bedrock Documentation

Release 1.0

Mozilla

Mar 25, 2025

CONTENTS

1 Contents **3**

1.1 Installing Bedrock . . . 3
1.2 Localization . . . 13
1.3 Developing on Bedrock . . . 28
1.4 How to contribute . . . 47
1.5 Continuous Integration & Deployment . . . 51
1.6 Front-end testing . . . 55
1.7 Managing Redirects . . . 62
1.8 Newsletters . . . 66
1.9 Contentful CMS (Content Management System) Integration (Deprecated) . . . 68
1.10 CMS . . . 82
1.11 Sitemaps . . . 94
1.12 Legal Docs . . . 94
1.13 Using External Content Cards Data . . . 96
1.14 Banners . . . 97
1.15 Mozilla.UITour . . . 98
1.16 Task queue . . . 99
1.17 Send to Device Widget . . . 99
1.18 Firefox Download Buttons . . . 101
1.19 Mozilla accounts helpers . . . 101
1.20 A/B Testing . . . 107
1.21 Mozilla VPN Subscriptions . . . 110
1.22 Attribution . . . 113
1.23 Architectural Decision Records . . . 135
1.24 Browser Support . . . 146

bedrock is the project behind www.mozilla.org. It is as shiny, awesome, and open-source as always. Perhaps even a little more.

bedrock is a web application based on Django, a Python web application framework.

Patches are welcome! Feel free to fork and contribute to this project on Github.

CHAPTER

ONE

CONTENTS

1.1 Installing Bedrock

1.1.1 Installation Methods

There are two primary methods of installing bedrock: Docker and Local. Whichever you choose, you'll start by getting the source.

The codebase lives at https://github.com/mozilla/bedrock/

Only Mozilla staff have write access to that repository; community contributors do not, so should instead make a fork of the repo to work from. You will still be able to make pull requests from your fork into `mozilla/bedrock`.

Get the source code:

```
# If you're a Mozilla staff member with write access to the repo
$ git clone https://github.com/mozilla/bedrock.git

# Or if you lack write access to the repo
$ git clone https://github.com/YOUR_GITHUB_USERNAME_HERE/bedrock.git
```

Once the codebase is cloned, switch into it:

```
$ cd bedrock
```

After these basic steps you can choose your install method below.

Docker is the easiest and recommended way, but local installation directly onto your machine is also possible and may be preferred, particularly if you're doing frontend work, which is currently slower when using Docker.

Note: You should also install our git pre-commit hooks. These are checks that automatically run before a git commit is allowed. You don't have to do this in order to get bedrock running locally, but it's recommended to do before you start making contributions.

The Bedrock project uses the pre-commit framework that makes managing git hooks easier across all contributors by ensuring everyone has the same ones set up.

Install the framework by running `pip install pre-commit`, then - ensuring you are in your `bedrock` directory - run `pre-commit install` in your terminal, followed by `pre-commit install-hooks`. This will set up the hooks that are specified in `bedrock/.precommit.yaml`

After that setup, whenever you try to make a commit, the 'hooks' will check/lint your Python, JS, and CSS files beforehand and report on problems that need to be fixed before the commit can be made. This will save you time waiting for

the tests to run in our CI (Continuous Integration) before noticing a linting error.

Docker Installation

Note: This method assumes you have Docker installed for your platform. If not please do that now or skip to the `Local Installation` section.

This is the simplest way to get started developing for bedrock. If you're on Linux or Mac (and possibly Windows 10 with the Linux subsystem) you can run a script that will pull our production and development docker images and start them:

```
$ make clean run
```

Note: You can start the server any other time with:

```
$ make run
```

You should see a number of things happening, but when it's done it will output something saying that the server is running at localhost:8000. Go to that URL in a browser and you should see the mozilla.org home page. In this mode the site will refresh itself when you make changes to any template or media file. Simply open your editor of choice and modify things and you should see those changes reflected in your browser.

Note: It's a good idea to run `make pull` from time to time. This will pull down the latest Docker images from our repository ensuring that you have the latest dependencies installed among other things. If you see any strange errors after a `git pull` then `make pull` is a good thing to try for a quick fix.

If you don't have or want to use Make you can call the docker and compose commands directly

```
$ docker compose pull
```

```
$ [[ ! -f .env ]] && cp .env-dist .env
```

Then starting it all is simply

```
$ docker compose up app assets
```

All of this is handled by the `Makefile` script and called by Make if you follow the above directions. You **DO NOT** need to do both.

These directions pull and use the pre-built images that our deployment process has pushed to the Docker Hub. If you need to add or change any dependencies for Python or Node then you'll need to build new images for local testing. You can do this by updating the requirements files and/or package.json file then simply running:

```
$ make build
```

Note: For Apple Silicon / M1 users

If you find that when you're building you hit issues compiling assets, try unchecking `Use Rosetta for x86_64/amd64 emulation on Apple Silicon` in the Docker Desktop settings.

Asset bundles

If you make a change to `media/static-bundles.json`, you'll need to restart Docker.

Note: Sometimes stopping Docker doesn't actually kill the images. To be safe, after stopping docker, run `docker ps` to ensure the containers were actually stopped. If they have not been stopped, you can force them by running `docker compose kill` to stop all containers, or `docker kill <container_name>` to stop a single container, e.g. `docker kill bedrock_app_1`.

Local Installation

These instructions assume you have Python, pip, and NodeJS installed. If you don't have *pip* installed (you probably do) you can install it with the instructions in the pip docs.

Bedrock currently uses Python 3.12.x. The recommended way to install and use that version is with pyenv and to create a virtualenv using pyenv-virtualenv that will isolate Bedrock's dependencies from other things installed on the system.

The following assumes you are on MacOS, using `zsh` as your shell and Homebrew as your package manager. If you are not, there are installation instructions for a variety of platforms and shells in the READMEs for the two pyenv projects.

Install Python 3.12.x with pyenv

1. Install `pyenv` itself

```
$ brew install pyenv
```

2. Configure your shell to init `pyenv` on start - this is noted in the project's own docs, in more detail, but omits that setting *PYENV_ROOT* and adding it to the path is needed:

```
$ echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.zshrc
$ echo 'export PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.zshrc
$ echo 'eval "$(pyenv init -)"' >> ~/.zshrc
```

3. Restart your login session for the changes to profile files to take effect - if you're not using `zsh`, the `pyenv` docs have other routes

```
$ zsh -l
```

4. Install the latest Python 3.12.x (e.g. 3.12.6), then test it's there:

```
$ pyenv install 3.12.6
```

If you'd like to make Python 3.12 your default globally, you can do so with:

```
$ pyenv global 3.12.6
```

If you only want to make Python 3.12 available in the current shell, while you set up the Python virtualenv (below), you can do so with:

```
$ pyenv shell 3.12.6
```

5. Verify that you have the correct version of Python installed:

```
$ python --version
Python 3.12.6
```

Install a plugin to manage virtualenvs via pyenv and create a virtualenv for Bedrock's dependencies

1. Install `pyenv-virtualenv`

```
$ brew install pyenv-virtualenv
```

2. Configure your shell to init `pyenv-virtualenv` on start - again, this is noted in the `pyenv-virtualenv` project's own documentation, in more detail. The following will slot in a command that will work as long as you have pyenv-virtualenv installed:

```
$ echo 'eval "$(pyenv virtualenv-init -)"' >> ~/.zshrc
```

3. Restart your login session for the changes to profile files to take effect

```
$ zsh -l
```

4. Make a virtualenv we can use - in this example we'll call it `bedrock` but use whatever you want

```
$ pyenv virtualenv 3.12.6 bedrock
```

Use the virtualenv

1. Switch to the virtualenv - this is the command you will use any time you need this virtualenv

```
$ pyenv activate bedrock
```

2. If you'd like to auto activate the virtualenv when you cd into the bedrock directory, and deactivate it when you exit the directory, you can do so with:

```
$ echo 'bedrock' > .python-version
```

3. Securely upgrade pip

```
$ pip install --upgrade pip
```

4. Install / update Python dependencies

```
$ make install-local-python-deps
```

Note: If you are on OSX and some of the compiled dependencies fails to compile, try explicitly setting the arch flags and try again. The following are relevant to Intel Macs only. If you're on Apple Silicon, 3.12.6 should 'just work':

```
$ export ARCHFLAGS="-arch i386 -arch x86_64"
```

```
$ make install-local-python-deps
```

If you are on Linux, you may need at least the following packages or their equivalent for your distro:

```
python3-dev libxslt-dev
```

Download a fresh copy of the sqlite database that Bedrock uses locally This contains product-details, security-advisories, credits, release notes, localizations, legal-docs etc. We also download the latest translations of site content in many languages:

```
$ bin/bootstrap.sh
```

Install the node dependencies to run the site. This will only work if you already have Node.js and npm installed:

```
$ npm install
```

Note: Bedrock uses npm to ensure that Node.js packages that get installed are the exact ones we meant (similar to pip hash checking mode for python). Refer to the npm documentation for adding or upgrading Node.js dependencies.

Note: As a convenience, there is a `make preflight` command which calls some of the commands above to bring your installed Python and NPM dependencies up to date and also fetches the latest DB containing the latest site content. This is a good thing to run after pulling in latest changes from the `main` branch.

IMPORTANT: if you do not want to replace your local DB with a fresher one, use `make preflight -- --retain-db` instead.

We also have an optional git hook that will alert you if `make preflight` needs to be run. You can install that with `make install-custom-git-hooks`.

1.1.2 Run the tests

Now that we have everything installed, let's make sure all of our tests pass. This will be important during development so that you can easily know when you've broken something with a change.

Docker

We manage our local docker environment with docker compose and Make. All you need to do here is run:

```
$ make test
```

If you don't have Make you can simply run `docker compose run test`.

If you'd like to run only a subset of the tests or only one of the test commands you can accomplish that with a command like the following:

```
$ docker compose run test pytest bedrock/firefox
```

This example will run only the unit tests for the `firefox` app in bedrock. You can substitute `pytest bedrock/firefox` with most any shell command you'd like and it will run in the Docker container and show you the output. You can also just run `bash` to get an interactive shell in the container which you can then use to run any commands you'd like and inspect the file system:

```
$ docker compose run test bash
```

Local

From the local install instructions above you should still have your virtualenv activated, so running the tests is as simple as:

```
$ pytest lib bedrock
```

To test a single app, specify the app by name in the command above. e.g.:

```
$ pytest bedrock/firefox
```

1.1.3 Run a local server

Attention: Regardless of whether you run Bedrock via Docker or directly on your machine, the URL of the site is `http://localhost:8000` - *not* `8080`

Docker

You can simply run the `make run` script mentioned above, or use docker compose directly:

```
$ docker compose up app assets
```

Local

To make the server run, make sure your virtualenv is activated with `pyenv activate bedrock`, and then run the server:

```
$ npm start
```

Wait for the server to start up and then browse to http://localhost:8000

Congratulations, you should now have your own copy of www.mozilla.org running locally!

Prod Mode

There are certain things about the site that behave differently when running locally in dev mode using Django's development server than they do when running in the way it runs in production. Static assets that work fine locally can be a problem in production if referenced improperly, and the normal error pages won't work unless `DEBUG=False` and doing that will make the site throw errors since the Django server doesn't have access to all of the built static assets. So we have a couple of extra Docker commands (via make) that you can use to run the site locally in a more prod-like way.

First you should ensure that your `.env` file is setup the way you need. This usually means adding `DEBUG=False` and `DEV=False`, though you may want `DEV=True` if you want the site to act more like www-dev.allizom.org in that all feature switches are `On` and all locales are active for every page. After that you can run the following:

```
$ make run-prod
```

This will run the latest bedrock image using your local bedrock files and templates, but not your local static assets. If you need an updated image just run `make pull`.

If you need to include the changes you've made to your local static files (images, css, js, etc.) then you have to build the image first:

```
$ make build-prod run-prod
```

Documentation

This is a great place for coders and non-coders alike to contribute! Please note most of the documentation is currently in reStructuredText but we also support Markdown files.

If you see a typo or similarly small change, you can use the "Edit in GitHub" link to propose a fix through GitHub. Note: you will not see your change directly committed to the main branch. You will commit the change to a separate branch so it can be reviewed by a staff member before merging to main.

If you want to make a bigger change or find a Documentation issue on the repo, it is best to edit and preview locally before submitting a pull request. You can do this with Docker or Local installations. Run the commands from your root folder. They will build documentation and start a live server to auto-update any changes you make to a documentation file.

Docker:

```
$ make docs
```

Local:

```
$ pip install -r requirements/docs.txt
```

```
$ make livedocs
```

1.1.4 Localization

Localization (or L10n) files were fetched by the *bootstrap.sh* command your ran earlier and are included in the docker images. If you need to update them or switch to a different repo or branch after changing settings you can run the following command:

```
$ ./manage.py l10n_update
```

You can read more details about how to localize content *here*.

1.1.5 Feature Flipping (aka Switches, or waffle switches)

Switches are managed using django-waffle and are stored in the database. These switches control behavior and/or features of select pages on Bedrock, and their state (active or inactive) is based on an `active` boolean field in the database.

Defining and Using Switches

The `switch()` template helper function allows you to check whether a specific switch is active. You pass a name to the function (using only letters, numbers, and dashes), which is automatically converted to uppercase and with dashes replaced by underscores for the lookup in the database. For example, `switch('the-dude')` will look for a switch named `THE_DUDE` in the database.

Locale-Specific Switches

You can provide a list of locale codes to limit the switch's activation to specific locales. If the page is viewed in a locale not included in the list, the switch will return False. You can also use "Locale Groups," which apply to all locales with a common prefix (e.g., "en-US, en-GB" or "zh-CN, zh-TW"). To use these groups, pass the prefix. For example, `switch('the-dude', ['en', 'de'])` will activate the switch for German and any English locale supported by the site.

Managing Switches

Switches are managed through the Django Admin interface, where you can add, edit, or remove switches from the database directly. This interface allows for easy management of feature toggles without modifying environment variables or code. There is also a Django management command to toggle switches from the command line, as detailed below.

Deploy switches with code

You can deploy switches directly through code by creating a data migration. This approach ensures switches are consistently created or updated during deployment, rather than requiring manual configuration through the Django Admin interface.

To implement a switch via data migration, create an empty migration file:

```
./manage.py makemigrations base --empty
```

Then add the following code to the generated migration file, which can be found in the `bedrock/base/migrations` directory:

```
from django.db import migrations

from waffle.models import Switch

# The name of the switch must be unique.
SWITCH_NAME = "RELEASE_THE_KRAKEN"


def create_switch(apps, schema_editor):
    Switch.objects.get_or_create(
        name=SWITCH_NAME,
        defaults={"active": True},  # Set initial state, True or False.
    )


def remove_switch(apps, schema_editor):
    Switch.objects.filter(name=SWITCH_NAME).delete()
```

(continues on next page)

(continued from previous page)

```
class Migration(migrations.Migration):
    dependencies = [
        (
            "base",
            "0001_initial",
        ),  # Keep whatever the makemigrations command generated here.
    ]

    operations = [
        migrations.RunPython(create_switch, remove_switch),
    ]
```

The migration will run during deployment and ensure the switch exists in the database. The `remove_switch` function allows the migration to be reversed if needed.

To test this locally, run the following command:

```
./manage.py migrate base
```

Verify the switch exists in the database by running:

```
./manage.py waffle_switch -l
```

You should see the switch listed in the output.

To test reversing the migration, run the following command but replace `0001` with whatever the previous migration number is:

```
./manage.py migrate base 0001
```

Example Usage in Templates

You can use the `switch()` helper function in your templates as follows:

```
{% if switch('the-dude') %}
    <!-- Feature-specific HTML goes here -->
{% endif %}
```

Example Usage in Python

You may also use switches in Python code (though locale support is unavailable in this context):

Note: **Avoid using switch() outside the request/response cycle** (e.g., during module-level imports or in a urls.py file), as the switch's state is managed in the database and can be changed via the admin interface. Using it outside the request cycle would prevent the switch value from reflecting real-time updates.

```
from bedrock.base.waffle import switch
```

(continues on next page)

(continued from previous page)

```
def home_view(request):
    title = "Staging Home" if switch("staging-site") else "Prod Home"
    ...
```

Testing

If the environment variable `DEV` is set to a "true" value, then all switches will be considered active unless they are explicitly set as not active in the database. `DEV` defaults to "true" in local development and demo servers.

To test switches locally, add the switch to the database. This can be done in one of two ways.

1. Add the switch via the Django management command:

   ```
   ./manage.py waffle_switch --create SWITCH_NAME on
   ```

   If the switch already exists, you can toggle it using:

   ```
   ./manage.py waffle_switch SWITCH_NAME on
   ./manage.py waffle_switch SWITCH_NAME off
   ```

   And you can view all the switches via:

   ```
   ./manage.py waffle_switch -l
   ```

   To delete a switch, run:

   ```
   ./manage.py waffle_delete --switches SWITCH_NAME
   ```

2. Add the switch in the Django admin at `/django-admin/`. There you will see the "Django-Waffle" module with the "Switches" table. Click through to view the switches and add/edit/delete as needed.

Traffic Cop

Currently, these switches are used to enable/disable Traffic Cop experiments on many pages of the site. We only add the Traffic Cop JavaScript snippet to a page when there is an active test.

To work with/test these experiment switches locally, you must add the switches to your local database.

Notes

A shortcut for activating virtual envs in zsh or bash is . *venv/bin/activate*. The dot is the same as *source*.

There's a project called pew that provides a better interface for managing/activating virtual envs, so you can use that if you want. Also if you need help managing various versions of Python on your system, the pyenv project can help.

1.2 Localization

Note: This document focuses on localization of hard-coded content. For localization of CMS-based content please see the CMS-specific L10N docs.

The site is fully localizable. Localization files are not shipped with the code distribution, but are available in separate GitHub repositories. The proper repos can be cloned and kept up-to-date using the `l10n_update` management command:

```
$ ./manage.py l10n_update
```

If you don't already have a `data/www-l10n` directory, this command will clone the git repo containing the .ftl translation files (either the dev or prod files depending on your `DEV` setting). If the folder is already present, it will update the repository to the latest version.

1.2.1 Fluent

Bedrock's Localization (l10n) system is based on Project Fluent. This is a departure from a standard Django project that relies on a gettext work flow of string extraction from template and code files, in that it relies on developers directly editing the default language (English in our case) Fluent files and using the string IDs created there in their templates and views.

The default files for the Fluent system live in the `l10n` directory in the root of the bedrock project. This directory houses directories for each locale the developers directly implement (mostly simplified English "en", and "en-US"). The simplified English files are the default fallback for every string ID on the site and should be strings that are plain and easy to understand English, as free from colloquialisms as possible. The translators are able to easily understand the meaning of the string, and can then add their own local flair to the ideas.

.ftl files

When adding translatable strings to the site you start by putting them all into an .ftl file in the `l10n/en/` directory with a path that matches or is somehow meaningful for the expected location of the template or view in which they'll be used. For example, strings for the `mozorg/mission.html` template would go into the `l10n/en/mozorg/mission.ftl` file. Locales are activated for a particular .ftl file, not template or URL, so you should use a unique file for most URLs, unless they're meant to be translated and activated for new locales simultaneously.

You can have shared .ftl files that you can load into any template render, but only the first .ftl file in the list of the ones for a page render will determine whether the page is active for a locale.

Activation of a locale happens automatically once certain rules are met. A developer can mark some string IDs as being "Required", which means that the file won't be activated for a locale until that locale has translated all of those required strings. The other rule is a percentage completion rule: a certain percentage (configurable) of the strings IDs in the "en" file must be translated in the file for a locale before it will be marked as active. We'll get into how exactly this works later.

Translating with .ftl files

The Fluent file syntax is well documented on the Fluent Project's site. We use "double hash" or "group" comments to indicate strings required for activation. A group comment only ends when another group comment starts however, so you should either group your required strings at the bottom of a file, or also have a "not required" group comment. Here's an example:

```
### File for example.html

## Required
example-page-title = The Page Title
# this is a note the applies only to example-page-desc
example-page-desc = This page is a test.

##
example-footer = This string isn't as important
```

Any group comment (a comment that starts with "##") that starts with "Required" (case does not matter) will start a required strings block, and any other group comment will end it.

Once you have your strings in your .ftl file you can place them in your template. We'll use the above .ftl file for a simple Jinja template example:

```
<!doctype html>
<html>
<head>
    <title>{{ ftl('example-page-title') }}</title>
</head>
<body>
    <h1>{{ ftl('example-page-title') }}</h1>
    <p>{{ ftl('example-page-desc') }}</p>
    <footer>
        <p>{{ ftl('example-footer') }}</p>
    </footer>
</body>
</html>
```

FTL (Fluent Translation List) String IDs

Our convention for string ID creation is the following:

1. String IDs should be all lower-case alphanumeric characters.
2. Words should be separated with hyphens.
3. IDs should be prefixed with the name of the template file (e.g. `firefox-new-skyline` for `firefox-new-skyline.html`)
4. If you need to create a new string for the same place on a page and to transition to it as it is translated, you can add a version suffix to the string ID: e.g. `firefox-new-skyline-main-page-title-v2`.
5. The ID should be as descriptive as possible to make sense to the developer, but could be anything as long as it adheres to the rules above.

Using brand names

Common Mozilla brand names are stored in a global brands.ftl file, in the form of terms. Terms are useful for keeping brand names separated from the rest of the translations, so that they can be managed in a consistent way across all translated files, and also updated easily in a global context. In general the brand names in this file remain in English and should not be translated, however locales still have the choice and control to make adjustments should it suit their particular language.

Only our own brands should be managed this way, brands from other companies should not. If you are concerned that the brand is a common word and may be translated, leave a comment for the translators.

Note: We are trying to phase out use of `{ -brand-name-firefox-browser }` please use `{ -brand-name-firefox } browser`.

```
-brand-name = Firefox

example-about = About { -brand-name }.
example-update-successful = { -brand-name } has been updated.
# "Safari" here refers to the competing web browser
example-compare = How does { -brand-name } compare to Safari?
```

Important: When adding a new term to `brands.ftl`, the new term should also be manually added to the mozilla-l10n/www-l10n repo for *all locales*. The reason for this is that if a term does not exist for a particular locale, then it does not fall back to English like a regular string does. Instead, the term variable name is displayed on the page.

Variables

Single hash comments are applied only to the string immediately following them. They should be used to provide additional context for the translators including:

1. What the values of variables are.
2. Context about where string appears on the page if it is not visible or references other elements on the page.
3. Explanations of English idioms and jargon that may be confusing to non-native speakers.

```
# Variables:
#   $savings (string) - the percentage saved from the regular price, not including the %␣
↪Examples: 50, 70
example-bundle-savings = Buy now for { $savings }% off.

# Context: Used as an accessible text alternative for an image
example-bookmark-manager-alt = The bookmark manager window in { -brand-name-firefox }.
# Context: This lists the various websites and magazines who have mentioned Firefox␣
↪Relay.
# Example: "As seen in: FORBES magainze and LifeHacker"
example-social-proof = As seen in:

example-privacy-on-every = Want privacy on every device?
# "You got it" here is a casual answer to the previous question, "Want privacy on every␣
```

(continues on next page)

(continued from previous page)

```
↪device?"
example-you-got-it = You got it. Get { -brand-name-firefox } for mobile.
```

HTML with attributes

When passing HTML tags with attributes into strings for translation, remove as much room for error as possible by putting all the attributes and their values in a single variable. (This is most common with links and their href attributes but we do occasionally pass classes with other tags.)

```
# Variables:
#   $attrs (attrs) - link to https://www.mozilla.org/about/
example-created = { -brand-name-firefox } was created by <a {$attrs}>{ -brand-name-
↪mozilla }</a>.

# Variables:
#   $class (string) - CSS class used to replace brand name with wordmark logo
example-firefox-relay = Add <span { $class }">{ -brand-name-firefox-relay }</span>
```

```
{% set created_attrs = 'href="%s" data-cta-text="created by Mozilla"'|safe|format(url(
↪'mozorg.about.index')) %}
<p>{{ ftl('example-created', attrs=created_attrs) }}</p>

{{ ftl('example-firefox-relay', class_name='class="mzp-c-wordmark mzp-t-wordmark-md mzp-
↪t-product-relay"') }}
```

Obsolete strings

When new strings are added to a page sometimes they update or replace old strings. Obsolete strings & IDs should be removed from ftl files immediately if they are not being used as a fallback. If they are being kept as a fallback they should be removed after 2 months.

When you add a comment marking a string as obsolete, add the date when it can be removed to the comment.

```
# Obsolete string (expires: 2024-03-18)
example-old-string = Fifty thousand year old twisted bark threads.
```

Fallback

If you need to create a new string for the same place on a page and would like to keep the old one as a fallback, you can add a version suffix to the new string ID: e.g. `firefox-new-skyline-main-page-title-v2`.

```
example-block-title-v2 = Security, reliability and speed - on every device, anywhere you␣
↪go.
# Obsolete string (expires: 2024-03-18)
example-block-title = Security, reliability and speed - from name you can trust.
```

The `ftl` helper function has the ability to accept a fallback string ID and is described in the next section.

Remove

If the new string is fundamentally different a new string ID should be created and the old one deleted.

For example, if the page is going from talking about the Google Translate extension to promoting our own Firefox Translate feature the old strings are not appropriate fall backs.

The old strings and IDs should be deleted:

```
example-translate-title = The To Google Translate extension makes translating the page␣
↪you're on easier than ever.
example-translate-content = Google Translate, with over 100 languages* at the ready, is␣
↪used by millions of people around the world.
```

The new strings should have different IDs and not be versioned:

```
example-translate-integrated-title = { -brand-name-firefox } now comes with an␣
↪integrated translation tool.
example-translate-integrated-content =  Unlike some cloud-based alternatives, { -brand-
↪name-firefox } translates text locally, so the content you're translating doesn't␣
↪leave your machine.
```

The `ftl_has_messages` jinja helper would be useful here and is described in the next section.

The `ftl` helper function

The `ftl()` function takes a string ID and returns the string in the current language, or simplified english if the string isn't translated. If you'd like to use a different string ID in the case that the primary one isn't translated you can specify that like this:

```
ftl("primary-string-id", fallback="fallback-string-id")
```

When a fallback is specified it will be used only if the primary isn't translated in the current locale. English locales (e.g. en-US, en-GB) will never use the fallback and will print the simplified english version of the primary string if not overridden in the more specific locale.

You can also pass in replacement variables into the `ftl()` function for use with fluent variables. If you had a variable in your fluent file like this:

```
welcome = Welcome, { $user }!
```

You could use that in a template like this:

```
<h2>{{ ftl('welcome', user='Dude') }}<h2>
```

For our purposes these are mostly useful for things that can change, but which shouldn't involve retranslation of a string (e.g. URLs or email addresses).

You may also request any other translation of the string (or the original English string of course) regardless of the current locale.

```
<h2>{{ ftl('welcome', locale='en', user='Dude') }}<h2>
```

This helper is available in Jinja templates and Python code in views. For use in a view you should always call it in the view itself:

```
# views.py
from lib.l10n_utils import render
from lib.l10n_utils.fluent import ftl


def about_view(request):
    ftl_files = "mozorg/about"
    hello_string = ftl("about-hello", ftl_files=ftl_files)
    render(request, "about.html", {"hello": hello_string}, ftl_files=ftl_files)
```

If you need to use this string in a view, but define it outside of the view itself, you can use the `ftl_lazy` variant which will delay evaluation until render time. This is mostly useful for defining messages shared among several views in constants in a `views.py` or `models.py` file.

Whether you use this function in a Python view or a Jinja template it will always use the default list of Fluent files defined in the `FLUENT_DEFAULT_FILES` setting. If you don't specify any additional Fluent files via the `fluent_files` keyword argument, then only those default files will be used.

The `ftl_has_messages` helper function

Another useful template tool is the `ftl_has_messages()` function. You pass it any number of string IDs and it will return `True` only if all of those message IDs exist in the current translation. This is useful when you want to add a new block of HTML to a page that is already translated, but don't want it to appear untranslated on any page.

```
{% if ftl_has_messages('new-title', 'new-description') %}
  <h3>{{ ftl('new-title') }}</h3>
  <p>{{ ftl('new-description') }}</p>
{% else %}
  <h3>{{ ftl('title') }}</h3>
  <p>{{ ftl('description') }}</p>
{% endif %}
```

If you'd like to have it return true when any of the given message IDs exist in the translation instead of requiring all of them, you can pass the optional `require_all=False` parameter and it will do just that.

There is a version of this function for use in views called `has_messages`. It works exactly the same way but is meant to be used in the view Python code.

```
# views.py
from lib.l10n_utils import render
from lib.l10n_utils.fluent import ftl, has_messages


def about_view(request):
    ftl_files = "mozorg/about"
    if has_messages("about-hello-v2", "about-title-v2", ftl_files=ftl_files):
        hello_string = ftl("about-hello-v2", ftl_files=ftl_files)
        title_string = ftl("about-title-v2", ftl_files=ftl_files)
    else:
        hello_string = ftl("about-hello", ftl_files=ftl_files)
        title_string = ftl("about-title", ftl_files=ftl_files)

    render(
```

(continues on next page)

(continued from previous page)

```
        request,
        "about.html",
        {"hello": hello_string, "title": title_string},
        ftl_files=ftl_files,
    )
```

Specifying Fluent files

You have to tell the system which Fluent files to use for a particular template or view. This is done in either the `page()` helper in a `urls.py` file, or in the call to `l10n_utils.render()` in a view.

Using the `page()` function

If you just need to render a template, which is quite common for bedrock, you will probably just add a line like the following to your `urls.py` file:

```
urlpatterns = [
    page("about", "about.html"),
    page("about/contact", "about/contact.html"),
]
```

To tell this page to use the Fluent framework for l10n you just need to tell it which file(s) to use:

```
urlpatterns = [
    page("about", "about.html", ftl_files="mozorg/about"),
    page(
        "about/contact",
        "about/contact.html",
        ftl_files=["mozorg/about/contact", "mozorg/about"],
    ),
]
```

The system uses the first (or only) file in the list to determine which locales are active for that URL. You can pass a string or list of strings to the `ftl_files` argument. The files you specify can include the `.ftl` extension or not, and they will be combined with the list of default files which contain strings for global elements like navigation and footer. There will also be files for reusable widgets like the newsletter form, but those should always come last in the list.

Using the class-based view

Bedrock includes a generic class-based view (CBV) that sets up l10n for you. If you need to do anything fancier than just render the page, then you can use this:

```
from lib.l10n_utils import L10nTemplateView


class AboutView(L10nTemplateView):
    template_name = "about.html"
    ftl_files = "mozorg/about"
```

Using that CBV will do the right things for l10n, and then you can override other useful methods (e.g. `get_context_data`) to do what you need. Also, if you do need to do anything fancy with the context, and you find that you need to dynamically set the fluent files list, you can easily do so by setting `ftl_files` in the context instead of the class attribute.

```
from lib.l10n_utils import L10nTemplateView


class AboutView(L10nTemplateView):
    template_name = "about.html"

    def get_context_data(self, **kwargs):
        ctx = super().get_context_data(**kwargs)
        ftl_files = ["mozorg/about"]
        if request.GET.get("fancy"):
            ftl_files.append("fancy")

        ctx["ftl_files"] = ftl_files
        return ctx
```

A common case is needing to use FTL files when one template is used, but not with another. In this case you would have some logic to decide which template to use in the `get_template_names()` method. You can set the `ftl_files_map` class variable to a dict containing a map of template names to the list of FTL files for that template (or a single file name if that's all you need).

```
# views.py
from lib.l10n_utils import L10nTemplateView


# class-based view example
class AboutView(L10nTemplateView):
    ftl_files_map = {"about_es.html": ["about_es"], "about_new.html": ["about"]}

    def get_template_names(self):
        if self.request.locale.startswith("en"):
            template_name = "about_new.html"
        elif self.request.locale.startswith("es"):
            template_name = "about_es.html"
        else:
            # FTL system not used
            template_name = "about.html"

        return [template_name]
```

If you need for your URL to use multiple Fluent files to determine the full list of active locales, for example when you are redesigning a page and have multiple templates in use for a single URL depending on locale, you can use the *activation_files* parameter. This should be a list of FTL filenames that should all be used when determining the full list of translations for the URL. Bedrock will gather the full list for each file and combine them into a single list so that the footer language switcher works properly.

Another common case is that you want to keep using an old template for locales that haven't yet translated the strings for a new one. In that case you can provide an `old_template_name` to the class and include both that template and `template_name` in the `ftl_files_map`. Once you do this the view will use the template in `template_name` only for requests for an active locale for the FTL files you provided in the map.

```
from lib.l10n_utils import L10nTemplateView


class AboutView(L10nTemplateView):
    template_name = "about_new.html"
    old_template_name = "about.html"
    ftl_files_map = {
        "about_new.html": ["about_new", "about_shared"],
        "about.html": ["about", "about_shared"],
    }
```

In this example when the `about_new` FTL file is active for a locale, the `about_new.html` template will be rendered. Otherwise the `about.html` template would be used.

Using in a view function

Lastly there's the good old function views. These should use `l10n_utils.render` directly to render the template with the context. You can use the `ftl_files` argument with this function as well.

```
from lib.l10n_utils import render


def about_view(request):
    render(request, "about.html", {"name": "Duder"}, ftl_files="mozorg/about")
```

Fluent File Configuration

In order for a Fluent file to be extracted through automation and sent out for localization, it must first be configured to go through one or more distinct pipelines. This is controlled via a set of configuration files:

- Vendor, locales translated by an agency, and paid for by Marketing (locales covered by staff are also included in this group).
- Pontoon, locales translated by Mozilla contributors.
- Special templates, for locales with dedicated templates that don't go through the localization process (not currently used).

Each configuration file consists of a pre-defined set of locales for which each group is responsible for translating. The locales defined in each file should not be changed without first consulting the with L10n team, and such changes should not be a regular occurrence.

To establish a localization strategy for a Fluent file, it needs to be included as a path in one or more configuration files. For example:

```
[[paths]]
    reference = "en/mozorg/mission.ftl"
    l10n = "{locale}/mozorg/mission.ftl"
```

You can read more about configuration files in the L10n Project Configuration docs.

Important: Path definitions in Fluent configuration files are not source order dependent. A broad definition using a wild card can invalidate all previous path definitions for example. Paths should be defined carefully to avoid exposing

.ftl files to unintended locales.

Using a combination of vendor and pontoon configuration offers a flexible but specific set of options to choose from when it comes to defining an l10n strategy for a page. The available choices are:

1. Staff locales.
2. Staff + select vendor locales.
3. Staff + all vendor locales.
4. Staff + vendor + pontoon.
5. All pontoon locales (for non-marketing content only).

When choosing an option, it's important to consider that vendor locales have a cost associated with them, and pontoon leans on the goodwill of our volunteer community. Typically, only non-marketing content should go through Pontoon for all locales. Everything that is marketing related should feature one of the staff/vendor/pontoon configurations.

Fluent File Activation

Fluent files are activated automatically when processed from the l10n team's repo into our own based on a couple of rules.

1. If a fluent file has a group of required strings, all of those strings must be present in the translation in order for it to be activated.
2. A translation must contain a minimum percent of the string IDs from the English file to be activated.

If both of these conditions are met the locale is activated for that particular Fluent file. Any view using that file as its primary (first in the list) file will be available in that locale.

Deactivation

If the automated system activates a locale but we for some reason need to ensure that this page remains unavailable in that locale, we can add this locale to a list of deactivated locales in the metadata file for that FTL file. For example, say we needed to make sure that the *mozorg/mission.ftl* file remained inactive for German, even though the translation is already done. We would add `de` to the `inactive_locales` list in the `metadata/mozorg/mission.json` file:

```
{
  "active_locales": [
    "de",
    "fr",
    "en-GB",
    "en-US",
  ],
  "inactive_locales": [
    "de"
  ],
  "percent_required": 85
}
```

This would ensure that even though `de` appears in both lists, it will remain deactivated on the site. We could just remove it from the active list, but automation would keep attempting to add it back, so for now this is the best solution we have, and is an indication of the full list of locales that have satisfied the rules.

Alternate Rules

It's also possible to change the percentage of string completion required for activation on a per-file basis. In the same metadata file as above, if a `percent_required` key exists in the JSON data (see above) it will be used as the minimum percent of string completion required for that file in order to activate new locales.

Note: Once a locale is activated for a Fluent file it will **NOT** be automatically deactivated, even if the rules change. If you need to deactivate a locale you should follow the *Deactivation* instructions.

Activation Status

You can determine and use the activation status of a Fluent file in a view to make some decisions; what template to render for example. The way you would do that is with the `ftl_file_is_active` function. For example:

```
# views.py
from lib.l10n_utils import L10nTemplateView
from lib.l10n_utils.fluent import ftl_file_is_active


# class-based view example
class AboutView(L10nTemplateView):
    ftl_files_map = {
        "about.html": ["about"],
        "about_new.html": ["about_new", "about"],
    }

    def get_template_names(self):
        if ftl_file_is_active("mozorg/about_new"):
            template_name = "about_new.html"
        else:
            template_name = "about.html"

        return [template_name]


# function view example
def about_view(request):
    if ftl_file_is_active("mozorg/about_new"):
        template = "mozorg/about_new.html"
        ftl_files = ["mozorg/about_new", "mozorg/about"]
    else:
        template = "about.html"
        ftl_files = ["mozorg/about"]

    render(request, template, ftl_files=ftl_files)
```

Active Locales

To see which locales are active for a particular .ftl file you can either look in the metadata file for that .ftl file, which is the one with the same path but in the `metadata` folder instead of a locale folder in the www-l10n repository. Or if you'd like something a bit nicer looking and more convenient there is the `active_locales` management command:

```
$ ./manage.py l10n_update
```

```
$ ./manage.py active_locales mozorg/mission
```

```
There are 91 active locales for mozorg/mission.ftl:
- af
- an
- ar
- ast
- az
- be
- bg
- bn
...
```

You get an alphabetically sorted list of all of the active locales for that .ftl file. You should run `./manage.py l10n_update` as shown above for the most accurate and up-to-date results.

String extraction

The string extraction process for both new .ftl content and updates to existing .ftl content is handled through automation. On each commit to `main` a command is run that looks for changes to the `l10n/` directory. If a change is detected, it will copy those files into a new branch in mozilla-l10n/www-l10n and then a bot will open a pull request containing those changes. Once the pull request has been reviewed and merged by the L10n team, everything is done.

To view the state of the latest automated attempt to open an L10N PR, see:

- Mozorg L10N PR action

(We also just try to open L10N PRs every 3 hours, to catch any failed jobs that are triggered by a commit to `main`)

CSS

If a localized page needs some locale-specific style tweaks, you can add the style rules to the page's stylesheet like this:

```
html[lang="it"] #features li {
  font-size: 20px;
}

html[dir="rtl"] #features {
  float: right;
}
```

If a locale needs site-wide style tweaks, font settings in particular, you can add the rules to `/media/css/l10n/{{LANG}}/intl.css`. Pages on Bedrock automatically includes the CSS in the base templates with the *l10n_css* helper function. The CSS may also be loaded directly from other Mozilla sites with such a URL: `//mozorg.cdn.mozilla.net/media/css/l10n/{{LANG}}/intl.css`.

Open Sans, the default font on mozilla.org, doesn't offer non-Latin glyphs. `intl.css` can have `@font-face` rules to define locale-specific fonts using custom font families as below:

- *X-LocaleSpecific-Light*: Used in combination with *Open Sans Light*. The font can come in 2 weights: normal and optionally bold
- *X-LocaleSpecific*: Used in combination with *Open Sans Regular*. The font can come in 2 weights: normal and optionally bold
- *X-LocaleSpecific-Extrabold*: Used in combination with *Open Sans Extrabold*. The font weight is 800 only

Here's an example of `intl.css`:

```
@font-face {
  font-family: X-LocaleSpecific-Light;
  font-weight: normal;
  font-display: swap;
  src: local(mplus-2p-light), local(Meiryo);
}

@font-face {
  font-family: X-LocaleSpecific-Light;
  font-weight: bold;
  font-display: swap;
  src: local(mplus-2p-medium), local(Meiryo-Bold);
}

@font-face {
  font-family: X-LocaleSpecific;
  font-weight: normal;
  font-display: swap;
  src: local(mplus-2p-regular), local(Meiryo);
}

@font-face {
  font-family: X-LocaleSpecific;
  font-weight: bold;
  font-display: swap;
  src: local(mplus-2p-bold), local(Meiryo-Bold);
}

@font-face {
  font-family: X-LocaleSpecific-Extrabold;
  font-weight: 800;
  font-display: swap;
  src: local(mplus-2p-black), local(Meiryo-Bold);
}
```

Localizers can specify locale-specific fonts in one of the following ways:

- Choose best-looking fonts widely used on major platforms, and specify those with the `src: local(name)` syntax
- Find a best-looking free Web font, add the font files to `/media/fonts/`, and specify those with the `src: url(path)` syntax
- Create a custom Web font to complement missing glyphs in *Open Sans*, add the font files to `/media/fonts/l10n/`, and specify those with the `src: url(path)` syntax. M+ 2c offers various international glyphs and

looks similar to Open Sans, while Noto Sans is good for the bold and italic variants. You can create subsets of these alternative fonts in the WOFF and WOFF2 formats using a tool found on the Web. See Bug 1360812 for the Fulah (ff) locale's example

Developers should use the `.open-sans` mixin instead of `font-family: 'Open Sans'` to specify the default font family in CSS. This mixin has both *Open Sans* and *X-LocaleSpecific* so locale-specific fonts, if defined, will be applied to localized pages. The variant mixins, `.open-sans-light` and `.open-sans-extrabold`, are also available.

1.2.2 All

Locale-specific Templates

While the `ftl_has_messages` template function is great in small doses, it doesn't scale particularly well. A template filled with conditional copy can be difficult to comprehend, particularly when the conditional copy has associated CSS and/or JavaScript.

In instances where a large amount of a template's copy needs to be changed, or when a template has messaging targeting one particular locale, creating a locale-specific template may be a good choice.

Locale-specific templates function simply by naming convention. For example, to create a version of `/firefox/new.html` specifically for the `de` locale, you would create a new template named `/firefox/new.de.html`. This template can either extend `/firefox/new.html` and override only certain blocks, or be entirely unique.

When a request is made for a particular page, bedrock's rendering function automatically checks for a locale-specific template, and, if one exists, will render it instead of the originally specified (locale-agnostic) template.

Note: Creating a locale-specific template for en-US was not possible when this feature was introduced, but it is now. So you can create your en-US-only template and the rest of the locales will continue to use the default.

Specifying Active Locales in Views

Normally we rely on activation tags in our translation files (.lang files) to determine in which languages a page will be available. This will almost always be what we want for a page. But sometimes we need to explicitly state the locales available for a page. The *impressum* page for example is only available in German and the template itself has German hard-coded into it since we don't need it to be translated into any other languages. In cases like these we can send a list of locale codes with the template context and it will be the final list. This can be accomplished in a few ways depending on how the view is coded.

For a plain view function, you can simply pass a list of locale codes to *l10n_utils.render* in the context using the name *active_locales*. This will be the full list of available translations. Use *add_active_locales* if you want to add languages to the existing list:

```
def french_and_german_only(request):
    return l10n_utils.render(request, "home.html", {"active_locales": ["de", "fr"]})
```

If you don't need a custom view and are just using the *page()* helper function in your *urls.py* file, then you can similarly pass in a list:

```
page("about", "about.html", active_locales=["en-US", "es-ES"]),
```

Or if your view is even more fancy and you're using a Class-Based-View that inherits from *LangFilesMixin* (which it must if you want it to be translated) then you can specify the list as part of the view Class definition:

```
class MyView(LangFilesMixin, View):
    active_locales = ["zh-CN", "hi-IN"]
```

Or in the *urls.py* when using a CBV:

```
url(r"about/$", MyView.as_view(active_locales=["de", "fr"])),
```

The main thing to keep in mind is that if you specify *active_locales* that will be the full list of localizations available for that page. If you'd like to add to the existing list of locales generated from the lang files then you can use the *add_active_locales* name in all of the same ways as *active_locales* above. It's a list of locale codes that will be added to the list already available. This is useful in situations where we would have needed the l10n team to create an empty .lang file with an active tag in it because we have a locale-specific-template with text in the language hard-coded into the template and therefore do not otherwise need a .lang file.

1.2.3 About L10N integrations

Bedrock manages an l10n pipeline that moves the l10n data (fluent `.ftl` files) between the l10n team and bedrock. This section describes how that works.

1. **FILE SETUP**

The source for Fluent files currently is `./l10n/`.

Here's a summary of the files within this directory:

```
./l10n/en/  # This is where source Fluent templates go
./l10n/configs/pontoon.toml  # Config if using community/Pontoon translations
./l10n/configs/vendor.toml  # Config if using a paid-for translation service such as␣
↪Smartling
./l10n/configs/special-templates.toml   # Only needed to exclude certain files from all␣
↪community AND vendor translation, e.g. we use staff translation only

./l10n/l10n-pontoon.toml  # Entrypoint for community localization.
./l10n/l10n-vendor.toml  # Entrypoint for vendor and staff localization

./data/l10n-team/  # populated via a git sync using data FROM the l10n team
```

The root `.toml` files point to the ones in `/configs/` and are a 'gateway' through which we specify which config files are relevant to which translation strategy (community or vendor - or neither if it's staff-only translation).

2. **REPO SETUP**

There are two repos, to hold the translation files as part of the pipeline.

- **A repo in where the files are sent to** for the L10N team's automation to pick up. For example, Mozorg uses `github.com/mozilla-l10n/www-l10n/`.
- **An optional repo where files are post-processed following translation**. For example, Mozorg uses `github.com/mozmeao/www-l10n/`.

Important: This repo is optional if not using Pontoon/community translations. Why? If the translations are done by the community (via Pontoon), there is a possibility that not enough of the strings will be translated in order to render the content in the relevant locale. We run a CI task to determine whether a locale has enough translated strings to be considered 'active'. If we used a vendor entirely, we would expect all locales to be 100% translated.

---

3. **CI SETUP**

Details of how MozMarRobot is hooked are best gleaned from looking at `https://gitlab.com/mozmeao/www-fluent-update`.

In short, once new translations land in the string-source repo (e.g. `github.com/mozilla-l10n/www-l10n`) they are cloned over to the activation-check repo `github.com/mozmeao/www-l10n/` by CI and later pulled into Bedrock from there.

4. **CONFIGURATION SETTINGS**.

There are many settings in `settings/base.py` that help bedrock know what remote repos and local folders to use. Search this file for settings starting with `FLUENT_` to find them.

5. **L10N UPDATE SCRIPT**.

Uploading strings for translation

Uploading `en`-locale source strings from Bedrock to the `github.com/mozilla-l10n/` repo is handled by `bedrock/bin/open-ftl-pr.sh`.

See the Github workflow in `.github/workflows/` for where this is triggered.

Downloading translated strings

The script `bedrock/lib/l10n_utils/management/commands/l10n_update.py` will pull down the appropriate translations.

1.3 Developing on Bedrock

1.3.1 Managing Dependencies

For Python we use `pip-compile` from pip-tools to manage dependencies expressed in our requirements files. `pip-compile` is wrapped up in Makefile commands, to ensure we use it consistently.

If you add a new Python dependency (eg to `requirements/prod.in` or `requirements/dev.in`) you can generate a pinned and hash-marked addition to our requirements files just by running:

```
make compile-requirements
```

and committing any changes that are made. Please re-build your docker image and test it with `make build test` to be sure the dependency does not cause a regression.

Similarly, if you *upgrade* a pinned dependency in an `*.in` file, run `make compile-requirements` then rebuild, test and commit the results

To check for stale Python dependencies (basically `pip list -o` but in the Docker container):

```
make check-requirements
```

For Node packages we use NPM, which should already be installed alongside Node.js.

Front-end Dependencies

Our team maintains a few dependencies that we serve on Bedrock's front-end.

- @mozilla-protocol/core: Bedrock's primary design system
- @mozmeao/cookie-helper: A complete cookies reader/writer framework
- @mozmeao/dnt-helper: Do Not Track (DNT) helper
- @mozmeao/trafficcop: Used for A/B testing page variants

Because they are all published on NPM, install the packages and keep up-to-date with the latest version of each dependency by running an `npm install`. For further documentation on installing NPM packages, check out the official documentation.

1.3.2 Asset Management and Bundling

Bedrock uses Webpack to manage front-end asset processing and bundling. This includes processing and minifying JavaScript and SCSS/CSS bundles, as well as managing static assets such as images, fonts, and other common file types.

When developing on bedrock you can start Webpack by running `make run` when using Docker, or `npm start` when running bedrock locally.

Once Webpack has finished compiling, a local development server will be available at localhost:8000. When Webpack detects changes to a JS/SCSS file, it will automatically recompile the bundle and then refresh any page running locally in the browser.

Webpack Configuration

We have two main Webpack config files in the root directory:

The `webpack.static.config.js` file is responsible for copying static assets, such as images and fonts, from the `/media/` directory over to the `/assets/` directory. This is required so Django can serve them correctly.

The `webpack.config.js` file is responsible for processing JS and SCSS files in the `/media/` directory and compiling them into the `/assets/` directory. This config file also starts a local development server and watches for file changes.

We use two separate config files to keep responsibilities clearly defined, and to make the configs both shorter and easier to follow.

Note: Because of the large number of files used in bedrock, only JS and SCSS files managed by `webpack.config.js` are watched for changes when in development mode. This helps save on memory consumption. The implication of this is that files handled by `webpack.static.config.js` are only copied over when Webpack first runs. If you update an image for example, then you will need to stop and restart Webpack to pick up the change. This is not true for JS and SCSS files, which will be watched for change automatically.

Asset Bundling

Asset bundles for both JS and SCSS are defined in `./media/static-bundles.json`. This is the file where you can define the bundle names that will get used in page templates. For example, a CSS bundle can be defined as:

```
"css": [
    {
        "files": [
            "css/firefox/new/basic/download.scss"
        ],
        "name": "firefox_new_download"
    }
]
```

Which can then be referenced in a page template using:

```
{{ css_bundle('firefox_new_download') }}
```

A JS bundle can be defied as:

```
"js": [
    {
        "files": [
            "protocol/js/protocol-modal.js",
            "js/firefox/new/basic/download.js"
        ],
        "name": "firefox_new_download"
    }
]
```

Which can then be referenced in a page template using:

```
{{ js_bundle('firefox_new_download') }}
```

Once you define a bundle in `static-bundles.json`, the `webpack.config.js` file will use these as entrypoints for compiling JS and CSS and watching for changes.

1.3.3 Writing JavaScript

Bedrock's Webpack configuration supports some different options for writing JavaScript:

Default Configuration

Write `example-script.js` using ES5 syntax and features. Webpack will bundle the JS as-is, without any additional pre-processing.

Babel Configuration

Write `example-script.es6.js` using ES2015+ syntax. Webpack will transpile the code to ES5 using Babel. This is useful when you want to write modern syntax but still support older browsers.

Important: Whilst Babel will transpile most modern JS syntax to ES5 when suitable fallbacks exist, it won't automatically include custom polyfills for everything since these can start to greatly increase bundle size. If you want to use `Promise` or `async/await` functions for example, then you will need to load polyfills for those. This can be done either at the page level, or globally in `lib.js` if it's something that multiple pages would benefit from. But please pick and choose wisely, and be concious of performance.

For pages that are served to Firefox browsers only, such as `/whatsnew`, it is also possible to write native modern JS syntax and serve that directly in production. Here there is no need to include the `.es6.js` file extension. Instead, you can simply use `.js`. The rules that define which files can do this can be found in our ESLint config.

1.3.4 Writing URL Patterns

URL patterns should be the entire URL you desire, minus any prefixes from URLs files importing this one, and including a trailing slash. You should also give the URL a name so that other pages can reference it instead of hardcoding the URL. Example:

```
path("channel/", channel, name="mozorg.channel")
```

If you only want to render a template and don't need to do anything else in a custom view, Bedrock comes with a handy shortcut to automate all of this:

```
from bedrock.mozorg.util import page

page("channel/", "mozorg/channel.html")
```

You don't need to create a view. It will serve up the specified template at the given URL (the first parameter. see the Django docs for details). You can also pass template data as keyword arguments:

```
page(
    "channel/",
    "mozorg/channel.html",
    latest_version=product_details.firefox_versions["LATEST_FIREFOX_VERSION"],
)
```

The variable `latest_version` will be available in the template.

1.3.5 Finding Templates by URL

General Structure

Bedrock follows the Django app structure and most templates are easy to find by matching URL path segments to folders and files within the correct app.

URL: `https://www.mozilla.org/en-US/firefox/features/private-browsing/`
Template path: `bedrock/bedrock/firefox/templates/firefox/features/private-browsing.html`

To get from URL to template path:

- Ignore `https://www.mozilla.org` and the locale path segment `/en-US`. The next path segment is the app name `/firefox`.
- From the root folder of bedrock, find the app's template folder at `bedrock/{app}/templates/{app}`
- Match remaining URL path segments (`/features/private-browsing`) to the template folder's structure (`/features/private-browsing.html`)

Note: `mozorg` is the app name for the home page and child pages related to Mozilla Corporation (i.e. About, Contact, Diversity).

Whatsnew and Firstrun

These pages are specific to Firefox browsers, and only appear when a user updates or installs and runs a Firefox browser for the first time. The URL and template depend on what Firefox browser and version are in use.

Note: There may be extra logic in the app's `views.py` file to change the template based on locale or geographic location as well.

Firefox release

Version number is digits only.

Whatsnew URL: https://www.mozilla.org/en-US/firefox/99.0/whatsnew/
Template path: https://github.com/mozilla/bedrock/tree/main/bedrock/firefox/templates/firefox/whatsnew

Firstrun URL: https://www.mozilla.org/en-US/firefox/99.0/firstrun/
Template path: https://github.com/mozilla/bedrock/blob/main/bedrock/firefox/templates/firefox/firstrun/firstrun.html

Firefox Nightly

Version number is digits and **a1**.

Whatsnew URL: https://www.mozilla.org/en-US/firefox/99.0a1/whatsnew/
Template path:
https://github.com/mozilla/bedrock/blob/main/bedrock/firefox/templates/firefox/nightly/whatsnew.html

Firstrun URL: https://www.mozilla.org/en-US/firefox/nightly/firstrun/
Template path: https://github.com/mozilla/bedrock/tree/main/bedrock/firefox/templates/firefox/nightly

Firefox Developer

Version number is digits and **a2**.

Whatsnew URL: https://www.mozilla.org/en-US/firefox/99.0a2/whatsnew/
Template path:
https://github.com/mozilla/bedrock/blob/main/bedrock/firefox/templates/firefox/developer/whatsnew.html

Firstrun URL: https://www.mozilla.org/en-US/firefox/99.0a2/firstrun/
Template path:
https://github.com/mozilla/bedrock/blob/main/bedrock/firefox/templates/firefox/developer/firstrun.html

Release Notes

Release note templates live here: https://github.com/mozilla/bedrock/tree/main/bedrock/firefox/templates/firefox/releases

Note: Release note content is pulled in from an external data source.

- Firefox release: https://www.mozilla.org/en-US/firefox/99.0.1/releasenotes/
- Firefox Developer and Beta: https://www.mozilla.org/en-US/firefox/100.0beta/releasenotes/
- Firefox Nightly: https://www.mozilla.org/en-US/firefox/101.0a1/releasenotes/
- Firefox Android: https://www.mozilla.org/en-US/firefox/android/99.0/releasenotes/
- Firefox iOS: https://www.mozilla.org/en-US/firefox/ios/99.0/releasenotes/

1.3.6 Optimizing Images

Images can take a long time to load and eat up a lot of bandwidth. Always take care to optimize images before uploading them to the site. There are a number of great online resources available to help with this:

- https://tinypng.com/
- https://jakearchibald.github.io/svgomg/
- https://squoosh.app/

We also bundle the svgo package as a dev dependency, which can optimize SVGs on the command line.

1.3.7 Embedding Images

Images should be included on pages using one of the following helper functions.

Primary image helpers

The following image helpers support the most common features and use cases you may encounter when coding pages:

static()

For a simple image, the `static()` function is used to generate the image URL. For example:

```
<img src="{{ static('img/firefox/new/firefox-wordmark-logo.svg') }}" alt="Firefox">
```

will output an image:

```
<img src="/media/img/firefox/new/firefox-wordmark-logo.svg" alt="Firefox">
```

resp_img()

For responsive images, where we want to specify multiple different image sizes and let the browser select which is best to use.

The example below shows how to serve an appropriately sized, responsive red panda image:

```
resp_img(
    url="img/panda-500.png",
    srcset={
        "img/panda-500.png": "500w",
        "img/panda-750.png": "750w",
        "img/panda-1000.png": "1000w",
    },
    sizes={
        "(min-width: 1000px)": "calc(50vw - 200px)",
        "default": "calc(100vw - 50px)",
    },
)
```

This would output:

```
<img src="/media/img/panda-500.png"
     srcset="/media/img/panda-500.png 500w,/media/img/panda-750.png 750w,/media/img/
↪panda-1000.png 1000w"
     sizes="(min-width: 1000px) calc(50vw - 200px),calc(100vw - 50px)" alt="">'
```

In the above example we specified the available image sources using the `srcset` parameter. We then used `sizes` to say:

- When the viewport is greater than `1000px` wide, the panda image will take up roughly half of the page width.
- When the viewport is less than `1000px` wide, the panda image will take up roughly full page width.

The default image `src` is what we specified using the `url` param. This is also what older browsers will fall back to using. Modern browsers will instead pick the best source option from `srcset` (based on both the estimated image size and screen resolution) to satisfy the condition met in `sizes`.

Note: The value `default` in the second `sizes` entry above should be used when you want to omit a media query. This makes it possible to provide a fallback size when no other media queries match.

Another example might be to serve a high resolution alternative for a fixed size image:

```
resp_img(url="img/panda.png", srcset={"img/panda-high-res.png": "2x"})
```

This would output:

```
<img src="/media/img/panda.png" srcset="/media/img/panda-high-res.png 2x" alt="">
```

Here we don't need a `sizes` attribute, since the panda image is fixed in size and small enough that it won't need to resize along with the browser window. Instead the `srcset` image includes an alternate high resolution source URL, along with a pixel density descriptor. This can then be used to say:

- When a browser specifies a device pixel ratio of `2x` or greater, use `panda-high-res.png`.
- When a browser specifies a device pixel ration of less than `2x`, use `panda.png`.

The `resp_img()` helper also supports localized images by setting the `'l10n'` parameter to `True`:

```
resp_img(
    url="img/panda-500.png",
    srcset={
        "img/panda-500.png": "500w",
        "img/panda-750.png": "750w",
        "img/panda-1000.png": "1000w",
    },
    sizes={
        "(min-width: 1000px)": "calc(50vw - 200px)",
        "default": "calc(100vw - 50px)",
    },
    optional_attributes={"l10n": True},
)
```

This would output (assuming `de` was your locale):

```
<img src="/media/img/l10n/de/panda-500.png"
     srcset="/media/img/l10n/de/panda-500.png 500w,/media/img/l10n/de/panda-750.png 750w,
↪/media/img/l10n/de/panda-1000.png 1000w"
     sizes="(min-width: 1000px) calc(50vw - 200px),calc(100vw - 50px)" alt="">'
```

Finally, you can also specify any other additional attributes you might need using `optional_attributes`:

```
resp_img(
    url="img/panda-500.png",
    srcset={
        "img/panda-500.png": "500w",
        "img/panda-750.png": "750w",
        "img/panda-1000.png": "1000w",
    },
```

(continues on next page)

(continued from previous page)

```
    sizes={
        "(min-width: 1000px)": "calc(50vw - 200px)",
        "default": "calc(100vw - 50px)",
    },
    optional_attributes={
        "alt": "Red Panda",
        "class": "panda-hero",
        "height": "500",
        "l10n": True,
        "loading": "lazy",
        "width": "500",
    },
)
```

picture()

For responsive images, where we want to serve different images, or image types, to suit different display sizes.

The example below shows how to serve a different image for desktop and mobile sizes screens:

```
picture(
    url="img/panda-mobile.png",
    sources=[
        {"media": "(max-width: 799px)", "srcset": {"img/panda-mobile.png": "default",
↪'width' : '160', 'height' : '320'}},
        {"media": "(min-width: 800px)", "srcset": {"img/panda-desktop.png": "default",
↪'width' : '640', 'height' : '1280'}},
    ],
)
```

This would output:

```
<picture>
    <source media="(max-width: 799px)" srcset="/media/img/panda-mobile.png" width="160"␣
↪height="320">
    <source media="(min-width: 800px)" srcset="/media/img/panda-desktop.png" width="640"␣
↪height="1280">
    <img src="/media/img/panda-mobile.png" alt="">
</picture>
```

In the above example, the default image `src` is what we specified using the `url` param. This is also what older browsers will fall back to using. We then used the `sources` parameter to specify one or more alternate image `<source>` elements, which modern browsers can take advantage of. For each `<source>`, `media` lets us specify a media query as a condition for when to load an image, and `srcset` lets us specify one or more sizes for each image. Each `srcset` also takes optional parameters for `height` and `width`. Defining height and width helps the Browser reserve space for the image and avoid content shifting around on the page. If the image will not change dimensions, these can be defined in the `optional_attributes` for the entire element instead of separately like this.

Note: The value `default` in the `srcset` entry above should be used when you want to omit a descriptor. In this example we only have one entry in `srcset` (meaning it will be chosen immediately should the media query be satisfied), hence we omit a descriptor value.

A more complex example might be when we want to load responsively sized, animated gifs, but also offer still images for users who set `(prefers-reduced-motion: reduce)`:

```
picture(
    url="img/dancing-panda-500.gif",
    sources=[
        {
            "media": "(prefers-reduced-motion: reduce)",
            "srcset": {
                "img/sleeping-panda-500.png": "500w",
                "img/sleepinng-panda-750.png": "750w",
                "img/sleeping-panda-1000.png": "1000w",
            },
            "sizes": {
                "(min-width: 1000px)": "calc(50vw - 200px)",
                "default": "calc(100vw - 50px)",
            },
        },
        {
            "media": "(prefers-reduced-motion: no-preference)",
            "srcset": {
                "img/dancing-panda-500.gif": "500w",
                "img/dancing-panda-750.gif": "750w",
                "img/dancing-panda-1000.gif": "1000w",
            },
            "sizes": {
                "(min-width: 1000px)": "calc(50vw - 200px)",
                "default": "calc(100vw - 50px)",
            },
        },
    ],
)
```

This would output:

```
<picture>
    <source media="(prefers-reduced-motion: reduce)"
            srcset="/media/img/sleeping-panda-500.png 500w,/media/img/sleeping-panda-750.
↪png 750w,/media/img/sleeping-panda-1000.png 1000w"
            sizes="(min-width: 1000px) calc(50vw - 200px),calc(100vw - 50px)">
    <source media="(prefers-reduced-motion: no-preference)"
            srcset="/media/img/dancing-panda-500.gif 500w,/media/img/dancing-panda-750.
↪gif 750w,/media/img/dancing-panda-1000.gif 1000w"
            sizes="(min-width: 1000px) calc(50vw - 200px),calc(100vw - 50px)">
    <img src="/media/img/dancing-panda-500.gif" alt="">
</picture>
```

In the above example we would default to loading animated gifs, but if a user agent specified `(prefers-reduced-motion: reduce)` then the browser would load static png files instead. Multiple image sizes are also supported for each <source> using `srcset` and `sizes`.

Another type of use case might be to serve different image formats, so capable browsers can take advantage of more efficient encoding:

```
picture(
    url="img/red-panda.png",
    sources=[{"type": "image/webp", "srcset": {"img/red-panda.webp": "default"}}],
)
```

This would output:

```
<picture>
    <source type="image/webp" srcset="/media/img/red-panda.webp">
    <img src="/media/img/red-panda.png" alt="">
</picture>
```

In the above example we use `sources` to specify an alternate image with a `type` attribute of `image/webp`. This lets browsers that support WebP to download `red-panda.webp`, whilst older browsers would download `red-panda.png`.

Like `resp_img()`, the `picture()` helper also supports L10n images and other useful attributes via the `optional_attributes` parameter:

```
picture(
    url="img/panda-mobile.png",
    sources=[
        {"media": "(max-width: 799px)", "srcset": {"img/panda-mobile.png": "default"}},
        {"media": "(min-width: 800px)", "srcset": {"img/panda-desktop.png": "default"}},
    ],
    optional_attributes={
        "alt": "Red Panda",
        "class": "panda-hero",
        "l10n": True,
        "loading": "lazy",
    },
)
```

Which image helper should you use?

This is a good question. The answer depends entirely on the image in question. A good rule of thumb is as follows:

- **Is the image a vector format (e.g. `.svg`)?**
  - If yes, then for most cases you can simply use `static()`.
- **Is the image a raster format (e.g. `.png` or `.jpg`)?**
  - Is the same image displayed on both large and small viewports? Does the image need to scale as the browser resizes? If yes to both, then use `resp_img()` with both `srcset` and `sizes`.
  - Is the image fixed in size (non-responsive)? Do you need to serve a high resolution version? If yes to both, then use `resp_img()` with just `srcset`.
- Does the source image need to change depending on a media query (e.g serve a different image on both desktop and mobile)? If yes, then use `picture()` with `media` and `srcset`.
- Is the image format only supported in certain browsers? Do you need to provide a fallback? If yes to both, then use `picture()` with `type` and `srcset`.

Secondary image helpers

The following image helpers are less commonly used, but exist to support more specific use cases. Some are also encapsulated as features inside inside of primary helpers, such as `l1n_img()`.

l10n_img()

Images that have translatable text can be handled with `l10n_img()`:

```
<img src="{{ l10n_img('firefox/os/have-it-all/messages.jpg') }}">
```

The images referenced by `l10n_img()` must exist in `media/img/l10n/`, so for above example, the images could include `media/img/l10n/en-US/firefox/os/have-it-all/messages.jpg` and `media/img/l10n/es-ES/firefox/os/have-it-all/messages.jpg`.

qrcode()

This is a helper function that will output SVG data for a QR Code at the spot in the template where it is called. It caches the results to the `data/qrcode_cache` directory, so it only generates the SVG data one time per data and box_size combination.

```
qrcode("https://accounts.firefox.com", 30)
```

The first argument is the data you'd like to encode in the QR Code (usually a URL), and the second is the "box size". It's a parameter that tells the generator how large to set the height and width parameters on the XML SVG tag, the units of which are "mm". This can be overriden with CSS so you may not need to use it at all. The `box_size` parameter is optional.

1.3.8 Using Large Assets

We don't want to (and if large enough GitHub won't let us) commit large files to the bedrock repo. Files such as large PDFs or very-high-res JPG files (e.g. leadership team photos), or videos are not well-tracked in git and will make every checkout after they're added slower and this diffs less useful. So we have another domain at which we upload these files: assets.mozilla.net

This domain is simply an AWS S3 bucket with a CloudFront CDN (Content Delivery Network) in front of it. It is highly available and fast. We've made adding files to this domain very simple using git-lfs. You simply install git-lfs, clone our assets.mozilla.net repo, and then add and commit files under the `assets` directory there as usual. Open a pull request, and once it's merged it will be automatically uploaded to the S3 bucket and be available on the domain.

For example, if you add a file to the repo under `assets/pdf/the-dude-abides.pdf`, it will be available as https://assets.mozilla.net/pdf/the-dude-abides.pdf. Once that is done you can link to that URL from bedrock as you would any other URL.

1.3.9 Writing Migrations

Bedrock uses Django's built-in Migrations framework for its database migrations, and has no custom database routing, etc. So, no big surprises here – write things as you regularly would.

However, as with any complex system, care needs to be taken with schema changes that drop or rename database columns. Due to the way the rollout process works (ask for details directly from the team), an absent column can cause some of the rollout to enter a crashloop.

To avoid this, split your changes across releases, such as below.

For column renames:

- Release 1: Add your new column
- Release 2: Amend the codebase to use it instead of the old column
- Release 3: Clean up - drop the old, deprecated column, which should not be referenced in code at this point.

For column drops:

- Release 1: Update all code that uses the relevant column, so that nothing interacts with it any more.
- Release 2: Clean up - drop the old, deprecated column.

With both paths, check for any custom schema or data migrations that might reference the deprecated column.

1.3.10 Writing Views

You should rarely need to write a view for mozilla.org. Most pages are static and you should use the `page` function documented above.

If you need to write a view and the page is translated or translatable then it should use the `l10n_utils.render()` function to render the template.

```
from lib import l10n_utils

from django.views.decorators.http import require_safe


@require_safe
def my_view(request):
    # do your fancy things
    ctx = {"template_variable": "awesome data"}
    return l10n_utils.render(request, "app/template.html", ctx)
```

Make sure to namespace your templates by putting them in a directory named after your app, so instead of templates/template.html they would be in templates/blog/template.html if `blog` was the name of your app.

The `require_safe` ensures that only `GET` or `HEAD` requests will make it through to your view.

If you prefer to use Django's Generic View classes we have a convenient helper for that. You can use it either to create a custom view class of your own, or use it directly in a `urls.py` file.

```
# app/views.py
from lib.l10n_utils import L10nTemplateView


class FirefoxRoxView(L10nTemplateView):
```

(continues on next page)

(continued from previous page)

```
    template_name = "app/firefox-rox.html"


# app/urls.py
urlpatterns = [
    # from views.py
    path("firefox/rox/", FirefoxRoxView.as_view()),
    # directly
    path(
        "firefox/sox/", L10nTemplateView.as_view(template_name="app/firefox-sox.html")
    ),
]
```

The `L10nTemplateView` functionality is mostly in a template mixin called `LangFilesMixin` which you can use with other generic Django view classes if you need one other than `TemplateView`. The `L10nTemplateView` already ensures that only `GET` or `HEAD` requests will be served.

Variation Views

We have a generic view that allows you to easily create and use a/b testing templates. If you'd like to have either separate templates or just a template context variable for switching, this will help you out. For example.

```
# urls.py

from django.urls import path

from bedrock.utils.views import VariationTemplateView

urlpatterns = [
    path(
        "testing/",
        VariationTemplateView.as_view(
            template_name="testing.html", template_context_variations=["a", "b"]
        ),
        name="testing",
    ),
]
```

This will give you a context variable called `variation` that will either be an empty string if no param is set, or `a` if `?v=a` is in the URL, or `b` if `?v=b` is in the URL. No other options will be valid for the `v` query parameter and `variation` will be empty if any other value is passed in for `v` via the URL. So in your template code you'd simply do the following:

```
{% if variation == 'b' %}<p>This is the B variation of our test. Enjoy!</p>{% endif %}
```

If you'd rather have a fully separate template for your test, you can use the `template_name_variations` argument to the view instead of `template_context_variations`.

```
# urls.py

from django.urls import path

from bedrock.utils.views import VariationTemplateView
```

(continues on next page)

(continued from previous page)

```
urlpatterns = [
    path(
        "testing/",
        VariationTemplateView.as_view(
            template_name="testing.html", template_name_variations=["1", "2"]
        ),
        name="testing",
    ),
]
```

This will not provide any extra template context variables, but will instead look for alternate template names. If the URL is `testing/?v=1`, it will use a template named `testing-1.html`, if `v=2` it will use `testing-2.html`, and for everything else it will use the default. It simply puts a dash and the variation value between the template file name and file extension.

It is theoretically possible to use the template name and template context versions of this view together, but that would be an odd situation and potentially inappropriate for this utility.

You can also limit your variations to certain locales. By default the variations will work for any localization of the page, but if you supply a list of locales to the `variation_locales` argument to the view then it will only set the variation context variable or alter the template name (depending on the options explained above) when requested at one of said locales. For example, the template name example above could be modified to only work for English or German like so

```
# urls.py

from django.urls import path

from bedrock.utils.views import VariationTemplateView

urlpatterns = [
    path(
        "testing/",
        VariationTemplateView.as_view(
            template_name="testing.html",
            template_name_variations=["1", "2"],
            variation_locales=["en-US", "de"],
        ),
        name="testing",
    ),
]
```

Any request to the page in for example French would not use the alternate template even if a valid variation were given in the URL.

Note: If you'd like to add this functionality to an existing Class-Based View, there is a mixin that implements this pattern that should work with most views: `bedrock.utils.views.VariationMixin`.

Geo Template View

Now that we have our CDN configured properly, we can also just swap out templates per request country. This is very similar to the above, but it will simply use the proper template for the country from which the request originated.

```
from bedrock.base.views import GeoTemplateView


class CanadaIsSpecialView(GeoTemplateView):
    geo_template_names = {
        "CA": "mozorg/canada-is-special.html",
    }
    template_name = "mozorg/everywhere-else-is-also-good.html"
```

Testing Geo Views

For testing purposes while you're developing or on any deployment that is not accessed via the production domain (www.mozilla.org) you can append your URL with a `geo` query param (e.g. `/firefox/?geo=DE`) and that will take precedence over the country from the request header. Remember to use a valid ISO country code as param value.

Other Geo Stuff

There are a couple of other tools at your disposal if you need to change things depending on the location of the user. You can use the `bedrock.base.geo.get_country_from_request` function in a view and it will return the country code for the request (either from the CDN or the query param, just like above).

```
from bedrock.base.geo import get_country_from_request


def dude_view(request):
    country = get_country_from_request(request)
    if country == "US":
        ...  # do a thing for the US
    else:
        ...  # do the default thing
```

The other convenience available is that the country code, either from the CDN or the query param, is avilable in any template in the `country_code` variable. This allows you to change anything about how the template renders based on the location of the user.

```
{% if country_code == "US" %}
    <h1>GO MURICA!</h1>
{% else %}
    <h1>Yay World!</h1>
{% endif %}
```

Reference:

- Officially assigned list of ISO country codes.

1.3.11 Metrics Collection with Markus

Markus is a metrics library that we use in our project for collecting and reporting statistics about our code's operation. It provides a simple and consistent way to record custom metrics from your application, which can be crucial for monitoring and performance analysis.

Markus supports a variety of backends, including Datadog, Statsd, and Logging. This means you can choose the backend that best fits your monitoring infrastructure and requirements. Each backend has its own set of features and capabilities, but Markus provides a unified interface to all of them.

Once the metrics are collected by Markus they are then forwarded to Telegraf. Telegraf is an agent for collecting and reporting metrics, which we use to process and format the data before it's sent to Grafana.

Grafana is a popular open-source platform for visualizing metrics. It allows us to create dashboards with panels representing the metrics we're interested in, making it easy to understand the data at a glance.

Here's an example of how to use Markus to record a metric:

```
from bedrock.base import metrics

# Counting events
metrics.incr("event_name")

# Timing events
metrics.timing("event_name", 123)

# Or timing events with context manager
with metrics.timer("event_name"):
    ...  # code to time goes here
```

In addition to recording the metric values, Markus also allows you to add tags to your metrics. Tags are key-value pairs that provide additional context about the metric, making it easier to filter and aggregate the data in Grafana. For example, you might tag a metric with the version of your application, the user's country, or the result of an operation. To add tags to a metric in Markus, you can pass them as a dictionary to the metric recording method. Here's an example:

```
# Counting events with tags
metrics.incr("event_name", tags=[f"version:{version}", f"country:{country}"])
```

For more information, refer to the Markus documentation.

1.3.12 Coding Style

Bedrock uses the following open source tools to follow coding styles and conventions, as well as applying automatic code formatting:

- ruff for Python style, code quality rules, and import ordering.
- black for Python code formatting.
- Prettier for JavaScript code formatting.
- ESLint for JavaScript code quality rules.
- Stylelint for Sass/CSS style and code quality rules.

For front-end HTML & CSS conventions, bedrock uses Mozilla's Protocol design system for building components. You can read the Protocol documentation site for more information.

Mozilla also has some more general coding styleguides available, although some of these are now rather outdated:

- Mozilla Python Style Guide
- Mozilla HTML Style Guide
- Mozilla JS Style Guide
- Mozilla CSS Style Guide

1.3.13 Test coverage

When the Python tests are run, a coverage report is generated, showing which lines of the codebase have tests that execute them, and which do not. You can view this report in your browser at `file:///path/to/your/checkout/of/bedrock/python_coverage/index.html`.

When adding code, please aim to provide solid test coverage, using the coverage report as a guide. This doesn't necessarily mean every single line needs a test, and 100% coverage doesn't mean 0% defects.

1.3.14 Configuring your Code Editor

Bedrock includes an `.editorconfig` file in the root directory that you can use with your code editor to help maintain consistent coding styles. Please see editorconfig.org. for a list of supported editors and available plugins.

1.3.15 Working with Protocol Design System

Bedrock uses the Protocol Design System to quickly produce consistent, stable components. There are different methods – depending on the component – to import a Protocol component into our codebase.

One method involves two steps:

1. Adding the *correct markup* or importing the *appropriate macro* to the page's HTML file.
2. Importing the necessary Protocol styles to a page's SCSS file.

The other method is to *import CSS bundles* onto the HTML file. However, this only works for certain components, which are listed below in the respective section.

Styles and Components

The base templates in Bedrock have global styles from Protocol that apply to every page. When we need to extend these styles on a page-specific basis, we set up Protocol in a page-specific SCSS file.

For example, on a Firefox product page, we might want to use Firefox logos or wordmarks that do not exist on every page.

To do this, we add Protocol `mzp-` classes to the HTML:

```
// bedrock/bedrock/firefox/templates/firefox/{specific-page}.html

<div class="mzp-c-wordmark mzp-t-wordmark-md mzp-t-product-firefox">
    Firefox Browser
</div>
```

Then we need to include those Protocol styles in the page's SCSS file:

```
/* bedrock/media/css/firefox/{specific-page}.scss */

/* if we need to use protocol images, we need to set the $image-path variable */
$image-path: '/media/protocol/img';
/* mozilla is the default theme, so if we want a different one, we need to set the
↪$brand-theme variable */
$brand-theme: 'firefox';

/* the lib import is always essential: it provides access to tokens, functions, mixins,␣
↪and theming */
@import '~@mozilla-protocol/core/protocol/css/includes/lib';
/* then you add whatever specific protocol styling you need */
@import '~@mozilla-protocol/core/protocol/css/components/logos/wordmark';
@import '~@mozilla-protocol/core/protocol/css/components/logos/wordmark-product-firefox';
```

Note: If you create a new SCSS file for a page, you will have to include it in that page's CSS bundle by updating *static-bundles.json* file.

Macros

The team has created several Jinja macros out of Protocol components to simplify the usage of components housing larger blocks of code (i.e. Billboard). The code housing the custom macros can be found in our protocol macros file. These Jinja macros include parameters that are simple to define and customize based on how the component should look like on a given page.

To use these macros in files, we simply import a macro to the page's HTML code and call it with the desired arguments, instead of manually adding Protocol markup. We can import multiple macros in a comma-separated fashion, ending the import with `with context`:

```
// bedrock/bedrock/firefox/templates/firefox/{specific-page}.html

{% from "macros-protocol.html" import billboard with context %}

{{ billboard(
    title='This is Firefox.',
    ga_title='This is Firefox',
    desc='Firefox is an awesome web browser.',
    link_cta='Click here to install',
    link_url=url('firefox.new')
  )}}
```

Because not all component styles are global, we still have to import the page-specific Protocol styles in SCSS:

```
/* bedrock/media/css/firefox/{specific-page}.scss */

$brand-theme: 'firefox';

@import '~@mozilla-protocol/core/protocol/css/includes/lib';
@import '~@mozilla-protocol/core/protocol/css/components/billboard';
```

Import CSS Bundles

We created pre-built CSS bundles to be used for some components due to their frequency of use. This method only requires an import into the HTML template. Since it's a separate CSS bundle, we don't need to import that component in the respective page CSS. The CSS bundle import only works for the following components:

- Split
- Card
- Picto
- Callout
- Article
- Newsletter form
- Emphasis box

Include a CSS bundle in the template's `page_css` block along with any other page-specific bundles, like so:

```
{% block page_css %}
    {{ css_bundle('protocol-split') }}
    {{ css_bundle('protocol-card') }}
    {{ css_bundle('page-specific-bundle') }}
{% endblock %}
```

1.4 How to contribute

Before diving into code it might be worth reading through the *Developing on Bedrock* documentation, which contains useful information and links to our coding guidelines for Python, Django, JavaScript and CSS.

1.4.1 Git workflow

When you want to start contributing, you should create a branch from main. This allows you to work on different project at the same time:

```
$ git switch main
```

```
$ git switch -c topic-branch
```

To keep your branch up-to-date, assuming the mozilla repository is the remote called mozilla:

```
$ git switch main
```

```
$ git pull --ff-only
```

More on Why you should use –ff-only. To make this the default update your Git config as described in the article.

```
$ git switch topic-branch
```

```
$ git rebase main
```

If you need more Git expertise, a good resource is the Git book.

Once you're done with your changes, you'll need to describe those changes in the commit message.

1.4.2 Git commit messages

Commit messages are important when you need to understand why something was done.

- First, learn how to write good git commit messages.
- All commit messages must include a bug number. You can put the bug number on any line, not only the first one.
- If you use the syntax `bug xxx`, Github will reference the commit into Bugzilla. With `fix bug xxx`, it will even close the bug once it goes into main.

If you're asked to change your commit message, you can use these commands:

```
$ git commit --amend
```

-f is doing a force push because you modified the history

```
$ git push -f my-remote topic-branch
```

1.4.3 Submitting your work

In general, you should submit your work with a pull request to main. If you are working with other people or you want to put your work on a demo server, then you should be working on a common topic branch.

Once your code has been positively reviewed, it will be deployed shortly after. So if you want feedback on your code but it's not ready to be deployed, you should note it in the pull request, or use a Draft PR. Also make use of an appropriate label, such as `Do Not Merge`.

1.4.4 Squashing your commits

Should your pull request contain more than one commit, sometimes we may ask you to squash them into a single commit before merging. You can do this with *git rebase*.

As an example, let's say your pull request contains two commits. To squash them into a single commit, you can follow these instructions:

```
$ git rebase -i HEAD~2
```

You will then get an editor with your two commits listed. Change the second commit from *pick* to *fixup*, then save and close. You should then be able to verify that you only have one commit now with *git log*.

To push to GitHub again, because you "altered the history" of the repo by merging the two commits into one, you'll have to *git push -f* instead of just *git push*.

1.4.5 Deploying your code

These are the websites that Bedrock is usually deployed to as part of development.

Demo sites

- ***Mozorg:***
  - Branch `mozorg-demo-1` -> https://www-demo1.allizom.org/
  - Branch `mozorg-demo-2` -> https://www-demo2.allizom.org/
  - Branch `mozorg-demo-3` -> https://www-demo3.allizom.org/
  - Branch `mozorg-demo-4` -> https://www-demo4.allizom.org/
  - Branch `mozorg-demo-5` -> https://www-demo5.allizom.org/
  - Branch `mozorg-demo-6` -> https://www-demo6.allizom.org/
  - Branch `mozorg-demo-7` -> https://www-demo7.allizom.org/
  - Branch `mozorg-demo-8` -> https://www-demo8.allizom.org/
  - Branch `mozorg-demo-9` -> https://www-demo9.allizom.org/

For example:

```
$ git push -f mozilla my-demo-branch:mozorg-demo-2
```

Deployment notification and logs

At the moment, there is no way to view logs for the deployment unless you have access to Google Cloud Platform.

If you *do* have access, the Cloud Build dashboard shows the latest builds, and Cloud Run will link off to the relevant logs.

There are Mozilla Slack notifications in `#www-notify` that show the status of demo builds. (Work is ticketed to make those notifications richer in data.)

Env vars

Rather than tweak env vars via a web UI, they are set in config files. We have specific demo-use-only env var files, which are only used by our GCP demo setup. They are:

- `bedrock/gcp/bedrock-demos/cloudrun/mozorg-demo.env.yaml`

If you need to set/add/remove an env var, you can edit the relevant file on your feature branch, commit it and push it along with the rest of the code, as above. There is a small risk of clashes, but these can be best avoided if you keep up to date with `bedrock/main` and can be resolved easily.

Secret values

Remember that the env vars files are public because they are in the Bedrock codebase, so sensitive values should not be added there, even temporarily.

If you need to add a secret value, this currently needs to be added at the GCP level by someone with appropriate permissions to edit and apply the Terraform configuration, and to edit the trigger YAML spec to pass through the new secret. Currently Web-SRE and the backend team have appropriate GCP access and adding a secret is relatively quick. (We can make this easier in the future if there's sufficient need, of course.)

Note: Always-on vs auto-sleep demo servers

The demo servers are on GCP Cloud Run, and by default they will be turned off if there is no traffic for 15 minutes. After this time, the demo app will be woken up if it receives a request.

Normally, a 'cold start' will not be a problem. However, if the branch you are demoing does things that alter the database (i.e contains migrations), then you may find the restarted demo app crashes because the new migrations have not been applied after a cold start.

The best current way to avoid that happening is:

- In your branch's demo-env-vars YAML file, set `LOCAL_DB_UPDATE=True` so that the Dev DB is not pulled down to the demo app
- Ask one of the backend team to set the Demo app to always be awake by setting 'Minimum instances' to 1 for the relevant Cloud Run service and restarting it. The app will always be on and will not sleep, so won't need a cold start. Once you have completed the feature work, please ask the backenders to restore the default sleepy behaviour. As an example with `mozorg-demo-1`:
  - To make it always-on: `gcloud run services update mozorg-demo-1 --min-instances 1`
  - To revert it to auto-sleeping: `gcloud run services update mozorg-demo-1 --min-instances 0`

(We'll try to make this a self-serve thing as soon as we can).

DEPRECATED: Heroku Demo Servers

Demos are now powered by Google Cloud Platform (GCP), and no longer by Heroku.

However, the Github Action we used to push code to Heroku may still be enabled. Pushing a branch to one of the *demo/** branches of the *mozilla/bedrock* repo will trigger this. However, note that URLs that historically used to point to Heroku will be pointed to the new GCP demos services instead, so you will have to look at Heroku's web UI to see what the URL of the relevant Heroku app is.

To push to launch a demo on Heroku:

```
$ git push -f mozilla my-demo-branch:demo/1
```

Pushing to production

We're doing pushes as soon as new work is ready to go out.

Code flows automatically to Dev, amd manually to Stage and to Production. See *Continuous Integration & Deployment* for details.

After doing a push, those who are responsible for implementing changes need to update the bugs that have been pushed with a quick message stating that the code was deployed.

If you'd like to see the commits that will be deployed before the push run the following command:

```
$ ./bin/open-compare.py
```

This will discover the currently deployed git hash, and open a compare URL at github to the latest main. Look at `open-compare.py -h` for more options.

We automate pushing to production via tagged commits (see *Continuous Integration & Deployment*)

1.5 Continuous Integration & Deployment

Bedrock runs a series of automated tests as part of continuous integration workflow and deployment pipeline. You can learn more about each of the individual test suites by reading their respective pieces of documentation:

- Python unit tests (see *Run the tests*).
- JavaScript unit tests (see *Front-end testing*).
- Redirect tests (see *Testing redirects*).
- Functional tests (see *Front-end testing*).

1.5.1 Deployed site URLs

Dev

- *Mozorg URL:* https://www-dev.allizom.org/
- *Bedrock locales:* dev repo
- *Bedrock Git branch:* main, deployed on git push
- *Firefox download URL:* https://bouncer-bouncer.stage.mozaws.net/

Staging

- *Mozorg URL:* https://www.allizom.org/
- *Bedrock locales:* prod repo
- *Bedrock Git branch:* stage, deployed on git push
- *Firefox download URL:* https://download.mozilla.org/

Production

- *Mozorg URL:* https://www.mozilla.org/
- *Bedrock locales:* prod repo
- *Bedrock Git branch:* prod, deployed on git push with date-tag
- *Firefox download URL:* https://download.mozilla.org/

Note: By default, the Demo servers on GCP point to the Bouncer Dev service at https://dev.bouncer.nonprod.webservices.mozgcp.net/ To change this, you will have adjust GCP Secrets - see the demo sites docs

---

You can check the currently deployed git commit by checking /revision.txt on any of these URLs.

1.5.2 Tests in the lifecycle of a change

Below is an overview of the tests during the lifecycle of a change to bedrock:

Local development

The change is developed locally, and page specific integration tests can be executed against a locally running instance of the application. If testing changes to the website as a whole is required, then pushing changes to the special `run-integration-tests` branch (see below) is much faster than running the full test suite locally.

Pull request

Once a pull request is submitted, a Unit Tests Github Action will run both the Python and JavaScript unit tests, as well as the suite of redirect headless HTTP(s) response checks.

Push to main branch

Whenever a change is pushed to the main branch, a new image is built and deployed to the dev environment, and the full suite of headless and UI tests are run. This is handled by the pipeline, and is subject to change according to the settings in the Github Action workflow defined in `bedrock/.github/workflows/integration_tests.yml`.

The tests for the dev environment are currently configured as follows:

- Firefox, Chromium, and Webkit headless browsers via Playwright.
- Chrome (latest) via local Selenium grid.
- Firefox (latest) via local Selenium grid.
- Internet Explorer 11 (smoke tests) via Sauce Labs.
- Headless tests.

Note: The deployment workflow runs like this

1. A push to the `main/stage/prod/run-integration-tests` branch of `mozilla/bedrock` triggers a webhook ping to the (private) `mozilla-sre-deploy/deploy-bedrock` repo.

2. A Github Action (GHA) in `mozilla-sre-deploy/deploy-bedrock` builds a "release"-ready Bedrock container image, which it stores in a private container registry (private because our infra requires container-image access to be locked down). Using the same commit, the workflow also builds an equivalent set of public Bedrock container images, which are pushed to Docker Hub.

3. The GHA deploys the relevant container image to the appropriate environment.

4. The GHA pings a webhook back in `mozilla/bedrock` to run integration tests against the environment that has just been deployed.

Push to stage branch

Whenever a change is pushed to the stage branch, a production docker image is built, published to Docker Hub, and deployed to a public staging environment. Once the new image is deployed, the full suite of UI tests is run against it again, but this time with the addition of the *headless download tests*.

Push to prod branch (tagged)

When a tagged commit is pushed to the `prod` branch, a production container image (private, see above) is built, and a set of public images is also built and pushed to Docker Hub if needed (usually this will have already happened as a result of a push to the `main` or `stage` branch). The production image is deployed to each production deployment.

Push to prod cheat sheet

1. Check out the `main` branch
2. Make sure the `main` branch is up to date with `mozilla/bedrock main`
3. **Check that dev deployment is green:**
    1. View the Integration Tests Github Action and look at the run labelled `Run Integration tests for main`
4. Check that stage deployment is also green (`Run Integration tests for stage`)
5. Tag and push the deployment by running `bin/tag-release.sh --push`

Note: By default the `tag-release.sh` script will push to the `origin` git remote. If you'd like for it to push to a different remote name you can either pass in a `-r` or `--remote` argument, or set the `MOZ_GIT_REMOTE` environment variable. So the following are equivalent:

```
$ bin/tag-release.sh --push -r mozilla
```

```
$ MOZ_GIT_REMOTE=mozilla bin/tag-release.sh --push
```

And if you'd like to just tag and not push the tag anywhere, you may omit the `--push` parameter.

1.5.3 What Is Currently Deployed?

You can look at the git log of the `main` branch to find the last commit with a date-tag on it (e.g. 2022-05-05): this commit will be the last one that was deployed to production. You can also use the whatsdeployed.io service to get a nice view of what is actually currently deployed to Dev, Stage, and Prod:

1.5.4 Instance Configuration & Switches

We have a separate repo for configuring our primary instances (dev, stage, and prod). The docs for updating configurations in that repo are on their own page, but there is a way to tell what version of the configuration is in use on any particular instance of bedrock. You can go to the `/healthz-cron/` URL on an instance (see prod for example) to see the current commit of all of the external Git repos in use by the site and how long ago they were updated. The info on that page also includes the latest version of the database in use, the git revision of the bedrock code, and how long ago the database was updated. If you recently made a change to one of these repos and are curious if the changes have made it to production, this is the URL you should check.

1.5.5 Updating Selenium

There are several components for Selenium, which are independently versioned. The first is the Python client, and this can be updated via the test dependencies. The other components are the Selenium versions used in both SauceLabs and the local Selenium grid. These versions are selected automatically based on the required OS / Browser configuration, so they should not need to be updated or specified independently.

1.5.6 Adding test runs

Test runs can be added by creating a new job in `bedrock/.github/workflows/integration_tests.yml` with the desired variables and pushing that branch to Github. For example, if you wanted to run the smoke tests in IE10 (using Saucelabs) you could add the following clause to the matrix:

```
- LABEL: test-ie10-saucelabs
  BROWSER_NAME: internet explorer
  BROWSER_VERSION: "10.0"
  DRIVER: SauceLabs
  PYTEST_PROCESSES: "8"
  PLATFORM: Windows 8
  MARK_EXPRESSION: smoke
```

You can use Sauce Labs platform configurator to help with the parameter values.

1.5.7 Pushing to the integration tests branch

If you have commit rights to our Github repo (mozilla/bedrock) you can simply push your branch to the branch named `run-integration-tests`, and the app will be deployed and the full suite of integration tests for that branch will be run. Please announce in our Slack channel (#www on mozilla.slack.com) that you'll be doing this so that we don't get conflicts. Also remember that you'll likely need to force push, as there may be commits on that branch which aren't in yours – so, if you have the `mozilla/bedrock` remote set as `mozilla`:

```
$ git push -f mozilla $(git branch --show-current):run-integration-tests
```

1.6 Front-end testing

Bedrock runs several different types of front-end tests to ensure that the site is working correctly and that new changes don't break existing functionality.

- Jasmine unit/behavioral tests are used to test JavaScript code that runs in the browser. These tests are run against both Firefox and Chrome browsers via a GitHub action, which is triggered against all pull requests and commits to the main branch.
- Playwright integration tests are used to run end-to-end tests in a real browser environment. These tests are run automatically as part of our CI deployment process against dev, stage, and prod. Playwright tests are run against Firefox, Chromium, and Webkit headless browsers for cross-engine coverage.
- Axe tests are used to test for accessibility issues on key pages. These tests are not run as part of our CI deployment process as they can contain a lot of information, but instead run once per day via a GitHub action against dev. Axe tests are run via Playwright as a subset of tests using the `@a11y` tag. Accessibility issues are reported in the GitHub action output, which can be downloaded and reviewed.
- Selenium tests are bedrock's older, legacy integration test suite, which will eventually be replaced by Playwright. These tests are run against Firefox, Chrome, and Internet Explorer (via a mix of both a local Selenium Grid and Sauce Labs) as part of our CI pipeline, and run alongside the Playwright tests.

Note: New integration tests should be written using Playwright, but we will continue to run the Selenium tests until they are all migrated over. We will also eventually retire the Internet Explorer tests.

The test specs for all of the above suites can be found in the root `./tests` directory:

- `./tests/unit/` for Jasmine tests.
- `./tests/playwright/` Playwright tests.
- `./tests/playwright/specs/a11y/` Axe tests.
- `./tests/functional/` for Selenium tests.

1.6.1 Automating the browser

Jasmine, Playwright and Selenium all require a browser to run. In order to automate browsers such as Firefox and Chrome, you may also need to have the appropriate drivers installed. To download `geckodriver` and `chromedriver` and have them ready to run in your system, there are a couple of ways:

Download geckodriver and add it to your system path:

```
$ cd /path/to/your/downloaded/files/
```

```
$ mv geckodriver /usr/local/bin/
```

If you're on MacOS, download `geckodriver` directly using Homebrew, which automatically places it in your system path:

```
$ brew install geckodriver
```

Download chromedriver and add it to your system path:

```
$ cd /path/to/your/downloaded/files/
```

```
$ mv chromedriver /usr/local/bin/
```

If you're on MacOS, download `chromedriver` directly using Homebrew/Cask, which automatically places it in your system path:

```
$ brew tap homebrew/cask
```

```
$ brew cask install chromedriver
```

1.6.2 Running Jasmine tests

Jasmine and its dependencies are installed via npm and are included when you run `make preflight` to install bedrock's main dependencies.

```
$ make preflight
```

Next, make sure you activate your bedrock virtual env.

```
$ pyenv activate bedrock
```

You can then run the full suite of Jasmine tests with the following command:

```
$ npm run test
```

This will also run all our front-end linters and formatting checks before running the Jasmine test suite. If you only want to run the tests themselves, you can run:

```
$ npm run jasmine
```

Writing Jasmine tests

See the Jasmine documentation for tips on how to write JS behavioral or unit tests. We also use Sinon for creating test spies, stubs and mocks.

Debugging Jasmine tests

If you need to debug Jasmine tests, you can also run them interactively in the browser using the following command:

```
$ npm run jasmine-serve
```

1.6.3 Running Playwright tests

Playwright test dependencies are installed via NPM but are not included in the `make preflight` command along with bedrock's core dependencies. This is because the dependencies are not required to run the site, and also include several large binary files for each headless browser engine.

To install the Playwright dependencies, run the following command:

```
$ cd tests/playwright && npm install && npm run install-deps
```

Specifying an environment

By default Playwright tests will run against `http://localhost:8000/`. Remember to have your development server running before starting the test suite locally.

You can also set `PLAYWRIGHT_BASE_URL` in your `.env` to point to a different environment, such as dev or stage. For example:

```
PLAYWRIGHT_BASE_URL=https://dev.bedrock.nonprod.webservices.mozgcp.net
```

Running the test suite

To run the full suite of tests (from the `/tests/playwright/` directory):

```
$ npm run integration-tests
```

This will run all tests against three different headless browser engines (Firefox, Chromium, WebKit).

Running specific tests

Tests are grouped using tags, such as `@mozorg`, `@firefox`, `@vpn`. To run only one group of tests, instead of the whole suite, you can use `--grep`:

```
$ npx playwright test --grep @firefox
```

To run only a specific test file, such as `firefox-new.spec.js`, you can pass the filename:

```
$ npx playwright test firefox-new
```

See the Playwright CLI docs for more options when running and debugging tests.

Writing tests

Playwright test spec files are found in the `./tests/playwright/specs/` directory.

See the Playwright docs for more examples on how to write tests.

Guidelines for writing integration tests

- Try and keep tests focused on user journeys and key functionality. For example, a test for the download page should focus on the download button, and not the footer or header.
- Test expected functionality from a **user perspective**. For example, if a user clicks a button, the test should check that the expected action occurs. Try to avoid testing implementation details, as these are both invisible to the user and likely to change more frequently.
- Keep tests organized and cleanly separated. Each page should have its own test spec file, and each test should be responsible for a specific purpose, or component of a page.
- Try and use `data-testid` attributes for a locator strategy, as these are less likely to change than other attributes.
- Avoid string checking as tests may break if strings are updated, or could change depending on the page locale.

- When writing tests, push them to the `run-integration-tests` branch to run them against the deployed environment prior to merging to `main`. This will allow you to catch any issues that may not be present in local testing. It's also worth running tests a few times to identify any potential intermittent failures.

User Agent string overrides

Playwright tests use User Agent string overrides to mock different browser and operating systems combinations. By default tests run with User Agent strings configured for Firefox and Chrome running on Windows, and Safari running on macOS. This is accomplished using an `OpenPage()` helper that can be imported in each test file:

```
const openPage = require('../scripts/open-page');
const url = '/en-US/firefox/new/';

test.beforeEach(async ({ page, browserName }) => {
    await openPage(url, page, browserName);
});
```

To mock a different browser or operating system combination, tests can manually load a different override instead of using `openPage`:

```
test.beforeEach(async ({ page, browserName }) => {
    if (browserName === 'webkit') {
        // Set macOS 10.14 UA strings.
        await page.addInitScript({
            path: `./scripts/useragent/mac-old/${browserName}.js`
        });
    } else {
        // Set Windows 8.1 UA strings (64-bit).
        await page.addInitScript({
            path: `./scripts/useragent/win-old/${browserName}.js`
        });
    }

    await page.goto(url + '?automation=true');
});
```

1.6.4 Accessibility testing (Axe)

Axe tests are run as a subset of Playwright tests using the `@a11y` tag. These tests are run against the dev environment once per day via a GitHub action. The axe spec files can be found in the `./tests/playwright/specs/a11y/` directory.

To run the Axe tests locally, you can use the following command from the `./tests/playwright/` directory:

```
$ npm run a11y-tests
```

The Axe tests consist of two different test types. One that scans an entire page for accessibility issues, and another that scans a specific element for issues (such as the main navigation and footer). These tests can also be run at both desktop and mobile viewport sizes.

Test results are output to the console, and any issues found will be created as HTML report files in the `./tests/playwright/test-results-a11y/` directory. When run via the GitHub action, the reports are also output to the annotation logs for each test job.

A list of all the Axe rules that are checked by the tests can be viewed in the axe-core repo.

1.6.5 Running Selenium tests

Note: Selenium tests are being retired in favour of the newer Playwright test suite. Whilst we will continue to run the Selenium tests until they are all migrated over, new tests should be written using Playwright.

Before running the Selenium tests, please make sure to follow the bedrock *installation docs*, including the database sync that is needed to pull in external data such as event/blog feeds etc. These are required for some of the tests to pass.

To run the full Selenium integration test suite against your local bedrock instance:

```
$ pytest --base-url http://localhost:8000 --driver Firefox --html tests/functional/
↪results.html tests/functional/
```

This will run all test suites found in the `tests/functional` directory and assumes you have bedrock running at `localhost` on port `8000`. Results will be reported in `tests/functional/results.html`.

Note: If you omit the `--base-url` command line option then a local instance of bedrock will be started, however the tests are not currently able to run against bedrock in this way.

By default, tests will run one at a time. This is the safest way to ensure predictable results, due to bug 1230105. If you want to run tests in parallel (this should be safe when running against a deployed instance), you can add `-n auto` to the command line. Replace `auto` with an integer if you want to set the maximum number of concurrent processes.

Note: There are some tests that do not require a browser. These can take a long time to run, especially if they're not running in parallel. To skip these tests, add `-m 'not headless'` to your command line.

To run a single test file you must tell pytest to execute a specific file e.g. `tests/functional/test_newsletter.py`:

```
$ pytest --base-url http://localhost:8000 --driver Firefox --html tests/functional/
↪results.html tests/functional/firefox/new/test_download.py
```

To run a single test you can filter using the `-k` argument supplied with a keyword e.g. `-k test_download_button_displayed`:

```
$ pytest --base-url http://localhost:8000 --driver Firefox --html tests/functional/
↪results.html tests/functional/firefox/new/test_download.py -k test_download_button_
↪displayed
```

You can also easily run the tests against any bedrock environment by specifying the `--base-url` argument. For example, to run all Selenium integration tests against dev:

```
$ pytest --base-url https://www-dev.allizom.org --driver Firefox --html tests/functional/
↪results.html tests/functional/
```

Note: For the above commands to work, Firefox needs to be installed in a predictable location for your operating system. For details on how to specify the location of Firefox, or running the tests against alternative browsers, refer to the pytest-selenium documentation.

For more information on command line options, see the pytest documentation.

Running tests in Sauce Labs

You can also run tests in Sauce Labs directly from the command line. This can be useful if you want to run tests against Internet Explorer when you're on Mac OSX, for instance.

1. Sign up for an account at https://saucelabs.com/opensauce/.
2. Log in and obtain your Remote Access Key from user settings.
3. Run a test specifying `SauceLabs` as your driver, and pass your credentials.

For example, to run the home page tests using Internet Explorer via Sauce Labs:

```
$ SAUCELABS_USERNAME=thedude SAUCELABS_API_KEY=123456789 SAUCELABS_W3C=true SELENIUM_
↪EXCLUDE_DEBUG=logs pytest --base-url https://www-dev.allizom.org --driver SauceLabs --
↪capability browserName 'internet explorer' --capability platformName 'Windows 10' --
↪html tests/functional/results.html tests/functional/test_home.py
```

Writing Selenium tests

Tests usually consist of interactions and assertions. Selenium provides an API for opening pages, locating elements, interacting with elements, and obtaining state of pages and elements. To improve readability and maintainability of the tests, we use the Page Object model, which means each page we test has an object that represents the actions and states that are needed for testing.

Well written page objects should allow your test to contain simple interactions and assertions as shown in the following example:

```
def test_sign_up_for_newsletter(base_url, selenium):
    page = NewsletterPage(base_url, selenium).open()
    page.type_email("noreply@mozilla.com")
    page.accept_privacy_policy()
    page.click_sign_me_up()
    assert page.sign_up_successful
```

It's important to keep assertions in your tests and not your page objects, and to limit the amount of logic in your page objects. This will ensure your tests all start with a known state, and any deviations from this expected state will be highlighted as potential regressions. Ideally, when tests break due to a change in bedrock, only the page objects will need updating. This can often be due to an element needing to be located in a different way.

Please take some time to read over the Selenium documentation for details on the Python client API.

Destructive tests

By default all tests are assumed to be destructive, which means they will be skipped if they're run against a sensitive environment. This prevents accidentally running tests that create, modify, or delete data on the application under test. If your test is nondestructive you will need to apply the `nondestructive` marker to it. A simple example is shown below, however you can also read the pytest markers documentation for more options.

```
import pytest
```

(continues on next page)

(continued from previous page)

```
@pytest.mark.nondestructive
def test_newsletter_default_values(base_url, selenium):
    page = NewsletterPage(base_url, selenium).open()
    assert "" == page.email
    assert "United States" == page.country
    assert "English" == page.language
    assert page.html_format_selected
    assert not page.text_format_selected
    assert not page.privacy_policy_accepted
```

Smoke tests

Smoke tests are considered to be our most critical tests that must pass in a wide range of web browsers, including Internet Explorer 11. The number of smoke tests we run should be enough to cover our most critical pages where legacy browser support is important.

```
import pytest


@pytest.mark.smoke
@pytest.mark.nondestructive
def test_download_button_displayed(base_url, selenium):
    page = DownloadPage(selenium, base_url, params="").open()
    assert page.is_download_button_displayed
```

You can run smoke tests only by adding `-m smoke` when running the test suite on the command line.

Waits and Expected Conditions

Often an interaction with a page will cause a visible response. While Selenium does its best to wait for any page loads to be complete, it's never going to be as good as you at knowing when to allow the test to continue. For this reason, you will need to write explicit waits in your page objects. These repeatedly execute code (a condition) until the condition returns true. The following example is probably the most commonly used, and will wait until an element is considered displayed:

```
from selenium.webdriver.support import expected_conditions as expected
from selenium.webdriver.support.ui import WebDriverWait as Wait

Wait(selenium, timeout=10).until(
    expected.visibility_of_element_located(By.ID, "my_element")
)
```

For convenience, the Selenium project offers some basic expected conditions, which can be used for the most common cases.

Debugging Selenium

Debug information is collected on failure and added to the HTML report referenced by the `--html` argument. You can enable debug information for all tests by setting the `SELENIUM_CAPTURE_DEBUG` environment variable to `always`.

1.6.6 Testing Basket email forms

When writing integration tests for front-end email newsletter forms that submit to Basket, we have some special case email addresses that can be used just for testing:

1. Any newsletter subscription request using the email address "success@example.com" will always return success from the basket client.
2. Any newsletter subscription request using the email address "failure@example.com" will always raise an exception from the basket client.

Using the above email addresses enables newsletter form testing without actually hitting the Basket instance, which reduces automated newsletter spam and improves test reliability due to any potential network flakiness.

Headless tests

There are targeted headless tests for the download pages. These tests and are run as part of the pipeline to ensure that download links constructed via product details are well formed and return valid 200 responses.

1.7 Managing Redirects

We have a redirects app in bedrock that makes it easier to add and manage redirects. Due to the size, scope, and history of mozilla.org we have quite a lot of redirects. If you need to add or manage redirects read on.

1.7.1 Add a redirect

You should add redirects in the app that makes the most sense. For example, if the source URL is `/firefox/...` then the `bedrock.firefox` app is the best place. Redirects are added to a `redirects.py` file within the app. If the app you want to add redirects to doesn't have such a file, you can create one and it will automatically be discovered and used by bedrock as long as said app is in the `INSTALLED_APPS` setting (see `bedrock/mozorg/redirects.py` as an example).

Once you decide where it should go you can add your redirect. To do this you simply add a call to the `bedrock.redirects.util.redirect` helper function in a list named `redirectpatterns` in `redirects.py`. For example:

```
from bedrock.redirects.util import redirect


redirectpatterns = [
    redirect(r"^rubble/barny/$", "/flintstone/fred/"),
]
```

This will make sure that requests to `/rubble/barny/` (or with the locale like `/pt-BR/rubble/barny/`) will get a 301 response sending users to `/flintstone/fred/`.

The `redirect()` function has several options. Its signature is as follows:

```
def redirect(
    pattern,
    to,
    permanent=True,
    locale_prefix=True,
    anchor=None,
    name=None,
    query=None,
    vary=None,
    cache_timeout=12,
    decorators=None,
    re_flags=None,
    to_args=None,
    to_kwargs=None,
    prepend_locale=True,
    merge_query=False,
):
    """
    Return a url matcher suited for urlpatterns.

    pattern: the regex against which to match the requested URL.
    to: either a url name that `reverse` will find, a url that will simply be returned,
        or a function that will be given the request and url captures, and return the
        destination.
    permanent: boolean whether to send a 301 or 302 response.
    locale_prefix: automatically prepend `pattern` with a regex for an optional locale
        in the URL. This locale (or None) will show up in captured kwargs as 'locale'.
    anchor: if set it will be appended to the destination URL after a '#'.
    name: if used in a `urls.py` the redirect URL will be available as the name
        for use in calls to `reverse()`. Does _NOT_ work if used in a `redirects.py` file.
    query: a dict of query params to add to the destination URL.
    vary: if you used an HTTP header to decide where to send users you should include␣
↪that
        header's name in the `vary` arg.
    cache_timeout: number of hours to cache this redirect. just sets the proper `cache-
↪control`
        and `expires` headers.
    decorators: a callable (or list of callables) that will wrap the view used to␣
↪redirect
        the user. equivalent to adding a decorator to any other view.
    re_flags: a string of any of the characters: "iLmsux". Will modify the `pattern` regex
        based on the documented meaning of the flags (see python re module docs).
    to_args: a tuple or list of args to pass to reverse if `to` is a url name.
    to_kwargs: a dict of keyword args to pass to reverse if `to` is a url name.
    prepend_locale: if true the redirect URL will be prepended with the locale from the
        requested URL.
    merge_query: merge the requested query params from the `query` arg with any query␣
↪params
        from the request.

    Usage:
    urlpatterns = [
        redirect(r'projects/$', 'mozorg.product'),
```

(continues on next page)

(continued from previous page)

```
        redirect(r'^projects/seamonkey$', 'mozorg.product', locale_prefix=False),
        redirect(r'apps/$', 'https://marketplace.firefox.com'),
        redirect(r'firefox/$', 'firefox.new', name='firefox'),
        redirect(r'the/dude$', 'abides', query={'aggression': 'not_stand'}),
    ]
    """
```

1.7.2 Differences

This all differs from `urlpatterns` in `urls.py` files in some important ways. The first is that these happen first. If something matches in a `redirects.py` file it will always win the race if another URL in a `urls.py` file would also have matched. Another is that these are matched before any locale prefix stuff happens. So what you're matching against in the redirects files is the original URL that the user requested. By default (unless you set `locale_prefix=False`) your patterns will match either the plain URL (e.g. `/firefox/os/`) or one with a locale prefix (e.g. `/fr/firefox/os/`). If you wish to include this locale in the destination URL you can simply use python's string `format()` function syntax. It is passed to the `format` method as the keyword argument `locale` (e.g. `redirect('^stuff/$', '{locale}whatnot/')`). If there was no locale in the URL the `{locale}` substitution will be an empty string. Similarly if you wish to include a part of the original URL in the destination, just capture it with the regex using a named capture (e.g. `r'^stuff/(?P<rest>.*)$'` will let you do `'/whatnot/{rest}'`).

1.7.3 Utilities

There are a couple of utility functions for use in the `to` argument of `redirect` that will return a function to allow you to match something in an HTTP header.

ua_redirector

`bedrock.redirects.util.ua_redirector` is a function to be used in the `to` argument that will use a regex to match against the `User-Agent` HTTP header to allow you to decide where to send the user. For example:

```
from bedrock.redirects.util import redirect, ua_redirector


redirectpatterns = [
    redirect(
        r"^rubble/barny/$",
        ua_redirector("firefox(os)?", "/firefox/", "/not-firefox/"),
        cache_timeout=0,
    ),
]
```

You simply pass it a regex to match, the destination URL (substitutions from the original URL do work) if the regex matches, and another destination URL if the regex does not match. The match is not case sensitive unless you add the optional `case_sensitive=True` argument.

Note: Be sure to include the `cache_timeout=0` so that you won't be bitten by any caching proxies sending all users one way or the other. Do not set the `Vary: User-Agent` header; this will not work in production.

---

header_redirector

This is basically the same as `ua_redirector` but works against any header. The arguments are the same as above except that thre is an additional first argument for the name of the header:

```
from bedrock.redirects.util import redirect, header_redirector


redirectpatterns = [
    redirect(
        r"^rubble/barny/$",
        header_redirector("cookie", "been-here", "/firefox/", "/firefox/new/"),
        vary="cookie",
    ),
]
```

1.7.4 Testing redirects

A suite of tests exists for redirects, which is intended as a reference of the redirects we expect to work on www.mozilla.org. This will become a base for implementing these redirects in the bedrock app and allow us to test them before release.

Installation

First follow the *installation instructions for bedrock*, which will guide you through installing pip and setting up a virtual environment for the tests. The additional requirements can then be installed by using the following commands:

```
$ source venv/bin/activate
```

```
$ pip install -r requirements/dev.txt
```

Running the tests

If you wish to run the full set of tests, which requires a deployed instance of the site (e.g. www.mozilla.org) you can set the `--base-url` command line option:

```
$ pytest --base-url https://www.mozilla.org tests/redirects/
```

By default, tests will run one at a time. If you intend to run the suite against a remote instance of the site (e.g. production) it will run a lot quicker by running the tests in parallel. To do this, you can add `-n auto` to the command line. Replace `auto` with an integer if you want to set the maximum number of concurrent processes.

1.8 Newsletters

Bedrock includes support for signing up for and managing subscriptions and preferences for Mozilla newsletters.

Many pages have a form to sign-up for the default newsletters, "Mozilla Foundation" and "Firefox & You". Other pages have more specific sign up forms, such as the contribute page, or Mozilla VPN wait-list page.

1.8.1 Features

- Ability to subscribe to a newsletter from a web form. Many pages on the site might include this form.
- Whole pages devoted to subscribing to one newsletter, often with custom text, branding, and layout.
- Newsletter preference center - allow user to change their email preferences (e.g. language, HTML vs. text), as well as which newsletters they're subscribed to, etc. Access is limited by requiring a user-specific token in the URL (it's a UUID). The full URL is included as a link in each newsletter sent to the user. Users can also recover a link to their token by visiting the newsletter recovery page and entering their email address.

1.8.2 Newsletters

Newsletters have a variety of characteristics. Some of these are implemented in Bedrock, others are transparent to Bedrock but implemented in the basket back-end that provides our interface to the newsletter vendor.

- Public name - the name that is displayed to users, e.g. "Firefox Weekly Tips".
- Internal name - a short string that is used internal to Bedrock and basket to identify a newsletter. Typically these are lowercase strings of words joined by hyphens, e.g. "firefox-tips". This is what we send to basket to identify a newsletter, e.g. to subscribe a user to it.
- Show publicly - pages like the newsletter preferences center show a list of unsubscribed newsletters and allow subscribing to them. Some newsletters aren't included in that list by default (though they are shown if the user is already subscribed, to let them unsubscribe). If the user has a Mozilla account, there are also some other related newsletters that will always be shown in the list.
- Languages - newsletters are available in a particular set of languages. Typically when subscribing to a newsletter, a user can choose their preferred language. We should try not to let them subscribe to a newsletter in a language that it doesn't support.

  The backend only stores one language for the user though, so whenever the user submits one of our forms, whatever language they last submitted is what is saved for their preference for everything.
- Welcome message - each newsletter can have a canned welcome message that is sent to a user when they subscribe to it. Newsletters should have both an HTML and a text version of this.
- Drip campaigns - some newsletters implement so-called drip campaigns, in which a series of canned messages are dribbled out to the user over a period of time. E.g. 1 week after subscribing, they might get message 1; a week later, message 2, and so on until all the canned messages have been sent.

  Because drip campaigns depend on the sign-up date of the user, we're careful not to accidentally change the sign-up date, which could happen if we sent redundant subscription commands to our backend.

1.8.3 Bedrock and Basket

Bedrock is the user-facing web application. It presents an interface for users to subscribe and manage their subscriptions and preferences. It does not store any information. It gets all newsletter and user-related information, and makes updates, via web requests to the Basket server. These requests are made typically made by Bedrock's front-end JavaScript modules.

The Basket server implements an HTTP API for the newsletters. The front-end (Bedrock) can make calls to it to retrieve or change users' preferences and subscriptions, and information about the available newsletters. Basket implements some of that itself, and other functions by calling the newsletter vendor's API. Details of that are outside the scope of this document, but it's worth mentioning that both the user token (UUID) and the newsletter internal name mentioned above are used only between Bedrock and Basket.

See the Basket docs for more information.

1.8.4 URLs

Here are a few important mozorg newsletter URLs. Some of these were established before Bedrock came along, and so are unlikely to be changed.

- `/newsletter/` - Subscribe to 'mozilla-and-you' newsletter (public name: "Firefox & You")
- `/newsletter/existing/{USERTOKEN}/` - User management of their preferences and subscriptions.
- `/newsletter/confirm/{USERTOKEN}/` - URL someone lands on when they confirm their email address after initially subscribing.
- `/newsletter/country/{USERTOKEN}/` - Allows users to change their country.
- `/newsletter/recovery/` - Allows users to recover a link containing their token so they can manage their subscriptions.
- `/newsletter/updated/` - A page users are redirected to after updating their details, or unsubscribing.

Note: URLs that contain `{USERTOKEN}` will have their path rewritten on page load so that they no longer contain the token e.g. `/newsletter/existing/{USERTOKEN}/` will be rewritten to just `/newsletter/existing/`. This helps to prevent accidental sharing of user tokens in URLS and also against referral information leakage.

1.8.5 Footer sign-up

In some common templates, you can customize the footer sign-up form by overriding the email_form template block. For example, to have no sign-up form:

```
{% block email_form %}{% endblock %}
```

The default is:

```
{% block email_form %}{{ email_newsletter_form() }}{% endblock %}
```

This will render a sign-up for "Firefox & You". You can pass parameters to the macro `email_newsletter_form` to change that. For example, the `newsletters` parameter controls which newsletter is signed up for, and `title` can override the text:

```
{% block email_form %}
    {{ email_newsletter_form('app-dev',
                             'Sign up for more news about the Firefox Marketplace.') }}
{% endblock %}
```

The *newsletters* parameter, the first positional argument, can be either a list of newsletter IDs or a comma separated list of newsletters IDs:

```
{% block email_form %}
    {{ email_newsletter_form('mozilla-foundation, mozilla-and-you') }}
{% endblock %}
```

Pages can control whether country or language fields are included by passing `include_language=[True|False]` and/or `include_country=[True|False]`.

1.9 Contentful CMS (Content Management System) Integration (Deprecated)

Important: We are no longer syncing content from Contentful, but we still hold that content frozen in our database and use it to render pages.

Pages previously managed with Contentful will be the first pages to be (re)implemented using our upcoming part-of-Bedrock CMS system. At that point, we will remove all Contentful-related code from the codebase.

In the meantime, if content changes are needed to pages formerly managed via Contentful, we can do this via data migration – just ask the backend team.

Please do not add new pages to Bedrock using Contentful.

1.9.1 Overview

Contentful is a headless CMS. It stores content for our website in a structured format. We request the content from Contentful using an API. Then the content gets made into Protocol components for display on the site.

We define the structure Contentful uses to store the data in **content models**. The content models are used to create a form for editors to fill out when they want to enter new content. Each chunk of content is called an **entry**.

For example: we have a content model for our "card" component. That model creates a form with fields like heading, link, blurb, and image. Each card that is created from the model is its own entry.

We have created a few different types of content models. Most are components that correspond to components in our design system. The smallest create little bits of code like buttons. The larger ones group together several entries for the smaller components into a bigger component or an entire page.

For example: The *Page: General* model allows editors to include a hero entry, body entry, and callout entry. The callout layout entry, in turn, includes a CTA (Call To Action) entry.

One advantage of storing the content in small chunks like this is that is can be reused in many different pages. A callout which focuses on the privacy related reasons to download Firefox could end up on the Private Browsing, Ad Tracker Blocking, and Fingerprinter Blocking pages. If our privacy focused tagline changes from "Keep it secret with Firefox" to "Keep it private with Firefox" it only needs to be updated in one entry.

So, when looking at a page on the website that comes from Contentful you are typically looking at several different entries combined together.

On the bedrock side, the data for all entries is periodically requested from the API and stored in a database.

When a Contentful page is requested the code in *api.py* transforms the information from the database into a group of Python dictionaries (these are like key/value pairs or an object in JS).

This data is then passed to the page template (either Mozilla or for Firefox themed as appropriate). The page template includes some files which take the data and feed it into macros to create Protocol components. These are the same macros we use on non-Contentful pages. There are also includes which will import the appropriate JS and CSS files to support the components.

Once rendered the pages get cached on the CDN as usual.

1.9.2 Contentful Apps

Important: We are no longer syncing content from Contentful – see the note at the top of this page.

Please do not add new pages to Bedrock using Contentful.

Installed on Environment level. Make sure you are in the environment you want to edit before accessing an app. Use *Apps* link in top navigation of Contentful Web App to find an environment's installed apps.

Compose

Compose provides a nicer editing experience. It creates a streamlined view of pages by combining multiple entries into a single edit screen and allowing field groups for better organization.

Any changes made to Compose page entries in a specific environment are limited to that environment. If you are in a sandbox environment, you should see an `/environments/sandbox-name` path at the end of your Compose URL.

Known Limitations

- Comments are not available on Compose entries
- It is not possible to edit embedded entries in Rich Text fields in Compose app. Selecting the "edit" option in the dropdown opens the entry in the Contentful web app.

Merge

Merge provides a UI for comparing the state of Content Models across two environments. You can select what changes you would like to migrate to a new environment.

Known Limitations

- Does not migrate Help Text (under Appearance Tab)
- Does not migrate any apps used with those Content Models
- Does not migrate Content Entries or Assets
- It can identify when Content Models should be available in Compose, but it cannot migrate the field groups

Others

- Launch allows creation of "releases", which can help coordinate publishing of multiple entries
- Workflows standardizes process for a specific Content Model. You can specify steps and permissions to regulate how content moves from draft to published.

1.9.3 Content Models

Emoji legend for content models

- this component is a page, it will include meta data for the page, a folder, and slug
- this is a layout wrapper for another component
- this component includes editable content, not just layout config
- this component is suitable for inclusion as an inline entry in a rich text field
- this component can be embedded without a layout wrapper

Naming conventions for content models

Note: For some fields it is important to be consistent because of how they are processed in bedrock. For all it is important to make the editor's jobs easier.

Name
: This is for the internal name of the entry. It should be set as the **Entry title**, required, and unique.

Preview (and Preview Title, Preview Blurb, Preview Image)
: These will be used in search results and social media sites. There's also the potential to use them for aggregate pages on our own sites. Copy configuration and validation from an existing page.

Heading (and Heading Level)
: Text on a page which provides context for information that follows it. Usually made into a H1-H4 in bedrock. Not: header, title, or name.

Image (and Image Size, Image Width)
: Not: picture, photo, logo, or icon (unless we are specifically talking about a logo or icon.)

Content
: Multi-reference

Product Icon
: Copy configuration and validation from an existing page.

Theme
Copy configuration and validation from an existing page.

Body (Body Width, Body Vertical Alignment, Body Horizontal Alignment)
Rich text field in a Component. Do not use this for multi reference fields, even if the only content on the page is other content entries. Do not use MarkDown for body fields, we can't restrict the markup. Copy configuration and validation from an existing page.

Rich Text Content
Rich text field in a Compose Page

CTA
The button/link/dropdown that we want a user to interact with following some content. Most often appearing in Split and Callout components.

Page

Pages in bedrock are created from page entries in Contentful's Compose App.

Homepage
The homepage needs to be connected to bedrock using a Connect component (see *Legacy*) and page meta data like title, blurb, image, etc come from bedrock.

General
Includes hero, text, and callout. The simplified list and order of components is intended to make it easier for editors to put a page together.

Versatile
No pre-defined template. These pages can be constructed from any combination of layout and component entries.

Resource Center
Includes product, category, tags, and a rich text editor. These pages follow a recognizable format that will help orient users looking for more general product information (i.e. VPN).

The versatile and general templates do not need bedrock configuration to be displayed. Instead, they should appear automatically at the folder and slug specified in the entry. These templates do include fields for meta data.

Layout

These entries bring a group of components together. For example: 3 picto blocks in a picto block layout. They also include layout and theme options which are applied to all of the components they bring together. For example: centering the icons in all 3 picto blocks.

These correspond roughly to Protocol templates.

The one exception to the above is the Layout: Large Card, which exists to attach a large display image to a regular card entry. The large card must still be included in the Layout: 5 Cards.

Component

We're using this term pretty loosely. It corresponds roughly to a Protocol atom, molecule, or organism.

These entries include the actual content, the bits that people write and the images that go with it.

If they do not require a layout wrapper there may also be some layout and theme options. For example, the text components include options for width and alignment.

Embed

These pre-configured content pieces can go in rich text editors when allowed (picto, split, multi column text…).

Embeds are things like logos, where we want tightly coupled style and content that will be consistent across entries. If a logo design changes, we only need to update it in one place, and all uses of that embed will be updated.

Adding a new Page

- Create the content model
  - Ensure the content model name starts with page (i.e. pageProductJournalismStory)
  - Add an SEO reference field which requires the **SEO Metadata** content type
  - In Compose, go to Page Types and click "Manage Page Types" to make your new content model available to the Compose editor.
    - If you have referenced components, you can choose whether they will be displayed as expanded by default.
    - Select "SEO" field for "Page Settings" field
  - If the page is meant to be localised, ensure all fields that need localisation have the "Enable localization of this field" checkbox checked in content model field settings
- Update `bedrock/contentful/constants`
  - Add content type constant
  - Add constant to default array
  - If page is for a single locale only, add to SINGLE_LOCALE_CONTENT_TYPES
  - If page is localised, add to LOCALISATION_COMPLETENESS_CHECK_CONFIG with an array of localised fields that need to be checked before the page's translation can be considered complete
- Update `bedrock/contentful/api.py`
  - If you're adding new embeddable content types, expand list of renderer helpers configured for the RichTextRenderer in the `ContentfulAPIWrapper`
  - Update `ContentfulAPIWrapper.get_content()` to have a clause to handle the new page type
- Create a custom view to pass the Contentful data to a template

Adding a new Component

Example: Picto

1. Create the content model in Contentful.
    - *Follow the naming conventions*.
    - You may need two models if you are configuring layout separately.
2. Add the new content model to the list of allowed references in other content models (At the moment this is just the "content" reference field on pages).
3. In bedrock create CSS and JS entries in static-bundles for the new component.
4. In api.py write a def for the component.
5. In api.py add the component name, def, and bundles to the CONTENT_TYPE_MAP.
6. Find or add the macro to macros-protocol.
7. Import the macro into all.html and add a call to it in the entries loop.

Note: Tips:

- can't define defaults in Contentful, so set those in your Python def.
- for any optional fields make sure you check the field exists before referencing the content.

Adding a new Embed

Example: Wordmark.

1. Create the content model in Contentful.
    - *Follow the naming conventions*.
2. Add the new content model to rich text fields (like split and text).
3. In bedrock include the CSS in the Sass file for any component which may use it (yeah, this is not ideal, hopefully we will have better control in the future).
4. Add a def to api.py to render the piece (like `_make_wordmark`).

Note: Tips:

- can't define defaults in Contentful, so set those in your Python def.
- for any optional fields make sure you check the field exists before referencing the content.

Adding a rich text field in a component

Disable everything then enable: B, I, UL, OL, Link to URL, and Inline entry. You will want to enable some some Headings as well, H1 should be enabled very rarely. Enable H2-H4 using your best judgement.

1.9.4 Adding support for a new product icon, size, folder

Many content models have drop downs with identical content. For example: the Hero, Callout, and Wordmark models all include a "product icon". Other common fields are width and folder.

There are two ways to keep these lists up to date to reflect Protocol updates:

1. By opening and editing the content models individually in Contentful
2. Scripting updates using the API

At the moment it's not too time consuming to do by hand, just make sure you are copy and pasting to avoid introducing spelling errors.

We have not tried scripting updates with the API yet. One thing to keep in mind if attempting this is that not all widths are available on all components. For example: the "Text: Four columns" component cannot be displayed in small content widths.

1.9.5 Rich Text Rendering

Contentful provides a helper library to transform the rich text fields in the API into HTML content.

In places were we disagree with the rendering or want to enhance the rendering we can provide our own renderers on the bedrock side. They can be as simple as changing <*b*> tags to <*strong*> tags or as complex as inserting a component.

A list of our custom renderers is passed to the *RichTextRenderer* helper at the start of the *ContentfulPage* class in api.py. The renderers themselves are also defined in api.py

---

Note:

- Built-in nodes cannot be extended or customized: *Custom node types and marks are not allowed*. Embed entry types are required to extend rich text functionality. (i.e. if you need more than one style of blockquote)

1.9.6 L10N

Important: We are no longer syncing content from Contentful – see the note at the top of this page.

Please do not add new pages to Bedrock using Contentful.

Smartling - our selected approach

When setting up a content model in Contentful, fields can be designated as available for translation.

Individual users can be associated with different languages, so when they edit entries they see duplicate fields for each language they can translate into. In addition - and in the most common case - these fields are automatically sent to Smartling to be translated there.

Once text for translation lands in Smartling, it is batched up into jobs for human translation. When the work is complete, Smartling automatically updates the relevant Contentful entries with the translations, in the appropriate fields.

Note that those translations are only visible in Contentful if you select to view that locale's fields, but if they are present in Contentful's datastore (and that locale is enabled in the API response) they will be synced down by Bedrock.

On the Bedrock side, the translated content is pulled down the same way as the default locale's content is, and is stored in a locale-specific ContentfulEntry in the database.

In terms of 'activation', or "Do we have all the parts to show this Contentful content"?, Contentful content is not evaluated in the same way as Fluent strings (where we will show a page in a given locale if 80% of its Fluent strings have been translated, falling back to en-US where not).

Instead, we check that all of the required fields present in the translated Entry have non-null data, and if so, then the entire page is viable to show in the given locale. (ie, we look at fields, not strings. It's a coarser level of granularity compared to Fluent, because the data is organised differently - most of Contentful-sourced content will be rich text, not individual strings).

The check about whether or not a Contentful entry is 'active' or 'localisation complete' happens during the main sync from Contentful. Note that there is no fallback locale for Contentful content other than a redirect to the en-US version of the page - either the page is definitely available in a locale, or it's not at all available in that locale.

Notes:

- The batching of jobs in Smartling is still manual, even though the data flow is automated. We need to keep an eye on how onerous this is, plus what the cost exposure could be like if we fully automate it.
- The Smartling integration is currently only set to use Mozilla.org's 10 most popular locales, in addition to en-US.
- No localisation of Contentful content happens via Pontoon.
- The Smartling setup is most effectively leveraged with Compose-based pages rather than Connect-based components, and the latter may require some code tweaks.
- Our Compose: SEO field in Contentful is configured for translation (and in use on the VPN Resource Center). All Compose pages require this field. If a Compose page type is *not* meant to be localised, we need to stop these SEO-related fields from going on to Smartling.

Fluent

NB: Not selected for use, but notes retained for reference

Instead of using the language translation fields in Contentful to store translations we could designate one of the locales to contain a fluent string ID. Bedrock could then use the string IDs and the English content to create Fluent files for submission into our current translation system.

Creation of the string IDs could be automated using Contentful's write API.

To give us the ability to use fallback strings the Contentful field could accept a comma separated list of values.

This approach requires significant integration code on the bedrock side but comes with the benefit of using our current translation system, including community contributions.

No English Equivalent

NB: Not selected for use, but notes retained for reference

Components could be created in the language they are intended to display in. The localized content would be written in the English content fields.

The down sides of this are that we do not know what language the components are written in and could accidentally display the wrong language on any page. It also means that localized content cannot be created automatically by English editors and translations would have to be manually associated with URLs.

This is the approach that will likely be used for the German and French homepages since that content is not going to be used on English pages and creating a separate homepage with different components is valuable to the German and French teams.

1.9.7 Assets

Important: We are no longer syncing content from Contentful – see the note at the top of this page.

Please do not add new pages to Bedrock using Contentful.

Images that are uploaded in Contentful will be served to site visitors from the Contentful CDN. The cost of using the CDN are not by request so we don't have to worry about how many times an image will be requested.

Using the Contentful CDN lets us use their Images API to format our images.

In theory, a large high quality image is uploaded in Contentful and then bedrock inserts links to the CDN for images which are cropped to fit their component and resized to fit their place on the page.

Because we cannot rely on the dimensions of the image uploaded to Contentful as a guide for displaying the image - bedrock needs to be opinionated about what size images it requests based on the component and its configuration. For example, hero images are fixed at 800px wide. In the future this could be a user configurable option.

1.9.8 Preview

Important: We are no longer syncing content from Contentful – see the note at the top of this page.

Please do not add new pages to Bedrock using Contentful.

Content previews are configured under *Settings* > *Content preview* on a per-content model basis. At the moment previews are only configured for pages, and display on demo5.

Once the code is merged into bedrock they should be updated to use the dev server.

Specific URLs will only update every 5 minutes as the data is pulled from the API but pages can be previewed up to the second at the *contentful-preview* URL. This preview will include "changed" and "draft" changes (even if there is an error in the data) not just published changes.

For previewing on localhost, see Development Practices, below.

1.9.9 Roles/Permissions

In general we are trusting people to check their work before publishing and very few guard rails have been installed. We have a few roles with different permissions.

Admin

Organization

- Define roles and permission
- Manage users
- Change master and sandbox environment aliases
- Create new environments

Master environment

- Edit content model
- Create, Edit, Publish, Archive, Delete content
- Install/Uninstall apps

Developer

Organization

- Create new environments

Master environment

- Create, Edit, Publish, Archive content

Sandbox environments (any non-master environment)

- Edit content model
- Create, Edit, Publish, Archive, Delete content
- Install/Uninstall apps

Editor (WIP)

Master environment (through Compose)

- Create, Edit, Publish, Archive content

1.9.10 Development practices

Important: We are no longer syncing content from Contentful – see the note at the top of this page.

Please do not add new pages to Bedrock using Contentful.

This section outlines tasks generally required if developing features against Contentful.

Get bedrock set up locally to work with Contentful

In your `.env` file for Bedrock, make sure you have the followign environment variables set up.

- `CONTENTFUL_SPACE_ID` - this is the ID of our Contentful integration
- `CONTENTFUL_SPACE_KEY` - this is the API key that allows you access to our space. Note that two types of key are available: a Preview key allows you to load in draft content; the Delivery key only loads published contnet. For local dev, you want a Preview key.
- `SWITCH_CONTENTFUL_HOMEPAGE_DE` should be set to `True` if you are working on the German Contentful-powered homepage
- `CONTENTFUL_ENVIRONMENT` Contentful has 'branches' which it calls environments. *master* is what we use in production, and *sandbox* is generally what we use in development. It's also possible to reference a specific environment - e.g. `CONTENTFUL_ENVIRONMENT=sandbox-2021-11-02`

To get values for these vars, please check with someone on the backend team.

If you are working on the Contentful Sync backed by the message-queue (and if you don't know what this is, you don't need it for local dev), you will also need to set the following env vars:

- `CONTENTFUL_NOTIFICATION_QUEUE_URL`
- `CONTENTFUL_NOTIFICATION_QUEUE_REGION`
- `CONTENTFUL_NOTIFICATION_QUEUE_ACCESS_KEY_ID`
- `CONTENTFUL_NOTIFICATION_QUEUE_SECRET_ACCESS_KEY`

How to preview your changes on localhost

When viewing a page in Contentful, it's possible to trigger a preview of the draft page. This is typically rendered on www-dev.allizom.org. However, that's only useful for code that's already in `main`. If you want to preview Contentful content on your local machine - e.g. you're working on a feature branch that isn't ready for merging - do the following:

Existing (master) Content Types

In the right-hand sidebar of the editor page in Contentful:

- Find the Preview section
- Select `Change` and pick `Localhost Preview`
- Click `Open preview`

New (non-master) Content Types

In bedrock:

- Update `class ContentfulPreviewView(L10nTemplateView)` in Mozorg Views with a render case for your new content type

In the right-hand sidebar of the editor page in Contentful:

- Click Info tab
- Find `Entry ID` section and copy the value

Manually create preview URL in browser:

- *http://localhost:8000/en-US/contentful-preview/{entry_id}/*

Note that previewing a page will require it to be pulled from Contentful's API, so you will need `CONTENTFUL_SPACE_ID` and `CONTENTFUL_SPACE_KEY` set in your `.env`. It may take a few seconds to get the data.

Also note that when you select `Localhost preview`, the choice sticks, so you should set it back to `Preview on web` when you're done.

How to update/refresh the sandbox environment

It helps to think of Contentful 'environments' as simply branches of a git-like repo full of content. You can take a particular environment and branch off it to make a new environment for WIP (Work in Progress) or experimental content, using the original one as your starting point. On top of this, Contentful has the concept of aliases for environments and we use two aliases in our setup:

- `master` is used for production and is an alias currently pointing to the *V1* environment. It is pretty stable and access to it is limited.
- `sandbox` is used for development and more team members have access to edit content. Again, it's an alias and is pointed at an environment (think, branch) with a name in the format `sandbox-YYYY-MM-DD`.

While updating `master` is something that we generally don't do (at the moment only a product owner and/or admin would do this), updating the sandbox happens more often, typically to populate it with data more recently added to master. To do this:

- Go to `Settings > Environments`
- Ensure we have at least one spare environment slot. If we don't delete the oldest `sandbox-XXXX-XX-XX` environment.
- Click the blue Add Environment button, to the right. Name it using the `sandbox-YYYY-MM-DD` pattern and base it on whatever environment is aliased to `master` - this will basically create a new 'branch' with the content currently in master.
- In the Environment Aliases section of the main page, find *sandbox* and click Change alias target, then select the `sandbox-XXXX-XX-XX` environment you just made.

Which environment is connected to where?

`master` is the environment used in Bedrock production, stage, dev and test `sandbox` may, in the future, be made the default environment for dev. It's also the one we should use for local development.

If you develop a new feature that adds to Contentful (e.g. page or component) and you author it in the sandbox, you will need to re-create it in master before the corresponding bedrock changes hit production.

Troubleshooting

If you run into trouble on an issue, be sure to check in these places first and include the relevant information in requests for help (i.e. environment).

Contentful Content Model & Entries

- What environment are you using?
- Do you have the necessary permissions to make changes?
- Do you see all the entry fields you need? Do those fields have the correct value options?

Bedrock API (api.py)

- What environment are you using?
- Can you find a Python function definition for the content type you need?
- Does it structure data as expected?

```
# example content type def


def get_section_data(self, entry_obj):
    fields = entry_obj.fields()
    # run `print(fields)` here to verify field values from Contentful

    data = {
        "component": "sectionHeading",
        "heading": fields.get("heading"),
    }

    # run `print(data)` here to verify data values from Bedrock API
    return data
```

Bedrock Render (all.html)

- Can you find a render condition for the component you need?

```
/* example component condition */

{% elif entry.component == 'sectionHeading' %}
```

- **If the component calls a macro:**
  - Does it have all the necessary parameters?
  - Is it passing the expected values as arguments?
- **If the component is custom HTML:**
  - Is the HTML structure correct?
  - Are Protocol-specific class names spelled correctly?
- Is the component CSS available?
- Is the component JS available?

Note: Component CSS and JS are defined in a `CONTENT_TYPE_MAP` from the Bedrock API (`api.py`).

Bedrock Database

Once content is synced into your local database, it can be found in the contentful_contentfulentry table. All the dependencies to explore the data are installed by default for local development.

Using sqlite (with an example query to get some info about en-US pages):

```
./manage.py dbshell
```

```
select id, slug, data from contentful_contentfulentry where locale='en-US';
```

Close the sqlite shell with `.exit`

Using Django shell (with an example query to get data from first entry of "pageProductJournalismStory" type):

```
./manage.py shell
```

```
from bedrock.contentful.models import ContentfulEntry

product_stories = ContentfulEntry.objects.filter(
    content_type="pageProductJournalismStory",
    localisation_complete=True,
    locale="en-US",
)

product_stories[0].data  # to see the data stored for the first story in the results
```

Close the Djanjo shell with `exit()` or `CTRL+D`

1.9.11 Useful Contentful Docs

Important: We are no longer syncing content from Contentful – see the note at the top of this page.

Please do not add new pages to Bedrock using Contentful.

---

https://www.contentful.com/developers/docs/references/images-api/#/reference/resizing-&-cropping/specify-focus-area

https://www.contentful.com/developers/docs/references/content-delivery-api/

https://contentful.github.io/contentful.py/#filtering-options

https://github.com/contentful/rich-text-renderer.py https://github.com/contentful/rich-text-renderer.py/blob/a1274a11e65f3f728c278de5d2bac89213b7470e/rich_text_renderer/block_renderers.py

1.9.12 Assumptions we still need to deal with

- image sizes

1.9.13 Legacy

Since we decided to move forward the the Compose App, we no longer need the Connect content model. The EN-US homepage is currently still using Connect. Documentation is here for reference.

- this component is referenced by ID in bedrock (at the moment that is just the homepage but could be used to connect single components for display on non-contentful pages. For example: the latest feature box on /new)

Connect

These are the highest level component. They should be just a name and entry reference.

The purpose of the connect is to create a stable ID that can be referenced in bedrock to be included in a jinja template. Right now we only do this for the homepage. This is because the homepage has some conditional content above and below the Contentful content.

Using a connect component to create the link between jinja template and the Contentful Page entry means an entire new page can be created and proofed in Contentful before the bedrock homepage begins pulling that content in.

In other contexts a connect content model could be created to link to entries where the ID may change. For example: the "Latest Firefox Features: section of /new could be moved to Contentful using a connect component which references 3 picto blocks.

Because the ID must be added to a bedrock by a dev, only devs should be able to make new connect entries.

1.10 CMS

From 2024, Bedrock's CMS will be powered by Wagtail CMS.

This page is currently a skeleton for documentation to be added as the work evolves.

1.10.1 High-level summary

Wagtail CMS will be used to make either entire pages or portions of pages (let's call them 'surfaces') content-editable on www.mozilla.org. It is not a free-for-all add-any-page-you-like approach, but rather a careful rollout of surfaces with appropriate guard rails that helps ensure the integrity, quality and security of www.mozilla.org.

Surfaces will be authored via a closed 'Editing' deployment and when those changes are published they will become visible on the main www.mozilla.org 'Web' deployment.

If you are new to Wagtail, it is recommended that you read the official docs and/or complete an online course to get a better understanding of how Wagtail works.

Useful resources:

- Wagtail Editor Guide.
- Wagtail Docs.
- The Ultimate Wagtail Developers Course.

1.10.2 Accessing the CMS on your local machine

SSO authentication setup

1. Become a member of `bedrock_cms_local_dev-access` on people.mozilla.org. Make sure you remember to accept the email invitation.
2. Extend your `.env` file with the development OIDC credentials that have been supplied to you. (make sure your `.env` file is based on a recent copy of `.env-dist` as several new variables exist).
3. In your `.env` file set the following variables:
   - `USE_SSO_AUTH=True`
   - `WAGTAIL_ENABLE_ADMIN=TRUE`
   - `WAGTAIL_ADMIN_EMAIL=YOUR_MOZILLA_LDAP_EMAIL@mozilla.com`
4. Run `make preflight` to update bedrock with the latest DB version. As part of this step, the make file will also create a local admin user for you, using the Mozilla LDAP email address you added in the previous step. **If you do not want to overwrite your local database, run** `make preflight -- --retain-db` **instead.**
5. Start bedrock running via `npm start` (for local dev) or `make build run` (for Docker).
6. Go to `http://localhost:8000/cms-admin/` and you should see a button to login with SSO. Click it and you should go through the OAuth flow and end up in the Wagtail admin.

Non-SSO authentication

1. In your `.env` file set `USE_SSO_AUTH=False`, and `WAGTAIL_ENABLE_ADMIN=TRUE`.
2. Run `make preflight` to update bedrock with the latest DB version. **If you do not want to overwrite your local database, run** `make preflight -- --retain-db` **instead.**
3. Create a local admin user with `./manage.py createsuperuser`, setting both the username, email and password to whatever you choose (note: these details will only be stored locally on your device).
4. Alternatively, if you define `WAGTAIL_ADMIN_EMAIL=YOUR_MOZILLA_LDAP_EMAIL@mozilla.com` and `WAGTAIL_ADMIN_PASSWORD=somepassword` in your `.env.` file, `make preflight` will automatically create a non-SSO superuser for you
5. Start bedrock running via `npm start` (for local dev) or `make build run` (for Docker).
6. Go to `http://localhost:8000/cms-admin/` and you should see a form for logging in with a username and password. Use the details you created in the previous step.

1.10.3 Fetching the latest CMS data for local work

Note: TL;DR version:

1. Get the DB with `make preflight`
2. If you need the images that the DB expects to exist, use `python manage.py download_media_to_local`

The CMS content exists in hosted cloud database and a trimmed-down version of this data is exported to a sqlite DB for use in local development and other processes. The exported database contains all the same content, but deliberately omits sensitive info like user accounts, unpublished drafts and outmoded versions of pages.

The DB export is generated twice a day and is put into the same public cloud buckets we've used for years. Your local Bedrock install will just download the *bedrock-dev* one as part of `make preflight`.

The DB will contain a table that knows the relative paths of the images uploaded to the CMS, but not the actual images. Those are in a cloud storage bucket, and if you want your local machine to have them available after you download the DB that expects them to be present, you can run `python manage.py download_media_to_local` which will sync down any images you don't already have.

Note: By default, `make preflight` and `./bin/run-db-download.py` will download a database file based on `bedrock-dev`. If you want to download from stage or prod, which are also available in sanitised form, you need to tell Bedrock which environment you want by prefixing the command with `AWS_DB_S3_BUCKET=bedrock-db-stage` or `AWS_DB_S3_BUCKET=bedrock-db-prod`.

```
AWS_DB_S3_BUCKET=bedrock-db-stage make preflight
```

```
python manage.py download_media_to_local --environment=stage
```

1.10.4 Editing current content surfaces

In terms of managing the content of an existing surface, please see the general Wagtail Editor Guide for now.

If you want to change the code-defined behaviour of an existing surface, that's similar to adding a new content surface, covered below. You may also find the Wagtail Docs and The Ultimate Wagtail Developers Course useful if you don't have experience of building with Wagtail yet.

1.10.5 Adding new content surfaces

This is an introduction to creating new content surfaces in the CMS. It is not a comprehensive guide, but rather a starting point to get you up and running with the basics.

The page types that you see in the CMS admin are defined as regular models in Django. As such, you can define new page types in the same way you would define any other Django model, using Wagtail's field types and panels to define the data that can be entered into the page.

When it comes to structuring CMS page models, there are some general guidelines to try and follow:

- Models and templates should be defined in the same Django app that corresponds to where the URL exists in Bedrock's information architecture (IA) hierarchy, similar to what we do for regular Jinja templates already. For example, a Mozilla themed page should be defined in `/bedrock/mozorg/models.py`, and a Firefox themed page model should be in `/bedrock/firefox/models.py`.
- Global `Page` models and `StreamField` blocks that are shared across many pages throughout the site should be defined in `/bedrock/cms/`.

Structuring code in this way should hopefully help to keep things organized and migrations in a manageable state.

Creating a new page model

Let's start by creating a new Wagtail page model called `TestPage` in `bedrock/mozorg/models.py`.

```
from django.db import models

from wagtail.admin.panels import FieldPanel
from wagtail.fields import RichTextField

from bedrock.cms.models.base import AbstractBedrockCMSPage


class TestPage(AbstractBedrockCMSPage):
    heading = models.CharField(max_length=255, blank=True)
    body = RichTextField(
        blank=True,
        features=settings.WAGTAIL_RICHTEXT_FEATURES_FULL,
    )

    content_panels = AbstractBedrockCMSPage.content_panels + [
        FieldPanel("heading"),
        FieldPanel("body"),
    ]

    template = "mozorg/test_page.html"
```

Some key things to note here:

- `TestPage` is a subclass of `AbstractBedrockCMSPage`, which is a common base class for all Wagtail pages in bedrock. Inheriting from `AbstractBedrockCMSPage` allows CMS pages to use features that exist outside of Wagtail, such as rendering Fluent strings and other L10n methods.
- The `TestPage` model defines two database field called `heading` and `body`. The `heading` field is a `CharField` (the most simple text entry field type), and `body` is a `RichTextField`. The HTML tags and elements that a content editor can enter into a rich text field are defined in `settings.WAGTAIL_RICHTEXT_FEATURES_FULL`.
- There is also a `title` field on the page model, which from `AbstractBedrockCMSPage` (which in turn comes from `wagtail.models.Page`). This doesn't make `heading` redundant, but it's worth knowing where `title` comes from.
- Both fields are added to the CMS admin panel by adding each as a `FieldPanel` to `content_panels`. If you forget to do this, that's usually why you don't see the field in the CMS admin.
- Finally, the template used to render the page type can be found at `mozorg/test_page.html`.
- If you don't set a custom template name, Wagtail will infer it from the model's name: `<app_label>/<model_name (in snake case)>.html`
- All new models must be added to the config for the DB exporter script. If you do not, the page will not be correctly exported for local development and will break for anyone using that DB export file. See *Add your new model to the DB export*, below.

Django model migrations

Once you have your model defined, it's then time to run create and run migrations to set up a database table for it:

```
./manage.py makemigrations
```

You can then run migrations using:

```
./manage.py migrate
```

Many times when you make changes to a model, it will also mean that the structure of the database table has changed. So as a general rule it's good to form a habit of running the above steps after making changes to your model. Each migration you make will add a new migration file to the `/migrations` directory. When doing local development for a new page you might find yourself doing this several times, so to help reduce the number of migration files you create you can also squash / merge them.

- Django migrations docs.
- Squashing migrations.

Rendering data in templates

This is a good time to test out your page model by adding data to it to see how it renders in your template.

The data can be rendered in `mozorg/test_page.html` as follows:

```
{% extends "base-protocol-mozilla.html" %}

{% block page_title %}{{ page.title }}{% endblock %}

{% block content %}
    <header>
    <h1>{{ page.heading }}</h1>
    <div class="w-rich-text">
        {{ page.body|richtext }}
    </div>
    </header>
{% endblock %}
```

Note the `|richtext` filter applied to the `page.body` field. This is a Wagtail-provided Jinja2 filter that will render the rich text field as HTML.

Previewing pages in the CMS admin

Next, restart your local server and log in to the CMS admin. Browse to a page and use the + icon or similar to add a new "child page". You should now see your new page type in the list of available pages. Create a new page using the `TestPage` type, give the page a title of `Test Page` and a slug of `test`, and then enter some data for the fields you defined. When you click the preview icon in the top right of the CMS page, you should hopefully see your template and data rendered successfully!

Using advanced page models, fields, and blocks

The example above was relatively simple in terms of data, but not very flexible. Now that you have the basics covered, the next step is to start thinking about your page requirements, and how to better structure your data models.

At this point, deep diving into the Wagtail Docs is very much recommended. In particular, reading up on more advanced concepts such as Stream Fields and Custom Block types will make it possible to make much more advanced CMS page types.

This is also a good time to start thinking about guardrails for your page and data. Some common things to consider:

- Are there rules around the type of content that should be allowed on the page, such as the minimum or maximum number of items in a block?
- Should there be a set order to content in a page, or can it be flexible?
- Are there rules that should be applied at the page level, such as where it should live in the site hierarchy?
- Should there be a limit to the number of instances of that page type? (e.g. it would be confusing to have more than one home page or contact page).

Writing tests

When it comes to testing CMS page models, wagtail_factories can be used to create mock data for tests to render. This can often be the trickiest part when testing more complex page models, so it takes some practice.

Factories for your page models and blocks should be defined in a `factories.py` file for your tests to import:

```
import factory
import wagtail_factories

from bedrock.mozorg import TestPage


class TestPageFactory(wagtail_factories.PageFactory):
    title = "Test Page"
    live = True
    slug = "test"

    heading = wagtail_factories.CharBlockFactory
    body = wagtail_factories.CharBlockFactory

    class Meta:
        model = models.TestPage
```

In your `test_models.py` file, you can then import the factory for your test and give it some data to render:

```
import pytest
from wagtail.rich_text import RichText

from bedrock.cms.tests.conftest import minimal_site  # noqa
from bedrock.mozorg.tests import factories

pytestmark = [
    pytest.mark.django_db,
]
```

(continues on next page)

(continued from previous page)

```
@pytest.mark.parametrize("serving_method", ("serve", "serve_preview"))
def test_page(minimal_site, rf, serving_method):  # noqa
    root_page = minimal_site.root_page

    test_page = factories.TestPageFactory(
        parent=root_page,
        heading="Test Heading",
        body=RichText("Test Body"),
    )

    test_page.save()

    _relative_url = test_page.relative_url(minimal_site)
    assert _relative_url == "/en-US/test/"
    request = rf.get(_relative_url)

    resp = getattr(test_page, serving_method)(request)
    page_content = str(resp.content)
    assert "Test Heading" in page_content
    assert "Test Body" in page_content
```

Add your new model to the DB export

When you add a new model, you must update the script that generates the sqlite DB export of our data, so that the model is included in the export. (It's an allowlist pattern, as requested by Mozilla Security).

If you do not, the page will not be correctly exported for local development and will break for anyone using that DB export file.

(It's down to Wagtail's multi-table inheritance pattern: if you don't specify your new model for export, Wagtail's core metadata `Page` is exported, but not the actual new data model that holds the content that's linked to that `Page`)

The script is `bin/export-db-to-sqlite.sh` and you need to add your new model to the list of models being exported. Search for `MAIN LIST OF MODELS BEING EXPORTED` and add your model (in the format `appname.ModelName`) there.

The `CMS_ALLOWED_PAGE_MODELS` setting

When you add a new page to the CMS, it will be available to add as a new child page immediately if `DEV=True`. This means it'll be on Dev (www-dev), but not in Staging or Prod.

So if you ship a page that needs to be used immediately in Production (which will generally be most cases), you must remember to add it to `CMS_ALLOWED_PAGE_MODELS` in Bedrock's settings. If you do not, it will not be selectable as a new Child Page in the CMS.

Why do we have this behaviour?

Two reasons:

1. This setting allows us to complete initial/eager work to add a new page type, but stop it being used in Production until we are ready for it (e.g. a special new campaign page type that we wanted to get ready in good time). While there will be guard rails and approval workflows around publishing, without this it could still be possible for part of the org to start using a new page without us realising it was off-limits, and possibly before it is allowed to be released.
2. This approach allows us to gracefully deprecate pages: if a page is removed in `settings.CMS_ALLOWED_PAGE_MODELS`, that doesn't mean it disappears from Prod or can't be edited - it just stops a NEW one being added in Prod.

Migrating Django pages to the CMS

Note: This is initial documentation, noting relevant things that exist already, but much fuller recommendations will follow

---

Migrating a surface to Wagtail is very similar to adding a new one, but some extra thought needs to be given to the switchover between old hardcoded content and new CMS-backed content.

The `@prefer_cms` decorator

If you have an existing Django-based page that you want to move to be a CMS-driven page, you are faced with a quandry.

Let's say the page exists at `/some/path/`; you can create it in the CMS with a branch of pages that mirror the same slugs (a parent page with a slug of `some` and a child page with a slug of `path`). However, in order for anyone to see the published page, you would have to remove the reference to the Django view from the URLconf, so that Wagtail would get a chance to render it (because Wagtail's page-serving logic comes last in all URLConfs). **BUT...** how can you enter content into the CMS fast enough replace the just-removed Django page? (Note: we could use a data migraiton here, but that gets complicated when there are images involved)

Equally, you may have a situation where the content for certain paths needs to be managed in the CMS for certain locales, while other locales (with rarely changing 'evergreen' content) may only exist as Django-rendered views drawing strings from Fluent.

The answer here is to use the `bedrock.cms.decorators.prefer_cms` decorator/helper.

A Django view decorated with `prefer_cms` will check if a live CMS page has been added that matches the same overall, relative path as the Django view. If it finds one, it will show the user *that* CMS page instead. If there is no match in the CMS, then the original Django view will be used.

The result is a graceful handover flow that allows us to switch to the CMS page without needing to remove the Django view from the URLconf, or to maintain a hybrid approach to page management. It doesn't affect previews, so the review of draft pages before publishing can continue with no changes. Once the CMS is populated with a live version of the replacement page, that's when a later changeset can remove the deprecated Django view if it's no longer needed.

The `prefer_cms` decorator can be used directly on function-based views, or can wrap views in the URLconf. It should not used with `bedrock.mozorg.util.page` due to the complexity of passing through what locales are involved, but instead the relevant URL route should be refactored as a regular Django view, and then decorated with `prefer_cms`

For more details, please see the docstring on `bedrock.cms.decorators.prefer_cms`.

1.10.6 Generating URLs for CMS pages in non-CMS templates

Pages in the CMS don't appear in the hard-coded URLConfs in Bedrock. Normally, this means there's no way to use *url()* to generate a path to it.

However, if there's a page in the CMS you need to generate a URL for using the `url()` template tag, *and you know what its path will be*, Bedrock contains a solution.

`bedrock.cms.cms_only_urls` is a special URLConf that only gets loaded during the call to the `url()` helper. If you expand it with a named route definition that matches the path you know will/should exist in the CMS (and most of our CMS-backed pages *do* have carefully curated paths), the `url()` helper will give you a path that points to that page, even though it doesn't really exist as a static Django view.

See the example in the `bedrock.cms.cms_only_urls.py` file.

Note: Moving a URL route to `cms_only_urls.py` is a natural next step after you've migrated a page to the CMS using the `@prefer_cms` decorator and now want to remove the old view without breaking all the calls to *url('some.view')* or *reverse('some.view')*.

1.10.7 Images

Using editor-uploaded images in templates

Images may be uploaded into Wagtail's Image library and then included in content-managed surfaces that have fields/spaces for images.

Images are stored in the same media bucket that fixed/hard-coded Bedrock images get put in, and coexist alongside them, being namespaced into a directory called `custom-media/`.

If a surface uses an image, images use must be made explicit via template markup — we need to state both *where* and *how* an image will be used in the template, including specifying the size the image will be. This is because — by design and by default — Wagtail can generate any size version that the template mentions by providing a "filter spec" e.g.

```
{% set the_image=image(page.product_image, "width-1200") %}
<img class="some-class" src="{{ the_image.url }})"/>
```

(More examples are available in the Wagtail Images docs.)

When including an image in a template we ONLY use filter specs between 2400px down to 200px in 200px steps, plus 100px.

Laying them out, these are the **only** filter specs allowed. **Using alternative ones will trigger an error in production.**

- `width-100`
- `width-200`
- `width-400`
- `width-600`
- `width-800`
- `width-1000`
- `width-1200`
- `width-1400`
- `width-1600`

- `width-1800`
- `width-2000`
- `width-2200`
- `width-2400`

Why are we limiting filter-specs to that set?

In a line: to balance infrastructure security constraints with site flexiblity, we have to pre-generate a known set of renditions.

Normally, if that `product_image` is not already available in `1024x1024`, Wagtail will resize the original image to suit, on the fly, and store this "rendition" (a resized version, basically) in the cloud bucket. It will also add a reference to the database so that Wagtail knows that the rendition already exists.

In production, the "Web" deployment has **read-only** access to the DB and to the cloud storage, so it will not be able to generate new renditions on the fly. Instead, we pre-generate those renditions when the image is saved.

This approach will not be a problem if we stick to image filter-specs from the 'approved' list. Note that extending the list of filter-specs is possible, if we need to.

I've downloaded a fresh DB and the images are missing!

That's expected: the images don't live in the DB, only references to them live there. CMS images are destined for public consumption, and Dev, Stage and Prod all store their images in a publicly-accessible cloud bucket.

We have a tool to help you sync down the images from the relevant bucket.

By default, the sqlite DB you can download to run bedrock locally is based on the data in Bedrock Dev. To get images from the cloud bucket for dev, run:

```
./manage.py download_media_to_local
```

This will look at your local DB, find the image files that it says should be available locally, copy them down to your local machine, then trigger the versions/renditions of them that should also exist.

The command will only download images you don't already have locally. You can use the `--redownload` option to force a redownload of all images.

If you have a DB from Stage you can pass the `--environment=stage` option to get the images from the Stage bucket instead. Same goes for Production.

1.10.8 L10N and Translation Management

Important: Localization via Wagtail is something we are ramping up on, so please do not assume the following notes are final, or that the workflows are currently all rock-solid. We're learning as we go.

Page-tree concept

Our Wagtail setup uses the official wagtail-localize package to manage localization of pages.

This package supports page-level localization rather than field-level localization, which means that each locale has its own distinct tree of pages, rather than each page having a stack of duplicate fields, one per destination language.

These language-specific trees can be "synchronised" with the default `en-US` page tree, so would have the same page structure, field by field) — or they can not be synchronised, so can have their own extra pages, or some specific pages in the tree can be made not "synchronised", while others are.

Basically, there is plenty of flexibility. The flipside of that flexibility is we may also create an edge-case situation that `wagtail-localize` won't work with, but we'll have to see and deal with it.

Note: It's worth investing 15 mins in watching the Wagtail Localize original demo to get a good feel of how it can work.

Locale configuration within Wagtail

While the list of available overall locales is defined in code in `settings.base.WAGTAIL_CONTENT_LANGUAGES`, any locale also needs enabling via the Wagtail Admin UI before it can be used.

When you go to `Settings > Locales` in the Wagtail fly-out menu, you will see which locales are currenly enabled. You can add new ones via the + icon.

Warning: When you add/edit a Locale in this part of the admin, you will see an option to enable syncronisation between locales. **Do not enable this**. If it is enabled, for every new page added in `en-US`, it will auto-create page aliases in every other enabled locale and these will deliver the `en-US` content under locale-specific paths, which is not what we want.

Localization process

Manual updates

At its most basic, there's nothing stopping us using copy-and-paste to enter translations into lang-specific pages, which might work well if we have a page in just one non-en-US lang and an in-house colleague doing the translation.

Automated via Smartling

However, we also have automation available to send source strings to translation vendor Smartling. This uses the `wagtail-localize-smartling` package.

Here's the overall workflow:

1. CMS page "MyPage" is created in the default lang (`en-US`)
2. The "Translate this page" option is triggered for MyPage, and relevant langs are selected from the configured langs that Smartling supports. (We don't have to translate into all of them)
3. A translation Job is created in Smartling, awaiting authorization by our L10N team.
4. A L10N team colleague authorizes the Job and selects the relevant translation workflow(s) for the relevant lang(s)

- Note that one Wagtail Page (or one Wagtail Snippet) creates one single Job, so if you select mutiple target languages for that Job and submit it, you won't get it back from Smartling until *all* languages involved are submitted by translators. One way around this is to submit each language as a separate Job, but that creates more work for our L10N team to coordinate. (We are looking to refine that experience in the future and to make it better for everyone.)

5. Once the job is completed, the localised strings flow back to Wagtail and populate a *draft* version of each language-specific page.
6. A human reviews these draft pages and publishes them
   - When a translation flows back, by default the relevant pages are *not* automatically published. At the moment, CMS admins are emailed for each language in a Job when it is synced back from Smartling, reminding them of this. (We may well move this to in-dashboard Wagtail `Tasks` for better UX.)
   - The CMS admin sidebar has a link to `Smartling Jobs`. You can use this to see what translations have landed, and also follow the link to the localized version of the page, which you can then Preview, visually check, then Publish like a regular page.

Notes:

- Smartling/`wagtail-localize-smartling` will only translate pages from the base lang (`en-US`) to another lang - it won't treat, say, a Page in `fr` as a source-language document.
- If a string is received from Smartling into the CMS and then manually edited on the CMS side, the change will *not* be overwritten by subsequent Smartling syncs and the manual edit needs to be added on the Smartling side for consistency and stability.
- If a page is translated from `en-US` once, then has new `en-US` content added that is sent for translation, that will trigger a new Smartling Job. When that job is complete, it *will* overwrite any manual edits made to a translation within the CMS. This is why it's important to make sure Smartling contains any manual tweaks done to translations in the CMS.

Automated via Pontoon

It should also be possible to use Pontoon with *wagtail-localize*. (There are notes on the Pontoon integration here, but we have not yet tried to enable this alongside *wagtail-localize-smartling*).

Additionally using Pontoon would let us benefit from community translations across a broad range of languages. However, we have yet to try to set this up and would need to agree which parts of the site do and do not use Pontoon.

1.10.9 Infrastructure notes

SSO authentication setup

When the env vars `OIDC_RP_CLIENT_ID` and `OIDC_RP_CLIENT_SECRET` are present and `USE_SSO_AUTH` is set to True in settings, Bedrock will use Mozilla SSO instead of Django's default username + password approach to sign in. The deployed sites will have these set, but we also have credentials available for using SSO locally if you need to develop something that needs it - see our password vault.

Note that Bedrock in SSO mode will *not* support 'drive by' user creation even if they have an `@mozilla.com` identity. Only users who already exist in the Wagtail admin as a User will be allowed to log in. You can create new users using Django's createsuperuser command, setting both the username and email to be your `flast@mozilla.com` LDAP address

Non-SSO authentication for local builds

If you just want to use a username and password locally, you can - ensure those env vars above are not set, and use Django's createsuperuser command to make an admin user in your local build.

1.11 Sitemaps

`bedrock` serves a root sitemap at `/sitemap.xml`, which links to localised sitemaps for each supported locale.

The sitemap data is (re)generated on a schedule by www-sitemap-generator and then is pulled into `bedrock`'s database, from which the XML sitemaps are rendered.

1.11.1 Quick summary

What does `www-sitemap-generator` do?

`www-sitemap-generator`, ultimately, produces an updated `sitemap.json` file if it detects changes in pages since the last time the sitemap was generated. It does this by loading every page and checking its ETag. This `sitemap.json` data is key to sitemap rendering by `bedrock`.

The update process is run on a schedule via our Gitlab CI setup.

Note: `www-sitemap-generator` uses the main `bedrock` release Docker image as its own base container image, which means it has access to all of `bedrock`'s code and data-loading utils.

Bear this in mind when looking at management commands in `bedrock`; `update_sitemaps` is actually only called by `www-sitemap-generator` even though it (currently) lives in `bedrock`

When is the sitemap data pulled into `bedrock`?

Bedrock's clock pod regularly runs `bin/run-db-update.sh`, which calls the `update_sitemaps_data` management command. This is what pulls in data from the `www-sitemap-generator` git repo and refreshes the `SitemapURL` records in Bedrock's database. It is from these `SitemapURL` records that the XML sitemap tree is rendered by `bedrock`.

1.12 Legal Docs

Privacy notices and their applicable translations are markdown files managed by the legal team in the legal-docs repository.

When the markdown files are imported into bedrock they are parsed using the default python markdown library and BeautifulSoup. BeautifulSoup allows content selection and manipulation using CSS selectors.

For example the code:

```
{{ doc.select('body > section > [datetime] ~ p')|join|safe }}
```

means something like:

From the document, find the `body` element, then find the `section` element that is its direct descendant, then look for a direct descendant with a `datetime` attribute, select all `p` that are its siblings. Join that array of data, and then print it without filtering it for unsafe characters.

If you would like to see the structure of the html you're working with, you can print it all to the page with `{{ doc.select('body') }}`.

1.12.1 Templates

There are three templates:

- base-notice
- base-notice-headings
- base-notice-paragraphs

base-notice

This is a basic template that will extract a lead-in section and add the appropriate classes to style lists. It will print the rest of the privacy notice un-altered.

base-notice-headings

This template is preferred for longer privacy notices. It collapses content under h3 headings.

The V1.2 subscription services notice is a good example of this template. It prefers content formatted like:

```
# Start the document with an h1, it will be used for the page title and first heading

Version 1.2, Effective February 5, 2024
{: datetime="2024-02-05" }

## The first heading will be the heading in the lead-in section

Paragraphs after the first heading but before later headings will be displayed as part␣
↪of the lead-in section.

A line will appear under the last paragraph of the lead-in.

## This will be displayed as an h2 heading and the sibling elements will not be␣
↪collapsed until an h3 is encountered

This paragraph will be visible

### This will be displayed as an h3 heading and will serve as a toggle for all its␣
↪siblings until the next h3 {: provide-an-id }

* This will be a bulleted list that is collapsed under the h3 heading

This will be a paragraph that is collapsed under the h3 heading

### Footnote {: footnote }

If a section contains a heading with an ID of `footnote` the section will be extracted␣
↪and output as the last thing on the page.

None of the elements in this section will be collapsed.
```

base-notice-paragraphs

This is an older style template that we would like to phase out in favour of the heading template. It assumes the notice will be formatted in a series of sections each with an introductory paragraph and then bulleted lists of further information. When the notice is rendered, the bulleted lists are collapsed behind a "learn more" button.

1.13 Using External Content Cards Data

The www-admin repo contains data files and images that are synced to bedrock and available for use on any page. The docs for updating said data is available via that repo, but this page will explain how to use the cards data once it's in the bedrock database.

1.13.1 Add to a View

The easiest way to make the data available to a page is to add the `page_content_cards` variable to the template context:

```
from bedrock.contentcards.models import get_page_content_cards


def view_with_cards(request):
    locale = l10n_utils.get_locale(request)
    ctx = {"page_content_cards": get_page_content_cards("home", locale)}
    return l10n_utils.render(request, "sweet-words.html", ctx)
```

The `get_page_content_cards` returns a dict of card data dicts for the given page (`home` in this case) and locale. The dict keys are the names of the cards (e.g. `card_1`). If the `page_content_cards` context variable is available in the template, then the `content_card()` macro will discover it automatically.

Note: The `get_page_content_cards` function is not all that clever as far as l10n is concerned. If you have translated the cards in the www-admin repo that is great, but you should have cards for every locale for which the page is active or the function will return an empty dict. This is especially tricky if you have multiple English locales enabled (en-US, en-CA, en-GB, etc.) and want the same cards to be used for all of them. You'd need to do something like `if locale.startswith('en-'):` then use `en-US` in the function call.

Alternately you could just wrap the section of the template using cards to be optional in an `{% if page_content_cards %}` statement, and that way it will not show the section at all if the dict is empty if there are no cards for that page and locale combination.

1.13.2 Add to the Template

Once you have the data in the template context, using a card is simple:

```
{% from "macros-protocol.html" import content_card with context %}

{{ content_card('card_1') }}
```

This will insert the data from the `card_1.en-US.md` file from the www-admin repo into the template via the `card()` macro normally used for protocol content cards.

If you don't have the `page_content_cards` variable in the template context and you don't want to create or modify a view, you can fetch the cards via a helper function in the template itself, but you have to pass the result to the macro:

```
{% from "macros-protocol.html" import content_card with context %}
{% set content_cards = get_page_content_cards('home', LANG) %}

{{ content_card('card_1', content_cards) }}
```

1.14 Banners

1.14.1 Creating page banners

Any page on bedrock can incorporate a top of page banner as a temporary feature. An example of such a banner is the MOFO (Mozilla Foundation) fundraising banner that gets shown on the home page several times a year.

Banners can be inserted into any page template by using the `page_banner` block. Banners can also be toggled on and off using a switch:

```
{% block page_banner %}
  {% if switch('fundraising-banner') %}
    {% include 'includes/banners/fundraiser.html' %}
  {% endif %}
{% endblock %}
```

Banner templates should extend the *basic banner template*, which provides very unopinionated structure but includes several helpful features such as a close button with localized text and exclusion of banner content from search result snippets.

Custom banner content can then be inserted using `banner_title` and `banner_content` blocks:

```
{% extends 'includes/banners/basic.html' %}

{% block banner_title %}We all love the web. Join Mozilla in defending it.{% endblock %}

{% block banner_content %}
    <!-- insert custom HTML here -->
{% endblock %}
```

To initiate a banner on a page, include `js/base/banners/mozilla-banner.js` in your page bundle and then initiate the banner using a unique ID. The ID will be used as a cookie identifier should someone dismiss a banner and not wish to see it again.

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

    function onLoad() {
        window.Mozilla.Banner.init('fundraising-banner');
    }

    window.Mozilla.run(onLoad);

})();
```

By default, page banners will be rendered directly underneath the primary page navigation. If you want to render a banner flush at the top of the page, you can pass a secondary `renderAtTopOfPage` parameter to the `init()` function with a boolean value:

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

    function onLoad() {
        window.Mozilla.Banner.init('fundraising-banner', true);
    }

    window.Mozilla.run(onLoad);

})();
```

L10n for page banners

Because banners can technically be shown on any page, they need to be broadly translated, or alternatively limited to the subset of locales that have translations. Each banner should have its own `.ftl` associated with it, and accessible to the template or view it gets used in.

1.15 Mozilla.UITour

1.15.1 Introduction

`Mozilla.UITour` is a JS library that exposes an event-based Web API for communicating with the Firefox browser chrome. It can be used for tasks such as opening menu panels, highlighting buttons, or querying Mozilla account signed-in state. It is supported in Firefox 29 onward, but some API calls are only supported in later versions.

For security reasons `Mozilla.UITour` will only work on white-listed domains and over a secure connection. The list of allowed origins can be found here: https://searchfox.org/mozilla-central/source/browser/app/permissions

The `Mozilla.UITour` library is maintained on Mozilla Central.

Important: The API is supported only on the desktop versions of Firefox. It doesn't work on Firefox for Android and iOS.

1.15.2 Local development

To develop or test using Mozilla.UITour locally you need to create some custom preferences in `about:config`.

- `browser.uitour.testingOrigins` (string) (value: local address e.g. `http://127.0.0.1:8000`)
- `browser.uitour.requireSecure` (boolean) (value: `false`)

Note that `browser.uitour.testingOrigins` can be a comma separated list of domains, e.g.

> 'http://127.0.0.1:8000, https://www-demo2.allizom.org'

Important: Prior to Firefox 36, the testing preference was called `browser.uitour.whitelist.add.testing` (Bug 1081772). This old preference does not accept a comma separated list of domains, and you must also exclude the domain protocol e.g. `https://`. A browser restart is also required after adding an allowed domain.

If you are working on Mozilla accounts integration, you can use the `identity.fxaccounts.autoconfig.uri` config property to change the Accounts server. For example, to change it to stage environment use this value: `https://accounts.stage.mozaws.net/`. Restart the browser and make sure the configuration updated. `identity.fxaccounts.remote.root` preference should now point to `https://accounts.stage.mozaws.net`. If it has not changed for some reason, update it manually. Ref: https://mozilla-services.readthedocs.io/en/latest/howtos/run-fxa.html

1.15.3 JavaScript API

The UITour API documentation can be found in the Mozilla Source Tree Docs.

1.16 Task queue

As part of the 2024 Wagtail CMS work, we have added a task queue to process longer-running jobs in the background.

The queue is supplied by `django-rq`, which sits on top of `rq` and uses Redis as the storage backend.

Once infra has been set up fully to support this, documentation will be expanded, but for local development you can start a worker to process jobs in the queue with:

```
make run-local-task-queue
```

Note that you will need Redis running on its default port of 6379. Redis is started by default when using Dockerized Bedrock locally.

1.17 Send to Device Widget

The *Send to Device* widget is a single form which facilitates the sending of a download link from a desktop browser to a mobile device. The form allows sending via email.

Important: This widget should only be shown to a limited set of locales who are set up to receive the emails. For those locales not in the list, direct links to the respective app stores should be shown instead. If a user is on iOS or Android, CTA buttons should also link directly to respective app stores instead of showing the widget. This logic should be handled on a page-by-page basis to cover individual needs.

Note: A full list of supported locales can be found in `settings/base.py` under `SEND_TO_DEVICE_LOCALES`, which can be used in the template logic for each page to show the form.

1.17.1 Usage

1. Make sure necessary files are in your CSS/JS bundles:

- `'css/protocol/components/send-to-device.scss'`
- `'js/base/send-to-device.es6.js'`

2. Include the macro in your page template:

```
{{ send_to_device() }}
```

3. Initialize the widget:

   In your page JS, initialize the widget using:

```
import SendToDevice from '/media/js/base/send-to-device.es6';

const form = new SendToDevice();
form.init();
```

   By default the `init()` function will look for a form with an HTML id of `send-to-device`. If you need to pass another id, you can do so directly:

```
const form = new SendToDevice('my-custom-form-id');
form.init();
```

Configuration

The Jinja macro supports parameters as follows (* indicates a required parameter)

Parameter name	Definition	Format	Example
platform*	Platform ID for the receiving device. Defaults to 'all'.	String	'all', 'android', 'ios'
message_set*	ID for the email that should be received. Defaults to 'default'.	String	'default', 'fx-mobile-download-desktop', 'fx-whatsnew'
dom_id*	HTML form ID. Defaults to 'send-to-device'.	String	'send-to-device'
class_name	CSS class name for form orientation. Defaults to 'vertical'	String	'horizontal', 'vertical'
include_title	Should the widget contain a title. Defaults to 'True'.	Boolean	'True', 'False'
title_text	Provides a custom string for the form title, overriding the default.	Localizable string	'Send Firefox Lite to your smartphone or tablet' .
input_label	Provides a custom label for the input field, overriding the default.	Localizable string	'Enter your email' .
legal_note_ema	Provides a custom legal note for email use.	Localizable String.	'The intended recipient of the email must have consented.'
spinner_color	Hex color for the form spinner. Defaults to '#000'.	String	'#fff'
button_class	Optional button CSS class string. Defaults to 'mzp-t-product'	String	'mzp-t-product mzp-t-dark'

1.18 Firefox Download Buttons

There are two Firefox download button helpers in bedrock to choose from. The first is a lightweight button that links directly to the `/firefox/download/thanks/` page. Its sole purpose is to facilitate downloading the main release version of Firefox.

```
{{ download_firefox_thanks() }}
```

The second type of button is more heavy weight, and can be configured to download any build of Firefox (e.g. Release, Beta, Developer Edition, Nightly). It can also offer functionality such as direct (in-page) download links, so it comes with a lot more complexity and in-page markup.

```
{{ download_firefox() }}
```

1.18.1 Which button should I use?

A good rule of thumb is to always use `download_firefox_thanks()` for regular landing pages (such as `/firefox/new/`) where the main release version of Firefox is the product being offered. For pages pages that require direct download links, or promote pre-release products (such as `/firefox/channel/`) then `download_firefox()` should be used instead.

1.18.2 Documentation

See helpers.py for documentation and supported parameters for both buttons.

1.19 Mozilla accounts helpers

Note: Since a rebranding in October 2023, we now refer to "Mozilla accounts" in our web pages instead of "Firefox accounts". This rebranding is so far superficial, and sign up flows still go to `accounts.firefox.com`. Because of this, our internal code and helpers still use `FxA` or `fxa` as a common abbreviation. However the language used around them should now be "Mozilla accounts" going forward.

Marketing pages often promote the creation of a Mozilla account as a common *call to action* (CTA). This is typically accomplished using either a sign-up form, or a prominent link/button. Other products such as Mozilla VPN use similar account auth flows to manage subscriptions. To accomplish these tasks, bedrock templates can take advantage of a series of Python helpers which can be used to standardize product referrals, and make supporting these auth flows easier.

Note: See the attribution docs (*Mozilla accounts attribution*) for more a detailed description of the analytics functions these helpers provide.

1.19.1 Mozilla account sign-up form

Use the `fxa_email_form` macro to display a Mozilla account sign-up form on a page.

Usage

To use the form in a Jinja template, first import the `fxa_email_form` macro:

```
{% from "macros.html" import fxa_email_form with context %}
```

The form can then be invoked using:

```
{{ fxa_email_form(entrypoint='mozilla.org-firefox-accounts') }}
```

The macro's respective JavaScript and CSS dependencies should also be imported in the page:

Javascript:

```
import FxaForm from './path/to/fxa-form.es6.js';

FxaForm.init();
```

The above JS is also available as a pre-compiled bundle, which can be included directly in a template:

```
{{ js_bundle('fxa_form') }}
```

CSS:

```
@import '../path/to/fxa-form';
```

The JavaScript files will automatically handle things such as adding metrics parameters for Firefox desktop browsers. The CSS file contains some default styling for the sign-up form.

Configuration

The sign-up form macro accepts the following parameters (* indicates a required parameter)

Parameter name	Definition	Format	Example
entry-point*	Unambiguous identifier for which page of the site is the referrer.	mozilla.org-directory-page	'mozilla.org-firefox-accounts'
entry-point_expe	Used to identify experiments.	Experiment ID	'whatsnew-headlines'
entry-point_varia	Used to track page variations in multivariate tests. Usually just a number or letter but could be a short keyword.	Variant identifier	'b'
style	An optional parameter used to invoke an alternatively styled page at accounts.firefox.com.	String	'trailhead'
class_name	Applies a CSS class name to the form. Defaults to: 'fxa-email-form'	String	'fxa-email-form'
form_title	The main heading to be used in the form (optional with no default).	Localizable string	'Join Firefox' .
intro_text	Introductory copy to be used in the form. Defaults to a well localized string.	Localizable string	'Enter your email address to get started.' .
button_text	Button copy to be used in the form. Defaults to a well localized string.	Localizable string	'Sign Up' .
button_class	CSS class names to be applied to the submit button.	String of one or more CSS class names	'mzp-c-button mzp-t-primary mzp-t-product'
utm_campa	Used to identify specific marketing campaigns. Defaults to `fxa-embedded-form`	Campaign name prepended to default value	'trailhead-fxa-embedded-form'
utm_term	Used for paid search keywords.	Brief keyword	'existing-users'
utm_conter	Declared when more than one piece of content (on a page or at a URL) links to the same place, to distinguish between them.	Description of content, or name of experiment treatment	'get-the-rest-of-firefox'

Invoking the macro will automatically include a set of default UTM (Urchin Tracking Module) parameters as hidden form input fields:

- `utm_source` is automatically assigned the value of the `entrypoint` parameter.
- `utm_campaign` is automatically set as the value of `fxa-embedded-form`. This can be prefixed with a custom value by passing a `utm_campaign` value to the macro. For example, `utm_campaign='trailhead'` would result in a value of `trailhead-fxa-embedded-form`.
- `utm_medium` is automatically set as the value of `referral`.

Note: When signing into a Mozilla account using this form on a Firefox Desktop browser, it will also activate the Sync feature.

1.19.2 Mozilla account links

Use the `fxa_button` helper to create a CTA button or link to https://accounts.firefox.com/.

Usage

```
{{ fxa_button(entrypoint='mozilla.org-firefox-sync-page', button_text='Sign In') }}
```

Note: There is also a `fxa_link_fragment` helper which will construct a valid `href` property. This is useful when constructing an inline link inside a paragraph, for example.

Note: When signing into a Mozilla account using this link on a Firefox Desktop browser, it will also activate the Sync feature.

For more information on the available parameters, read the "Common Parameters" section further below.

1.19.3 Mozilla Monitor links

Use the `monitor_fxa_button` helper to link to https://monitor.mozilla.org/ via a Mozilla accounts auth flow.

Usage

```
{{ monitor_fxa_button(entrypoint=_entrypoint, button_text='Sign Up for Monitor') }}
```

For more information on the available parameters, read the "Common Parameters" section further below.

1.19.4 Pocket links

Use the `pocket_fxa_button` helper to link to https://getpocket.com/ via a Mozilla accounts auth flow.

Usage

```
{{ pocket_fxa_button(entrypoint='mozilla.org-firefox-pocket', button_text='Try Pocket Now
↪', optional_parameters={'s': 'ffpocket'}) }}
```

For more information on the available parameters, read the "Common Parameters" section below.

1.19.5 Common Parameters

The `fxa_button`, `pocket_fxa_button`, and `monitor_fxa_button` helpers all support the same standard parameters:

Parameter name	Definition	Format	Example
entry-point*	Unambiguous identifier for which page of the site is the referrer. This also serves as a value for 'utm_source'.	'mozilla.org-firefox-pocket'	'mozilla.org-firefox-pocket'
button_text*	The button copy to be used in the call to action.	Localizable string	'Try Pocket Now'
class_nam	A class name to be applied to the link (typically for styling with CSS).	String of one or more class names	'pocket-main-cta-button'
is_button_	A boolean value that dictates if the CTA should be styled as a button or a link. Defaults to 'True'.	Boolean	True or False
include_me	A boolean value that dictates if metrics parameters should be added to the button href. Defaults to 'True'.	Boolean	True or False
optional_par	An dictionary of key value pairs containing additional parameters to append the the href.	Dictionary	{'s': 'ffpocket'}
optional_attı	An dictionary of key value pairs containing additional data attributes to include in the button.	Dictionary	{'data-cta-text': 'Try Pocket Now', 'data-cta-type': 'pocket','data-cta-position': 'primary'}

Note: The `fxa_button` helper also supports an additional `action` parameter, which accepts the values `signup`, `signin`, and `email` for configuring the type of authentication flow.

1.19.6 Mozilla VPN (Virtual Private Network) Links

Use the `vpn_subscribe_link` helpers to create a VPN subscription link via a Mozilla accounts auth flow.

Usage

```
{{ vpn_subscribe_link(entrypoint='www.mozilla.org-vpn-product-page', link_text='Get␣
↪Mozilla VPN') }}
```

Common VPN Parameters

Both helpers for Mozilla VPN support the same parameters (* indicates a required parameter)

Parameter name	Definition	Format	Example
entrypoint*	Unambiguous identifier for which page of the site is the referrer. This also serves as a value for 'utm_source'.	'www.mozilla.c page-name'	'www.mozilla.org-vpn-product-page'
link_text*	The link copy to be used in the call to action.	Localizable string	'Get Mozilla VPN'
class_nam	A class name to be applied to the link (typically for styling with CSS).	String of one or more class names	'vpn-button'
lang	Page locale code. Used to query the right subscription plan ID in conjunction to country code.	Locale string	'de'
country_code	Country code provided by the CDN. Used to determine the appropriate subscription plan ID.	Two digit, uppercase country code	'DE'
optional_par	An dictionary of key value pairs containing additional parameters to append the the href.	Dictionary	{'utm_campaign': 'vpn-product-page'}
optional_attı	An dictionary of key value pairs containing additional data attributes to include in the button.	Dictionary	{'data-cta-text': 'VPN Sign In', 'data-cta-type': 'fxa-vpn', 'data-cta-position': 'navigation'}

The `vpn_subscribe_link` helper has an additional `plan` parameter to support linking to different subscription plans.

Parameter name	Definition	Format	Example
plan	Subscription plan ID. Defaults to 12-month plan.	'12-month'	'12-month' or 'monthly'

1.19.7 Firefox Sync and UITour

Since Firefox 80 the accounts link and email form macros use *UITour* to show the Mozilla accounts page and log the browser into Sync or an account. For non-Firefox browsers or if UITour is not available, the flow uses normal links that allow users to log into the Mozilla accounts website only, without connecting the Firefox Desktop client. This transition was introduced to later migrate Firefox Desktop to an OAuth based client authentication flow.

The script that handles this logic is `/media/js/base/fxa-link.js`, and will automatically apply to any link with a `js-fxa-cta-link` class name. The current code automatically detects if you are in the supported browser for this flow and updates links to drive them through the UITour API. The UITour `showFirefoxAccounts` action supports flow id parameters, UTM parameters and the email data field.

1.19.8 Testing Sign-up Flows

Testing the Mozilla account sign-up flows on a non-production environment requires some additional configuration.

Configuring bedrock:

Set the following in your local `.env` file:

```
FXA_ENDPOINT=https://accounts.stage.mozaws.net/
```

For Mozilla VPN links you can also set:

```
VPN_ENDPOINT=https://stage.guardian.nonprod.cloudops.mozgcp.net/
VPN_SUBSCRIPTION_URL=https://accounts.stage.mozaws.net/
```

Note: The above values for staging are already set by default when `Dev=True`, which will also apply to demo servers. You may only need to configure your `.env` file if you wish to change a setting to something else.

1.20 A/B Testing

1.20.1 Traffic Cop experiments

More complex experiments, such as those that feature full page redesigns, or multi-page user flows, should be implemented using Traffic Cop. Traffic Cop small javascript library which will direct site traffic to different variants in a/b experiments and make sure a visitor always sees the same variation.

It's possible to test more than 2 variants.

Traffic Cop sends users to experiments and then we use Google Analytics (GA) to analyze which variation is more successful. (If the user has DNT (Do Not Track) or GPC (Global Privacy Control) enabled they do not participate in experiments.)

All a/b tests should have a mana page detailing the experiment and recording the results.

Coding the variants

Traffic cop supports two methods of a/b testing. Executing different on page javascript or redirecting to the same URL with a query string appended. We mostly use the redirect method in bedrock. This makes testing easier.

Create a variation view for the a/b test.

The view can handle the URL redirect in one of two ways:

1. the same page, with some different content based on the *variation* variable
2. a totally different page

Content variation

Useful for small focused tests.

This is explained on the variation view page.

New page

Useful for large page changes where content and assets are dramatically different.

Create the variant page like you would a new page. Make sure it is `noindex` and does not have a `canonical` URL.

```
{% block canonical_urls %}<meta name="robots" content="noindex,follow">{% endblock %}
```

Configure as explained on the variation view page.

Traffic Cop

Create a .js file where you initialize Traffic Cop and include that in the experiments block in the template that will be doing the redirection. Wrap the extra js include in a switch.

```
{% block experiments %}
  {% if switch('experiment-berlin-video', ['de']) %}
    {{ js_bundle('firefox_new_berlin_experiment') }}
  {% endif %}
{% endblock %}
```

Switches

See the traffic cop section of the switch docs for instructions.

Recording the data

Note: If you are measuring installs as part of your experiment be sure to configure custom stub attribution as well.

Send the experiment view events to GA4 with the event name `experiment_view`. The `id` of all variants should be the same and all `variant` values should be unique.

Make sure any buttons and interaction which are being compared as part of the test will report into GA (Google Analytics).

```
if (href.indexOf('v=a') !== -1) {
    // GA4
    window.dataLayer.push({
        event: 'experiment_view',
        id: 'Berlin-Campaign-Landing-Page',
        variant: 'de-page',
    });
} else if (href.indexOf('v=b') !== -1) {
    // GA4
```

(continues on next page)

(continued from previous page)

```
    window.dataLayer.push({
        event: 'experiment_view',
        id: 'Berlin-Campaign-Landing-Page',
        variant: 'campaign-page',
    });
}
```

Viewing the data

We have not figured this out for GA4 yet.

Tests

Write some tests for your a/b test. This could be simple or complex depending on the experiment.

Some things to consider checking:

- Requests for the default (non variant) page call the correct template.
- Requests for a variant page call the correct template.
- Locales excluded from the test call the correct (default) template.

1.20.2 Avoiding experiment collisions

To ensure that Traffic Cop doesn't overwrite data from any other externally controlled experiments (for example Ad campaign tests, or in-product Firefox experiments), you can use the experiment-utils helper to decide whether or not Traffic Cop should initiate.

```
import TrafficCop = from '@mozmeao/trafficcop';
import { isApprovedToRun } from '../../base/experiment-utils.es6';

if (isApprovedToRun()) {
    const cop = new TrafficCop({
        variations: {
            'entrypoint_experiment=experiment-name&entrypoint_variation=a': 10,
            'entrypoint_experiment=experiment-name&entrypoint_variation=b': 10
        }
    });

    cop.init();
}
```

The `isApprovedToRun()` function will check the page URL's query parameters against a list of well-known experimental params, and return `false` if any of those params are found. It will also check for some other cases where we do not want to run experiments, such as if the page is being opened in an automated testing environment.

1.21 Mozilla VPN Subscriptions

The Mozilla VPN landing page displays both pricing and currency information that is dependant on someone's physical location in the world (using geo-location). If someone is in the United States, they should see pricing in $USD, and if someone is in Germany they should see pricing in Euros. The page is also available in multiple languages, which can be viewed independently of someone's physical location. So someone who lives in Switzerland, but is viewing the page in German, should still see pricing and currency displayed in Swiss Francs (CHF).

Additionally, it is important that we render location specific subscription links, as purchasing requires a credit card that is registered to each country where we have a plan available. We are also legally obligated to prevent both purchasing and/or downloading of Mozilla VPN in certain countries. In countries where VPN is not yet available, we also rely on geo-location to hide subscription links, and instead to display a *call to action* to encourage prospective customers to sign up to the VPN wait list.

To facilitate all of the above, we rely on our CDN to return an appropriate country code that relates to where a visitor's request originated from (see *Geo Template View*). We use that country code in our helpers and view logic for the VPN landing page to decide what to display in the pricing section of the page (see *Mozilla VPN Links*).

1.21.1 Server architecture

Bedrock is configured so that when `dev=True`, VPN subscription links will point to the Mozilla accounts staging environment. When `dev=False`, they will point to the Fxa production environment.

So our different environments are mapped like so:

- `http://localhost:8000` -> `https://accounts.stage.mozaws.net/`
- `https://www-dev.allizom.org/products/vpn/` -> `https://accounts.stage.mozaws.net/`
- `https://www.allizom.or/products/vpn/` -> `https://accounts.firefox.com/`
- `https://www.mozilla.org/products/vpn` -> `https://accounts.firefox.com/`

This allows the product and QA teams to routinely test changes and new VPN client releases on https://www-dev.allizom.org/products/vpn/, prior to being available in production.

1.21.2 Adding new countries for VPN

When launching VPN in new countries there is a set process to follow.

Launch steps

1. All the code changes below should be added **behind a feature switch**.
2. Once the PR is reviewed and merged, the product QA team should be notified and they can then perform testing on https://www-dev.allizom.org/products/vpn/. Often the QA team will request a date for code to be ready for testing to begin.
3. Code can be pushed to production ahead of time (but will be disabled behind the feature switch by default).
4. Once QA gives the green light on launch day, the feature switch can then be enabled in production.
5. QA will then do a final round of post-launch QA to verify subscriptions / purchasing works in the new countries in production.

Code changes

Reference: officially assigned list of ISO country codes. Reference: list of *ISO 4217 currency codes* `_

The majority of config changes need to happen in `bedrock/settings/base.py`:

1. Add new pricing plan configs to `VPN_PLAN_ID_MATRIX` for any new countries that require newly created plan IDs (these will be provided by the VPN team). Separate plan IDs for both dev and prod are required for each new currency / language combination (this is because the product QA team need differently configured plans on dev to routinely test things like renewal and cancellation flows). Meta data such as price, total price and saving for each plan / currency should also be provided.

   Example pricing plan config for $USD / English containing both 12-month and monthly plans:

```
VPN_PLAN_ID_MATRIX = {
    "usd": {
        "en": {
            "12-month": {
                "id": (
                    "price_1J0Y1iKb9q6OnNsLXwdOFgDr"
                    if DEV
                    else "price_1Iw85dJNcmPzuWtRyhMDdtM7"
                ),
                "price": "US$4.99",
                "total": "US$59.88",
                "saving": 50,
                "analytics": {
                    "brand": "vpn",
                    "plan": "vpn",
                    "currency": "USD",
                    "discount": "60.00",
                    "price": "59.88",
                    "period": "yearly",
                },
            },
            "monthly": {
                "id": (
                    "price_1J0owvKb9q6OnNsLExNhEDXm"
                    if DEV
                    else "price_1Iw7qSJNcmPzuWtRMUZpOwLm"
                ),
                "price": "US$9.99",
                "total": None,
                "saving": None,
                "analytics": {
                    "brand": "vpn",
                    "plan": "vpn",
                    "currency": "USD",
                    "discount": "0",
                    "price": "9.99",
                    "period": "monthly",
                },
            },
        }
    },
```

(continues on next page)

(continued from previous page)

```
    # repeat for other currency / language configs.
}
```

See the *Begin Checkout* section of the *analytics docs* for more a detailed description of what should be in the analytics objects.

2. Map each new country code to one or more applicable pricing plans in `VPN_VARIABLE_PRICING`.

   Example that maps the `US` country code to the pricing plan config above:

```
VPN_VARIABLE_PRICING = {
    "US": {
        "default": VPN_PLAN_ID_MATRIX["usd"]["en"],
    },
    # repeat for other country codes.
}
```

3. Once every new country has a mapping to a pricing plan, add each new country code to the list of supported countries in `VPN_COUNTRY_CODES`. Because new countries need to be added behind a feature switch, you may want to create a new variable temporarily for this until launched, such as `VPN_COUNTRY_CODES_WAVE_VI`. You can then add these to `VPN_COUNTRY_CODES` in `products/views.py` using a simple function like so:

```
def vpn_available(request):
    country = get_country_from_request(request)
    country_list = settings.VPN_COUNTRY_CODES

    if switch("vpn-wave-vi"):
        country_list = settings.VPN_COUNTRY_CODES + settings.VPN_COUNTRY_
↪CODES_WAVE_VI

    return country in country_list
```

   The function could then be used in the landing page view like so:

```
vpn_available_in_country = (vpn_available(request),)
```

4. If you now test the landing page locally, you should hopefully see the newly added pricing for each new country (add the `?geo=[INSERT_COUNTRY_CODE]` param to the page URL to mock each country). If all is well, this is the perfect time to add new unit tests for each new country. This will help give you confidence that the right plan ID is displayed for each new country / language option.

```
def test_vpn_subscribe_link_variable_12_month_us_en(self):
    """Should contain expected 12-month plan ID (US / en-US)"""
    markup = self._render(
        plan="12-month",
        country_code="US",
        lang="en-US",
    )
    self.assertIn("?plan=price_1Iw85dJNcmPzuWtRyhMDdtM7", markup)


def test_vpn_subscribe_link_variable_monthly_us_en(self):
    """Should contain expected monthly plan ID (US / en-US)"""
    markup = self._render(
```

(continues on next page)

(continued from previous page)

```
        plan="monthly",
        country_code="US",
        lang="en-US",
    )
    self.assertIn("?plan=price_1Iw7qSJNcmPzuWtRMUZpOwLm", markup)
```

5. Next, update `VPN_AVAILABLE_COUNTRIES` to the new total number of countries where VPN is available. Again, because this needs to be behind a feature switch you may want a new temporary variable that you can use in `products/views.py`:

```
available_countries = settings.VPN_AVAILABLE_COUNTRIES

if switch("vpn-wave-vi"):
    available_countries = settings.VPN_AVAILABLE_COUNTRIES_WAVE_VI
```

6. Finally, there is also a string in `l10n/en/products/vpn/shared.ftl` that needs updating to include the new countries. This should be a new string ID, and behind a feature switch in the template:

```
vpn-shared-available-countries-v6 = We currently offer { -brand-name-mozilla-vpn }␣
↪in Austria, Belgium, Canada, Finland, France, Germany, Ireland, Italy, Malaysia,␣
↪the Netherlands, New Zealand, Singapore, Spain, Sweden, Switzerland, the UK, and␣
↪the US.
```

```
{% if switch('vpn_wave_vi') %}
    {{ ftl('vpn-shared-available-countries-v6', fallback='vpn-shared-available-
↪countries-v5') }}
{% else %}
    {{ ftl('vpn-shared-available-countries-v5') }}
{% endif %}
```

7. After things are launched in production and QA has verified that all is well, don't forget to file an issue to tidy up the temporary variables and switch logic.

1.21.3 Excluded countries

For a list of country codes where we are legally obligated to prevent purchasing VPN, see `VPN_EXCLUDED_COUNTRY_CODES` in `bedrock/settings/base.py`.

For a list of country codes where we are also required to prevent downloading the VPN client, see `VPN_BLOCK_DOWNLOAD_COUNTRY_CODES`.

1.22 Attribution

Attribution is the practice of recording the main touch points that a website visitor encounters on their path to downloading or signing up for one of our products. It often involves a multi-step user journey, sometimes across multiple properties, but the goal is to end up with informative data that tells us where the user of a product initially came from, and what their journey looked like along the way.

These documents define how attribution works for the different products on our websites.

1.22.1 Mozorg analytics

Google Analytics

We use Google Analytics (GA) to track how our website is used.

Implementation

Google calls a 3rd party JavaScript library that is imported by adding a script tag to a website, Google names these "tags".

We include the tag for Google Tag Manager (GTM) on our website. GTM is a service meant to simplify and centralize the tags included on a website in cases where multiple tags are needed. Their web-based user interface can then be used to watch for user actions on the website and then trigger events to be sent to tags.

We do no not use any of GTM's most powerful features. Our use of GTM is a historical artifact of our Universal Analytics (UA) (aka GA3) implementation. GTM is a tool to let non-technical users add code to a website, however, there is no one outside of the websites team who is trained to do that.

The only tag we include with GTM is the Google Tag (GTag), and that in turn connects to our Google Analytics (GA) account.

There are three ways we send events from our site to GA:

1. Google's Enhanced Event Measurement logs certain events automatically.
2. GTM watches elements with specific data-attributes for user interaction
3. Bedrock watches for specific events, formats them, and sends them to GTM using the dataLayer

Implementation Principles

The current implementation tries very hard to keep any kind of logic and formatting of our events in bedrock where it can be tested, code reviewed, and version controlled.

Many user interactions will trigger multiple events. For example: clicking the "get subscription" button on the VPN page will trigger `link`, `cta_click`, and `begin_checkout` events. This is totally fine.

If you're creating a new event use the two word noun_verb scheme that Google came up with and use snake_case, even if one of your noun or verb is more than one word.

Debugging

- GTM is managed with Tag Manager https://tagmanager.google.com/
- Our gTag is also managed with Tag Manager https://tagmanager.google.com/#/home#tags
- GA, GTM, and gTag can be debugged using Tag Assistant https://tagassistant.google.com/
- GA also has a debug view https://support.google.com/analytics/answer/7201382?hl=en

How can visitors opt out of GA?

Visitors to the website can opt-out of loading Google Analytics on our website by enabling Do Not Track (DNT) in their web browser. We facilitate this by using a DNT helper that our team maintains. We also respect Global Privacy Control (GPC) as an additional opt-out signal.

Alternatively, visitors can also opt-out of GA by visiting the cookie settings page, which is linked to in the main site footer on every page.

Event Tracking

Enhanced Event Measurement

Pageviews, scrolls, video events, and outbound link clicks are being collected using GA4's enhanced event measurement.

Some form submissions are also being collected but newsletter signups are not (see bug #13348) . They are instead being tracked with the `newsletter_subscribe` event.

See the full list of events GA collects by default. https://support.google.com/analytics/answer/9216061

Data Attributes

One place where we do still rely on GTM to watch for an trigger events is when we use data-attributes to pass information. There are only two events we do this with and they are primarily used on buttons and links.

CTA Click

CTA ("Call to Action") click is intented to track the one or two main actions the page is designed to get the user to do. This data-attribute can be used on either <button> or <a>. Links can go to or away from the site, a button might trigger a JS function instead of going away from the page, that's still a cta.

The attribute `data-cta-text` must be present to trigger the event. All links to accounts.mozilla.com must also use `data-cta-type`.

Data Attribute	Expected Value
`data-cta-text` *	Text or name of the link (e.g. `Sign Up`, `Start Here`, `Get Relay`, `See your report`, `Read the Manifesto`). • This does not need to exactly match the text displayed to the user • Defining this is useful to group the link clicks across locales • – this attribute is required
`data-cta-position`	Location of CTA on the page (e.g. `primary`, `secondary`, `banner`, `pricing`)
`data-cta-type`	fxa-servicename (e.g. `fxa-sync`, `fxa-monitor`, `fxa-vpn`, `monitor`, `relay`, `pocket`) • This is to group CTAs by their destination • Do not use this to identify the element (ie. link, button)
`data-cta-name`	A identifier for this cta that is unique across the entire site. (e.g. `fx20-primarycta`, `wnp118-sfaq-so-special-features`). This is to help with reporting since it is difficult to filter on more than one parameter at a time.

```
<a href="https://monitor.firefox.com/" data-cta-text="Check for breaches" data-cta-type=
↪"fxa-monitor">Check for breaches</a>

<a href="{{ url('firefox.browsers.mobile.get-app') }}" data-cta-text="Send Link for␣
↪Firefox Mobile" data-cta-position="banner">Send me a link</a>

<a href="{{ url('firefox.browsers.mobile.ios') }}" data-cta-text="Firefox for iOS">
↪Firefox for iOS</a>
```

Link Click

Link click is intended to track links that are of interest but not the focus of the page. Examples include links in paragraphs, lists, FAQs, supplemental content, or in a navigation menu. Links can go to or away from the site.

The attribute `data-link-text` must be present to trigger the event.

Data Attribute	Expected Value
`data-link-text` *	Text or name of the link (e.g. `Monitor`, `Features`, `Instagram (mozilla)`, `Mozilla VPN`). - * this attribute is required
`data-link-posit`	Location of CTA on the page (e.g. `topnav`, `subnav`, `body`, `features`)

```
<p>This is text with a <a href="#" data-link-text="simple">simple</a>example.</p>

<li><a href="{{ url('firefox.features.pdf-editor') }}" data-link-text="Edit PDFs">Edit␣
↪PDFs</a> on the go within your Firefox browser window.</li>
```

Link click is also commonly used for navigation menus. If you wish to indicate that a link is nested you can include the categories seperated by a dash (`topnav - firefox`, `footer - company`)

```
<li><a href="{{ url('firefox.developer.index') }}" data-link-text="Firefox Developer␣
↪Edition" data-link-position="footer">{{ ftl('footer-developer-edition') }}</a></li>

<li><a href="{{ url('firefox.browsers.mobile.android') }}" data-link-text="Firefox for␣
↪Android" data-link-position="topnav - firefox"></li>
```

Data Layer Events

The data layer is a JS object we can push events to and GTM will read from it.

```
window.dataLayer.push({'event': 'event_name'});
```

We push a mix of recommended events and custom events to the data layer. When creating a new custom event please follow the Implementation Principles outlined above. Remember, both GTM and GA must be configured to recieve new events.

https://developers.google.com/analytics/devguides/collection/ga4/reference/events?client_type=gtag#begin_checkout

https://mozilla-hub.atlassian.net/wiki/spaces/EN/pages/430866463/GA4+Custom+Events

https://developers.google.com/tag-platform/tag-manager/datalayer

Events that bedrock will send to GTM include:

- begin_checkout
- cta_click
- default_browser_set
- dimension_set
- experiment_view
- link_click
- newsletter_subscribe
- product_download (firefox_download, firefox_mobile_download, etc)
- send_to_device
- social_share
- stub_session_set
- widget_action

Begin Checkout

We are using GA4's recommended eCommerce event begin_checkout for VPN referrals to the FxA Subscription Platform with purchase intent. This event can accept values for other products but we are not currently using it for anything other than VPN.

Note: Any link to Mozilla accounts should also be using *mozilla accounts attribution*

`datalayer-begincheckout.es6.js` contains generic functions that can be called on to push the appropriate information to the dataLayer. The script is expecting the following values:

Property	Value
`item_id`	Text or name of the link (e.g. `Sign Up`, `Join Now`, `Start Here`).
`brand`	fxa-servicename (e.g. `fxa-sync`, `fxa-monitor`)
`plan`	Location of CTA on the page (e.g. `primary`, `secondary`, `header`)
`period`	Location of CTA on the page (e.g. `primary`, `secondary`, `header`)
`price`	Location of CTA on the page (e.g. `primary`, `secondary`, `header`)
`currency`	Location of CTA on the page (e.g. `primary`, `secondary`, `header`)
`discount`	Location of CTA on the page (e.g. `primary`, `secondary`, `header`)

- item_id: Stripe Plan ID
- brand: `relay`, `vpn`, or `monitor`
- **plan:**
  - `vpn-monthly`
  - `vpn-yearly`
  - `vpn-relay-yearly`
  - `relay-email-monthly`
  - `relay-email-yearly`

  - relay-phone-monthly
  - relay-phone-yearly
  - monitor-monthly
  - monitor-yearly
- period: monthly or yearly
- price: cost displayed at checkout, pre tax (example: 119.88)
- currency: in 3-letter ISO 4217 format (examples: USD, EUR)
- discount: value of the discount in the same currency as price (example: 60.00)

There are two ways to use TrackBeginCheckout:

1) Call the function passing the values directly.

```
TrackBeginCheckout.getEventObjectAndSend(item_id, brand, plan, period, price, currency,␣
↪discount)
```

2) Pass the values as a data attribute.

The vpn_subscribe_link will automatically generate a data-ga-item object and add the ga-begin-checkout class to links they create – as long as there is analytics information associated with the plan in its lookup table.

To use this method you will need to include datalayer-begincheckout-init.es6.js in the page bundle.

```
<a href="{{ fxa link }}"
    class="ga-begin-checkout"
    data-ga-item="{
        'id' : 'price_1Iw7qSJNcmPzuWtRMUZpOwLm',
        'brand' : 'vpn',
        'plan' : 'vpn',
        'period' : 'monthly',
        'price' : '9.99',
        'discount' : '0',
        'currency' : 'USD'
    }"
>
    Get monthly plan
</a>
```

Default Browser

Trigger this event when a user sets their default browser to Firefox. It's an important conversion for us!

```
window.dataLayer.push({
    event: 'default_browser_set',
});
```

Newsletter Subscribe

Product Downloads

Important: VPN support has not been added. Firefox, Firefox Mobile, Focus, Klar, and Pocket are currently supported.

When the user signals their intent do install one of our products we log a download event named for the product. This intent could be: clicking an app store badge, triggering a file download, or sending themselves the link using the send to device widget. The events are in the format [product name]_download and all function the same. So they use the same JavaScript "TrackProductDownload". For this documentation the following custom events will be talked about as *product_download* :

- *firefox_download*
- *firefox_mobile_download*
- *focus_download*
- *klar_download*
- *pocket_download*

Properties for use with *product_download* (not all products will have all options):

- product (one of: firefox, firefox_mobile, focus, klar, pocket, vpn)
- platform **optional** (one of: win, win-msi, win64, win64-msi, win64-aarch64, macos, linux, linux64, android, ios)
- method (one of: site, store, or adjust)
- release_channel **optional** (one of: release, esr, devedition, beta, nightly)
- download_language **optional** (example: en-CA)

There are two ways to use TrackProductDownload:

1) Call the function, passing it the same URL you are sending the user to:

```
TrackProductDownload.sendEventFromURL(downloadURL);
```

2) Add a class to the link:

```
<a href="{{ link }}" class="ga-product-download">Link text</a>
```

You do NOT need to include `datalayer-productdownload-init.es6.js` in the page bundle, it is already included in the site bundle.

Note: Most apps listed in *appstores.py* are supported but you may still want to check that the URL you are tracking is identified as valid in `` `isValidDownloadURL` `` and will be recognized by `` `getEventFromUrl` ``.

If you would like to track something as a download that is not currently in the *appstores.py* you can get and send the event object manually. This most often happens with adjust links generated for specific campaigns:

```
let customEventObject = TrackProductDownload.getEventObject(
        'firefox_mobile',
        '', // if you are not redirecting to a specific store, leave platform empty
        'adjust'
    );
TrackProductDownload.sendEvent(customEventObject);
```

Note: Calling TrackProductDownload will also fire an event named *product_download* so two events are being logged for each product download. This is because prior to Feb 2024 we only used one unified product download event and did not have the individual product download events yet. The split events are considered easier to deal with for reporting purposes inside GA4. Some data science dashboards use *product_download* because it has existed longer. Ideally, we will remove it some day.

Send to Device

Social Share

Stub Session Set

Widget Action

We are using the custom event `widget_action` to track the behaviour of javascript widgets.

How do you chose between ``widget_action`` and ``cta_click``?

widget_action	cta_click
The action is specific or unique. *(Only the language switcher changes the page language.)*	The action is "click".
The user does not leave the page.	It sends the user somewhere else.
It requires Javascript to work.	No JS required.
It can perform several actions. *(A modal can be opened and closed.)*	It does one action.
There could be several on the page doing different things. *(An accordion list of FAQs)*	There could be several on the page doing the same thing. *(A download button in the header and footer.)*

Properties for use with *widget_action* (not all widgets will use all options):

- **type**
  - **Required.**
  - The type of widget.
  - Examples: "modal", "protection report", "affiliate notification", "help icon".
  - *Avoid "button" or "link". If you want to track a link or button use `cta_click`.*
- **action**
  - **Required.**

  - The thing that happened.
  - Examples: “open”, “accept”, “timeout”, “vote up”.
  - *Avoid “click”. If you want to track a click use `cta_click`.*

- **text**
  - How is this action labeled to the user?
  - Examples: “Okay”, “Check your protection report”, “Get the app”
- **name**
  - Give the widget a name.
  - You probably only need this optional attribute if the *text* value is not enough to tell the widgets apart.
  - This can help you group actions from the same widget, or make it easier to find the widget in the reports.
  - The dashes are not required but they’re allowed if you want to match the element class or ID.
  - Examples: “dad-joke-banner”, “focus-qr-code”, “Join Firefox Modal”
- **non_interaction (boolean)**
  - True if the action was triggered by something other than a user gesture.
  - If it’s not included we assume the value is *false*

To use `widget_action` push your event to the `dataLayer`:

```
window.dataLayer.push({
    event: 'widget_action',
    type: 'banner',
    action: 'accept',
    name: 'dad-jokes-banner'
});

window.dataLayer.push({
    event: 'widget_action',
    type: 'modal',
    action: 'open',
    name: 'help-icon'
    text: 'Get Browser Help'
});

window.dataLayer.push({
    event: 'widget_action',
    type: 'vote',
    action: 'helpful',
    name: 'vpn-resource-center'
    text: 'What is an IP address?'
});

window.dataLayer.push({
    event: 'widget_action',
    type: 'details',
    action: 'open',
    name: 'relay-faq'
```

(continues on next page)

(continued from previous page)

```
    text: 'Where is Relay available?'
});
```

Dimension Set

When using GA4 through GTM there isn't a way to set user scoped custom dimensions without an accompanying event. The custom event we use for this is *dimension_set*.

```
window.dataLayer.push({
    event: 'dimension_set',
    firefox_is_default: true
});
```

User scoped custom dimensions must be configured in GA4. The list of supported custom dimensions is:

- *firefox_is_default* (boolean)
- *firefox_is_signed_in* (boolean)

Glean

In addition to GA, Bedrock also runs a parallel web analytics implementation using Mozilla's own Glean telemetry SDK (Software Development Kit).

One advantage to Glean is that it is a first-party solution, meaning that we have full control over the data we collect and how it is used. It is also less likely to be blocked by ad blockers or privacy tools.

Using Glean's standardized schema for data collection, we can also take advantage of automated dashboard creation in Looker (see below), which makes it easier to query data than some other tools such as GA4's default dashboard.

Where can I query Glean data?

The easiest place to view Glean data is in Looker:

- Website sessions dashboard
- Event monitoring dashboard

If you need more detailed queries, you can click "Explore from here" from within each visualization to create your own queries.

It is also possible to create more complex queries for raw Glean events using any of our standard Telemetry tools. The easiest way to do this is via the Glean Dictionary. For example, if you view the page load ping, you will see a table in the "Access" section (see screenshot below) that contains different links to query the event data.

Access

GLAM	This application is not supported by GLAM.
Looker	In event_counts as glean.page_load On the Event Monitoring Dashboard for the page_load event
BigQuery	In bedrock.events and bedrock.events > bedrock.events (event.category=glean and event.name=page_load)
Data Catalog	metrics.event.glean_page_load
STMO	Start a query in STMO with the following SQL ➡ Generate SQL

Filtering out non-production pings

Bedrock automatically sets an `app_channel` tag with a value of either `prod` or `non-prod`, depending on the environment. This is present in all pings in the `client_info` section, and is useful for filtering out non-production data in telemetry dashboards.

If you are viewing one of the dashboards linked above, make sure you set the `app_channel` filter to `prod` to only see production data.

Recording page load events

Glean automatically records a page load event when the page is loaded. This event contains basic information about the page, such as the URL, the page title, and the referrer. The page load event is recorded in the `glean.page_load` event. Each page load event is associated with a unique `glean.page_id` metric, which is used to group all events related to a single page view / session.

Recording click events

Glean will automatically record click events on any HTML element that has at least one of the following data attributes:

- `data-glean-id`: A string indicating an identifier of the clicked element.
- `data-glean-type`: A string indicating the type of the clicked element.
- `data-glean-label`: A string indicating the label of the clicked element.

Each click event will also record a `glean.page_id` metric, so that we can associate the click event with the page view that triggered it.

Bedrock also has a custom `Mozilla.Glean.clickEvent()` helper that can be used to record click events directly via JavaScript:

```
if (typeof window.Mozilla.Glean !== 'undefined') {
    window.Mozilla.Glean.clickEvent({
        id: 'firefox_download',
        type: 'macos',
        label: 'release'
    });
}
```

Important: When calling `Mozilla.Glean.clickEvent()` directly, make sure to always check if the `Mozilla.Glean` object is defined first.

Defining additional metrics and pings

Outside of the standard page load and click event metrics recorded by Glean, any additional metrics we send to the Glean pipeline is defined in YAML (Yet Another Markup Language) schema files in the `./glean/` project root directory. The `metrics.yaml` file defines all the different metrics types and events we record.

Note: Before running any Glean commands locally, always make sure you have first activated your virtual environment by running `pyenv activate bedrock`.

When Bedrock starts, we automatically run `npm run glean` which parses these schema files and then generates some JavaScript library code in `./media/js/libs/glean/`. This library code is not committed to the repository on purpose, in order to avoid people altering it and becoming out of sync with the schema. This library code is then imported into our Glean analytics code in `./media/js/glean/`, which is where we initiate page views and capture click events.

Running `npm run glean` can also be performed independently of starting bedrock. It will also first lint the schema files.

Important: All metrics and events we add to the YAML file first undergo a data review before being recorded in production. Additionally changes or updates to existing metrics should also undergo a data review.

Debugging pings

Glean supports debugging pings via a set of flags that can be enabled directly in the browser's web console.

- `window.Glean.setLogPings(true)` (enable verbose ping logging in the web console).
- `window.Glean.setDebugViewTag('bedrock')` (send pings to the Glean debug dashboard with the tag name `bedrock`).
- You can also use `window.Glean.debugSession()` for automatically opening a link to the Debug Ping Viewer with your current session selected.

Note: After enabling Glean debugging in the web console, it will be remembered when navigating across pages using `sessionStorage`. To stop debugging, you need to either close the browser tab, or delete the items from `sessionStorage`. You can disable ping logging by calling `window.Glean.setLogPings(false)`.

How can visitors opt out of Glean?

Website visitors can opt out of Glean by visiting the cookie settings page, which is linked to in the main site footer on every page. Clicking opt-out will set a cookie which Glean checks for before initializing on page load.

1.22.2 Firefox desktop attribution

Firefox Desktop Attribution (often referred to as Stub Attribution) is a system that enables Mozilla to link website attributable referral data (including Google Analytics data) to a user's Firefox profile. When a website visitor lands on www.mozilla.org and clicks to download Firefox, we pass attribution data about their visit to the Firefox installer for inclusion in Telemetry. This is to enable Mozilla to better understand how changes to our website and different marketing campaigns can affect installation rates, as well as overall product retention. The data also gives us an insight into how many installations originate from www.mozilla.org, as opposed to elsewhere on the internet.

Scope and requirements

- Attribution was originally only possible via the Firefox stub installer on Windows (hence the name *stub attribution*), however it now also works on full installer links, and across all desktop release channels.
- Attribution now also works on macOS. The flow does not yet work for Linux, Android or iOS devices.
- Attribution will only be passed if a website visitor has their Do Not Track (DNT) and Global Privacy Control (GPC) preferences disabled in their browser. Visitors can opt-out by enabling DNT or GPC. This is covered in our privacy policy.

How does attribution work?

See the Application Logic Flow Chart for a more detailed visual representation of the steps below (Mozilla access only).

1. A user visits a page on www.mozilla.org. On page load, a JavaScript function collects referral and analytics data about from where their visit originated (see the table below for a full list of attribution data we collect).
2. Once the attribution data is validated, bedrock then generates an attribution session ID. This ID is included in the user's attribution data, and is also sent to Google Analytics as a non-interaction event.
3. Next we send the attribution data to an authentication service that is part of bedrock's back-end server. The data is validated again, then base64 encoded and returned to the client together with an signed, encrypted signature to prove that the data came from www.mozilla.org.
4. The encoded attribution data and signature are then stored as cookies in the user's web browser. The cookies have the IDs `moz-stub-attribution-code` (the attribution code) and `moz-stub-attribution-sig` (the encrypted signature). Both cookies have a 24 hour expiry.
5. Once the user reaches a Firefox download page, bedrock then checks if both attribution cookies exist, and if so appends the authenticated data to the Firefox download link. The query parameters are labelled `attribution_code` and `attribution_sig`.
6. When the user clicks the Firefox download link, another attribution service hosted at `download.mozilla.org` then decrypts and validates the attribution signature. If the secret matches, a unique download token is generated. The service then stores both the attribution data (including the Google Analytics client ID) and the download token in Mozilla's private server logs.
7. The service then passes the download token and attribution data (excluding the GA client ID) into the installer being served to the user.

8. Once the user installs Firefox, the data that was passed to the installer is then stored in the users' Telemetry profile.
9. During analysis, the download token can be used to join Telemetry data with the corresponding GA data in the server logs.

Note: The stub attribution script uses the attribute `data-download-version` to identify what links are download links.

Attribution data

Name	Description	Example
`utm_source`	Query param identifying the referring site which sent the visitor.	`utm_source=google`
`utm_medium`	Query param identifying the type of link, such as referral, cost per click, or email.	`utm_medium=cpc`
`utm_campaign`	Query param identifying the specific marketing campaign that was seen.	`utm_campaign=fast`
`utm_content`	Query param identifying the specific element that was clicked.	`utm_content=getfirefox`
`referrer`	The domain of the referring site when the link was clicked.	`google.com`
`ua`	Simplified browser name parsed from the visitor's User Agent string.	`chrome`
`experiment`	Query param identifying an experiment name that visitor was a cohort of.	`taskbar`
`variation`	Query param identifying the experiment variation that was seen by the visitor.	
`client_id_ga`	Google Analytics 4 Client ID.	`1715265578.1681917481`
`session_id`	A random 10 digit string identifier used to associate attribution data with GA session.	`9770365798`
`dlsource`	A hard-coded string ID used to distinguish mozorg downloads from archive downloads	`mozorg`

Note: If any of the above values are not present then a default value of `(not set)` will be used.

Cookies

The cookies created during the attribution flow are as follows:

Name	Value	Domain	Path	Expiry
`moz-stub-attribution-code`	Base64 encoded attribution string	`www.mozilla.org`	`/`	24 hours
`moz-stub-attribution-sig`	Base64 encoded signature	`www.mozilla.org`	`/`	24 hours

Measuring campaigns and experiments

Firefox Desktop Attribution was originally designed for measuring the effectiveness of marketing campaigns where the top of the funnel was outside the remit of www.mozilla.org. For these types of campaigns, stub attribution requires zero configuration. It just works in the background and passes along any attribution data that exists.

It is also possible to measure the effectiveness of experiments on installation rates and retention. This is achieved by adding optional `experiment` and `variation` parameters to a page URL. Additionally, these values can also be set via JavaScript using:

```
Mozilla.StubAttribution.experimentName = 'experiment-name';
Mozilla.StubAttribution.experimentVariation = 'v1';
```

Note: When setting a experiment parameters using JavaScript like in the example above, it must be done prior to calling `Mozilla.StubAttribution.init()`.

Return to addons.mozilla.org (RTAMO)

Return to AMO (RTAMO) is a Firefox feature whereby a first-time installation onboarding flow is initiated, that redirects a user to install the extension they have chosen whilst browsing AMO using a different browser. RTAMO works by leveraging the existing stub attribution flow, and checking for specific `utm_` parameters that were passed if the referrer is from AMO.

Specifically, the RTAMO feature looks for a `utm_content` parameter that starts with `rta:`, followed by an ID specific to an extension. For example: `utm_content=rta:dUJsb2NrMEByYXltb25kaGlsbC5uZXQ`. The stub attribution code in bedrock also checks the referrer before passing this on, to make sure the links originate from AMO. If RTAMO data comes from a domain other than AMO, then the attribution data is dropped.

RTAMO initially worked for only a limited subset of addons recommended by Mozilla. This functionality was recently expanded by the AMO team to cover all publically listed addons, under a project called *Extended RTAMO (ERTAMO)*.

Important: Because RTAMO is a user facing feature, expressly requested by the user from the AMO page, we deem the RTAMO flow as an essential/necessary use of attribution data. We do however limit the amount of data we collect to only what's essential for the RTAMO to function as a feature. Non-essential fields such as the Google Analytics client ID are omitted. We also continue to respect Do Not Track (DNT) as a valid opt-out signal.

How can visitors opt out?

Visitors to the website can opt-out of desktop attribution on our website by enabling Do Not Track (DNT) in their web browser. We facilitate this by using a DNT helper that our team maintains.

Local testing

For stub attribution to work locally or on a demo instance, a value for the HMAC key that is used to sign the attribution code must be set via an environment variable e.g.

```
STUB_ATTRIBUTION_HMAC_KEY=thedude
```

Note: This value can be anything if all you need to do is test the bedrock functionality. It only needs to match the value used to verify data passed to the stub installer for full end-to-end testing via Telemetry.

Manual testing for code reviews

You might not need to test all these depending on what is changing this is an exhaustive testing guide. This guide assumes demo1, make sure you're testing on the right URL.

1. Use Chrome on Windows or MacOS with DNT and adblocking disabled.
2. Open https://www-demo1.allizom.org/en-US/?utm_source=ham&utm_campaign=pineapple
3. Using Dev Tools, open the Application tab and inspect cookies.
4. Look for a cookie called *moz-stub-attribution-code* and copy the value (it should be a base64 encoded string).
5. **Decode the base64 string (e.g. using https://base64decode.org) and check that:**
    - *dlsource* parameter value is mozorg
    - *client_id_ga4* and *session_id* parameters exist
    - *client_id_ga4* should look something like 0700077325.1656063224 (the numbers will differ but the format with the middle period should look the same).
    - *source* and *campaign* have the values ham and pineapple, respectively.
    - The ua value should be chrome (assuming you tested in Chrome).
    - Everything else should be (not set).
6. Inspect the "Download Firefox" button in the top right and verify the download URL contains *attribution_code* and *attribution_sig* params.
7. Click "Download Firefox".
8. Inspect the "Try downloading again" link and check for the *attribution_code* and *attribution_sig* params. - decode the value of *attribution_code* to check it has the expected values

Other places on the site you may want to check:

- firefox/all (inspect the network request to check that the attribution params were added on click)
- firefox/new
- firefox/enterprise

1.22.3 Firefox mobile attribution

For Firefox mobile referrals we use native app store web links with additional campaign parameters to help measure download to install rates.

App store url helpers

To help streamline creating app store referral links we have *app_store_url()* and *play_store_url()* helpers, which accept a *product* ` name and an optional *campaign* ` parameter.

For example:

```
play_store_url('firefox', 'firefox-home')
app_store_url('firefox', 'firefox-home')
```

Would render:

```
https://play.google.com/store/apps/details?id=org.mozilla.firefox&referrer=utm_source
↪%3Dwww.mozilla.org%26utm_medium%3Dreferral%26utm_campaign%3Dfirefox-home&hl=en
https://apps.apple.com/us/app/apple-store/id989804926?pt=373246&ct=firefox-home&mt=8
```

For Firefox Focus:

```
play_store_url('focus', 'firefox-browsers-mobile-focus')
app_store_url('focus', 'firefox-browsers-mobile-focus')
```

Would render:

```
https://play.google.com/store/apps/details?id=org.mozilla.focus&referrer=utm_source
↪%3Dwww.mozilla.org%26utm_medium%3Dreferral%26utm_campaign%3Dfirefox-browsers-mobile-
↪focus&hl=en
https://apps.apple.com/us/app/apple-store/id1055677337?pt=373246&ct=firefox-browsers-
↪mobile-focus&mt=8
```

App store redirects

Occasionally we need to create a link that can auto redirect to either the Apple App Store or the Google Play Store depending on user agent. A common use case is to embed inside a QR Code, which people can then scan on their phone to get a shortcut to the app. To make this easier bedrock has a special redirect URL to which you can add product and campaign query strings. When someone hits the redirect URL, bedrock will attempt to detect their mobile platform and then auto redirect to the appropriate app store.

The base redirect URL is `https://www.mozilla.org/firefox/browsers/mobile/app/`, and to it you can add both a `product` and `campaign` query parameter. For example, the following URL would redirect to either Firefox on the Apple App Store or on the Google Play Store, with the specified campaign parameter.

```
https://www.mozilla.org/firefox/browsers/mobile/app/?product=firefox&campaign=firefox-
↪whatsnew
```

Note: The `product` and `campaign` parameters are limited to a set of strictly trusted values. To add more product and campaign options, you can add those values to the `mobile_app` helper function in firefox/redirects.py.

Where can I find mobile attribution data?

You can find Firefox Android client attribution data in Looker. Firefox iOS data is currently only available in App Store Connect, however this will also be added to Looker in the near future.

1.22.4 Mozilla accounts attribution

For products such as Mozilla VPN, Relay, and Monitor, we use Mozilla account as an authentication and subscription service. In addition to Google Analytics for basic conversion tracking, we attribute web page visits and clicks and through to actual subscriptions and installs by passing a specific allow-list of known query parameters through to the subscription platform. This is accomplished by adding referral data as parameters to sign up links on product landing pages.

How does attribution work?

When using any of the *Mozilla accounts helpers* in bedrock, a default set of attribution parameters are added to each account sign-in / subscription link on a product landing page. Here's what we set for Mozilla VPN, as an example:

Name	Description	Example value
`utm_source`	Query param identifying the referring site which sent the visitor.	`www.mozilla.org-vpn-product-page`
`utm_medium`	Query param identifying the type of link, such as referral, cost per click, or email.	`referral`
`utm_campaign`	Query param identifying the specific marketing campaign that was seen.	`vpn-product-page`
`entrypoint`	ID for which page of the website the request originates from (used for funnel analysis).	`www.mozilla.org-vpn-product-page`
`device_id`	ID that correlates to the active device being used (used for funnel analysis).	Alpha numeric string
`flow_id`	The flow identifier. A randomly-generated opaque ID (used for funnel analysis).	Alpha numeric string
`flow_begin_t`	The time at which a flow event occurred (used for funnel analysis).	Timestamp
`service`	Product ID used for data analysis in BigQuery (optional).	Alpha numeric string

When performing data analysis, the default UTM values above are what we equate to "direct" traffic (i.e. someone came to the landing page directly then subscribed. They did not arrive from a specific marketing campaign or other channel).

If we do detect that someone came from a marketing campaign or other form of referral, then we have logic in place that will replace the default UTM parameters on each link with more specific referral data, so that we can attribute subscriptions to individual campaigns.

We also support passing several other optional referral parameters:

Name	Description	Example value
coupon	A coupon code that can be automatically applied at checkout (case sensitive).	VPN20
entrypoint_experime	Experiment name ID.	Alpha numeric string
entrypoint_variatio	Experiment variation ID	Alpha numeric string

Attribution logic

See the Application Logic Flow Chart for a visual representation of the steps below (Mozilla access only).

1. A website visitor loads a product landing page in their web browser.
2. A JavaScript function then checks for and validates attribution data via a list of known URL parameters (see tables above).
3. If there are UTM parameters in the referral data, then those are used to replace the default values in each sign up link. Additionally if `coupon` or `entrypoint_experiment` params found, those are also appended.
4. If no UTM params exist, but there is a referrer cookie set, then the cookie value is used for `utm_campaign` and `utm_source` is set to `www.mozilla.org`. This cookie is often set when we display a "Get Mozilla VPN" promo on another mozorg page, such as `/whatsnew`.
5. If there's no referrer cookie, we next look at `document.referrer` to see if the visitor came from a search engine. If found, we set `utm_medium` as `organic` and `utm_source` as the search engine name.
6. Next, an metrics function makes a flow API request to the Mozilla accounts authentication server. The request returns a series of metrics parameters that are used to track progress through the sign-up process. These "flow" parameters are also appended to each sign up link in addition to the existing attribution data.
7. When someone clicks through and completes the sign up process, attribution data we passed through is emitted as event logs. This data is then joined to a person's Mozilla account data during the Data Science team's ETL process (Extract, Transform, Load), where data is then brought together in Big Query.

Note: UTM parameters on sign up links will only be replaced if the page URL contains both a valid `utm_source` and `utm_campaign` parameter. All other UTM parameters are considered optional, but will still be passed through, as long as the required parameters exist. This is to avoid mixing referral data from different campaigns.

Attribution referrer cookie

In situations where we want to try and track a visitor's first entry point, say if someone lands on a `/whatsnew` page and then clicks on a "Get Mozilla VPN" promo link, then we can set a referral cookie in someone's browser when they click a same-site link (step 4 in the list above).

The cookie can be set simply by adding the class name `js-fxa-product-referral-link` to a same-site link, along with a `data-referral-id` attribute. When clicked, our attribution logic will use the value of `data-referral-id` to augment `utm_campaign` when someone click through to the product page.

For example, a referral with `data-referral-id="navigation"` would result in the following utm parameters being set on sign up links in the product landing page:

- `utm_source=www.mozilla.org`.

- `utm_campaign=navigation`.
- `utm_medium=referral`.

Mozilla VPN referral link helper

For Mozilla VPN, there's a `vpn_product_referral_link` helper built specifically to help implement account referral links to the VPN landing page:

```
{{ vpn_product_referral_link(
    referral_id='navigation',
    link_to_pricing_page=True,
    link_text='Get Mozilla VPN',
    class_name='mzp-t-secondary mzp-t-md',
    page_anchor='#pricing',
    optional_attributes= {
        'data-cta-text' : 'Get Mozilla VPN',
        'data-cta-type' : 'button',
        'data-cta-position' : 'navigation',
    }
) }}
```

The helper supports the following parameters:

Parameter name	Definition	Format	Example
`referral_i`	The ID for the referring page / component. This serves as a value for 'utm_campaign'.	String	'navigation'
`link_to_pr`	Link to the pricing page instead of the landing page (defaults to `False`).	Boolean	True
`link_text`	The link copy to be used in the call to action.	Localized string	'Get Mozilla VPN'
`class_name`	A class name to be applied to the link (typically for styling with CSS).	String of one or more class names	'mzp-t-secondary mzp-t-md'
`page_anchc`	An optional page anchor for the link destination.	String	'#pricing'
`optional_a`	An dictionary of key value pairs containing additional data attributes to include in the button.	Dictionary	{'data-cta-text': 'Get Mozilla VPN', 'data-cta-type': 'vpn', 'data-cta-position': 'navigation'}

The cookie has the following configuration:

Cookie name	Value	Domain	Expiry
`fxa-product-referral-id`	Campaign identifier	`www.mozilla.org`	1 hour

Flow metrics

Whilst UTM parameters are passed through to sign up links automatically for any page of the website, in order for flow metrics to be added to links, a specific JavaScript bundle needs to be manually run in the page that requires it. The reason why it's separate is that depending on the situation, flow metrics need to get queried and added at specific times and conditions (more on that below).

To add flow metrics to links, a page's respective JavaScript bundle should import and initialize the `FxaProductButton` script.

```
import FxaProductButton from './path/to/fxa-product-button.es6.js';

FxaProductButton.init();
```

The above JS is also available as a pre-compiled bundle, which can be included directly in a template:

```
{{ js_bundle('fxa_product_button') }}
```

When *init()* is called, flow metrics will automatically be added to add account sign up links on a page.

Important: Requests to metrics API endpoints should only be made when an associated CTA is visibly displayed on a page. For example, if a page contains both a Mozilla accounts sign-up form and a Mozilla Monitor button, but only one CTA is displayed at any one time, then only the metrics request associated with the visible CTA should occur.

Note: For links generated using the `fxa_link_fragment` helper, you will also need to manually add a CSS class of `js-fxa-product-button` to trigger the script.

Google Analytics guidelines

For GTM (Google Tag Manager) `datalayer` attribute values in Mozilla account links, please use the *analytics* documentation.

1.22.5 Mozilla CJMS affiliate attribution

The CJMS affiliate attribution flow comprises an integration between the Commission Junction (CJ) affiliate marketing event system, bedrock, and the Security and Privacy team's CJ micro service (CJMS).

The system allows individuals who partner with Mozilla, via CJ, to share referral links for Mozilla with their audiences. When people subscribe using an affiliate link, the partner can be attributed appropriately in CJ's system.

How does attribution work?

For a more detailed breakdown you can view the full flow diagram (Mozilla access only), but at a high level the logic that bedrock is responsible for is as follows:

1. On pages which include the script, on page load, a JavaScript function looks for a `cjevent` query parameterin the page URL.
2. If found, we validate the query param value and then `POST` it together with a Firefox Account `flow_id` to the CJMS.

3. The CJMS responds with an affiliate marketing ID and expiry time, which we then set as a first-party cookie. This cookie is used to maintain a relationship between the `cjevent` value and an individual `flow_id`, so that successful subscriptions can be properly attributed to CJ.
4. If a website visitor later returns to the page with an affiliate marketing cookie already set, then we update the `flow_id` and `cjevent` value (if a new one exists) via `PUT` on their repeat visit. This ensures that the most recent CJ referral is attributed if/when someone decides to purchase a subscription.
5. The CJMS then responds with an updated ID / expiry time for the affiliate marketing cookie.

How can visitors opt out?

1. To facilitate an opt-out of attribution, we display a cookie notification with an opt-out button at the top of the page when the flow initiates.
2. If someone clicks "Reject" to opt-out, we generate a new `flow_id` (invalidating the existing `flow_id` in the CJMS database) and then delete the affiliate marketing cookie, replacing it with a "reject" preference cookie that will prevent attribution from initiating on repeat visits. This preference cookie will expire after 1 month.
3. If someone clicks "OK" or closes the opt-out notification by clicking the "X" icon, here we assume the website visitor is OK with attribution. We set an "accept" preference cookie that will prevent displaying the opt-out notification on future visits (again with a 1 month expiry) and allow attribution to flow.

Cookies

The affiliate cookie has the following configuration:

Cookie name	Value	Domain	Expiry
`moz-cj-affiliate`	Affiliate ID	`www.mozilla.org`	30 days

Note: To query what version of CJMS is currently deployed at the endpoint bedrock points to, you can add `__version__` at the end of the base URL to see the release number and commit hash. For example: https://stage.cjms.nonprod.cloudops.mozgcp.net/__version__

1.22.6 Managing Data Privacy and Consent

Please see the internal Confluence page on Mozilla's overall approach to handling data privacy and cookie consent management on www.mozilla.org. This page will include detailed legal guidance and FAQs on the permitted use of tracking technologies.

Here we will cover bedrock's technical approach to implementing data consent per that legal guidance, whilst also balancing UX considerations and input from other teams.

In EU and EAA countries where explicit consent to cookies and analytics is required, there are certain web page URLs where bedrock will display a cookie consent banner. These URLs are stored in a strict allow-list. URLs that are not in this list will neither show a banner, nor load any non-necessary cookies / analytics in the EU/EAA. The intent here is to provide as little disruption for our website visitors as possible, whilst still allowing opt-in to analytics URLs such as campaign pages. It is also possible to force the banner to show on any EU page by adding a query parameter *?mozcb=y* (used for specific campaign traffic sources such as advertisements).

Visitors in the EU/EAA countries can also send an opt-out signal by enabling either Global Privacy Control (GPC) and Do Not Track (DNT) in their browser. If either of these signals are enabled then we do not show a banner. Individual cookie preferences can also be updated via a dedicated cookie settings page linked in the main footer.

In non-EU/EAA countries, non-necessary cookies and analytics are loaded by default. Visitors can still opt out via the cookie settings page. Enabling GPC / DNT will also act as an opt-out signal where needed.

There is a Figma flowchart detailing the general flow of logic. The code that implements this logic can be found in the `media/js/base/consent` directory.

Related dependencies

For more detail documentation on dependencies used for consent management, see their respective GitHub repositories.

- Cookie consent banner repo
- Cookie helper repo
- DNT helper repo

1.23 Architectural Decision Records

We record major architectural decisions for bedrock in Architecture Decision Records (ADR), as described by Michael Nygard. Below is the list of our current ADRs.

1.23.1 1. Record architecture decisions

Date: 2019-01-07

Status

Accepted

Context

We need to record the architectural decisions made on this project.

Decision

We will use Architecture Decision Records, as described by Michael Nygard.

Consequences

See Michael Nygard's article, linked above. For a lightweight ADR toolset, see Nat Pryce's adr-tools.

1.23.2 2. Move CI/CD Pipelines to Gitlab

Date: 2019-10-09

Status

Superseded by 0010

Context

Our current CI/CD pipelines are implemented in Jenkins. We would like to decommission our Jenkins server by the end of this year. We have implemented CI/CD pipelines using Gitlab in other projects, including basket, nucleus and the snippets-service.

Decision

We will move our existing CI/CD pipeline implementation from Jenkins to Gitlab.

Consequences

We will continue to use www-config to version control our Kubernetes yaml files, but we will replace the use of git-sync-operator and its branch with self-managed instances of gitlab runner executing jobs defined in a new .gitlab-ci.yml file leveraging what we have learned implementing similar solutions in nucleus-config, basket-config, and snippets-config. We will also eliminate our last dependency on Deis Workflow, which we have been using for dynamic demo deployments based on the branch name, in favor of a fixed number of pre-configured demo deployments, potentially supplemented by Heroku Review Apps.

1.23.3 3. Use Cloudflare Workers and Convert for multi-variant testing

Date: 2019-10-09

Status

Accepted

Context

Our current method for implementing multi-variant tests involves frequent, often non-trivial code changes to our most high traffic download pages. Prioritizing and running concurrent experiments on such pages is also often complex, increasing the risk of accidental breakage and making longer-term changes harder to roll out. Our current tool, Traffic Cop, also requires significant custom code to accomodate these types of situations. Accurately measuring and reporting on the outcome of experiments is also a time consuming step of the process for our data science team, often requiring custom instrumentation and analysis.

We would like to make our end-to-end experimentation process faster, with increased capacity, whilst also minimizing the performance impact and volume of code churn related to experiments running on our most important web pages.

Decision

We will use Cloudflare Workers to redirect a small percentage of traffic to standalone, experimental versions of our download pages. The worker code will live in the www-workers repository. We will implement a (vetted and approved) third-party experimentation tool called Convert for use on those experimental pages.

Consequences

Convert experiment code will be separated from our main web pages, where the vast majority of our traffic is routed. This will minimize code churn on our most important pages, and also reduce the performance impact and risks involved in using a third-party experimentation tool. Using Cloudflare Workers to redirect traffic to experimental pages also has significant performance benefits over handling redirection client-side.

In terms of features, Convert offers a custom dashboard for configuring, prioritizing, and running multi-variant tests. It also has built-in analysis and reporting tools, which are all areas where we hope to see significant savings in time and resources.

1.23.4 4. Use Fluent For Localization

Date: 2019-12-16

Status

Accepted

Context

The current localization (l10n) system uses the outdated and unsupported .lang format, which our l10n team would prefer to no longer support. Mozilla's current l10n standard for products and websites is Fluent.

Decision

In order to update our l10n practices and technology and support from Mozilla's existing l10n infrastructure and teams we will decomission the .lang system in bedrock and implement one based on Fluent. We will support both during a transition period.

Consequences

Dealing with strings and templates is very different in Fluent (see the updated bedrock docs). There will be a period of developer training and adjustment to the new way of writing and previewing templates. The biggest change is that strings are no longer in the templates at all, and are instead referenced by string IDs which are in Fluent files (.ftl files).

The positive side of this change is that the developer has total control over the strings in the translation files and there are no string extraction or merge steps.

1.23.5 5. Use a Single Docker Image For All Deployments

Date: 2020-07-07

Status

Accepted

Context

We currently build an individual docker image for each deployment (dev, stage, and prod) that contains the proper data for that environment. It would save time and testing if we only built a single image that could be promoted to each environment and loaded with the proper data at startup.

Decision

We will use a Kubernetes DaemonSet to ensure that a data updater pod is running on each node in a cluster. This pod will keep the database and l10n files updated in a volume that will be used by the other bedrock pods to access the data.

GitHub issue

Consequences

This change means that bedrock will be more simple to run because each pod will no longer need to be responsible for keeping its data updated, and so it will run only the bedrock web process and not also the updater daemon. It also means that there is a risk of a bedrock pod being run on a node that hasn't had the updater pod run yet, so there would be no available data. We will handle this by ensuring that bedrock won't start when the data isn't available, and so k8s will not send traffic to those pods until they're successfully up and responding, and will keep trying to start pods on the node untill they succeed.

1.23.6 6. Revise tooling for Python dependency management

Date: 2022-02-25

Status

Superseded by 0007, but the context in this ADR is still useful

Context

At the moment of revisiting our dependency-management approach, Bedrock's Python dependencies were installed from a hand-cut `requirements/*.txt` files which (sensibly) included hashes so that we could be sure about what our Python package installer, `pip`, was actually installing.

However, this process was onerous:

- We had a number of requirements files, `base`, `prod`, `dev`, `migration` (no longer required but still being processed at installation time) and `docs` - all of which had to be hand-maintained.
- Hashes needed to be generated when adding/updating a dependency. This was done with a specific tool `hashin` and needed to be done for each requirement.
- When `pip` detects hashes in a requirements file, it automatically requires hashes for *all* packages it installs, including subdependencies of dependecies mentioned in `requirements/*.txt`. This in turn meant that adding or updating a new dep often required hashing-in one or more subdeps – and at worst, a change or niggle with `pip` would result in a new subdep being implicitly required, which would then fail to install because it was not hashed in to the requirements file.

Other projects (both within MEAO and across Mozilla) used more sophisticated dependency management tools, including:

- `pip-tools` - which draws reqs from an input file and generates a requirements.txt complete with hashes
- `pip-compile-multi` - which extends pip-tools' behaviour to support multiple output files and shared input files
- `poetry` - which combines a lockfile approach with a standalone virtual environment
- `pipenv` - which similarly combines a lockfile with a virtual environment
- `conda` - a language-agnostic package manager and environment management system
- simply `pip`

The ideal solution would support all of the following:

- Simple input file format/syntax
- Ability to pin dependencies
- Support for installing with hash-checking of packages
- Automatic hashing of requirements, rather than having to manually do it with `hashin` et al.
- Support for multiple build configurations (eg prod, dev, docs)
- Dependabot compatibility, so we still get alerts and updates
- An unopinionated approach to virtualenvs – can work with and without them, so that developers can use the virtualenv tooling they prefer and we don't have to use a virtualenv in our containers if we don't want to
- Sufficiently active maintenance of the project
- Use/knowledge of the tooling elsewhere in the broader organisation

Decision

After evaluating the above, including `pip-tools`, `pip-compile-multi` and `poetry` in greater depth, `pip-compile-multi` was selected.

Significant factors were how allows us to pin our top-level dependencies in a clutter-free input format, supports inheritance between files and miltiple output files with ease, and it automatically generates hashes for subdependencies.

Consequences

`pip-compile-multi` has been easily integrated into the Bedrock workflow, but there is one non-trivial downside: Github's Dependabot service does not play well with the combination of multiple requirements files and inheritance between them. As such, does not currently produce reliable updates (either partial updates or some requirements files seem to be ignored entirely). See https://github.com/dependabot/dependabot-core/issues/536

Strictly, though, we don't *need* the convenience of Dependabot - we have a `make` command to identify stale deps and recompiling is another, single, `make` command. Also, we're more likely to compile a bunch of Dependabot PRs into one changeset (eg with `paul-mclendahand`), than to merge them straight to `master`/`main` one at at time. As long as we're getting Github security alerts for vulnerable dependencies, we'll be OK.

That said, if we did find we needed Dependabot compatibility, `pip-tools` and some extra legwork in the Makefile to deal with prod, dev and docs deps separately would likely be a viable alternative.

1.23.7 7. Further revise tooling for Python dependency management

Date: 2022-03-02

Status

Proposed

Context

While pip-compile-multi gave us plenty fot benefits (see ADR 0006) the lack of Dependabot support was an annoyance and replacing it with alternatives seemed fairly involved.

Decision

We've downgraded to regular `pip-compile` and instead are doing the extra legwork in the Makefile instead. The input files are indentical, so we do not need to pin sub-dependencies, and we still get automatic hash generation for all packages.

Consequences

There should be no downsides to switching away from pip-compile-multi in this context. If Dependabot still does not manage to parse our multiple requirements files, we should look to renaming them in case that tips the balance (as has been suggested by a colleague)

1.23.8 8. Move Demos To GCP

Date: 2022-07-14

Status

Accepted

Context

Previously, demos for Bedrock were run on Heroku. This worked fine, but Heroku's recent security incident there meant our integration had to be disabled, prompting discussion of self-managed demo instances.

In addition, while it was possible to demo Bedrock in Pocket Mode on Heroku, by amending the settings via the Heroku web UI, the domains set up (www-demoX.allizom.org) were originally set up for Mozorg, and as such may be confusing for colleagues reviewing Pocket changes. Flipping and un-flipping settings in Heroku to enable Mozorg Mode or Pocket Mode was also extra legwork that we ideally would do without, too.

Decision

We have implemented a new, self-managed, approach to running demos, using a handful of Google Cloud Platform services. Cloud Build and Cloud Run are the most significant ones.

Cloud Build has triggers which monitor pushes to specific branches, then builds a Bedrock container from the branch, using the appropriate env vars for Pocket or Mozorg use, including the SITE_MODE env var that specifies the mode Bedrock runs in.

Cloud Run then deploys the built container as a 'serverless' webapp. By default, supervisord runs in the container, so it updates DB and L10N files automatically.

This process is triggered by a simple push to a specific target branch. e.g. pushing code to mozorg-demo-2 will result in the relevant code being deployed in Mozorg mode to www-demo2.allizom.org, while pushing to pocket-demo-4 will deploy it to www-demo4.tekcopteg.com in Pocket mode.

Environment variables can also be configured by developers, via two dedicated env files in the Bedrock codebase, which are only used for demo services. Clashes are unlikely, and can still be managed with common sense.

Consequences

Upsides:

It is now easier to stand up Pocket demos in addition to existing Mozorg demos, plus we have full control over the infrastructure our demos are run on.

We will no longer need to use Heroku for demos. In the future, we may also be able to support ad-hoc 'review apps', which we have also used Heroku for in the past.

Downsides:

1) If a new secret value is required on a demo instance, and so that value cannot go into the demo env vars file because our codebase is public, some SRE-like devops is needed to add that secret value to GCP's Secret Manager Service. This can be quick, but requires understanding how that side fits together, plus access, so may need a backender to add them.

2) At the moment, only the MEAO Backend team have GCP access, which is handy to monitor whether a demo has successfull be pushed out, or to amend secrets, etc. Both of these issues can be addressed without a lot of work.

1.23.9 9. Manage Contentful schema state via migrations

Date: 2022-09-09

Status

Superseded by 0012

Context

Our chosen CMS Contentful is powerful and can be configured via its UI quite easily. However, wanted to bring this under control using migrations so that changes are explicit, reviewable, repeatable and stored. This would be a key part of moving to a "CMS-as-Code" approach to using Contentful, where content-type changes and data migrations (outside of regular content entry) are managed via code.

Decision

We wanted to have as close as possible to the experience provided by the excellent Django Migrations framework, where we would:

- be able to script migrations, rather than resort to "clickops"
- be able to apply them individually or en masse
- be able to store the state of which migrations have/have not been applied in a central datastore (and ideally Contentful)

We experimented with hand-cutting our own framework, which was looking viable, but then we came across https://github.com/jungvonmatt/contentful-migrations which does all of the above. We've evaluated it and it seems fit for purpose, even if it has some gaps, so we've adopted it as our current way to manage and apply migrations to our Contentful setup.

Consequences

We've gained a tool that enables code-based changes to Contentful, which helps in two ways:

1) It enables and eases the initial work to migrate from Legacy Compose to new Compose (these are both ways of structuring pages in Contentful)

2) It lays tracks for moving to CMS-as-Code

1.23.10 10. Move CI to Github Actions for Unit and Integration tests

Date: 2023-04-06

Status

Accepted

Context

Prior to this work, Bedrock's CI/CD pipeline involved Github, Gitlab and CircleCI. We were mirroring from Github to Gitlab to benefit from Gitlab's CI tooling for our functional integration tests, including private (i.e. Mozilla-managed) runners.

Additionally, we were using a third party (CircleCI) to run our Python and JS unit tests.

Since then, two things have changed:

1. Github Actions (GHA) have arrived
2. We are now able to use private runners with GHA

Decision

We will move our CI/CD pipeline from being a combination of Github + Gitlab + CircleCI to just Github, using GHA.

This will mean:

1. The mirroring to Gitlab will no longer be necessary.
2. Unit tests move from CircleCI to GHA. They will continue to be run on every PR raised against `mozilla/bedrock`.
3. Functional/integration tests move from Gitlab to GHA. They will still be triggered by a successful deployment to dev/test/stage/prod.

This work will be carried out in parallel with changes to how our deployment pipeline works, as that side is also being moved out of Gitlab and into GHA + GCP. When a deployment succeeds, a GHA in the deployment repo will trigger a GHA in `mozilla/bedrock`, which will then run the functional integration tests.

Consequences

Pros

- We're no longer mirroring from Github to Gitlab, which will make understanding the deployment pipeline easier for new (and current) developers
- We will no longer have Gitlab in our pipeline, removing a potential point of failure that could block releases
- We can still use private runners for our functional integration tests and more (just via GHA instead of Gitlab), giving us control over security and machine resource spec

Cons

- There's a risk that there will still be new race conditions or CI kick-off failures if the webhook from the deployment repo to mozilla/bedrock fails.
- We will not all get visibility of a failed webhook ping from the deployment repo's GHA, because that's locked down to be private. We can mitigate this risk with a sensible pattern of Slack notifications (e.g. Start, Success, Failure), so a missing notification will itself be a significant thing.

1.23.11 11. Use StatsD for metrics collection

Date: 2023-05-19

Status

Accepted

Context

We need to implement a metrics collection solution to gain insights into the performance and behavior of bedrock. Metrics play a crucial role in understanding system health, identifying bottlenecks, and making informed decisions for optimization and troubleshooting.

Decision

StatsD is a proven open-source solution that provides a lightweight and scalable approach to capturing, aggregating, and visualizing application metrics. It offers numerous benefits that align with bedrock's needs:

1. **Simplicity and Ease of Integration**: StatsD is easy to install and integrate into our existing Python codebase. It provides a simple API that allows us to instrument our code and send metrics with minimal effort.
2. **Aggregation and Sampling**: StatsD supports various aggregation methods, such as sum, average, maximum, and minimum, which can be applied to collected metrics. Additionally, it provides built-in support for sampling, allowing us to reduce the volume of metrics collected while still maintaining statistical significance.
3. **Scalability**: StatsD is designed to handle high volumes of metrics and can easily scale horizontally to accommodate increasing demands. It relies on a fire-and-forget mechanism, where the metrics are sent asynchronously, ensuring minimal impact on the performance of our application.
4. **Integration with Monitoring and Visualization Tools**: At Mozilla we already have a stack available and configured by SRE that uses StatsD along with Telegraf to send metrics to Grafana for visualization and monitoring. This integration will enable us to analyze and visualize our metrics, create dashboards, and set up alerts for critical system thresholds.

Overview of how StatsD, Telegraf, and Grafana work together.

Here's an overview of how these tools fit into the workflow:

- **StatsD**: StatsD is responsible for collecting and aggregating metrics data within the application. It provides a simple API that allows us to instrument our code and send metrics to a StatsD server. StatsD operates over UDP and uses a lightweight protocol for sending metrics.
- **Telegraf**: Telegraf is an agent-based data collection tool that can receive metrics from various sources, including StatsD. Telegraf acts as an intermediary between the data source (StatsD) and the data visualization tool (Grafana). It can collect, process, and forward metrics data to different destinations.
- **Grafana**: Grafana is a popular open-source data visualization and monitoring tool. It provides a rich set of features for creating dashboards, visualizing metrics, and setting up alerts. Grafana can connect to Telegraf to retrieve metrics data and display it in a user-friendly and customizable manner.

Consequences

1. **Metrics Design and Instrumentation**: Proper metrics design and instrumentation are crucial to deriving meaningful insights. We need to invest time and effort in identifying the key metrics to capture and strategically instrument our codebase to provide actionable data for analysis.
2. **Operational Overhead**: Introducing a new tool requires additional operational effort for monitoring, maintaining, and scaling the StatsD infrastructure. However, since this infrastructure is in use currently by other projects within Mozilla, this overhead is already being assumed and is spread out across projects.
3. **Integration Effort**: While integrating StatsD into bedrock is relatively straightforward, we will need to allocate development time to instrument our codebase and ensure that metrics are captured at relevant points within the application.

Considerations and best practices for metrics design

- **Identify Key Metrics**: Identify the key aspects of our website that we want to monitor and measure. These could include response times, error rates, database query performance, and cache hit ratios.
- **Granularity and Context**: Determine the appropriate level of granularity for our metrics. We can choose to measure metrics at the application level, specific Django views, individual API endpoints, or even down to specific functions or code blocks within bedrock.
- **Define Consistent Metric Names**: Choose meaningful and consistent names for our metrics. This helps in easily understanding and interpreting the collected data.
- **Timing Metrics**: Use timing metrics to measure the duration of specific operations. This can include measuring the time taken to render a template, execute a database query, or process a request. StatsD provides a timing metric type that captures the duration and calculates statistics such as average, maximum, and minimum durations.
- **Counting Metrics**: Use counting metrics to track occurrences of specific events. This can include counting the number of requests received or the number of errors encountered. StatsD supports counting metric types that increments a value each time an event occurs.
- **Sampling**: Consider implementing sampling to reduce the number of metrics collected while still maintaining statistical significance. We can selectively sample a subset of requests or events to ensure a representative sample of data for analysis if a particular metric is of high volume.
- **Re-evaluate often**: Continuously evaluate our metrics and refine them based on changing requirements and insights gained from analysis.

1.23.12 12. Use Wagtail as Bedrock's CMS

Date: 2024-04-15

Status

Accepted

Context

As Bedrock evolves, expanding the number of content-managed pages will give us greater agility. We needed to evaluate our options and pick a best-fit solution.

Decision

We previously used Contentful as a headless CMS, but have decided (https://docs.google.com/document/d/1icqCOtCIMhducdrlKKYRBfGsbwsyrFTUH1wvjVldbKo/edit) to move to Wagtail CMS (wagtail.org), which we'll integrate with the Bedrock codebase (https://docs.google.com/document/d/1aQc-FRhI69XQwoaXmvbp9s7zy8UaCVQhZyF6RGTt4Lk/edit)

Consequences

- The integration of a Django-based CMS into the Bedrock codebase will allow for a significantly faster and clearer developer experience when creating content-managed pages, plus the option (over time) for members of the org to create new pages based on CMS templates with no development needed, unless the pages have new designs.
- There is a significant amount of engineering work needed, including:
  - We'll need to integrate Wagtail into Bedrock, which first necessitates refactoring away our bespoke i18n mechanism and using Django's own i18n logic.
  - We'll need to develop workflows around adding Wagtail-managed pages that the whole team understands
  - We'll need to integrate Wagtail with our chosen localization vendor, which requires a custom integration
  - Because we have stopped using Contentful as a source of data, we have the last exported state of the data in our DB, and will need to migrate pages that previously used Contentful to the new CMS

1.24 Browser Support

We seek to provide usable experiences of our most important web content to all user agents. But newer browsers are far more capable than older browsers, and the capabilities they provide are valuable to developers and site visitors. We **will** take advantage of modern browser capabilities. Older browsers **will** have a different experience of the website than newer browsers. We will strike this balance by generally adhering to the core principles of Progressive Enhancement:

- Basic content should be accessible to all web browsers
- Basic functionality should be accessible to all web browsers
- Sparse, semantic markup contains all content
- Enhanced layout is provided by externally linked CSS
- Enhanced behavior is provided by unobtrusive, externally linked JavaScript

- End-user web browser preferences are respected

Some website experiences may require us to deviate from these principles – imagine *a marketing campaign page built under timeline pressure to deliver novel functionality to a particular locale for a short while* – but those will be exceptions and rare.

1.24.1 Browser Support Matrix

Last updated: Updated July 19, 2023

Firefox

It is important for website visitors to be able to download Firefox on a very broad range of desktop operating systems. As such, we aim to deliver enhanced support to user agents in our browser support matrix below.

Enhanced support:

Windows 11 and above

- All evergreen browsers
  - Firefox
  - Firefox ESR
  - Chrome
  - Edge
  - Brave
  - Opera

Windows 10

- All evergreen browsers

macOS 10.15 and above

- All evergreen browsers
- Safari

Linux

- All evergreen browsers

Degraded support:

Website visitors on slightly older browsers fall under degraded support, which means that the website should be fully readable and accessible, but they may not get enhanced CSS layout or JS features.

Windows 10

- Internet Explorer 11

Windows 8.1 and below

- Firefox 115
- Chrome 109
- Internet Explorer 10

macOS 10.14 and below

- Firefox 115
- Chrome 114
- Safari 12.1

Note: As of Firefox 116 (released August 1st 2023), support for Firefox has been ended on Windows 8.1 and below, as well as on macOS 10.14 and below. Website visitors on these outdated operating systems now fall under degraded support, and we offer them to download Firefox ESR instead.

Basic support:

Website visitors on very old versions of Internet Explorer will get only a very basic universal CSS style sheet, and a basic no-JS experience.

Windows 7

- Internet Explorer 9
- Internet Explorer 8

Unsupported:

Even older versions of Internet Explorer are now unsupported.

Windows XP / Vista

- Internet Explorer 7
- Internet Explorer 6

Note: Firefox ended support for Windows XP and Vista in 2017 with Firefox 53. Since then, we have continued to serve those users Firefox ESR 52 instead. However, since then support for downloading has been discontinued. The SSL certificates on download.mozilla.org no longer support TLS 1.0.

Privacy & security products

Browser support for our privacy and security products (such as VPN, Relay, Monitor etc) is thankfully a simpler story. Since all these product use a Firefox account for authentication, we can simply follow the Firefox Ecosystem Platform browser support documentation.

The most notable thing here for bedrock is that Internet Explorer 11 does not need to be supported.

1.24.2 Delivering basic support

On IE browsers that support conditional comments (IE9 and below), basic support consists of no page-specific CSS or JS. Instead, we deliver well formed semantic HTML, and a universal CSS stylesheet that gets applied to all pages. We do not serve these older browsers any JS, with the exception of the following scripts:

- Google Analytics / GTM snippet.
- HTML5shiv for parsing modern HTML semantic elements.
- Stub Attribution script (IE8 / IE9).

Conditional comments should instead be used to handle content specific to IE. To hide non-relevant content from IE users who see the universal stylesheet, a `hide-from-legacy-ie` class name can also be applied directly to HTML:

```
<p class="hide-from-legacy-ie">See what Firefox has blocked for you</p>
```

1.24.3 Delivering degraded support

On other legacy browsers where conditional comments are not supported, developers should instead rely on feature detection to deliver a degraded experience where appropriate.

Note: The following feature detection helpers will return true for all browsers that get enhanced support, but will also return true for IE11 currently, even though that has now moved to degraded support. The reason for this is that whilst many of our newer products don't support IE at all (e.g. Mozilla VPN, Mozilla Monitor, Firefox Relay), we do still need to provide support so that IE users can easily download Firefox. We can decide to update the feature detect in the future, at a time when we think makes sense.

Feature detection using CSS

For CSS, enhanced experiences can be delivered using feature queries, whilst allowing older browsers to degrade gracefully using simpler layouts when needed.

Additionally, there is also a universal CSS class hook available that gets delivered via a site-wide JS feature detection snippet:

```
.is-modern-browser {
    /* Styles will only be applied to browsers that get enhanced support. */
}
```

Feature detection using JavaScript

For JS, enhanced support can be delivered using a helper that leverages the same feature detection snippet:

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

    function onLoad() {
        // Code that will only be run on browsers that get enhanced support.
    }

    window.Mozilla.run(onLoad);
})();
```

The `site.isModernBrowser` global property can also be used within conditionals like so:

```
if (window.site.isModernBrowser) {
    // Code that will only be run on browsers that get enhanced support.
}
```

1.24.4 Exceptions (Updated 2019-06-11)

Some pages of the website provide critical functionality to older browsers. In particular, the Firefox desktop download funnel enables users on older browsers to get a modern browser. To the extent possible, we try to deliver enhanced experiences to all user agents on these pages.

The following pages get enhanced experiences for a longer list of user agents:

- `/firefox/`
- `/firefox/new/`
- `/firefox/download/thanks/`

Note: An enhanced experience can be defined as a step above basic support. This can be achieved by delivering extra page-specific CSS to legacy browsers, or allowing them to degrade gracefully. It does not mean everything needs to look the same in every browser.
