bedrock Documentation

Release 1.0

Mozilla

Apr 09, 2020

Contents

bedrock is the project behind www.mozilla.org. It is as shiny, awesome, and open-sourcy 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 1

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:

```
$ git clone --recursive git://github.com/mozilla/bedrock.git
$ cd bedrock
```

After these basic steps you can choose your install method below. Docker is the easiest and recommended way, but local is also possible and may be preferred by people for various reasons.

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
$ git submodule sync
$ git submodule update --init --recursive
$ [[ ! -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
```

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 3.6+, 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.

You need to create a virtual environment for Python libraries:

```
$ python3 -m venv venv                          # create a virtual env in the folder␣
↪`venv`
$ source venv/bin/activate                      # activate the virtual env. On Windows,
↪ run: venv\Scripts\activate.bat
$ pip install --upgrade pip                     # securely upgrade pip
$ pip install -r requirements/dev.txt           # installs dependencies
```

If you are on OSX and some of the compiled dependencies fails to compile, try explicitly setting the arch flags and try again:

```
$ export ARCHFLAGS="-arch i386 -arch x86_64"
$ pip install -r requirements/dev.txt
```

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

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

Sync the database and all of the external data locally. This gets product-details, security-advisories, credits, release notes, localizations, legal-docs etc:

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

Next, you need to have Node.js and Yarn installed. The node dependencies for running the site can be installed with `yarn`:

```
$ yarn
```

You'll also need to install the Gulp cli globally:

```
$ npm install -g gulp-cli
```

Note: Bedrock uses yarn 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 yarn documentation for adding or upgrading Node.js dependencies.

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 py.test bedrock/firefox
```

This example will run only the unit tests for the `firefox` app in bedrock. You can substitute `py.test 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:

```
$ py.test lib bedrock
```

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

```
$ py.test bedrock/firefox
```

Note: If your local tests run fine, but when you submit a pull-request the tests fail in CircleCI, it could be due to the difference in settings between what you have in `.env` and what CircleCI uses: `docker/envfiles/demo.env`. You can run tests as close to Circle as possible by moving your `.env` file to another name (e.g. `.env-backup`), then copying `docker/envfiles/demo.env` to `.env`, and running tests again.

1.1.3 Make it run

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, and then run the server:

```
$ npm start
```

If you are not inside a virtualenv, you can activate it by doing:

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

Browsersync

Both the Docker and Local methods of running the site use Browsersync to serve the development static-assets (CSS, JS, etc.) as well as refresh the browser tab for you when you change files. The refreshing of the page works by injecting a small JS snippet into the page that listens to the browsersync service and will refresh the page when it receives a signal. It also injects a script that shows a small notification in the top-right corner of the page to inform you that a refresh is happening and when the page connects to or is disconnected from the browsersync service. We've not seen issues with this, but since it is modifying the page it is possible that this could conflict with something on the page itself. Please let us know if you suspect this is happening for you. This notification can be disabled in the browsersync options in the `gulpfile.js` by setting `notify: false` in the `browser-sync` task.

1.1.4 Legal Docs

Legal docs (for example: the privacy policy) are generated from markdown files in the legal-docs repo.

To view them or update to a more recent version update the submodule:

```
$ git submodule update --init --recursive
```

To add a new commit of the git submodule to bedrock:

> $ cd vendor-local/src/legal-docs $ git checkout master $ git pull $ cd .. (back to project root) $ git commit -am "Update legal-docs git submodule"

1.1.5 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*.

1.1.6 Feature Flipping (aka Switches)

Environment variables are used to configure behavior and/or features of select pages on bedrock via a template helper function called `switch()`. It will take whatever name you pass to it (must be only numbers, letters, and dashes), convert it to uppercase, convert dashes to underscores, and lookup that name in the environment. For example: `switch('the-dude')` would look for the environment variable `SWITCH_THE_DUDE`. If the value of that variable is any of "on", "true", "1", or "yes", then it will be considered "on", otherwise it will be "off".

You can also supply a list of locale codes that will be the only ones for which the switch is active. If the page is viewed in any other locale the switch will always return `False`, even in `DEV` mode. This list can also include a "Locale Group", which is all locales with a common prefix (e.g. "en-US, en-GB" or "zh-CN, zh-TW"). You specify these with just the prefix. So if you used `switch('the-dude', ['en', 'de'])` in a template, the switch would be active for German and any English locale the site supports.

You may also use these switches in Python in `views.py` files (though not with locale support). For example:

```
from bedrock.base.waffle import switch

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 "on" unless they are explicitly "off" in the environment. `DEV` defaults to "true" in local development and demo servers.

To test switches locally:

1. Set `DEV=False` in your `.env` file.
2. Enable the switch in your `.env` file.
3. Restart your web server.

To configure switches for a demo branch. Follow the configuration instructions here.

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. You can see the current state of these switches and other configuration values in our configuration repo.

To work with/test these experiment switches locally, you must add the switches to your local environment. For example:

```
# to switch on firstrun-copy-experiment you'd add the following to your ``.env`` file
SWITCH_FIRSTRUN_COPY_EXPERIMENT=on
```

To do the equivalent in one of the bedrock apps see the www-config documentation.

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

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 `locale` directory, this command will clone the git repo containing the .lang 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. It do the same thing for the repository for the .ftl translation files in `git-repos/www-l10n` directory.

1.2.1 Fluent

Bedrock's Localization (l10n) system is based on Project Fluent. This is a departure from the legacy system (see below) that relied 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.

Note: We have some *fluent tools* to aid in the transition from the legacy system.

.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 document 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
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 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.

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).

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(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'),
```

(continues on next page)

(continued from previous page)

```
    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
```

(continues on next page)

(continued from previous page)

```
# 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]
```

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 occurence.

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.

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
```

(continues on next page)

(continued from previous page)

```
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.

1.2.2 Legacy

This section describes the legacy l10n system based on .lang files, which will be frozen and no longer supported for new translations in January of 2020.

.lang files

Bedrock supports a workflow similar to gettext. You extract all the strings from the codebase, then merge them into each locale to get them translated.

The files containing the strings are called ".lang files" and end with a `.lang` extension.

To extract all the strings from the codebase, run:

```
$ ./manage.py l10n_extract
```

If you'd only like to extract strings from certain files, you may optionally list them on the command line:

```
$ ./manage.py l10n_extract bedrock/mozorg/templates/mozorg/contribute.html
```

Command line glob matching will work as well if you want all of the HTML files in a directory, for example:

```
$ ./manage.py l10n_extract bedrock/mozorg/templates/mozorg/*.html
```

That will use gettext to get all the needed localizations from Python and HTML files, and will convert the result into a series of .lang files inside `locale/templates`. This directory represents the "reference" set of strings to be translated, and you are free to modify or split up .lang files here as needed (just make sure they are being referenced correctly, from the code, see *Which .lang file should it use?*).

Once you have extracted .lang files locally, they can then be added via pull request to the mozilla.org l10n repository for translation. You can read the full documentation for more information on the extraction workflow.

Translating with .lang files

To translate a string from a .lang file, simply use the gettext interface.

In a jinja2 template:

```
<div>{{ _('Hello, how are you?') }}<div>

<div>{{ _('<a href="%s">Click here</a>')|format('http://mozilla.org/') }}</div>

<div>{{ _('<a href="%(url)s">Click here</a>')|format(url='http://mozilla.org/') }}</
↪div>
```

Note the usage of variable substitution in the latter examples. It is important not to hardcode URLs or other parameters in the string. jinja's *format* filter lets us apply variables outsite of the string.

You can provide a one-line comment to the translators like this:

```
{# L10n: "like" as in "similar to", not "is fond of" #}
{{ _('Like this:') }}
```

The comment will be included in the .lang files above the string to be translated.

In a Python file, use `lib.l10n_utils.dotlang._` or `lib.l10n_utils.dotlang._lazy`, like this:

```
from lib.l10n_utils.dotlang import _lazy as _

sometext = _('Foo about bar.')
```

You can provide a one-line comment to the translators like this:

```
# L10n: "like" as in "similar to", not "is fond of"
sometext = _('Like this:')
```

The comment will be included in the .lang files above the string to be translated.

There's another way to translate content within jinja2 templates. If you need a big chunk of content translated, you can put it all inside a *trans* block.

```
{% trans %}
  <div>Hello, how are you</div>
{% endtrans %}

{% trans url='http://mozilla.org' %}
  <div><a href="{{ url }}">Click here</a></div>
{% endtrans %}
```

Note that it also allows variable substitution by passing variables into the block and using template variables to apply them.

Which .lang file should it use?

Translated strings are split across several .lang files to make it easier to manage separate projects and pages. So how does the system know which one to use when translating a particular string?

- All translations from Python files are put into main.lang. This should be a very limited set of strings and most likely should be available to all pages.
- Templates always load *main.lang* and *download_button.lang*.
- Additionally, each template has its own .lang file, so a template at *mozorg/firefox.html* would use the .lang file at *<locale>/mozorg/firefox.lang*.

- Templates can override which .lang files are loaded. The above global ones are always loaded, but instead of loading *<locale>/mozorg/firefox.lang*, the template can specify a list of additional lang files to load with a template block:

```
{% add_lang_files "foo" "bar" %}
```

That will make the page load *foo.lang* and *bar.lang* in addition to *main.lang* and *download_button.lang*.

When strings are extracted from a template, they are added to the template-specific .lang file. If the template explicitly specifies .lang files like above, it will add the strings to the first .lang file specified, so extracted strings from the above template would go into *foo.lang*.

You can similarly specify extra .lang files in your Python source as well. Simply add a module-level constant in the file named *LANG_FILES*. The value should be either a string, or a list of strings, similar to the *add_lang_files* tag above.

```
# forms.py

from lib.l10n_utils.dotlang import _

LANG_FILES = ['foo', 'bar']

sometext = _('Foo about bar.')
```

This file's strings would be extracted to *foo.lang*, and the lang files *foo.lang*, *bar.lang*, *main.lang* and *download_button.lang* would be searched for matches in that order.

l10n blocks

Bedrock also has a block-based translation system that works like the `{% block %}` template tag, and marks large sections of translatable content. This should not be used very often; lang files are the preferred way to translate content. However, there may be times when you want to control a large section of a page and customize it without caring very much about future updates to the English page.

A Localizers' guide to l10n blocks

Let's look at how we would translate an example file from **English** to **German**.

The English source template, created by a developer, lives under *apps/appname/templates/appname/example.html* and looks like this:

```
{% extends "base-pebbles.html" %}

{% block content %}
  <img src="someimage.jpg">

  {% l10n foo, 20110801 %}
  <h1>Hello world!</h1>
  {% endl10n %}

  <hr>

  {% l10n bar, 20110801 %}
  <p>This is an example!</p>
  {% endl10n %}
{% endblock %}
```

The `l10n` blocks mark content that should be localized. Realistically, the content in these blocks would be much larger. For a short string like above, please use lang files. We'll use this trivial code for our example though.

The `l10n` blocks are named and tagged with a date (in ISO format). The date indicates the time that this content was updated and needs to be translated. If you are changing trivial things, you shouldn't update it. The point of l10n blocks is that localizers completely customize the content, so they don't care about small updates. However, you may add something important that needs to be added in the localized blocks; hence, you should update the date in that case.

When the command `./manage.py l10n_extract` is run, it generates the corresponding files in the `locale` folder (see below for more info on this command).

The German version of this template is created at `locale/de/templates/appname/example.html`. The contents of it are:

```
{% extends "appname/example.html" %}

{% l10n foo %}
<h1>Hello world!</h1>
{% endl10n %}

{% l10n bar %}
<p>This is an example!</p>
{% endl10n %}
```

This file is an actual template for the site. It extends the main template and contains a list of l10n blocks which override the content on the page.

The localizer just needs to translate the content in the l10n blocks.

When the reference template is updated with new content and the date is updated on an l10n block, the generated l10n file will simply add the new content. It will look like this:

```
{% extends "appname/example.html" %}

{% l10n foo %}
<h1>This is an English string that needs translating.</h1>
{% was %}
<h1>Dies ist ein English string wurde nicht.</h1>
{% endl10n %}

{% l10n bar %}
<p>This is an example!</p>
{% endl10n %}
```

Note the `was` block in `foo`. The old translated content is in there, and the new content is above it. The `was` content is always shown on the site, so the old translation still shows up. The localizer needs to update the translated content and remove the `was` block.

Generating the locale files

```
$ ./manage.py l10n_check
```

This command will check which blocks need to be translated and update the locale templates with needed translations. It will copy the English blocks into the locale files if a translation is needed.

You can specify a list of locales to update:

```
$ ./manage.py l10n_check fr
$ ./manage.py l10n_check fr de es
```

Currency

When dealing with currency, make a separate gettext wrapper, placing the amount inside a variable. You should also include a comment describing the intent. For example:

```
{# L10n: Inserts a sum in US dollars, e.g. '$100'. Adapt the string in your␣
↪translation for your locale conventions if needed, ex: %(sum)s US$ #}
{{ _('$%(sum)s')|format(sum='15') }}
```

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);
```

(continues on next page)

(continued from previous page)

```
}

@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.

Staging Copy Changes

The need will often arise to push a copy change to production before the new copy has been translated for all locales. To prevent locales not yet translated from displaying English text, you can use the `l10n_has_tag` template function. For example, if the string "Firefox benefits" needs to be changed to "Firefox features":

```
{% if l10n_has_tag('firefox_products_headline_spring_2016') %}
  <h1>{{ _('Firefox features') }}</h1>
{% else %}
  <h1>{{ _('Firefox benefits') }}</h1>
{% endif %}
```

This function will check the .lang file(s) of the current page for the tag `firefox_products_headline_spring_2016`. If it exists, the translation for "Firefox features" will be displayed. If not, the pre-existing translation for "Firefox benefits" will be displayed.

When using `l10n_has_tag`, be sure to coordinate with the localization team to decide on a good tag name. Always use underscores instead of hyphens if you need to visually separate words.

1.2.3 All

Locale-specific Templates

While the `l10n_has_tag` or `ftl_has_messages` template functions are great in small doses, they don'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.

Important: Note that the presence of an L10n template (e.g. `locale/de/templates/firefox/new.html`) will take precedence over a locale-specific template in bedrock.

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.

Development

In local development environments and on demo servers all `l10n_has_tag` calls evaluate to true. If the content has not been translated it will display the English strings.

To test l10n locally you can set `DEV=False` in your `.env` file.

If you are running your local server you will need to restart it after altering your `.env` file.

1.3 L10n Fluent Conversion

There are some tools that exist now but are only useful during our conversion from .lang to .ftl files. This document will cover the usage of these.

If you've got a translated page you'd like to convert as-is to the Fluent system then you can follow this procedure to get a big head start.

The key concept in converting a page to Fluent is a recipe. The recipe is a piece of python code that can programmatically generate a Fluent file. It can use existing Fluent files as templates, and .lang localizations as data sources.

All the functionality is provided by the `fluent` management command via subcommands for each phase.

1.3.1 Create a recipe from a template

The first step is to create the recipe. Let's say you want to convert `bedrock/mozorg/templates/mozorg/mission.html`, then you'll run

```
$ make run-shell
$ ./manage.py fluent recipe bedrock/mozorg/templates/mozorg/mission.html
```

This will parse all of the calls to `_()` and the `trans` blocks, process the strings in the same way the old string extraction process did, and create new Fluent string IDs. It will generate the recipe in `lib/fluent_migrations/mozorg/mission.py`. The recipe name is based on the template name after `templates`.

1.3.2 Sanitize recipe

The recipe creates migrations for each localizable string in the template, with some possibly bad string IDs. At this point, you want to change the IDs to be conforming with the best practices laid out in the l10n docs. The existing recipe will already have the template name as prefix, though.

You can also choose to remove strings from the conversion, if you just want to convert a subset of the strings.

If you want to handle things such as updating existing translations to use brand terms as placeholders, you can update the recipe to replace existing string content by using the `REPLACE()` helper:

```
ctx.add_transforms(
    "firefox/switch.ftl",
    "firefox/switch.ftl",
    [
        FTL.Message(
            id=FTL.Identifier("switch-switching-to-firefox-is-fast"),
            value=REPLACE(
                "firefox/switch.lang",
                "Switching to Firefox is fast, easy and risk-free, because Firefox␣
↪imports your bookmarks, autofills, passwords and preferences from Chrome.",
                {
                    "Firefox": TERM_REFERENCE("brand-name-firefox"),
                    "Chrome": TERM_REFERENCE("brand-name-chrome")
                }
            )
        ),
    ]
)
```

Once you're happy with the recipe, you can create the Fluent files and the template.

1.3.3 Convert a .lang file to English .ftl

The next step is to convert an existing english .lang file into an equivalent .ftl file in the `en` folder of the `l10n` directory in bedrock. Let's continue with the example of `mission.html`; you would run:

```
$ make run-shell
$ ./manage.py fluent ftl bedrock/mozorg/templates/mozorg/mission.html
```

This should create `l10n/en/mozorg/mission.ftl` which has all of the strings in the order in which they appear in the template.

You want to sanitize this Fluent file by adding license headers, file comments with staging URLs, as well as comments individual strings or groups of strings.

It's a good idea to add the new Fluent file to a project config, and validate it for errors like duplicated IDs.

```
$ moz-l10n-lint l10n/l10n-pontoon.toml
$ moz-l10n-lint l10n/l10n-vendor.toml
```

1.3.4 Convert the template

With the recipe created in the first step, you'll do the following (assuming your docker shell is still running):

```
$ ./manage.py fluent template bedrock/mozorg/templates/mozorg/mission.html
```

This will reparse the template much in the same way it did when creating the recipe. It will inspect the recipe to see which legacy strings map to which ID, that you've chosen when you sanitized the recipe. It will then take this mapping of IDs and replace all of the old calls with new calls to `ftl()`. If there are any issues you should see warnings printed to your screen, but always inspect the new template and give the page a test run to make sure all is working as expected.

Convert the View or URL

To get it working on the site you do have to do a bit more. The above step creates a new template with a `_ftl.html` suffix instead of overwriting the old one so that you can compare them before removing the old one. You can then either delete the old one and rename the new one with the original name, or keep them both for a while if you may need to quickly switch back. You then need to specify which .ftl file to use by passing it (or them) to the `l10n_utils.render` function in the view, or the `page()` function in urls.py. See the *Specifying Fluent files* section for more details.

```
# urls.py
urlpatterns = [
    page('mission', 'mozorg/mission.html', ftl_files=['mozorg/mission']),
]

# views.py
def mission_view(request):
    return l10n_utils.render(request, 'mozorg/mission.html', ftl_files=['mozorg/
↪mission'])
```

Note: If you are using the `page()` helper and switch to the new template name that will also change the name of the URL referenced by calls to `url` and `reverse` around the site. To avoid this you can pass the original name to the page function, e.g. `url_name='mozorg.mission'`.

1.3.5 Port the translations

The remaining step is to port all of the existing translation in the .lang files over to .ftl files in our fluent files repo.

```
$ ./manage.py fluent ftl bedrock/mozorg/templates/mozorg/mission.html de it
$ ./manage.py fluent ftl lib/fluent_migrations/mozorg/mission.py de it
```

This is the same command we used to create the original `en` Fluent file. As you can see, you can specify both the template path here as well as the recipe path.

Before you run this, make sure to update the local clones of your l10n repositories.

This command will use the Fluent file you edited as template, read the legacy translations from `locale` and write the generated Fluent files for each locale into the `git-repos/www-l10n/` directory.

1.3.6 Commit

After that it's up to you to commit all of these changes and push them to where they need to be: a pull request to bedrock for the template and English .ftl file changes, and a pull request to the www-l10n repo for the translated .ftl files and activation metadata.

1.4 Developing on Bedrock

1.4.1 Writing URL Patterns

URL patterns should be as strict as possible. It should begin with a ^ and end with */$* to make sure it only matches what you specifiy. It also forces a trailing slash. You should also give the URL a name so that other pages can reference it instead of hardcoding the URL. Example:

```
url(r'^channel/$', channel, name='mozorg.channel')
```

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 even need to create a view. It will serve up the specified template at the given URL (the first parameter). 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.

1.4.2 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.

The script *img.sh* can be used to optimize images locally on the command line:

1. Before you run it for the first time you will need to run *yarn* to install dependencies
2. Add the image files to git's staging area *git add **
3. Run the script *./bin/img.sh*
4. The optimized files will not automatically be staged, so be sure to add them before commiting

The script will:

- **optimize JPG and PNG files using tinypng (**
  - this step is optional since running compression on the same images over and over degrades them)
  - you will be prompted to add a TinyPNG API key
- optimize SVG images locally with svgo
- check that SVGs have a viewbox (needed for IE support)
- check that images that end in *-high-res* have low res versions as well

1.4.3 Embedding images

Images should be included on pages using helper functions.

static()

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

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

will output an image:

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

high_res_img()

For images that include a high-resolution alternative for displays with a high pixel density, use the *high_res_img()* function:

```
high_res_img('img/firefox/new/firefox-logo.png', {'alt': 'Firefox', 'width': '200',
↪'height': '100'})
```

The *high_res_img()* function will automatically look for the image in the URL parameter suffixed with *'-high-res'*, e.g. *img/firefox/new/firefox-logo-high-res.png* and switch to it if the display has high pixel density.

high_res_img() supports localized images by setting the *'l10n'* parameter to *True*:

```
high_res_img('img/firefox/new/firefox-logo.png', {'l10n': True, 'alt': 'Firefox',
↪'width': '200', 'height': '100'})
```

When using localization, *high_res_img()* will look for images in the appropriate locale folder. In the above example, for the *de* locale, both standard and high-res versions of the image should be located at *media/img/l10n/de/firefox/new/*.

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*.

platform_img()

Finally, for outputting an image that differs depending on the platform being used, the *platform_img()* function will automatically display the image for the user's browser:

```
platform_img('img/firefox/new/browser.png', {'alt': 'Firefox screenshot'})
```

platform_img() will automatically look for the images *browser-mac.png*, *browser-win.png*, *browser-linux.png*, etc. Platform image also supports hi-res images by adding *'high-res': True* to the list of optional attributes.

platform_img() supports localized images by setting the *'l10n'* parameter to *True*:

```
platform_img('img/firefox/new/firefox-logo.png', {'l10n': True, 'alt': 'Firefox␣
↪screenshot'})
```

When using localization, *platform_img()* will look for images in the appropriate locale folder. In the above example, for the *es-ES* locale, all platform versions of the image should be located at *media/img/l10n/es-ES/firefox/new/*.

1.4.4 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

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.

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):
    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*.

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.conf.urls import url

from bedrock.utils.views import VariationTemplateView

urlpatterns = [
    url(r'^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.conf.urls import url

from bedrock.utils.views import VariationTemplateView

urlpatterns = [
    url(r'^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.conf.urls import url

from bedrock.utils.views import VariationTemplateView

urlpatterns = [
    url(r'^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 Redirect View

We sometimes need to have a special page variation for people visiting from certain countries. To make this easier we have a redirect view class that will allow you to define URLs per country as well as a default for everyone else. This redirector URL must only be a redirector since it must be uncachable by our CDN so that all visitors will hit the server and see the correct page for their location.

```
from bedrock.base.views import GeoRedirectView

class CanadaIsSpecialView(GeoRedirectView):
    geo_urls = {
        'CA': 'app.canada-is-special',
    }
    default_url = 'app.everyone-else'
```

In this example people in Canada would go to the URL that Django returns using *reverse()* (i.e. the name of the URL) and everyone else would go to the *app.everyone-else* URL. You may also use full URLs instead of URL names if you want to. It will look for strings that start with "http(s)://" and use it as is. The country code must be 2 characters and upper case. If the patterns for the redirect and the destination(s) have URL parameters they will be passed to the reverse call for the URL pattern name. So for example if you're doing this for a Firefox page with a version number in the URL, as long as the view and destination URLs use the same URL parameter names it will be preserved in the resulting destination URL. So */firefox/70.0beta/whatsnew/* would redirect to */firefox/70.0beta/whatsnew/canada/* for example. The redirector will also preserve query parameters by default. You can turn that off by setting the *query_string = False* class variable.

1.4.5 Coding Style Guides

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

Use the `.open-sans`, `.open-sans-light` and `.open-sans-extrabold` mixins to specify font families to allow using international fonts. See the *CSS* section in the l10n doc for details.

Use the `.font-size()` mixin to generate root-relative font sizes. You can declare a font size in pixels and the mixin will convert it to an equivalent `rem` (root em) unit while also including the pixel value as a fallback for older browsers that don't support `rem`. This is preferable to declaring font sizes in either fixed units (`px`, `pt`, etc) or element-relative units (`em`, `%`). See this post by Jonathan Snook for more info.

When including CSS blocks, use `{% block page_css %}` for page specific inclusion of CSS. `{% block site_css %}` should only be touched in rare cases where base styles need to be overwritten.

1.4.6 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.

1.5 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.

1.5.1 Git workflow

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

```
git checkout master
git checkout -b topic-branch
```

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

```
git fetch mozilla
git checkout master
git merge mozilla/master
git checkout topic-branch
git rebase master
```

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.

1.5.2 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 master.

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
```

1.5.3 Submitting your work

In general, you should submit your work with a pull request to master. 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.

1.5.4 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*.

1.5.5 Server architecture

Demos

- *URLs:* - http://www-demo1.allizom.org/ - http://www-demo2.allizom.org/ - http://www-demo3.allizom.org/ - http://www-demo4.allizom.org/ - http://www-demo5.allizom.org/
- *Bedrock locales:* dev repo
- *Bedrock Git branch:* `demo/1`, `demo/2`, etc.

On-demand demos

- *URLs:* Demo instances can also be spun up on-demand by pushing a branch to the mozilla bedrock repo that matches a specific naming convention (the branch name must start with `demo/`). Jenkins will then automate spinning up a demo instance based on that branch. For example, pushing a branch named `demo/feature` would create a demo instance with the following URL: `https://bedrock-demo-feature.oregon-b.moz.works/`
- *Bedrock locales:* dev repo
- *Bedrock Git branch:* any branch named starting with `demo/`

Note: Deployed demo instances are not yet automatically cleaned up when branches are deleted, so to avoid lots of instances piling up it is currently recommended to try and limit a single demo instance per developer, reusing a branch such as *demo/<your_username>*.

Dev

- *URL:* http://www-dev.allizom.org/
- *Bedrock locales:* dev repo
- *Bedrock Git branch:* master, deployed on git push

Stage

- *URL:* http://www.allizom.org/
- *Bedrock locales:* prod repo
- *Bedrock Git branch:* prod, deployed on git push with date-tag

Production

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

You can check the currently deployed git commit by checking https://www.mozilla.org/revision.txt.

1.5.6 Pushing to production

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

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 master. Look at `open-compare.py -h` for more options.

We automate pushing to production via tagged commits (see *Push to prod branch (tagged)*)

1.6 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*).

1.6.1 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 all integration tests can be executed against a locally running instance of the application.

Pull request

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

Push to master branch

Whenever a change is pushed to the master branch, a new image is built and deployed to the dev environment, and the full suite of headless and UI tests are then run against Firefox on Windows 10 using Sauce Labs. This is handled by the pipeline, and is subject to change according to the settings in the .gitlab-ci.yml file in the www-config repository.

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 with Firefox, Chrome, and Internet Explorer on Windows 10, and the sanity suite is run with IE9.

Push to prod branch (tagged)

When a tagged commit is pushed to the prod branch, a production docker image is built and published to Docker Hub if needed (usually this will have already happened as a result of a push to the stage branch), and deployed to each production deployment. After each deployment is complete, the full suite of UI tests is run against Firefox, Chrome and Internet Explorer on Windows 10, and the sanity suite is run against IE9. As with untagged pushes, this is all handled by the pipeline, and is subject to change according to the settings in the .gitlab-ci.yml file in the www-config repository.

Push to prod cheat sheet

1. Check out the `master` branch
2. Make sure the `master` branch is up to date with `mozilla/bedrock master`
3. **Check that dev deployment is green:**
    1. View deployment pipeline and look at `master` branch
4. 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.

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.

Updating Selenium

There are two components for Selenium, which are independently versioned. The first is the Python client, and this can be updated via the test dependencies. The other component is the server, which in the pipeline is either provided by a Docker container or Sauce Labs. The `SELENIUM_VERSION` environment variable controls both of these, and they should ideally use the same version, however it's possible that availability of versions may differ. You can check the Selenium Docker versions available. If needed, the global default can be set and then can be overridden in the individual job configuration.

Adding test runs

Test runs can be added by creating a new job in the .gitlab-ci.yml file in the www-config repository with the desired variables For example, if you wanted to run tests in Firefox on both Windows 10 and OS X, against our dev environment, you could create the following clauses:

```
```

dev-test-firefox-osx:

extends:

- .dev
- .test

variables: BROWSER_NAME: firefox BROWSER_VERSION: latest PLATFORM: OS X 10.11

dev-test-firefox-win10:

extends:

- .dev
- .test

variables: BROWSER_NAME: firefox BROWSER_VERSION: latest PLATFORM: Windows 10

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

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 `bedrock-integration-tests` app will be deployed and all of the integration tests defined in the `jenkins.yml` file for that branch will be run. Please announce in our IRC channel (#www on irc.mozilla.org) that you'll be doing this so that we don't get conflicts.

1.7 Front-end testing

Bedrock runs a suite of front-end Jasmine behavioral/unit tests, which use Karma as a test runner. We also have a suite of functional tests using Selenium and pytest. This allows us to emulate users interacting with a real browser. All these test suites live in the `tests` directory.

The `tests` directory comprises of:

- `/functional` contains pytest tests.
- `/pages` contains Python page objects.
- `/unit` contains the Jasmine tests and Karma config file.

1.7.1 Installation

First follow the *installation instructions for bedrock*, which will install the specific versions of Jasmine/Karma which are needed to run the unit tests, and guide you through installing pip and setting up a virtual environment for the functional tests. The additional requirements can then be installed by using the following commands:

```
$ source venv/bin/activate
$ pip install -r requirements/dev.txt
```

1.7.2 Running Jasmine tests using Karma

To perform a single run of the Jasmine test suite using Firefox, type the following command:

```
$ gulp js:test
```

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.

1.7.3 Running functional tests

Note: Before running the functional 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 tests using Firefox, you must also first download geckodriver and make it available in your system path. You can alternatively specify the path to geckodriver using the command line (see the pytest-selenium documentation for more information).

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

```
$ py.test --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 functional 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 py.test to execute a specific file e.g. `tests/functional/test_newsletter.py`:

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

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

```
$ py.test --base-url http://localhost:8000 --driver Firefox --html tests/functional/
↪results.html tests/functional/test_newsletter.py -k test_successful_sign_up
```

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

```
$ py.test --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 py.test --base-url https://
↪www-dev.allizom.org --driver SauceLabs --capability browserName 'internet explorer'␣
↪-n auto --html tests/functional/results.html tests/functional/test_home.py
```

1.7.4 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

@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
```

Sanity tests

Sanity tests are considered to be our most critical tests that must pass in a wide range of web browsers, including old versions of Internet Explorer. Sanity tests are run automatically post deployment on a wider range of browsers & platforms than we run the full suite against. The number of sanity tests we run should remain small, but cover our most critical pages where legacy browser support is important. Sanity tests are typically run after a tagged commit to master (see *Push to prod branch (tagged)*).

```
import pytest

@pytest.mark.sanity
@pytest.mark.nondestructive
def test_click_download_button(base_url, selenium):
    page = FirefoxNewPage(base_url, selenium).open()
    page.download_firefox()
    assert page.is_thank_you_message_displayed
```

You can run sanity tests only by adding `-m sanity` 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.

1.7.5 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`.

1.7.6 Guidelines for writing functional tests

- Try and keep tests organized and cleanly separated. Each page should have its own page object and test file, and each test should be responsible for a specific purpose, or component of a page.
- Avoid using sleeps - always use waits as mentioned above.
- Don't make tests overly specific. If a test keeps failing because of generic changes to a page such as an image filename or `href` being updated, then the test is probably too specific.
- Avoid string checking as tests may break if strings are updated, or could change depending on the page locale.
- When writing tests, try and run them against a staging or demo environment in addition to local testing. It's also worth running tests a few times to identify any intermittent failures that may need additional waits.

See also the Web QA style guide for Python based testing.

1.7.7 Testing Basket email forms

When writing functional 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.

1.7.8 Link Checks

A full link checker is run over the production environments, which uses a tool named LinkChecker to crawl the entire website and reports any broken or malformed links both internally and externally. These jobs are run once a day in the Jenkins instance and are named with the `bedrock_linkchecker_` prefix.

In addition, there are targeted functional tests for the download and localized download pages. These tests do not use the LinkChecker tool, and are run as part of the pipeline, which ensures that any broken download links are noticed much earlier, and also do not depend on a crawler to find them.

1.8 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.

1.8.1 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):
    """
    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.

    Usage:
    urlpatterns = [
        redirect(r'projects/$', 'mozorg.product'),
        redirect(r'^projects/seamonkey$', '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'}),
    ]
    """
```

1.8.2 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}'`).

1.8.3 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/'),
```

(continues on next page)

(continued from previous page)

```
            vary='cookie'),
]
```

1.8.4 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:

```
$ py.test --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.

1.9 JavaScript Libraries

1.9.1 Mozilla Image Lazy Loader

mozilla-lazy-load.js

1.10 Newsletters

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

By default, every page's footer has a form to signup for the default newsletter, "Firefox & You".

1.10.1 Features

- ability to subscribe to a newsletter from a page's footer area. Many pages on the site might include this.
- whole pages devoted to subscribing to one newsletter, often with custom text, branding, and layout

- newsletter preference center - allow user to change their email address, preferences (e.g. language, HTML vs. text), 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, which is the only way (currently) they can get the token.
- landing pages that user ends up on after subscribing. These can vary depending on where they're coming from.

1.10.2 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).
- 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 signup date of the user, we're careful not to accidentally change the signup date, which could happen if we sent redundant subscription commands to our backend.

1.10.3 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.

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.

1.10.4 URLs

Here are a few important URLs implemented. These were established before Bedrock came along and so are unlikely to be changed.

(Not all of these might be implemented in Bedrock yet.)

/newsletter/ - subscribe to 'mozilla-and-you' newsletter (public name: "Firefox & You")

/newsletter/existing/USERTOKEN/ - user management of their preferences and subscriptions

1.10.5 Configuration

Currently, information about the available newsletters is configured in Basket. See Basket for more information.

1.10.6 Footer signup

Customize the footer signup form by overriding the email_form template block. For example, to have no signup form:

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

The default is:

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

which gives a signup for Firefox & You. You can pass parameters to the macro `email_newsletter_form` to change that. For example, the `newsletter_id` 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 %}
```

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

You can also use the same form outside a page footer by passing `footer=False` to the macro.

You can also specify one of three color variants for the "Sign Up Now" button. The options are:

- default - Which sets the border and font color to a light blue [#00afe5]
- dark - Which sets the border and font color to the dark Firefox blue [00539F]
- white - Which sets the border and font color to white [#fff]

This is done in a template as follows:

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

# dark
{% block email_form %}
    {{ email_newsletter_form(button_class='button-dark') }}
{% endblock %}

# white
{% block email_form %}
    {{ email_newsletter_form(button_class='button-light') }}
{% endblock %}
```

1.11 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.

1.11.1 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.

1.11.2 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) }}
```

1.12 Banners

1.12.1 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 fundraising form 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 *base banner template*, and content can then be inserted using `banner_title` and `banner_content` blocks:

```
{% extends 'includes/banners/base.html' %}

{% block banner_title %}We all love the web. Join Mozilla in defending it.{% endblock
↪%}

{% block banner_content %}
    <!-- insert custom HTML here -->
{% endblock %}
```

CSS styles for banners should be located in `media/css/base/banners/`, and should extend common base banner styles:

```
@import 'includes/base';
```

To initiate a banner on a page, include `media/js/base/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('fundraiser-sept2019');
    }

    window.Mozilla.run(onLoad);

})();
```

L10n for page banners

Becuase 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 `.lang` or `.ftl` associated with it, and accessible to the template or view it gets used in.

1.12.2 Fundraising banner

The fundraising banner typically gets shown on the home page, but technically can be shown on any page in bedrock. The donation parameters that get passed to the form require some extra context data that needs to get passed to the template via the view in order to work. For example:

```
def home_view(request):
    locale = l10n_utils.get_locale(request)
    donate_params = settings.DONATE_PARAMS.get(
        locale, settings.DONATE_PARAMS['en-US'])

    # presets are stored as a string but, for the home banner
    # we need it as a list.
    donate_params['preset_list'] = donate_params['presets'].split(',')

    ctx = {
        'donate_params': donate_params
    }

    return l10n_utils.render(request, 'mozorg/home/home.html', ctx)
```

The HTML and CSS assets for the fundraising banner are located in:

- `bedrock/base/templates/includes/banners/fundraiser.html`
- `media/css/base/banners/fundraiser.scss`

Note: Strings for the fundraising banner are currently in a bit of a mess. Some are in `main.lang`, whilst others are in the homepage `.lang` file. This means it can't be shown outside of the home page currently, unless in English only. This needs fixing when we migrate over to Fluent.

1.13 Mozilla.UITour

1.13.1 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 Firefox 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.

1.13.2 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 a whitelisted domain.

1.13.3 JavaScript API

The UITour API documentation can be found in the Mozilla Source Tree Docs.

1.14 Send to Device Widget

The *Send to Device* widget is a single macro form which facilitates the sending of a download link from a desktop browser to a mobile device. The form allows sending via SMS or email, although the SMS copy & messaging is shown only to those in the configured countries. Geo-location is handled in JavaScript using a bedrock view that gets the request country from CloudFlare CDN headers. For users without JavaScript, the widget falls back to a standard email form.

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. The countries enabled for SMS for each message are defined in the running configuration per the environment names in `SEND_TO_DEVICE_MESSAGE_SETS` in the settings file.

1.14.1 Usage

1. Include this macro:

```
{% from "macros.html" import send_to_device with context %}
```

2. Add the appropriate lang file to the page template:

```
{% add_lang_files "firefox/sendto" %}
```

3. Make sure necessary files are in your CSS/JS bundles:

- `'css/protocol/components/send-to-device.scss'`
- `'js/base/send-to-device.js'`

4. Include the macro in your page template:

```
{{ send_to_device() }}
```

5. Initialize the widget:

   In your page JS, initialize the widget using:

```
var form = new Mozilla.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:

```
var form = new Mozilla.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'
product*	Product ID for what should be downloaded. Defaults to 'firefox'.	String	'firefox', 'focus', 'klar'
message_set*	ID for the email that should be received. Defaults to 'default'.	String	'default', 'fx-mobile-download-desktop', 'download-firefox-rocket'
id*	HTML form ID. Defaults to 'send-to-device'.	String	'send-to-device'
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_email	Provides a custom legal note for email use.	Localizable String.	_('The intended recipient of the email must have consented.')
legal_note_sms	Provides a custom legal note for SMS or email.	Localizable string	_('SMS service available in select countries only.')
spinner_color	Hex color for the form spinner. Defaults to '#000'.	String	'#fff'

1.14.2 Geolocation Callback

You can piggy-back on the widget's geolocation call by providing a callback function to be executed when the lookup has completed:

```
var form = new Mozilla.SendToDevice();
form.geoCallback = function(countryCode) {
  console.log(countryCode);
```

(continues on next page)

(continued from previous page)

```
}
form.init();
```

The callback function will be passed a single argument - the country code returned from the geolocation lookup.

If the geolocation lookup fails, the country code passed to the callback function will be an empty string.

1.15 Firefox Accounts Referrals

Marketing pages often promote the creation of a Firefox Account as a common *call to action* (CTA). This is typically accomplished using either a signup form, or a prominent link/button. To accomplish this, bedrock templates can take advantage of a series of helpers which can be used to standardize referrals.

Note: The helpers below can typically be shown to all browsers, but some also feature logic specific to Firefox, such as signing users into Sync.

1.15.1 Conventions

When choosing URL parameter values, the following conventions help to support uniformity in code and predictability in retroactive analysis.

- Use lower case characters in parameter values.
- Separate words in parameter values with hyphens.
- Follow parameter naming patterns established in previous iterations of a page.

Important: All query string parameters also need to pass the validation rules applied by the Firefox Accounts server.

1.15.2 Signup Form

Use the `fxa_email_form` macro to display an account signup 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 template's respective JavaScript and CSS bundles should also include the following dependencies:

Javascript:

```
js/base/mozilla-fxa-form.js
js/base/mozilla-fxa-form-init.js
```

CSS:

```
css/base/mozilla-fxa-form.scss
```

The JavaScript files will automatically handle things adding metrics parameters, as well as configuring Sync and distribution ID (e.g. the China re-pack) for Firefox browsers. The CSS file contains some default styling for the signup form.

Configuration

The signup form macro accepts the following parameters (* indicates a required parameter)

Parameter name	Definition	Format	Example
entrypoint*	Unambiguous identifier for which page of the site is the referrer.	mozilla.org-directory-page	'mozilla.org-firefox-accounts'
entrypoint_experiment	Used to identify experiments.	Experiment ID	'whatsnew-headlines'
entrypoint_variation	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_campaign	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_content	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 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`.

1.15.3 Linking to accounts.firefox.com

Use the `fxa_button` helper to create a CTA button or link to https://accounts.firefox.com/.

Usage

```
{{ fxa_button(entrypoint='mozilla.org-firefox-accounts', button_text='Sign In') }}
```

Note: There is also a `fxa_link_fragment` helper which will construct only valid `href` and `data-mozillaonline-link` properties. This is useful when constructing an inline link inside a paragraph, for example.

For more information on the available parameters, read the "CTA button parameters" section further below.

1.15.4 Linking to monitor.firefox.com

Use the `monitor_fxa_button` helper to link to https://monitor.firefox.com/ via a Firefox Accounts auth flow.

Usage

```
{{ monitor_fxa_button(entrypoint=_entrypoint, button_text='Sign Up for Monitor') }}
```

For more information on the available parameters, read the "CTA button parameters" section further below.

1.15.5 Linking to getpocket.com

Use the `pocket_fxa_button` helper to link to https://getpocket.com/ via a Firefox 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 "CTA button parameters" section below.

1.15.6 CTA button parameters

The `fxa_button`, `pocket_fxa_button`, and `monitor_fxa_button` helpers all support the same standard parameters:

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'.	'mozilla.org-firefox-pocket'	'mozilla.org-firefox-pocket'
button_text*	The button copy to be used in the call to action. Default to a well localized string.	Localizable string	_('Try Pocket Now')
class_name	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_class	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_metrics	A boolean value that dictates if metrics parameters should be added to the button href. Defaults to 'True'.	Boolean	True or False
optional_parameters	An dictionary of key value pairs containing additional parameters to append the the href.	Dictionary	{'s': 'ffpocket'}
optiona_attributes	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': 'activate 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.

1.15.7 CTA button dependencies

When using any of the FxA button helpers, a templates's respective JavaScript bundle should also include the following dependencies:

```
js/base/mozilla-fxa-product-button.js
js/base/mozilla-fxa-product-button-init.js
```

This script automatically adds metrics parameters to the button `href`:

- `deviceId`
- `flowId`
- `flowBeginTime`

These are values are fetched from an API endpoint, and are instered back into the destination link along with the other standard referral parameters.

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 Firefox Accounts signup form and a Firefox Monitor button, but only one CTA is displayed at any one time, then only the metrics request associated with that CTA should occur. 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.

1.15.8 Tracking External Referrers

If the URL of a bedrock page contains existing UTM parameters on page load, bedrock will attempt to automatically use those values to replace the inline UTM parameters in Firefox Accounts links. This is handled using a client side script in the site common bundle which can be found in `/media/js/base/fxa-utm-referral.js`.

The behavior is as follows:

- UTM paramters 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 as long as the required parameters exist.
- If the above criteria is satisfied, then UTM parameters on FxA links will be replaced in their entirety with the UTM parameters from the page URL. This is to avoid mixing referral data from different campaigns.

Important: Links generated by the FxA button helpers will automatically be covered by this script. For links generated using the `fxa_link_fragment` helper, you will need to manually add a CSS class of `js-fxa-cta-link` to trigger the behavior.

1.15.9 Handling Distribution (aka China Repack)

The China repack of Firefox points to https://accounts.firefox.com.cn/ by default for accounts signups. To compensate for this on https://www.mozilla.org (so we don't send those visitors to the wrong place), we rely on *UITour* to check the distribution ID of the browser. If the distribution ID is `mozillaonline` (i.e. China repack), then we replace our accounts endpoints with the alternate domain specified in the `data-mozillaonline-link` attribute. The logic to handle this is self contained in the associated helper scripts and handled automatically.

1.15.10 Testing Signup Flows

Testing the Firefox Account signup flows on a non-production environment requires some additional configuration.

Configuring bedrock:

Set the following in your local `.env` file:

```
FXA_ENDPOINT=https://latest.dev.lcip.org/
```

Configuring a demo Server:

Demo servers must have the same `.env` setting as above. See the configure-demo-servers docs.

Local and demo server testing:

Follow the instructions provided by the FxA team. These instructions will launch a new Firefox instance with the necessary config already set. In the new instance of Firefox:

1. Navigate to the page containing the Firefox Accounts CTA.
2. If testing locally, be sure to use `127.0.0.1` instead of `localhost`

1.15.11 Google Analytics Guidelines

For GTM datalayer attribute values in FxA links, please use the *analytics* documentation.

1.16 Funnel cakes and Partner Builds

1.16.1 Funnel cakes

In addition to being an American delicacy funnel cakes are what we call special builds of Firefox. They can come with extensions preinstalled and/or a custom first-run experience.

> "The whole funnelcake system is so marred by history at this point I don't know if anyone fully understands what it's supposed to do in all situations" - pmac

Funnelcakes are configured by the Release Engineering team. You can see the configs in the funnelcake git repo

Currently bedrock only supports funnelcakes for "stub installer platforms". Which means they are windows only. However, funnelcakes can be made for all platforms so bedrock support may expand.

We signal to bedrock that we want a funnelcake when linking to the download page by appending the query variable *f* with a value equal to the funnelcake number being requested.

```
https://www.mozilla.org/en-US/firefox/download/thanks/?f=137
```

Bedrock checks to see if the funnelcake is configured (this is handled in the www-config repo)

```
FUNNELCAKE_135_LOCALES=en-US
FUNNELCAKE_135_PLATFORMS=win,win64
```

Bedrock then converts that into a request to download a file like so:

Windows:

```
https://download.mozilla.org/?product=firefox-stub-f137&os=win&lang=en-US
```

Mac (You can see the mac one does not pass the funnelcake number along.):

```
https://download.mozilla.org/?product=firefox-latest-ssl&os=osx&lang=en-US
```

Someone in Release Engineering needs to set up the redirects on their side to take the request from here.

Places things can go wrong

As with many technical things, the biggest potential problems are with people:

- Does it have executive approval?
- Did legal sign off?
- Has it had a security review?

On the technical side:

- Is the switch enabled?
- Is the variable being passed?

1.16.2 Partner builds

Bedrock does not have an automated way of handling these, so you'll have to craft your own download button:

```
<a href="https://download.mozilla.org/?product=firefox-election-edition&os=win&
↪lang=en-US">
Download</a>
```

Bugs that might have useful info:

- https://bugzilla.mozilla.org/show_bug.cgi?id=1450463
- https://bugzilla.mozilla.org/show_bug.cgi?id=1495050

PRs that might have useful code:

- https://github.com/mozilla/bedrock/pull/5555

1.17 A/B Testing

1.17.1 Convert experiments

Conversion rate optimization (CRO) experiments on bedrock are run on separate, experimental versions of our main landing pages using a tool called Convert. Convert experiments are for relatively simple multivariate experiments, such as testing changes to headlines, images, or button copy.

Experimental versions of our main product pages can be found under the `/exp/` app within bedrock. We keep our main product pages separate from experiment code for the following reasons:

- To maintain a separation of concerns. Code changes to our main landing pages do not need to be complicated by experiment code. Reducing the frequency of code changes related to experiments on our main landing pages also reduces the risk of accidental breakage.
- To minimize any negative performance impact of experimentation. Convert requires additional scripts that need to be downloaded on the client. Experiments also aren't always as optimized as production code will be.
- To minimize the percentage of our traffic exposed to experimentation. Keeping experimental versions of our pages separate means that only people who are entered into an experiment need to download and run code associated with it.

Creating experimental pages

Templates within the `/exp/` app should have the following characteristics:

- They should be simple clones of our main pages. They should look the same by duplicating CSS & JS where needed, but they do not necessarily need the same level of complex functionality (if that exists in a page). Only what's needed to facilitate basic experimentation.
- They should not be part of our localization system. Experimental pages are mostly `en-US` only, so strings do not need to be wrapped for translation. Hard-coded, locale specific templates can be created if there's a need to test in German, for example.
- They should extend the base template within the `/exp/` app. This facilitates loading the Convert JS in a uniform way, that respects DNT.
- If a page is indexed by search engines, then the experimental equivalent should have a canonical URL that points back to the original page. If the page is considered to be an in-product page, then the experimental page should retain the same noindex meta tag.
- Experimental pages should also contain an Open Graph sharing URL that points back to the original page.

Once an experimental page exists, A/B tests can then be implemented using the Convert dashboard and editor. Convert experiments should be coded and tested against staging, before being reviewed and scheduled to run in production.

QA for Convert experiments

The process for QA'ing Convert experiments is as follows:

1. Experiments are completely built and configured to run on `https://www.allizom.org/*`
2. In the Github issue for an experiment, someone will request review by an engineer.

An engineer reviewing the experiment will:

1. Verify that the experiment is not configured to run on https://www.mozilla.org/ (production) yet.
2. Activate the experiment to run on stage.

During review, the engineer will compare the following to the experiment plan:

1. The experiment's logic.
2. Any JS included (in Convert editor's JS field).
3. Any CSS included (in Convert editor's CSS field).
4. The target audience is configured.
5. The goals are configured.
6. The distribution percentages are configured.
7. The target URLs are configured.

Once the engineer is satisfied, the engineer (or someone else with write privileges) will:

1. Add `https://www.mozilla.org/*` to the list of urls the experiment can run on.
2. Reset the experiment (eliminating any data gathered during QA).
3. Activate (or schedule) the experient.

Note: `*` should be replaced by the exact URL pathname for the experiment page.

Routing traffic to experimental pages

Once an experimental page has been created, a predefined percentage of traffic can then be redirected from the main product page, to the experimental page using a Cloudflare Worker script. We do this using a worker because it has significant performance improvements over doing redirection on the client or the server, and can also be done independently of a bedrock deployment.

The code for redirecting to experimental pages lives in the www-workers repository.

Adding a new redirect requires making a pull request to add a new object to the `experimentalPages` array in the `workers/redirector.js` file. An existing configuration to route 6% of traffic from `/firefox/new/` to `/exp/firefox/new/` can be seen below:

```
const experimentPages = [
    {
        'targetPath': `/en-US/firefox/new/`,
        'sandboxPath': `/en-US/exp/firefox/new/`,
```

(continues on next page)

(continued from previous page)

```
        'sampleRate': 0.06
    }
];
```

Important: When implementing new changes to the redirector, make sure to test and verify that things are working as expected on dev and stage before pushing to production. See the documentation in the www-workers repository for more information.

1.17.2 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 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.

Including the `data-ex-variant` and `data-ex-name` in the analytics reporting will add the test to an auto generated report in GA. The variable values may be provided by the analytics team.

```
if(href.indexOf('v=a') !== -1) {
    window.dataLayer.push({
        'data-ex-variant': 'de-page',
        'data-ex-name': 'Berlin-Campaign-Landing-Page'
    });
} else if (href.indexOf('v=b') !== -1) {
    window.dataLayer.push({
        'data-ex-variant': 'campaign-page',
        'data-ex-name': 'Berlin-Campaign-Landing-Page'
    });
}
```

Make sure any buttons and interaction which are being compared as part of the test and will report into GA.

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.

A/B Test PRs that might have useful code to reuse

- https://github.com/mozilla/bedrock/pull/5736/files
- https://github.com/mozilla/bedrock/pull/4645/files

- https://github.com/mozilla/bedrock/pull/5925/files
- https://github.com/mozilla/bedrock/pull/5443/files
- https://github.com/mozilla/bedrock/pull/5492/files
- https://github.com/mozilla/bedrock/pull/5499/files

1.18 Analytics

1.18.1 Google Tag Manager (GTM)

Bedrock uses Google Tag Manager (GTM) to manage and organize its Google Analytics solution.

GTM is a tag management system that allows for easy implementation of Google Analytics tags and other 3rd party marketing tags in a nice GUI experience. Tags can be added, updated, or removed directly from the GUI. GTM allows for a *one source of truth* approach to managing an analytics solution in that all analytics tracking can be inside GTM.

Bedrock's GTM solution is CSP compliant and does not allow for the injection of custom HTML or JavaScript but all tags use built in templates to minimize any chance of introducing a bug into Bedrock.

1.18.2 The GTM DataLayer

How an application communicates with GTM is via the `dataLayer` object, which is a simple JavaScript array GTM instantiates on the page. Bedrock will send messages to the `dataLayer` object by means of pushing an object literal onto the `dataLayer`. GTM creates an abstract data model from these pushed objects that consists of the most recent value for all keys that have been pushed to the `dataLayer`.

The only reserved key in an object pushed to the `dataLayer` is `event` which will cause GTM to evaluate the firing conditions for all tag triggers.

1.18.3 DataLayer Push Example

If we wanted to track clicks on a carousel and capture what the image was that was clicked, we might write a dataLayer push like this:

```
dataLayer.push({
    'event': 'carousel-click',
    'image': 'house'
});
```

In the dataLayer push there is an event value to have GTM evaluate the firing conditions for tag triggers, making it possible to fire a tag off the dataLayer push. The event value is descriptive to the user action so it's clear to someone coming in later what the dataLayer push signifies. There is also an image property to capture the image that is clicked, in this example it's the house picture.

In GTM, a tag could be setup to fire when the event `carousel-click` is pushed to the dataLayer and could consume the image value to pass on what image was clicked.

1.18.4 The Core DataLayer Object

For the passing of contextual data on the user and page to GTM, we've created what we call the Core DataLayer Object. This object passes as soon as all required API calls for contextual data have completed. Unless there is a

significant delay to when data will be available, please pass all contextual or meta data on the user or page here that you want to make available to GTM.

1.18.5 GTM Listeners & Data Attributes

GTM also uses click and form submit listeners to gather context on what is happening on the page. Listeners push to the dataLayer data on the specific element that triggered the event, along with the element object itself.

Since GTM listeners pass the interacted element object to the dataLayer, the use of data attributes works very well when trying to identify key elements that you want to be tracked and for storing data on that element to be passed into Google Analytics. We use data attributes to track clicks on all downloads, buttons elements, and nav, footer, and CTA/button link elements.

Important: When adding any new elements to a Bedrock page, please follow the below guidelines to ensure accurate analytics tracking.

For all generic CTA links and <button> elements, add these data attributes (* indicates a required attribute):

Data Attribute	Expected Value (lowercase)
data-cta-type *	Link type (e.g. 'navigation', 'footer', or 'button')
data-cta-text	name or text of the link
data-cta-position	Location of CTA on the page (e.g. 'primary', 'secondary', 'header')

For all download buttons, add these data attributes:

Data Attribute	Expected Value
data-link-type	'Desktop', 'Android', or 'iOS'
data-download-os	name or text of the link
data-download-version	'standard', 'developer', 'beta'

For all links to accounts.firefox.com use these data attributes (* indicates a required attribute):

Data Attribute	Expected Value
data-cta-type *	fxa-servicename (e.g. 'fxa-sync', 'fxa-monitor', 'fxa-lockwise')
data-cta-text	Name or text of the link (e.g. 'Sign Up', 'Join Now', 'Start Here'). We use this when the link text is not useful, as is the case with many FxA forms that say, 'Continue'. We replace 'Continue' with 'Register'.
data-cta-position	Location of CTA on the page (e.g. 'primary', 'secondary', 'header')

For all conditional banners, add the following calls.

When a banner is shown:

```
dataLayer.push({
    'eLabel': 'Banner Impression',
    'data-banner-name': '<banner name>', //ex. Fb-Video-Compat
```

(continues on next page)

(continued from previous page)

```
    'data-banner-impression': '1',
    'event': 'non-interaction'
});
```

When an element in the banner is clicked:

```
dataLayer.push({
    'eLabel': 'Banner Clickthrough',
    'data-banner-name': '<banner name>', //ex. Fb-Video-Compat
    'data-banner-click': '1',
    'event': 'in-page-interaction'
});
```

When a banner is dismissed:

```
dataLayer.push({
    'eLabel': 'Banner Dismissal',
    'data-banner-name': '<banner name>', //ex. Fb-Video-Compat
    'data-banner-dismissal': '1',
    'event': 'in-page-interaction'
});
```

When doing a/b tests configure something like the following.

```
if(href.indexOf('v=a') !== -1) {
    window.dataLayer.push({
        'data-ex-variant': 'de-page',
        'data-ex-name': 'Berlin-Campaign-Landing-Page'
    });
} else if (href.indexOf('v=b') !== -1) {
    window.dataLayer.push({
        'data-ex-variant': 'campaign-page',
        'data-ex-name': 'Berlin-Campaign-Landing-Page'
    });
}
```

1.18.6 Some notes on how this looks in GA

`data-cta-type=""` and `data-cta-name=""` trigger a generic link / buton click with the following structure:

Event Category: {{page ID}} Interactions
Event Action: {{data-cta-type}} click
Event Label: {{data-cta-name}}

1.19 Stub Attribution

Stub Attribution is a process that enables the construction and transmission of marketing attribution code on www.mozilla.org. When a user visits a mozilla.org download page, the website constructs an attribution code, which is then passed to download.mozilla.org via a URL parameter. This attribution code is then passed to the Windows stub installer, where it finally gets passed to Firefox for use in Telemetry. This attribution funnel enables Mozilla to better understand how different marketing campaigns effect overall retention in Firefox.

Note: The AMO team also relies on Stub Attribution on the /new page to inform their Return to AMO feature.

1.19.1 How does it work in bedrock?

The base template in bedrock contains a stub attribution script that runs on every page of the website. The script only runs on Windows (since the stub installer is a Windows feature) and if cookies are enabled. The script also respects Do Not Track, so it will not run if the user agent has this flag set.

The stub attribution script looks for *utm_** params on the page URL as well as the referrer, and then creates a hash of that data by passing it to an authentication service in bedrock. This hash is then accompanied by a signed, encrypted signature to prove that the data came from a trusted source. Both pieces of authenticated data are then stored in a cookie.

Once the user reaches the download page, bedrock then checks if the cookie exists, and if so appends the authenticated data to the download URL. The rest of the process is handled by the download service and stub installer.

1.19.2 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.

1.19.3 Measuring campaigns and experiments

Stub 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 (as configured in */media/js/base/stub-attribution.js*) in the background and passes along any attribution data that exists.

More recently, the ability to measure the effectiveness of experiments was also added as a feature. 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()`.

1.20 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.

1.20.1 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.

1.20.2 2. Move CI/CD Pipelines to Gitlab

Date: 2019-10-09

Status

Accepted

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.

1.20.3 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.

1.20.4 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.

1.21 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.

1.21.1 Browser Support Matrix (Updated 2019-06-11)

We deliver enhanced CSS & JS to browsers in our browser support matrix (below). We deliver degraded support to all other user agents, except legacy IE browsers, which get basic support.

The following browsers have enhanced support:

- All evergreen browsers (Firefox, Chrome, Safari, Edge, Opera, etc.)
- IE11 and above.

The following browsers have degraded support:

- Outdated evergreen browser versions.
- IE10.

The following browsers have basic support:

- IE9 and below.

1.21.2 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, a universal CSS stylesheet that gets applied to all pages, and a universal JS bundle that only handles downloading Firefox (click a button, get a file), and Google Analytics.

To hide non-relevant content from legacy IE users who see the universal stylesheet, a `hide-from-legacy-ie` class name can be applied directly to HTML:

```
<p class="hide-from-legacy-ie">See what Firefox has blocked for you</p>
```

1.21.3 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.

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.
}
```

1.21.4 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 or JS to legacy browsers, or allowing them to degrade gracefully. It does not mean everything needs to look the same in every browser.
