a list of dashboards.
2. The user chooses a graph on the dashboard.
3. The user drills down (accesses data at a lower level in the hierarchy of the data structure) to see more details about the graph.

Defining what good means

From a user□s perspective □ good□ in □ choose a Grafana dashboard□ means the:

- Grafana (web service) is available
- data shown on the graph are near-real-time (live) and accurate
- Grafana response is fast enough

Mapping out high-level system components

The team mapped out the system components to complete the user journey □ choose a Grafana dashboard□ :

1. A user views a Grafana dashboard on their computer.
2. The computer fetches data from a Grafana server running on GOV.UK PaaS (https://www.cloud.service.gov.uk/).
3. The Grafana server fetches data from a Prometheus database (https://prometheus.io).

Choosing a Grafana dashboard: high-level system components

The team de□ned the SLI over the last hour the service collected metrics, identifying:

- the importance of successful requests
- latency for the users

Defining the SLIs

The first sets of SLIs are a percentage of:

- successful (status code is not 5xx (https://en.wikipedia.org/wiki/List_of_HTTP_status_codes#5xx_server_errors)) requests
- requests that the service responds to within 2.5s

You could also represent this relationship using a formula:

The availability per hour of the SLI is the number of total requests minus the number of 5xx requests divided by the number of total requests multiplied by 100%.

The latency per hour of the SLI is the number of requests within 2.5 seconds divided by the number of total requests multiplied by 100%.

Choosing a Grafana dashboard: points of measurement

The team looked at the components to figure out the best place to collect data for their SLIs. They looked for the point closest to the end user, so that the metrics would be representative of their experience. For example, if the dashboard was loading slowly for users, the SLI would reflect this accurately. In this case, the team collected data using a component called the PaaS Prometheus Exporter, which was closest to the end user.

Creating tasks and iterating SLIs

The team created 2 related stories to gather metrics, displaying the SLIs on Grafana dashboards.

The team has since refined the percentage of successful requests responded within 2.5s to better reflect service status.

Further reading

Find out more about SLIs and how to identify and create metrics by reading:

- The Google Site Reliability Engineering (SRE) (https://sre.google/sre-book/service-level-objectives/) manual
- SRE fundamentals: SLIs, SLAs and SLOs (https://cloud.google.com/blog/products/devops-sre/sre-fundamentals-slis-slas-and-slos) - Google Cloud Platform blog
- The Sysadmin Approach to Service Management (https://sre.google/sre-book/production-environment/) - Site Reliability Engineering by Google

This page was last reviewed on 20 February 2025. It needs to be reviewed again on 20 November 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Make data-driven decisions with service level objectives

Service level objectives (SLOs) set out a target level for your service's reliability. Using SLOs helps you make data-driven decisions about the opportunity cost (https://www.investopedia.com/terms/o/opportunitycost.asp) of reliability work.

You can use SLOs and error budgets alongside service level indicators (/standards/slis.html) (SLIs) to help you make better decisions.

SLOs help you decide how to prioritise reliability work, such as bug fixes and architectural improvements, by stating the problem in a user-centric way. For example, 'this bug affects 10% of users' or 'this will provide a faster experience for the users by 500ms'.

What you'll need to know before setting your SLOs

Current SLI levels

Current performance based on SLIs is usually a good place to start, especially if you do not have any other information. It also helps to set a baseline that you can improve to reflect service objectives.

Your dependencies

You'll need to know the SLOs for services your service relies on. For example, your hosting provider (/standards/hosting.html) or any other components that are important to your users' experience.

User satisfaction

You'll need to be aware of current user satisfaction levels. SLOs give users an expectation about the level of reliability they can expect from your service. Users are more likely to complain or stop using the service if it drops below the target level.

If your users are happy with the availability and latency of your service you can set your SLOs based on your current SLIs.

In other cases, your user expectations will lead to a set of SLOs the service's current architecture may not achieve. You can reduce the gap between expectations and your SLOs by using techniques like ratcheting (https://martinfowler.com/articles/useOfMetrics.html#MetricsAsARatchet).

Setting your first SLOs

Use the SLIs as a reference

Your SLIs will give you a good idea of how your current platform is doing and give you a baseline reference.

Choose a time window

You should report SLOs over a set period of time. For most purposes, a 4-week rolling window works well.

A shorter time window lets you make decisions and iterate development more quickly, such as prioritising bug fixes. You could also set your time window over a longer period of time. This can be better for more strategic decisions and can align with your business calendar.

Choose a SLO Target

You should consider setting an SLO target with technical, product and business implications.

Although your current performance (SLIs) should be taken into account, you should not set an SLO target based on the current performance to lock you into supporting the system with overachieving goals. Balance the SLO target with your product and business objectives.

Create error budgets

Error budgets help you know when to prioritise reliability work and when to prioritise new features. They provide an absolute value to a process and help determine when you need to take action.

You can work out the error budgets from your SLOs. For example, if your SLO on availability is 99.9%, your error budget is 0.1% of the request volume. For example, if your service had 1 million requests, your error budget will be 1,000. This works out at 40 minutes of outage in your 4-week rolling window.

An error budget is usually expressed as a percentage rather than an absolute value. This can help you to focus on what the error means to your users. For example, if your service has 435 errors during the SLO period, it would have used up:

435/1000x100% = 43.5%

of its error budget.

If your error rate, which is the number of bad requests in this case, is less than 1,000 you've met your error budget.

SLO burn rate and alerts

SLO burn rate is how fast, relative to the SLO, the service consumes the error budget. An alerting window is defined to capture errors accumulated within this period before alerting.

Team should capture both sudden, fast SLO burn, which may be caused by an incident or bug; and gradual, slow SLO burn, which may be caused by system degradation and scalability issues.

Using error budgets to create policy

You can use error budgets to create policies, which are the actions you need to take when your service has used up, or nearly used up, its error budget for the period.

For example, your policy could state that you:

- stop launching features until the SLO is met again
- devote 80% of developer time to reliability-related bug fixing

SLO and error budgets can also help data-driven decision making.

This table is an example of how you can use a combination of your SLO and user satisfaction to decide what action to take.

SLOs	Toil	Customer satisfaction	Action
Met	Low	High	Choose to (a) relax release and deployment processes and increase velocity, or (b) step back from the engagement and focus engineering time on services that need more reliability
Met	Low	Low	Tighten SLO
Met	High	High	If alerting is generating false positives, reduce sensitivity. Otherwise, temporarily loosen the SLOs (or

			of□bad toil) and fix product and/or improve automated fault mitigation
Met	High	Low	Tighten SLO
Missed	Low	High	Loosen SLO
Missed	Low	Low	Increase alerting sensitivity
Missed	High	High	Loosen SLO
Missed	High	Low	Of□bad toil and fix product and/or improve automated fault mitigation

Table credit: https://sre.google/books/ (https://sre.google/books/)

Your team should agree on the SLOs and error budget policies. It's useful to get agreement from the:

- product manager
- delivery manager
- technical architect
- developers
- site reliability engineers

Continuous improvement of SLOs

You can use your user research and outage records to improve your SLOs. Eventually you'll be able to set your SLOs by user experience. For example, if your users are happy with the service's performance, you can experiment with relaxing your SLOs and measure the resulting user satisfaction.

Further reading

- Service Level Objectives (https://sre.google/sre-book/service-level-objectives/) - chapter 4 of the Google SRE handbook (https://sre.google/books/)
- Alerting on SLOs (https://sre.google/workbook/alerting-on-slos/) - chapter 5 of the Google SRE handbook (https://sre.google/books/)
- Liz Fong-Jones - Adopting SRE and Error Budgets (https://www.youtube.com/watch?v=7VeU6LnOUms)

This page was last reviewed on 20 February 2025. It needs to be reviewed again on 20 November 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Use GitHub

GDS uses the alphagov (https://github.com/alphagov/) organisation on GitHub (https://technology.blog.gov.uk/2016/05/31/how-we-use-git-at-the-government-digital-service/) to collaborate on code. The GOV.UK One Login programme uses govuk-one-login (https://github.com/govuk-one-login).

Getting access to alphagov

You can use your personal GitHub account to access alphagov if you wish.

All github user accounts added to [redacted] *must* be connected (https://docs.github.com/en/account-and-profile/setting-up-and-managing-your-personal-account-on-github/managing-email-preferences/adding-an-email-address-to-your-github-account) with a valid [redacted] email address. Accounts not connected to a valid email address will be removed from [redacted].

You must also set up two-factor authentication on your account.

To join [redacted] ask your tech lead or technical architect to invite you. Make sure you've connected your GDS email address to your account first, otherwise your account will be removed.

Configuring GitHub repositories

Consider protecting your main branch (https://docs.github.com/en/github/administering-a-repository/defining-the-mergeability-of-pull-requests/about-protected-branches) to prevent changes being committed without a suitable review.

You can also consider backing up your Git repositories to another location (this should be a team responsibility). If you are using AWS to host your service AWS CodeCommit (https://aws.amazon.com/codecommit/) is one option.

How to retire applications

If an application is no longer used in production, you should archive its repository (https://docs.github.com/en/github/creating-cloning-and-archiving-repositories/archiving-a-github-

repository/archiving-repositories).

Update the application's README to explain why the repository has been archived, and link to a new location if the application has been superseded.

Using Github Actions and workflows

Alphagov repositories can be configured to use GitHub Actions (https://docs.github.com/en/actions) for CI/CD jobs, for example running unit tests or deploying a static site.

Actions and workflows can be powerful, so take care to follow best security practice.

Ensure the repository permissions follow the principle of least privilege (/standards/principle-least-access.html) by:

- disabling Actions entirely on repositories that do not use it (this will hide the 'Actions' tab from the repository menu)
- setting workflow permissions to read-only by default
- setting granular permissions in the workflow YAML (https://docs.github.com/actions/reference/authentication-in-a-workflow#modifying-the-permissions-for-the-github_token) where appropriate
- using environments to restrict access to secrets (https://docs.github.com/en/actions/deployment/targeting-different-environments/using-environments-for-deployment#environment-secrets)

If your repository has external contributors, ensure they do not have permissions to add workflows or trigger workflow runs (https://docs.github.com/en/actions/managing-workflow-runs/approving-workflow-runs-from-public-forks).

If your workflow interacts with another resource (for example AWS or DockerHub), consider setting up a dedicated account or role, with permissions limited to the scope of the action. Use temporary credentials in preference to storing long-lived credentials in a secret. In particular, when accessing AWS you must authenticate using the OpenID Connect token (https://docs.github.com/en/actions/deployment/security-hardening-your-deployments/configuring-openid-connect-in-amazon-web-services) and not using an IAM User's access key and secret access key. You should also specify the branch you expect to be deploying from (for example, [illegible]) in your IAM role to make sure code cannot be deployed from untrusted branches.

Consider protecting the .[illegible]/[illegible] folder by using a CODEOWNERS file (https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-code-owners) and requiring review from CODEOWNERS for merges into protected branches.

Consider creating a Workflow Template in the alphagov workflow folder (https://github.com/alphagov/.github/tree/main/workflow-templates) if you need to share a similar

workflow between many repositories.

Create your own local actions (https://docs.github.com/en/actions/creating-actions/about-actions) wherever possible.

If using GitHub-owned actions, pin to a specific version (https://docs.github.com/en/actions/learn-github-actions/finding-and-customizing-actions#using-release-management-for-your-custom-actions) and configure Dependabot to keep your actions up to date (https://docs.github.com/en/code-security/dependabot/working-with-dependabot/keeping-your-actions-up-to-date-with-dependabot#example-dependabotyml-file-for-github-actions) by adding a comment on the same line with the tag the commitsha represents. For example:

```
- uses: actions/checkout@01aecccf739ca6ff86c0539fbc67a7a5007bbc81 # v2.1.0
```

Pinned versions should include the semver version in a comment next to the SHA, helping humans understand which versions we are pinned to. Where possible, allow automated dependency management tools to scan these version comments and suggest updates.

Please double-check that the digest and the commented version are consistent each time you upgrade, as dependabot does not have perfect capability at either identifying the magnitude of the upgrade, or necessarily updating the commented pin.

Third-party actions should only be used if:

- The provider is verified by GitHub (for example, aws-actions (https://github.com/aws-actions))
- The action is complex enough that you cannot write your own local action
- You have fully reviewed the code in the version of the third-party action you will be using
- You have pinned the specific version in your workflow and in the repository settings, using a Git commit SHA
- You have included the semver version in a comment next to the SHA, helping humans understand the version and automated tools report on what is out of date (for example dependabot)
- The third-party action is actively maintained, well-documented and tested (follow the guidance on third party dependencies (/standards/tracking-dependencies.html)).

You can enforce this in the settings for Actions by choosing 'Allow select actions' (https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/enabling-features-for-your-repository/managing-github-actions-settings-for-a-repository#allowing-specific-actions-to-run) and then 'Allow actions created by GitHub' and 'Allow Marketplace actions by verified creators' as required.

Note that for public repositories, the output of workflow runs is visible to everyone. Do not use workflows if this output could be considered sensitive.

Access GitHub support

The alphagov organisation is covered under the Cabinet Office's GitHub enterprise support agreement (https://help.github.com/en/github/working-with-github-support/github-enterprise-cloud-support). Under this agreement GitHub will respond to support requests within eight hours, Monday to Friday.

To access enterprise support you need either to be an enterprise admin or have been granted a support entitlement by an enterprise admin. There can only be a maximum of 20 people across the enterprise who have the support entitlement, so not everyone can have this.

Request support

If you are not already an enterprise admin or have a support entitlement on your GitHub user you will first need to ask one of the enterprise admins to be given permissons to access the support portal. You can do this by emailing the GDS GitHub enterprise owners google group.

Once you've been given permission, you can view and raise support requests using GitHub's support portal (https://support.github.com/).

You should use your [redacted] email during the sign up process to ensure your ticket is prioritised. You should also state that you are part of alphagov organisation in your request.

See also

- How to store source code (/standards/source-code/)
- Working with Git (/standards/source-code/working-with-git.html)
- Updating actions with DependaBot (https://docs.github.com/en/code-security/dependabot/working-with-dependabot/keeping-your-actions-up-to-date-with-dependabot)

This page was last reviewed on 20 November 2024. It needs to be reviewed again on 20 May 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Working with Git

Default branch name

Your repository's default branch should be called `main`.

To configure git to use `main` for new repositories, first make sure you are using git 2.28 or later, then run:

```
git config --global init.defaultBranch main
```

To migrate existing repositories from `master` to `main`, you can use GitHub's branch renaming tool (https://github.com/github/renaming#renaming-existing-branches).

Before you rename your default branch, you should consider the impact this will have on:

- internal users
- external users
- your continuous integration (CI) system (for example, Concourse pipelines or GitHub actions)

You can configure some CI systems to check out the default branch rather than hardcoding a branch name:

- GitHub Actions (https://github.blog/changelog/2020-07-22-github-actions-better-support-for-alternative-default-branch-names/) supports a `$default-branch` macro
- concourse-git-resource (https://github.com/concourse/git-resource): if you do not set `branch`, it will check out the default branch

Commits

Atomic commits

Each commit should be atomic - this means that each commit:

- is self-contained
- only includes changes that achieve a specific step
- should be in a logical order

As Anna Shipman describes it, they should tell a story (https://www.annashipman.co.uk/jfdi/good-pull-requests.html#make-the-pull-request-tell-a-story).

Example

From Anna's blogpost, we see an example of how a single commit with 51 files changed (https://github.com/alphagov/paas-alpha-tsuru-ansible/commit/7547ac0d35a) was broken down into a series of 8 commits implementing those changes (https://github.com/alphagov/paas-alpha-tsuru-ansible/compare/d891857...d14bb2f44) instead.

Commit messages

Writing good commit messages is important. Not just for yourself, but for other developers on your project. This includes:

- new (or recently absent) developers who want to get up to speed on progress
- interested external parties who want to follow progress of the project
- people in the public (remember, we code in the open) who want to see our work, or learn from our practices
- any future developers (including yourself) who want to see why a change was made

Capturing context around a change allows people to understand why a particular implementation decision was made, much like an architecture decision record (https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions). We're being kinder to our future selves.

Recommended blog posts on commit messages

- How to Write a Git Commit Message (https://chris.beams.io/posts/git-commit/)
- 5 useful tips for a better commit message (https://thoughtbot.com/blog/5-useful-tips-for-a-better-commit-message)
- Every line of code is always documented (https://mislav.net/2014/02/hidden-documentation/)

Content

A good commit message briefly summarises the "what" for scanning purposes, but also includes the "why". If the "what" in the message is not enough, the diff is there as a fallback. This is not true for the "why" of a change - this can be much harder or impossible to reconstruct, but is often of great significance.

Example

`Set cache headers`

prefer:

```
Set cache headers

IE 6 was doing foo, so we need to do X.
See http://example.com/why-is-this-broken for more details.
```

Links to issue trackers

A link to a ticket in an issue tracker should not be seen as an alternative to writing a commit message.

While a link can add some extra context for people reviewing a pull-request, the commit message should stand on its own. There's no guarantee that the link will continue to work in the future when someone is looking through the commit history to understand why a change was made.

If you are adding a link to a publicly viewable repository, ensure that the linked ticket is publicly viewable (and likely to remain so).

Structure

Commit messages should start with a one-line summary no longer than 50 characters. Various Git tools (including GitHub) use this as the commit summary, so you should format it like an email subject, with a leading capital and no full stop. The Git convention is to write these in the imperative mood, as if you are issuing a command to the repository.

For example:

> Leverage best-of-breed synergies going forward

It can help to imagine a silent "please" at the beginning of your message:

> [Please] Leverage best-of-breed synergies going forward

You should leave a blank line before the rest of the commit message, which you should wrap at around 72 characters: this makes it easier to view commit messages in a terminal.

Example

Taken from Tim Pope's guidelines (https://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html).

> Capitalized, short (50 chars or less) summary
>
> More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of an email and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely); tools like rebase can get confused if you run the two together.
>
> Write your commit message in the present tense: "Fix bug" and not "Fixed bug." This convention matches up with commit messages generated by commands like git merge and git revert.
>
> Further paragraphs come after blank lines.
>
> - Bullet points are okay, too
> - Typically a hyphen or asterisk is used for the bullet, preceded by a single space, with blank lines in between, but conventions vary here
> - Use a hanging indent

Branching/merging conventions

You should develop on a short-lived "feature" branch (https://git-scm.com/book/en/Git-Branching-Basic-Branching-and-Merging) rather than directly on the trunk branch, which is usually `main` or `master`, unless there is a compelling reason to do otherwise.

For example if your repository has many binary files, the `git diff` will not provide useful change information, and you may choose to collaborate directly on the trunk branch.

You should prefer organising your commits as small, self-contained changes rather than having a single commit per pull-request. Ideally, each commit should represent a coherent set of changes and pass all of your project's tests and static analysis. This will make your changes easier for other people to understand them at review time.

You may need to update your feature branch with the latest changes from the trunk branch when you have multiple people contributing to the same repository. In this case, you should use `git rebase` (https://docs.github.com/en/get-started/using-git/about-git-rebase) with

the trunk branch name. This applies your local changes on top of the most recent changes to the trunk branch, rather than creating a merge commit on your feature branch, which keeps the history linear and makes it easier to review.

Git allows you to rewrite local changes (https://git-scm.com/book/en/Git-Tools-Rewriting-History) by combining commits, splitting commits apart, changing commit messages and many more operations.

You should not rewrite commits that have been merged or pushed to the trunk branch unless there is a compelling reason such as committing personally identifiable information or application secrets. In this case you should communicate this to affected staff within GDS, so they know to update their local working copy.

You should use `git merge` with `--no-ff` if you are manually merging a feature branch into your repository's trunk branch rather than GitHub's own merge capability. This option preserves evidence of your feature branch in the repository history by creating a merge commit even if your changes could be simply applied to the trunk branch.

You should use the merge strategy that best suits the context of your team and project. Rebasing changes onto your trunk branch maintains a linear history but commits are not guaranteed to be in time order. Merging changes creates a merge commit which is useful as a known integration point for a change but the repository history becomes non-linear.

Do not use `git push -f`, use `--force-with-lease` instead

Force pushing in git is a subject that attracts all kinds of religious battles. This advice is not about the merits of force pushing. This is about how to use `git push -f` if and when you do use it.

Let's say you're working on a branch, 'foobar', and you decide to force push to the remote. So you do this:

```
$ git push -f
```

If anyone else has committed anything to your branch since you last pulled, you will blow their changes away.

So for a bit of safety, if you ever need to force push please instead do:

```
$ git push --force-with-lease
```

`--force-with-lease` refuses to update a branch (https://blog.developer.atlassian.com/force-with-lease/) unless it is the state that we expect, which is that nobody has updated the remote branch.

See also

- How to store source code (/standards/source-code/)
- Working with Git (/standards/source-code/working-with-git.html)

This page was last reviewed on 18 February 2025. It needs to be reviewed again on 18 November 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

How to store source code

At GDS, we follow the principles set out in the Service Manual for managing the code we write by:

- using version control (https://www.gov.uk/service-manual/technology/maintaining-version-control-in-coding)
- making source code open (https://www.gov.uk/service-manual/technology/making-source-code-open-and-reusable)

Publish open source code

Wherever possible, we make our source code open and reusable. This means other government departments and people in outside organisations can bene□t from our work. We also maintain several open source projects developed for use on GOV.UK and with other work we do, such as GOV.UK Frontend (https://github.com/alphagov/govuk-frontend).

It□s not always appropriate to open code. There are sometimes grounds for keeping some data and code closed (https://www.gov.uk/government/publications/open-source-guidance/when-code-should-be-open-or-closed), for example:

- keys and credentials
- algorithms used to detect fraud
- code or data that makes clear details of unannounced policy

The Service Manual explains how to open previously closed code and your responsibilities for maintaining open code (https://www.gov.uk/service-manual/technology/making-source-code-open-and-reusable).

When you publish open source code, your project must:

- include a README, using guidance for writing a README (/manuals/readme-guidance.html)
- have useful and informative commit messages (/standards/source-code/working-with-git.html#commit-messages) about why a change was made

- include an MIT and OGL licence file (/manuals/licensing.html)
- have an email address to submit security related bug reports

If your changes will impact third parties, your project must also:

- provide a changelog, optionally also providing a template for making changes. See the GOV.UK Frontend changelog template (https://github.com/alphagov/govuk-frontend/blob/main/docs/contributing/CHANGELOG_TEMPLATE.md) for an example
- provide release notes (/manuals/release-notes.html)
- link to a public list of known issues and bugs, for example GOV.UK Frontend issues (https://github.com/alphagov/govuk-frontend/issues)
- list a version number compatible with Semantic Versioning (https://semver.org/spec/v2.0.0.html)

Your open source code project should:

- when appropriate, publish packages to relevant language specific repositories such as PyPI - the Python Package Index (https://pypi.org/) or RubyGems (https://rubygems.org/)
- post contributors' guidelines in a contributing file, like the GOV.UK Frontend repository (https://github.com/alphagov/govuk-frontend/blob/main/CONTRIBUTING.md)
- set up any tests to run in a public continuous integration environment using tools such as Github Actions (/standards/source-code/use-github.html#using-github-actions-and-workflows)

You could also provide a mailing list so people can discuss your project.

See also

- Use Github (/standards/source-code/use-github.html)
- Working with Git (/standards/source-code/working-with-git.html)

This page was last reviewed on 18 February 2025. It needs to be reviewed again on 18 February 2026 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

How to store credentials

Depending on how you manage your accounts (/standards/accounts-with-third-parties.html), you, your team and the service you run may have credentials or other secrets that you need to store securely.

Personal credentials

Personal credentials belong only to you. They uniquely identify you and grant access to your GitHub, AWS, and GOV.UK Signon accounts.

If possible, use the password manager built into your browser (https://lock.cmpxchg8b.com/passmgrs.html#conclusion). This is simpler than setting up an extra account with a third party and avoids the potential issues below.

If you are unable to use your browser's password manager then you should use a third-party password manager. This could be necessary if your browser has an accessibility issue, or if you work with multiple browsers.

Third-party password managers used by people at GDS include:

- **1Password** (https://1password.com/)
- **BitWarden** (https://bitwarden.com/) - An open source password manager.
- **KeePassXC** (https://keepassxc.org/) - An offline password store, which you may want to backup somewhere.
- **QtPass** (https://qtpass.org/) - Another offline store, integrated with Git and GPG / pass.

There is a security trade-off involved in using browser extensions to autofill credentials.

Auto-filling credentials can protect against phishing attacks. Your password manager will refuse to autofill credentials for the wrong site, such as `exxample.com` attempting to impersonate `example.com`. However, it can be difficult to implement this functionality securely in an extension (https://lock.cmpxchg8b.com/passmgrs.html#interprocess-communication).

Team credentials

Credentials sometimes need to be shared across a team or programme. Software repositories (NPM, RubyGems, Maven Central) and admin portals (Fastly, DockerHub) will often have shared credentials.

You should follow the guidance for managing team credentials. (/standards/accounts-with-third-parties.html).

We have no established best practice for storing shared credentials at GDS. Some teams have a “credentials” GitHub repository and use pass (https://www.passwordstore.org/) or blackbox (https://github.com/StackExchange/blackbox) to store credentials encrypted with GPG.

Make sure you configure the repo as internal or ideally private (https://docs.github.com/en/github/creating-cloning-and-archiving-repositories/creating-a-repository-on-github/about-repository-visibility) to help prevent accidental disclosure

Investigate alternatives before adopting GPG-based credential stores for new teams.

- We cannot revoke access to GPG-based credentials unless we change them. Anyone with access can still decrypt credentials using their local copy of the repo and its commit history.
- It creates a high barrier to entry as GPG tools are generally difficult to use (https://latacora.micro.blog/2019/07/16/the-pgp-problem.html) and key-servers are unreliable.

The GOV.UK Design System and Prototype Kit teams have recently set up a Bitwarden (https://bitwarden.com) organisation as a replacement for their credentials repository.

Service credentials

Deployed services sometimes need sensitive configuration such as API keys and IP block lists.

Use the secret management feature of your infrastructure or cloud provider e.g. AWS Secrets Manager (https://aws.amazon.com/secrets-manager/getting-started/), HashiCorp Vault (https://www.vaultproject.io/). This should make it easy to control and audit access to the credentials.

Older projects in GDS might have a “secrets” repository and use GPG to encrypt files, which are then decrypted during the deployment process. This approach should not be used unless there is no better option.

This page was last reviewed on 18 February 2025. It needs to be reviewed again on 18 August 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

How to track technical debt

While planning for the technical work we do in GDS, we need a common language between technologists and product people to describe the scale and scope of technical debt.

The language should enable communication of the bene□ts of individual pieces of work that pay down tech debt. It should also enable us to be clearer about the consequences of not doing something, or leaving a piece of work in a certain state, and should allow us to track the reduction or accumulation of technical debt over time.

The process used for tracking debt should not add signi□cant overhead to planning, and assigning a rating to a piece of technical debt should be quick and simple.

Example consequences of tech debt

- Harder to patch security vulnerabilities
- Harder to implement new features
- Harder to onboard new developers
- Consumes too many compute resources
- Too closely coupled to an architecture
- Causes too many support tickets
- Creates too much chore work
- Dif□cult to debug issues
- Inconsistent architecture

Example causes of tech debt

- Out of date dependencies
- Hardcoded con□guration
- Component has too many responsibilities
- Long running test suite

- Manual deployment task
- Missing documentation
- No admin interface
- Not using lessons learnt since build

Classifying and measuring

Classifying tech debt uses two factors: the current impact of the consequence and the effort required to remove the cause. Both are subjectively measured as high (red), medium (amber) or low (green) with a justification and explanation of the cause and consequence.

These are subjectively combined into an overall high/medium/low risk rating, with the reason for the relative weighting also recorded. This rating signifies the risk and associated cost with not dealing with the item right now. In some circumstances (such as an upcoming planned rewrite or retirement) we would accept a high risk as the item would go away naturally anyway. For this reason, the risks do not necessarily map directly to priorities.

This is similar to how risks are recorded in a Risk Register. High, medium and low ratings are given for impact and likelihood along with a combined overall rating based on a decision about the relative importance of the impact and likelihood. The higher the overall rating, the more the programme should be concerned about it.

The impact of a piece of technical debt, and the effort required to remove the cause of it, will change over time. Review recorded items of technical debt at appropriate periods. For example: if a problematic system was not being actively worked on at the time a piece of debt was originally recorded, it may have a low impact rating; if it subsequently becomes worked on again, you should review the impact.

Process

The exact process for tracking and managing technical debt can vary per team and product but should be in line with the following guidance.

Items of technical debt should be recorded in the appropriate teams work management tool, for example, Trello, Jira or GitHub Projects. The product's technical leadership should triage these on a fortnightly basis and agree the assigned ratings. All items should be periodically reviewed to check that the rating given is still relevant and correct.

The product's technical leadership (for example, Technical Architect, Technical Leads, Lead Developers) should use these lists during conversations with Product and Delivery Managers when prioritising work. Technical debt existing on the register doesn't necessarily mean it will be paid down at any point, only that the whole team is conscious of its existence and will take it into account in product decisions.

Tracking debt as it gets created will follow the same process, with a link to a card on a team's backlog for the story that created it. The debt review process doesn't make decisions on whether the creation of the debt is the correct thing to do, that decision stays with the product team.

Example

> ### Whitehall has its own upload management system
>
> Cause: Whitehall's system for handling uploads of PDFs, CSVs and images from publishing users is different to the Asset Manager application used by Specialist Publisher and others. This is due to it being developed in isolation from another system.
>
> Consequences:
>
> - Inconsistent architecture
> - Too closely coupled to physical disk
> - Causes support tickets due to delay in processing attachments
>
> Impact of debt: High due to confusion, operational restrictions and support burden.
>
> Effort to pay down: High due to the technical effort involved in changing Whitehall to talk to Asset Manager, implementing missing features in Asset Manager and migrating existing assets.
>
> Overall rating: High

This page was last reviewed on 14 February 2025. It needs to be reviewed again on 14 August 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

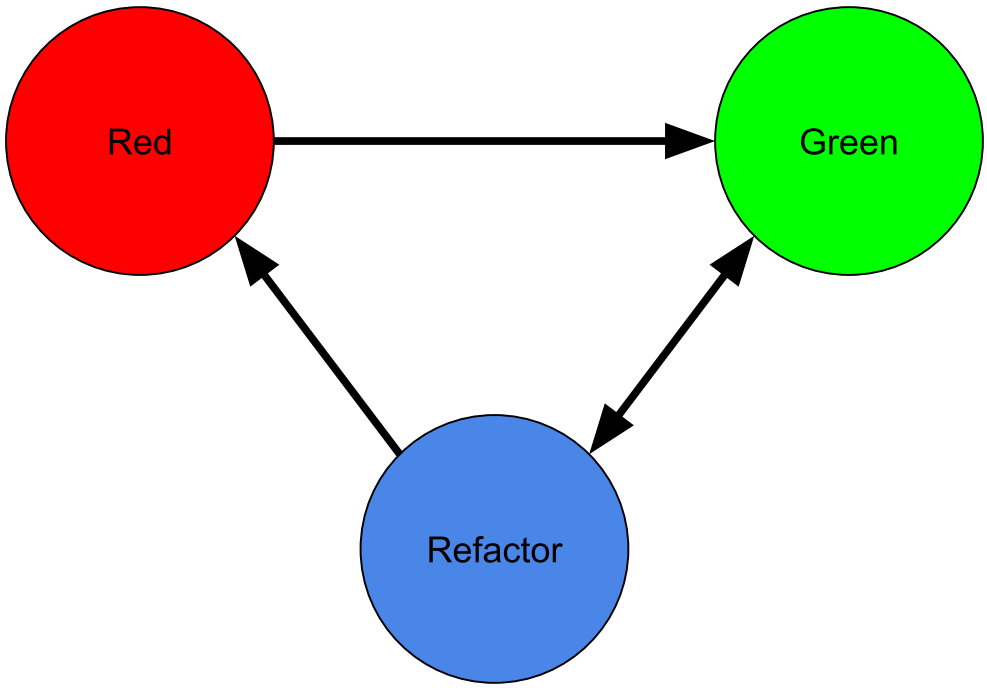
Test-driven development (TDD)

GDS advocates for agile software development practices because we believe that □exibility and responsiveness-to-change are key to building software that meets the needs of our users.

Test-driven development (TDD) is a core agile software development practice which aims to build □exibility and correctness into production software.

The process starts with a set of desired behaviours for a piece of code. This list could include happy-path behaviours, error cases, and interaction points with other components.

The inner-loop of TDD is also commonly referred to as “Red-Green-Refactor”; red to represent a failing test, green to represent all passing tests, and refactoring only when all tests are passing.



Red - □ rite a failing test

Turn exactly one item on the list into an actual, concrete, runnable test. This test should fail when you run it, and it should fail in the way you expect.

If it fails unexpectedly, that could indicate a functionality gap or a misimplementation. For example, setting a field as immutable when you expect to mutate it on later interactions.

Green - make all tests pass

Change the code to make all tests, including the new test, pass. If you discover that you are missing a behaviour or test, add it to the list.

Tests should pass in a timely manner and slow-running tests should be investigated in the refactoring stage. You may choose to reduce the size of your test case or decompose the functionality into multiple, separately testable, components.

Refactor - iterate on □ our design

Improve the design and implementation of the code by removing duplication, utilising well known patterns and structures.

This is where the majority of software design happens. You have code that behaves as specified in the tests and that is your safety net for iterating the models and structures of the code to best represent the software's needs at that particular time.

You should also let yourself be guided by the tests you've written; if a test is hard to write or you find yourself having to update many tests to support a new test without using automated refactoring tools, then take a step back examine why.

These steps are repeated until your set of behaviours is exhausted.

Wh□ we recommend it

The advantage of writing the tests before the implementation is that the behaviour of the code you will write is defined first, rather than implementing the change and retroactively applying tests.

A test-first approach gives you quick feedback on the design of your code - if your code is becoming difficult to test or has many dependencies, these are signals that you likely need to refactor.

> For each desired change, make the change easy (warning: this may be hard), then make the easy change - Kent Beck

TDD naturally lends itself to our recommended approach to [breaking down work (/standards/breaking-down-work.html)](/standards/breaking-down-work.html). If you are careful about introducing feature toggles so

that code can be “dark launched” into production environments, TDD can give you a robust and reliable iterative workflow.

For example, if you are defining a new component, you could first define the interface, integrate a “do nothing” or noop version of that component into the live codebase, and then iterate on that implementation or another implementation safely.

TDD is also a great tool for cultivating a testing mindset in teams. Practicing these techniques and concepts will teach how to approach a problem with the curiosity (what does this code do if I give it these inputs?) and skepticism (does this code really do what it’s supposed to?) necessary for building robust and resilient user-facing software.

What if we can't do TDD?

The most valuable output of the TDD process is not the code itself but the tight feedback loops that allow you to reflect and the iterative design decisions that you make along the way.

If doing what Kent Beck calls canon TDD (https://tidyfirst.substack.com/p/canon-tdd) is not possible, for whatever reason, that’s okay as long as you have another mechanism for realising the fast feedback and incremental delivery advantages that TDD does provide.

Beyond TDD

TDD works best when the feedback loops are fast. For unit testing, which should be subsecond, and component integration testing, the TDD process is very effective.

However, it is less effective for testing processes with longer feedback loops such as feature acceptance testing across multiple components, especially if you are unable to run those tests locally before pushing your code.

As systems get more complex, we usually cannot enumerate and emulate all possible interactions within unit and integration tests. Exploratory testing, done collaboratively and built on a solid foundation of fast tests, can help a team tease out unknown or unexpected behaviours of their system.

Useful resources

Getting started

If you’re struggling to write your first test, consider the approach of Zero-One-Many (https://www.agilealliance.org/resources/sessions/test-driven-development-guided-by-zombies/):

start with the empty case, then consider the singular case, then generalise to many kinds of inputs.

For example, if you were processing some kind of file input, handle the empty case first (what if the file has no lines?), then the singular case (what if the file has a single line?), and then generalise.

Getting better

TDD is a skill like any other, and one which you can develop your □uency with through practice.

Try working through some example problems while adding a constraint such as TDD-As-If-You-Meant-It (https://github.com/sf105/tdd-as-if-you-meant-it), where you can only write code in your test package, and any production code must be refactored out rather than written directly. This constraint reframes the TDD approach by forcing you to write tests first rather than retrofitting them onto the existing production code.

You could also look at GeePaw Hill's Many More Much Smaller Steps (https://www.geepawhill.org/2021/10/26/mmmss-a-closer-look-at-steps/), which advocates for framing work as a composition of a lot of smaller, and therefore safer, changes than a few large, and therefore risky, changes.

Books and blogposts

- Growing Object-Oriented Software, Guided By Tests (https://www.oreilly.com/library/view/growing-object-oriented-software/9780321574442/) - Nat Pryce & Steve Freeman
- Test-Driven Development By Example (https://www.oreilly.com/library/view/test-driven-development/0321146530/) - Kent Beck
- TDD (https://codemanship.co.uk/papers.html) - Jason Gorman

This page was last reviewed on 21 October 2024. It needs to be reviewed again on 21 April 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Threat Modelling

What's a threat?

A threat is an action that causes denial, alteration or disclosure of your assets that you are trying to protect. It could be a user database, a deployment pipeline or the integrity of web form submissions.

What's threat modelling?

Threat modelling aims at identifying, prioritising and mitigating threats to a service.

Attack Tree workshops will help you:

- Understand threats that are unique to your service, helping you to adopt security conscious behaviours during its design, development and operation
- Focus mitigation efforts on the threats that matter – that is, threats that pose the greatest risk to the normal operation of your service
- Ensure the right security controls are in place to match the threats your service faces
- Adopt secure by design approach to your service throughout the service's lifecycle

The best time to perform threat modelling activities is during the design phase; however, it can be done anytime and should become a continuous process in your service team.

Within the Cabinet Office, the Cyber Security Team can 🔒 support you with threat modelling your service (https://intranet.cabinetoffice.gov.uk/it-data-and-security/cyber-and-information-security-services/threat-modelling/), as well as advising you should you decide to carry it out yourself or through a third party.

Within the Cabinet Office and GDS, we follow the Threat Modeling Manifesto (https://www.threatmodelingmanifesto.org/)'s four questions:

1. What are we working on?

2. What can go wrong?
3. What are we going to do about it?
4. Did we do a good enough job?

We answer most of the above over three sessions, with time between sessions allowing for asynchronous input.

1. Threat Discovery

Threat discovery aims to answer the “What are we working on?” and part of the “What can go wrong?” question.

We use an interactive exercise to talk through and explain the system or architecture. Someone who is comfortable with explaining the system from the service team will need to lead this part. Try to keep the scope small; it may be more effective to break a service down and perform multiple threat modelling activities.

The group of people involved typically includes the development team, cyber security and information assurance colleagues; however, working with other people, such as user researchers and product managers, often provides more varied perspectives.

As part of the interactive exercise, we surface hypothetical threats or vulnerabilities in a service. We use STRIDE to help us look at different aspects and make sure we get a rounded view of all threats.

1.1 Interactive Exercise

The purpose of this exercise is to talk through the service, architecture or the user journey with the people in the session, allowing for questions and answers.

Remotely, digital whiteboard software, such as Jamboard (https://jamboard.google.com/), Mural (https://www.mural.co/), can be used to add virtual post-its on top of the digital diagrams. Make sure the tools are accessible so all colleagues can participate and ensure access control is sufficient as the diagrams will likely be at least OFFICIAL-SENSITIVE when complete.

Another useful tool you could use is diagrams.net (https://www.diagrams.net/) with its threat modelling library (https://michenriksen.com/blog/drawio-for-threat-modeling/).

In person, which may be more appropriate for some teams and systems, diagrams are typically drawn up on a whiteboard, followed by questions and post-its of potential threats stuck at relevant points.

2. Threat Analysis

Threat analysis aims to finalise the answer to the “What can go wrong?” question. We use a scoring methodology to determine if a threat is valid and prioritise threats against each

other.

You should aim to cover all potential attack vectors (https://searchsecurity.techtarget.com/definition/attack-vector).

2.1 Scoring

After the discovery stage, you can make a copy of the Threat Modelling Scoring template (https://docs.google.com/spreadsheets/d/1j_RiCXz2anKpybu9e3i2hvtWE9OBzfkkymUcbyp6Wts/edit?usp=sharing) to use for noting down threats, the most likely actor, scoring, and tracking mitigation.

Scoring is useful as it helps prioritise what's valid and what you should look into first.

The difficulty is a rating between 0 (very easy) to 5 (hard); it should take into account how hard it would be to gain access and any costs associated with performing the attack.

The reward is a rating between 0 (low) to 5 (high); it's how valuable the outcome is to the threat actor.

The final score equals reward minus difficulty, so a high score is worse than a low one.

3. Threat Response

Threat response aims to answer the "What are we going to do about it?" question.

It may be helpful to add the outputs to your bug or ticket tracking system.

There are three typical responses to threats:

3.1 Mitigate

Implementing a preventive control, such as an architectural change, eliminates the vulnerability or blocks the threat.

3.2 Monitor

Monitoring is a reactive control to detect if a threat occurs; it controls a situation and limits damage or loss by detecting an attack early.

3.3 Acknowledge the risk

Acknowledging the risk but not planning any immediate action can help with resourcing any future mitigation or deciding to accept a risk because the cost of the mitigation outweighs the maximum cost of a threat.

4. Did we do a good enough job?

After completing the threat modelling activity sessions, you should set aside some time to review the threat modelling outputs regularly. This is to see whether you are improving the security and taking the mitigations forward; it's also important to consider any new threats.

A recommendation is to perform this review within your service team every two months; the time you need may vary depending on the scope of your threat modelling, but 90 minutes is usually sufficient.

You should consider running the threat modelling sessions every year or when there are significant architectural changes planned.

Why's it necessary?

Adam Shostack has produced an informative short video on this: Why Threat Model? (https://www.youtube.com/watch?v=YP4mNRXGcks)

Adam explains that we threat model to anticipate problems when it's inexpensive to deal with them.

How do you know what a problem is? Information and computer security is often broken up into three main pillars, known as the CIA triad:

Confidentiality

- Information accessible to authorised people

Integrity

- Information that's correct

Availability

- Information available when needed

Abuse or neglect of one or more of these pillars could lead to a security incident.

The opposite of the CIA triad is sometimes known as the DAD triad:

- If confidentiality is affected, disclosure occurs.
- If integrity is affected, alteration occurs.
- If availability is affected, denial occurs.

Here is a fun, alternative look at the CIA and DAD triads, explained with Lord of the Rings references (https://squirreltb.wordpress.com/2015/11/17/the-cia-and-dad-triad/).

STRIDE

STRIDE is a model of threats developed by Microsoft.

It's a helpful tool for thinking about different potential attacks or vulnerabilities. If you're finding it hard coming up with potential attacks, you can use STRIDE to help stimulate ideas.

Note that STRIDE is *not* useful as a taxonomy. Some attacks don't neatly fit into "spoofing" versus "tampering" versus "elevation of privilege". That doesn't matter: the important thing is to come up with threats and write them down, not to ensure they are "correctly" categorised.

Threat	Security Control
Spoofing	Authenticity
Tampering	Integrity
Repudiation	Non-repudiability
Information Disclosure	Confidentiality
Denial of Service	Availability
Elevation of Privilege	Authorisation

Spoofing

Spoofing occurs where malicious actors use another person's user credentials without their knowledge.

It also includes attacks where systems mimic or disguise themselves as other systems, such as IP address spoofing in DNS amplification attacks.

The desired property is authenticity, meaning an event or action is trustworthy and genuine.

Tampering

Only authorised users should be able to modify a system or the data it uses.

The desired property is integrity, meaning an event or action is valid and correct.

Repudiation

Repudiation means an action or event cannot be associated with what triggered it, whether that is a person or another system component.

The desired property is non-repudiation, which means an action or event is difficult to deny by a person or system.

Information Disclosure

Data such as passwords or personal information is made available to those who should not have access to it.

The desired property is confidentiality, where data is only available to those who are authorised to view or alter it.

Denial of Service (DoS)

Data is unavailable for its intended audience.

The desired property is availability, where data is always available when those authorised to view or alter it expect it to be.

Distributed Denial of Service (DDoS) is a form of DoS where multiple entities are responsible for inappropriate traffic leading to overwhelmed services.

Elevation of Privilege

Data is available to unintended or unauthenticated users.

The desired property is authorisation, where data is only available to those expected to view or alter it.

Examples

Looking at the GOV.UK Pay service as an example; Pay is likely to attract attention from organised criminal groups and lower-level hackers/script kiddies who have financial motivations. Organised criminal groups are capable of a level of attack sophistication that would mean zero-day exploits, and more advanced techniques like replay attacks, and would be willing to use supply chain attacks as a route for access. State actors are likely to be less interested in Pay, except for disruptive actions to render the service unavailable.

Applying this threat model, Pay requires a good level of cyber security (in addition to it's PCI-DSS requirements) and regular scanning/penetration testing is necessary to ensure that both basic cyber hygience and more advanced attacks are accounted for.

This would contrast with a service like GOV.UK, where the threat is likely to be state sponsored or affiliated threats and lower-level hackers/script kiddies using common toolsets/bots with political/reputational motivations.

Additional tools and links

- OWASP Application Threat Modelling (https://owasp.org/www-community/Threat_Modeling)
- Mario Areias - Threat Modelling the Death Star (https://www.youtube.com/watch?v=ivmfZ6vGkEs&feature=youtu.be) YouTube video example

This page was last reviewed on 3 October 2024. It needs to be reviewed again on 3 April 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

How to manage third party software dependencies

When you develop and operate a service, it's important to keep any third party dependencies you use up-to-date. By doing this, you can avoid potential security vulnerabilities.

Any automated tools you use to manage third party dependencies should be compatible with GDS supported programming languages (/standards/programming-languages.html#content). The tools you use should neither slow down your development process nor disclose potential security vulnerabilities to the public.

You can read more about managing software dependencies in the Service Manual (https://www.gov.uk/service-manual/technology/managing-software-dependencies), where you will find a list of common dependency management tools.

Our programming language style guides (/manuals/programming-languages.html) also contain language-specific advice about managing dependencies (for example, managing Python dependencies (/manuals/programming-languages/python/python.html#dependencies)).

Update dependencies frequently

Update your dependencies frequently rather than in 'big bang' batches. This works well with continuous delivery (/standards/continuous-delivery.html) principles and makes sure the changes introduced are small and can be automatically tested.

There are tools which scan GitHub repositories and raise pull requests (PRs) when they find dependency updates. Teams at GDS are using:

- Dependabot (https://dependabot.com/) - used by GOV.UK, and GOV.UK Pay. The GOV.UK docs contain guidance on using Dependabot (https://docs.publishing.service.gov.uk/manual/manage-ruby-dependencies.html) and how the PRs raised should be reviewed (https://docs.publishing.service.gov.uk/manual/merge-pr.html#dependabot)

> Note: this is separate from the security-only updates provided automatically by GitHub Dependabot (https://docs.github.com/en/code-security/supply-chain-security/managing-vulnerabilities-in-your-projects-dependencies/about-alerts-for-vulnerable-dependencies).
>
> Note: repos requiring at least one approving review for PRs cannot, and should not, use Dependabot's auto-approve-and-merge facility (https://docs.github.com/en/code-security/supply-chain-security/keeping-your-dependencies-updated-automatically/automating-dependabot-with-github-actions).
>
> Note: we have not enabled "Treat PR approval as a request to merge", as this would lead to a surprising behaviour at the point of approval.
>
> Note: GOV.UK has implemented RFC-167 (https://github.com/alphagov/govuk-rfcs/blob/main/rfc-167-auto-patch-dependencies.md) which allows automatic patching of all dependencies in certain cases.

- PyUp (https://pyup.io/) - a Python dependency checker. Used by GOV.UK Notify, PyUp will monitor for updates and vulnerabilities

All the above tools are free to use on public repositories.

The COD Cyber Security team will review the repositories that do not have dependency management in use and will turn on Dependabot where required. Service teams are free to use a different tool such as Snyk (https://snyk.io/), but will need to add a `no-dependabot` tag to their repository for monitoring purposes. You can 🔒 contact Cyber Security (https://gds.slack.com/archives/CCMPJKFDK) if you have any questions or need help.

Monitor for vulnerabilities

You should monitor for potential vulnerabilities in every layer of your technology stack. This is not straightforward but tooling exists to help. The Service Manual provides guidance on browser technology, tooling and mailing lists you can subscribe to (https://www.gov.uk/service-manual/technology/managing-software-dependencies#managing-risks-in-third-party-code).

Most of the tooling that helps you stay on top of dependency updates will also highlight vulnerabilities. Additional tooling includes:

GitHub security alerts (https://docs.github.com/en/code-security/supply-chain-security/managing-vulnerabilities-in-your-projects-dependencies/about-alerts-for-vulnerable-dependencies)

When GitHub discovers or is informed about a vulnerability, it will email an alert to the repository owner and users with admin access. GitHub security alerts will be turned on for all Alphagov repositories. If services wish to opt out of security advisories on their repository, they can contact Cyber Security and then add the `no-security-advisories` tag to their repository.

Snyk (https://snyk.io/)

Snyk is capable of detecting vulnerabilities in a variety of languages including all the GDS supported programming languages (/standards/programming-languages.html#content). You can configure Snyk to raise PRs, email regular reports and alert you when new vulnerabilities are detected.

If you have an old repository that is receiving security alerts but is not being worked on or maintained, you may wish to archive your repository instead.

Splunk dashboards

The Cyber Security team has created a Splunk dashboard. This gives service teams visibility of vulnerabilities in their repositories. To allow for your repositories to be categorised correctly in the Splunk dashboard, please make sure you tag your repository with the service name. If you would like to access the dashboard, 🔒 contact Cyber Security (https://gds.slack.com/archives/CCMPJKFDK).

Managing Docker dependencies

Rebuild Docker base images

Like dependencies, Docker base images are also frequently updated. If you run containers as part of your service, you should regularly rebuild your images (and base images) to include the latest updates. Automate this process where possible.

Specifying Docker image tags

The GDS Way Dockerfile guidance contains advice on how to use Docker image tags (/manuals/programming-languages/docker.html#using-tags-and-digests-in-from-instructions) to specify the exact container image version to use.

Use official Docker base images

Always use official base images (https://docs.docker.com/docker-hub/official_images/). Docker Hub regularly scans official images and you can view the results by logging into Docker Hub. If you do not regularly update your base image, you must make sure a manual process exists to monitor and prioritise fixes for detected vulnerabilities.

The Digital Marketplace team uses Snyk container vulnerability management tooling (https://snyk.io/product/container-vulnerability-management/) to regularly scan base images.

Minimise what you need to monitor

Minimise the things you need to monitor by minimising dependencies where possible. For example, if running containers, make sure you pick the smallest base images.

Also consider managed solutions where possible. For example:

- Use GOV.UK PaaS buildpacks (https://docs.cloud.service.gov.uk/deploying_apps.html#buildpacks) so you do not have to monitor for Operating System (OS), runtime or programming language vulnerabilities
- Consider managed container orchestration technology such as AWS Fargate (https://aws.amazon.com/fargate/) or similar so you do not have to monitor for vulnerabilities in your OS or control plane

This page was set to be reviewed before 27 December 2024 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way). This might mean the content is out of date.

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Understand the risks to your service

When you build, maintain or change your service, you must have a clear understanding of any associated risks because they will impact your service design and affect your users.

You should work with 🔒 GDS Information Security IA (https://sites.google.com/a/digital.cabinet-office.gov.uk/gds/directorates-and-groups/cto-and-ciso-office/information-security) to design appropriate solutions for your service's risks. IA may need to obtain risk acceptance from your Senior Risk Owner (SRO). You can also work with the 🔒 COD Cyber Security Team (https://intranet.cabinetoffice.gov.uk/it-data-and-security/cyber-and-information-security-services/) to get advice on the threats applicable to your service, and how to best mitigate them.

The Service Manual has some recommendations which can reduce risk to your service, for example, how to:

- protect against fraud (https://www.gov.uk/service-manual/technology/protecting-your-service-against-fraud) when you design and manage your service
- secure your information (https://www.gov.uk/service-manual/technology/securing-your-information) if you handle 'official' classified data

The government security hub security.gov.uk (https://www.security.gov.uk/) provides links to the policies and standards that we have to follow.

Model security threats

Modelling threats can help you gain a clearer understanding of threats against your service, see threat modelling (/standards/threat-modelling.html).

Further Reading

The National Cyber Security Centre (NCSC) (https://www.ncsc.gov.uk/) provides guidance about cyber security. The Service Manual has advice about securing your information (https://www.gov.uk/service-manual/technology/securing-your-information) and securing your

cloud environment (https://www.gov.uk/service-manual/technology/securing-your-cloud-environment).

This page was last reviewed on 3 October 2024. It needs to be reviewed again on 3 April 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

How GDS provides user support

GDS teams follow Service Manual guidance on setting up and managing user support (https://www.gov.uk/service-manual/helping-people-to-use-your-service/set-up-and-manage-user-support).

GDS teams operate support models that are aligned to the needs of the users that they are supporting. For example, the GOV.UK Pay (https://www.payments.service.gov.uk/) team supports users working in government departments that are using GOV.UK Pay or considering doing so. GOV.UK (https://www.gov.uk/) and One Login (https://www.sign-in.service.gov.uk/) support both government and non-government users.

Support lines for your service

However the support model varies, teams will have multiple 'lines' of support.

1st line user support

1st line support provide resolution for basic user enquiries and triage frequent, repeatable incidents. For example, issues signing in.

The GDS User Support team provides 1st line support for many teams in GDS. Contact the User Support team using the 🔒 #user-support Slack channel (https://gds.slack.com/archives/CADFJBDQU).

2nd line technical support

2nd line support provide direct incident resolution (see incident management (/standards/incident-management.html)) and triage user requests. For example, when users receive timeout errors or performance issues.

3rd line product teams

3rd line support are technical specialists associated with an existing product team. Their involvement in incident management is limited to the diagnosis and resolution of more

complex incidents, such as a data or security breach, or a complete service outage.

Document routes to support

Document the routes into each support line for your service. For example, users might contact 1st line support using an online form or by email. 2nd line support respond to issues raised through 1st line support and also alerts from failed tests, monitoring and vulnerability scans.

Understanding your service's routes into support will make sure support and product teams monitor the relevant channels. It will also make sure your service users and team understand how to reach the level of support they need.

Support hours and support rotas

Document and share your support team's operating hours. Many GDS teams operate a rota for their 2nd line support and escalation contacts.

Zendesk

Most GDS teams use Zendesk to manage communications with service users. The User Support team work with GDS teams to help manage Zendesk setup and configuration, for example with:

- adding and removing Zendesk accounts
- choosing agent types, groups and 'pass-through' to other government departments
- General Data Protection Regulation (GDPR) (https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/) considerations, such as data disclosure and data retention
- general best practice guidance

For help using Zendesk in GDS, contact the User Support team using the 🔒 #user-support Slack channel (https://gds.slack.com/archives/CADFJBDQU)

This page was last reviewed on 19 November 2024. It needs to be reviewed again on 19 May 2025 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Vulnerability Disclosure and security.txt

Vulnerability Disclosure

The 🔒 Cabinet Office Cyber Security team (https://intranet.cabinetoffice.gov.uk/it-data-and-security/cyber-and-information-security-services/) runs a vulnerability disclosure programme with HackerOne (https://www.hackerone.com) and NCC Group (https://www.nccgroup.com) to triage reports from security researchers. This is not a sign post for security researchers to 'hack' our systems; we advocate secure disclosure so we can find out about issues and fix them before they cause a security incident.

The public security policy is here: https://www.gov.uk/help/report-vulnerability (https://www.gov.uk/help/report-vulnerability)

GDS services are within scope of this programme and should participate by:

- publishing a `security.txt`
- having a plan for how you would respond to a vulnerability notification (triage, escalation, etc.).

security.txt

A `security.txt` file is a way of telling researchers how to get in contact with us. As per the current policy, we only accept reports from services that have a `security.txt` file pointing to the security policy (https://www.gov.uk/help/report-vulnerability).

We have a central deployment of the `security.txt` file so that we only have to keep one place up to date. The public alphagov/security.txt (https://github.com/alphagov/security.txt) repo is where it's maintained.

You should use https://vdp.cabinetoffice.gov.uk/.well-known/security.txt (https://vdp.cabinetoffice.gov.uk/.well-known/security.txt) in either:

- the origin for your site's `/.well-known/security.txt`
- the destination of a redirect for `/.well-known/security.txt`

Try implementing in the following order to ensure the best capability with all user agents:

1. Server-side redirect (302 status and `Location` header in response)
2. Client-side HTML (meta `http-equiv=refresh` tag in the head)
3. Client-side JavaScript redirect (`window.location.href`) - this won't work if JavaScript is disabled, so you should display a link as well

As well as `/.well-known/security.txt` you may optionally configure `/security.txt` .

We do not recommend hosting the security.txt file yourself, but if you are hosting it yourself, you should host at `/.well-known/security.txt` and optionally `/security.txt` . You should use a `text/plain` content type and follow the current security.txt guidance (https://github.com/securitytxt/security-txt).

thanks.txt

The central `security.txt` file contains an acknowledgements page, which is used for thanking researchers for valid reports. The page is a simple text file and is hosted at: https://vdp.cabinetoffice.gov.uk/thanks.txt (https://vdp.cabinetoffice.gov.uk/thanks.txt)

The `thanks.txt` file is also maintained in the alphagov/security.txt (https://github.com/alphagov/security.txt) repo.

If your vulnerability report comes to the Cyber Security team, the team will engage with the researcher and ask if they would like to be added to the page.

If you receive and manage a report directly and want to acknowledge a researcher, check with them first and ask which name they wish to have displayed.

This page was last reviewed on 18 February 2025. It needs to be reviewed again on 18 February 2026 by the page owner 🔒 #gds-way (https://gds.slack.com/messages/gds-way) .

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

Use a web application firewall (WAF)

A web application firewall (WAF) (https://owasp.org/www-community/Web_Application_Firewall) is an application layer protection for bi-directional web-based traffic. With a WAF, you can track web traffic and use specific tools to configure access control for your web content. Doing this improves your service's security monitoring and security position.

Why you should use a WAF

Your continuous integration (CI) and continuous deployment (CD) pipelines should include security tests in their workflows to identify any common vulnerabilities in your code. Some common vulnerabilities like Cross-site Scripting (XSS) (https://owasp.org/www-community/attacks/xss/) and XML command injection attacks (https://wiki.owasp.org/index.php/Testing_for_XML_Injection_(OTG-INPVAL-008)) are still possible in your production environments due to human error.

Combining a WAF with CI and CD tools reduces the risk of these attacks being successful, and provides enhanced layered security coverage for your service.

You may also need to use a WAF because of:

- GDS or departmental policies or standards
- HMG Standards such as the Cyber Assessment Framework (https://www.ncsc.gov.uk/collection/cyber-assessment-framework/introduction-to-caf) (NCSC) or Secure by Design Principles (https://www.security.gov.uk/guidance/secure-by-design/) (CDDO)
- 🔒 Information Security (https://sites.google.com/a/digital.cabinet-office.gov.uk/gds/directorates-and-groups/cto-and-ciso-office/information-security) requirements to minimise risk
- Payment Card Industry Data Security Standard (PCI DSS) (https://www.pcisecuritystandards.org/) compliance
- they are generally considered basic practice for protecting public web applications

When and how to use a WAF

Set up a baseline of tests in your project's alpha phase to identify any security vulnerabilities. As your service's features grow, extend your tests to cover new vulnerabilities you identify. For example, through exercises like application threat modelling (/standards/threat-modelling.html)

Good development practices (https://www.ncsc.gov.uk/collection/developers-collection) should detect and fix common vulnerabilities before they reach production environments. Use your WAF to track digital service vulnerabilities an attacker could exploit.

You should:

- have an independent security audit in place
- use established logging techniques (/standards/logging.html)
- encrypt data at rest as well as in transit
- subscribe to and apply security patches
- use query variables instead of plain text (stored procedure) to prevent SQL injections (https://owasp.org/www-community/attacks/SQL_Injection)

You should monitor the Open Web Application Security Project (OWASP) top 10 most critical web application vulnerabilities (https://owasp.org/www-project-top-ten/) to keep up to date with the latest threats.

Using a WAF should align with your other security monitoring features, like Security Information Event Management (SIEM) (https://en.wikipedia.org/wiki/Security_information_and_event_management). When developing use cases you should also factor in the extra time and resources needed to configure WAF rules.

Managing your WAF

Identify any areas in your app not covered by your WAF and define measures to protect them, such as using:

- alerts - as part of your incident management strategy
- threat modelling (/standards/threat-modelling.html) - to assess potential weaknesses in your environment
- reviews - to manage your security controls

Alerts

It's not always possible to block a detected attack because some services need to process transactions for any user of that service. In these situations you should raise events as alerts (/standards/alerting.html).

When WAF alerts are raised, make sure you already have an incident policy in place, including:

- who should manage an alert
- what their responsibilities are
- how to investigate an alert
- how to tune out false positives

Reviews

Review your WAF after each application change against the risks in the OWASP top 10 category rules.

This should be similar to how you use an IT Health Check (ITHC) (/standards/how-to-do-penetration-tests.html) to test and con□rm the effectiveness of security controls in your environment.

Case study GOV.UK PaaS

A GOV.UK PaaS (https://www.cloud.service.gov.uk/) tenant uses a pattern with Amazon Web Services (AWS) WAF (https://docs.aws.ama□ on.com/waf/latest/developerguide/waf-chapter.html) before forwarding traf□c to their apps with enabled shield advance (/manuals/security-overview-for-websites.html#12-aws-shield-response-team) for extra protection.

For more information read the proposed architecture for implementing a DDoS-resistant Website using AWS Services (https://docs.aws.ama□ on.com/waf/latest/developerguide/tutorials-ddos-cross-service.html).

Case study GOV.UK Pay

GOV.UK Pay (https://www.payments.service.gov.uk/) uses a NAXSI WAF (pay-nginx-proxy) (https://github.com/alphagov/pay-nginx-proxy) for its Nginx, forked from the Home Of□ce.

GOV.UK Pay operates under the governance of PCI compliance and DSS point 6.6 (https://www.pcisecuritystandards.org/pdfs/infosupp_6_6_application□rewalls_codereviews.pdf) which states the need for web application scanning.

Contact GDS Information Security or CO:D Cyber Security

Contact GDS 🔒 Information Security (https://sites.google.com/a/digital.cabinet-of□ce.gov.uk/gds/directorates-and-groups/cto-and-ciso-of□ce/information-security) or the security

architects in the CO:D Cyber Security team (https://intranet.cabinetoffice.gov.uk/it-data-and-security/cyber-and-information-security-services/threat-modelling/) or use the #cyber-security-help Slack channel (https://gds.slack.com/messages/CCMPJKFDK/) for help and advice.

This page was set to be reviewed before 24 January 2025 by the page owner #gds-way (https://gds.slack.com/messages/gds-way). This might mean the content is out of date.

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright

The GDS Way and its content is intended for internal use by the GDS community.

The GDS Way

The GDS Way guides teams to build and operate brilliant, cost-effective digital services.

It documents the specific technology, tools and processes that Government Digital Service (GDS) teams use.

It's not intended as guidance for anyone working outside GDS (though some other Cabinet Office teams use it too) - you'll find that in the Service Manual (https://www.gov.uk/service-manual).

About The GDS Way

The GDS Way shares agreed ways of working so service teams benefit from:

- using similar tools
- central procurement
- costs savings as new teams will be cheaper to spin up

The GDS Way makes it easier for projects to get started while still giving teams flexibility to do something different if their project needs it.

The GDS Way includes consistent:

- terminology
- ways of working
- technology and tools
- measures

All decisions are made in alignment with Service Manual (https://www.gov.uk/service-manual), which covers service design more broadly, and the Technology Code of Practice (https://www.gov.uk/guidance/the-technology-code-of-practice).

Products at GDS in discovery or alpha development phases must follow agile delivery principles (https://www.gov.uk/service-manual/agile-delivery) and also have the option to follow

the standards in this repository.

Products in beta and live phases must follow both the instructions set out in the Service Manual and the standards in this repository. They must be secure by design (https://www.security.gov.uk/guidance/secure-by-design/).

How to add new guidance

Contribute to this repository by making a pull request in GitHub (https://github.com/alphagov/gds-way) for discussion at the GDS Way Forum.

You can also read the service manual to find out about learning about and writing user needs (https://www.gov.uk/service-manual/user-research/start-by-learning-user-needs).

Thank you for your contributions as we develop this repository.

Submission template

When you create a new Markdown file follow this pattern and then make a pull request:

```
---
title: Thing you're writing a standard about
last_reviewed_on: yyyy-mm-dd
review_in: 6 months
---

# <%= current_page.data.title %>

Introduction of a couple of paragraphs to explain why the thing you're
writing a standard about is important.

## User needs

Why do we do this thing? Who is it helping?

## Principles

What broad approaches do we follow when we do this thing?

## Tools
```

```
What specific bits of software (commercial or open source) do
we use to help us do this thing?
```

The GDS Way Forum

This site documents some of the decisions agreed at the GDS Way Forum about the products we operate.

The GDS Forum meets once a month. The Forum is lead developers and technical architects representing GDS programmes who are responsible for communicating and implementing the GDS Way.

The Forum reviews:

- all GDS Way open and closed PRs
- expired guidance and if it should be continued
- possible subject areas and ownership of new content

Contact The GDS Way Forum

Contact the GDS Way Forum using the 🔒 #gds-way Slack channel (https://gds.slack.com/messages/gds-way/) or by email at the-gds-way@digital.cabinet-office.gov.uk.

Software development

- How to name software products (/standards/naming-software-products.html)
- Choosing a programming language (/standards/programming-languages.html)
- Style guides (/manuals/programming-languages.html)
- How to manage third party software dependencies (/standards/tracking-dependencies.html)
- Building accessible services (/manuals/accessibility.html)
- Pair programming (/standards/pair-programming.html)
- Test-driven Development (TDD) (/standards/test-driven-development.html)
- Breaking down work (/standards/breaking-down-work.html)
- Supporting different browsers (/manuals/browser-support.html)
- How to optimise frontend performance (/standards/optimise-frontend-perf.html)
- Documenting architecture decisions (/standards/architecture-decisions.html)
- Diagrams as code (/standards/diagrams-as-code.html)
- Anticipate security issues using threat modelling (/standards/threat-modelling.html)

Version control and deployments

- How to store source code (/standards/source-code/index.html)
- Using Pull Requests (/standards/pull-requests.html)
- Writing READMEs (/manuals/readme-guidance.html)
- Writing release notes (/manuals/release-notes.html)
- Licensing (/manuals/licensing.html)
- Use continuous delivery (/standards/continuous-delivery.html)

Hosting and infrastructure

- How to host a service (/standards/hosting.html)
- How to manage DNS records for your service (/standards/dns-hosting.html)
- Use configuration management (/standards/con□guration-management.html)
- Operating systems for virtual machines (/standards/operating-systems.html)
- Use a web application firewall (WAF) (/standards/web-application-□rewall.html)
- Security overview for websites (/manuals/security-overview-for-websites.html)
- Tagging AWS resources (/manuals/aws-tagging.html)
- Working with AWS accounts (/manuals/working-with-aws-accounts.html)

Logging, monitoring and alerting

- Store and query logs (/standards/logging.html)
- How to monitor your service (/standards/monitoring.html)
- How to manage alerts (/standards/alerting.html)
- Make data-driven decisions with Service Level Objectives (SLOs) (/standards/slo.html)
- Run a Service Level Indicator (SLI) workshop (/standards/slis.html)

Operating a service

- Understand the risks to your service (/standards/understanding-risks.html)
- How to track technical debt (/standards/technical-debt.html)
- How to manage access to your third-party service accounts (/standards/accounts-with-third-parties.html)
- How to store credentials (/standards/storing-credentials.html)
- How to do penetration testing (/standards/how-to-do-penetration-tests.html)

- Performance testing (/standards/performance-testing.html)
- How to send email notifications (/standards/sending-email.html)
- How GDS provides user support (/standards/user-support.html)
- Incident management (/standards/incident-management.html)
- Disaster Recovery (/standards/disaster-recovery.html)
- Principle of least privilege (/standards/principle-least-access.html)
- Tracking access control (/standards/secrets-acl.html)
- Secret auditing (/standards/secrets-auditing.html)
- Vulnerability disclosure and security.txt (/standards/vulnerability-disclosure.html)

View source Report problem GitHub Repo

Accessibility

OGL
All content is available under the Open Government Licence v3.0, except where otherwise stated

© Crown copyright