bedrock Documentation

Release 1.0

Mozilla

May 24, 2017

Contents

bedrock is the code name of the new mozilla.org. It is bound to be as shiny, awesome, and open sourcy as always. Perhaps even a little more.

bedrock is a web application based on Django.

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

CHAPTER 1

Contents

Installing Bedrock

Installation

These instructions assume you have *git* and *pip* installed. If you don't have *pip* installed, you can install it with *easy_install pip*.

Start by getting the source:

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

(Make sure you use –recursive so that legal-docs are included)

You need to create a virtual environment for Python libraries. Skip the first instruction if you already have virtualenv installed:

```
$ pip install virtualenv                        # installs virtualenv, skip if already␣
↪have it
$ virtualenv -p python2.7 venv                  # create a virtual env in the folder␣
↪`venv`
$ source venv/bin/activate                      # activate the virtual env
$ pip install -U pip                            # securely upgrade pip
$ pip install -r requirements/test.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/test.txt
```

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

python-dev libxslt-dev

Now configure the application to run locally by creating your local settings environment file:

```
$ cp .env-dist .env
```

You shouldn't need to customize anything in there yet.

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

```
$ bin/sync-all.sh
```

Lastly, you need to have Node.js and NPM installed. The node dependencies for running the site can be installed with `npm`:

```
$ npm install
```

You may also need to install the Gulp cli globally.

Note: Bedrock uses npm-lockdown to ensure that Node.js packages that get installed are the exact ones we meant (similar to peep.py for python). Refer to the lockdown documentation for adding or upgrading Node.js dependencies.

Run the tests

Important: We're working on fixing this, but for now you need the localization files for the tests to pass. See the *Localization* section below for instructions on checking those out.

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. 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 lib 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.

Make it run

To make the server run, make sure you are inside a virtualenv, and then run the server:

```
$ ./manage.py runserver
```

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

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

If you get the error "NoneType is not iterable", you didn't check out the latest product-details. See the above section for that.

Next, in a new terminal tab run gulp to watch for local file changes:

```
$ gulp
```

This will automatically copy over CSS, JavaScript and image files to the /static directory as and when they change, which is needed for Django Pipeline to serve the assets as pages are requested.

If you have problems with gulp, or you for some reason don't want to use it you can set:

```
PIPELINE_COLLECTOR_ENABLED=True
```

in your `.env` file or otherwise set it in your environment and it will collect media for you as you make changes. The reason that this is not the preferred method is that it is much slower than using gulp.

Localization

If you want to install localizations, run the following command:

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

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

Feature Flipping

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". If the environment variable `DEV` is set to one of those "true" values, then all switches will be considered "on" unless they are explicitly "off" in the environment.

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, en-ZA" 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'
    ...
```

Currently, these switches are used to enable/disable Optimizely on many pages of the site. We only add the Optimizely JavaScript snippet to a page when there is an active test to minimize the security risk of the service. We maintain a page on the Mozilla wiki detailing our use of Optimizely and these switches.

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

```
# to switch on firefox-new-optimizely you'd add the following to your ``.env`` file
SWITCH_FIREFOX_NEW_OPTIMIZELY=on
```

You then must set an Optimizely project code in `.env`:

```
# Optimize.ly project code
OPTIMIZELY_PROJECT_ID=12345
```

Note: You are not required to set up Optimizely as detailed above. If not configured, bedrock will treat the switches as set to `off`.

To do the equivalent in one of the bedrock apps deployed with Deis, use deis config. To continue the example above with a Deis app named `bedrock-demo-switch`:

```
deis config:set SWITCH_FIREFOX_NEW_OPTIMIZELY=on -a bedrock-demo-switch
```

Note: We have multiple Deis clusters with independent configurations, and recommend using the DEIS_PROFILE environment variable to switch between clusters.

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 virtualenvwrapper that provides a better interface for managing/activating virtual envs, so you can use that if you want.

Localization

The site is fully localizable. Localization files are not shipped with the code distribution, but are available in a separate GitHub repository. The proper repo 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 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.

.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 apps/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 apps/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?*).

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.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; src: local(mplus-2p-light), local(Meiryo);
}
@font-face { font-family: X-LocaleSpecific-Light; font-weight: bold; src: local(mplus-2p-medium), local(Meiryo-Bold);
}
@font-face { font-family: X-LocaleSpecific; font-weight: normal; src: local(mplus-2p-regular), local(Meiryo);
}
@font-face { font-family: X-LocaleSpecific; font-weight: bold; src: local(mplus-2p-bold), local(Meiryo-Bold);
}
@font-face { font-family: X-LocaleSpecific-Extrabold; font-weight: 800; 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/`, and specify those with the `src: url(path)` syntax. The M+ font family offers various international glyphs and looks similar to *Open Sans*. You can create a subset of the *M+ 2c* font using a tool found on the Web. See Bug 776967 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 features" needs to be changed to "Firefox benefits":

```
{% 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.

Locale-specific Templates

While the `l10n_has_tag` template function is great in small doses, it doesn't scale particularly well. A template filled with conditional copy can be difficult to comprehend, particularly when the conditional copy has associated CSS and/or JavaScript.

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

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

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

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.

Developing on Bedrock

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.

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('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. *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('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('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('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/*.

Writing Views

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

If you need to write a view and the page has a newsletter signup form in the footer (most do), make sure to handle this in your view. Bedrock comes with a function for doing this automatically:

```
from bedrock.mozorg.util import handle_newsletter
from django.views.decorators.csrf import csrf_exempt

@csrf_exempt
def view(request):
    ctx = handle_newsletter(request)
    return l10n_utils.render(request, 'app/template.html', ctx)
```

You'll notice a few other things in there. You should use the *l10n_utils.render* function to render templates because it handles special l10n work for us. Since we're handling the newsletter form post, you also need the *csrf_exempt* decorator.

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.

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 tempalte 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*.

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 :ref: *CSS<l10n>* 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.

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.

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.

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.

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
```

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.

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*.

Getting a new Bedrock page online

On our servers, Bedrock pages are accessible behind the `/b/` prefix. So if a page is accessible at this URL locally:

```
http://localhost:8000/foo/bar
```

then on our servers, it will be accessible at:

```
http://www.mozilla.org/b/foo/bar
```

When you're ready to make a page available to everyone, we need to remove that /b/ prefix. We handle that with Apache RewriteRule. Apache config files that are included into the server's config are in the bedrock code base in the `etc/httpd` directory. In there you'll find a file for each of the environments. You'll almost always want to use `global.conf` unless you have a great reason for only wanting the config to stay on one of the non-production environments.

In that file you'll add a RewriteRule that looks like the following:

```
# bug 123456
RewriteRule ^/(\w{2,3}(?:-\w{2}(?:-mac)?)?/)?foo/bar(/?)$ /b/$1foo/bar$2 [PT]
```

This is a lot simpler than it looks. The first large capture is just what's necessary to catch every possible locale code. After that it's just your new path. Always capture the trailing slash as we want that to hit django so it will redirect.

Note: It's best if the RewriteRule required for a new page is in the original pull request. This allows it to flow through the push process with the code and for it to go live as soon as it's on the production server. It's also one less review and pull-request for us to manage.

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:* master, updated via a webhook on pushes
- *Bedrock Git branch:* any branch we want, manually updated

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.us-west.moz.works/`
- *Bedrock locales dev repo:* master branch, updated via a webhook on pushes
- *Bedrock Git branch:* any branch we want, manually updated

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:* master branch, updated via a webhook on pushes
- *Bedrock Git branch:* master, updated every 10 minutes

Stage

- *URL:* http://www.allizom.org/
- *Bedrock locales dev repo:* master branch, updated via a webhook on pushes
- *Bedrock Git branch:* master, updated manually

Production

- *URL:* http://www.mozilla.org/
- *Bedrock locales production repo:* master branch, updated via a webhook on pushes

- *Bedrock Git branch:* master, updated manually

We use Chief for the manual deploys. You can check the currently deployed git commit by checking https://www.mozilla.org/media/revision.txt.

If you want to know more and you have an LDAP account, you can check the IT documentation.

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)*)

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*).

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 smoke suite of redirect headless HTTP(s) response checks.

Push to master branch

Whenever a change is pushed to the master branch, the smoke suite of headless (see *Testing redirects*) and UI tests (see *Smoke tests*) are run against Firefox on Linux. If successful, the change is pushed 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 master.yml file in the repository.

Push to prod branch (tagged)

When a tagged commit is pushed to the prod branch, everything that happens during a untagged push is still run. In addition, the full suite of UI tests is run against Chrome and Internet Explorer on Windows 10, and the sanity suite is run against older versions of Internet Explorer (currently IE6 & IE7). If successful, the change is pushed to staging, tested, and then to production and the same tests are then run against production. As with untagged pushes, this is all handled by the pipeline, and is subject to change according to the settings in the prod.yml file in the 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:**
    (a) 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.

Pipeline integration

Our Jenkinsfile will run the integration tests based on information in our branch-specific YAML files. These files specify various test names per branch that will cause it to use different parameters, allowing it to be called in many different ways to cover the testing needs. The job executes this script, which then runs this Docker image, and ultimately runs another script. The two scripts can also be executed locally to replicate the way Jenkins operates.

During the **Test Images** stage, the Test Runner job is called without a `BASE_URL`. This means that a local instance of the application will be started, and the URL of this instance will be used for testing. The `DRIVER` parameter is set to `Remote`, which causes a local instance of Selenium Grid to be started in Docker and used for the browser-based functional UI tests.

The test scripts above will be run once for each properties name specified in the branch-specific YAML files for the branch being built and tested. Pushes to *master* will run different tests than pushes to *prod* for example.

Configuration

Many of the options are configured via environment variables passed from the initial script to the Docker image and onto the final script. Many of these options can be set in the branch-specific YAML files in the repository. In the

branch-specific YAML files folder you can copy any file there to match the name of your branch and modify it to set how it should be built by jenkins. Take the following example:

This configuration would cause commits pushed to a branch named `change-all-the-things` to have docker images built for them, have the smoke and unit tests run, and deploy to a deis app named `bedrock-probably-broken` in our us-west deis cluster. If you'd like it to create the deis app and pre-fill a local database for your app, you can set `demo: true` in the file. Note that if the app already exists it must have the `jenkins` user added via the `deis perms:create jenkins -a <your app name>` command.

The available branch configuration options are as follows:

- `smoke_tests`: boolean. Set to `true` to cause the unit and smoke test suites run against the docker images.
- `push_public_registry`: boolean. Set to `true` to cause the built images to be pushed to the public docker hub.
- `require_tag`: boolean. Set to `true` to require that the commit being built have a git tag in the format YYYY-MM-DD.X.
- `regions`: list. A list of strings indicating the deployment regions for the set of apps. The valid values are in the `regions` area of the `jenkins/global.yml` file. If omitted a deployment to only `usw` is assumed.
- `apps`: list. A list of strings indicating the deis app name(s) to which to deploy. If omitted no deployments will occur.
- `demo`: boolean. Set to `true` to have the deployed app in demo mode, which means it will have a pre-filled local database and the deis app will be created and configured for you if it doesn't already exist.
- `integration_tests`: list. A list of strings indicating the types of integration tests to run. If omitted no tests will run.

You can also set app configuration environment variables via deployment as well for demos. The default environment variables are set in *jenkins/branches/demo/default.env*. To modify your app's settings you can create an env file named after your branch (e.g *jenkins/branches/demo/pmac-l10n.env* for the branch *demo/pmac-l10n.env*). The combination of values from *demo/default.env*, your branch specific env file, and a region specific env file (e.g. *jenkins/regions/virginia.env*) will be used to configure the app. So you only need to add the variables that differ from the default files to your file, and you can override any values from the default files as well.

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 properties section in the integration tests script with the parameters of the new test run. This is simply a bash script and you can duplicate a clause of the case staement. For example, if you wanted to run tests in Firefox on both Windows 10 and OS X, you could create the following clauses:

```
case $1 in
  osx-firefox)
    BROWSER_NAME=firefox
    PLATFORM="OS X 10.11"
    ;;
```

```
  win10-firefox)
    BROWSER_NAME=firefox
    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.

Known issues in Jenkins

Jenkins stalls after global configuration changes

When using the IRC plugin for notifications, global configuration changes can cause Jenkins to become unresponsive. To make such changes it can be necessary to first restart Jenkins, as this issue only appears some time after Jenkins has been started. A bug for the IRC plugin has been raised.

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.

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/test.txt
```

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.

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 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
```

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
```

Smoke tests

Smoke tests are run on every commit to master as part of bedrocks deployment pipeline. These should be considered as critical baseline functional tests. (Note: we only run the full suite of cross-browser functional tests on tagged commits. See *Push to prod branch (tagged)*). If your test should be considered a smoke test you will need to apply a `smoke` marker to it.

```
import pytest

@pytest.mark.smoke
@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
```

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

Note: Tests that rely on long-running timeouts, cron jobs, or that test for locale specific interactions should not be marked as a smoke test. We should try and ensure that the suite of smoke tests are quick to run, and they should not have a dependency on checking out and building the full site.

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.

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`.

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.

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.

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.

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.

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'}),
    ]
    """
```

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}'`).

Utilities

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

ua_redirector

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

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


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

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

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

header_redirector

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

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


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

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/test.txt
```

Running the tests

Smoke tests are run against a local bedrock instance using `-m smoke`. These are also run automatically against GitHub pull requests.

```
$ py.test -r a -m smoke tests/redirects/
```

This will start a local instance of bedrock, run the smoke tests, and then stop the instance. 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.

JavaScript Libraries

Mozilla Pager

mozilla-pager.js

Mozilla Accordion

mozilla-accordion.js

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".

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.

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.

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.

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

Configuration

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

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 %}
```

Creating a signup page

Start with a template that extends `'newsletter/one_newsletter_signup.html'`. It's probably simplest to copy an existing one, like `'newsletter/mobile.html'`.

Set the `newsletter_title` and `newsletter_id` variables and override at least the `page_title` and `newsletter_content` blocks:

```
    {% set newsletter_title = _('Firefox and You') %}
    {% set newsletter_id = 'mozilla-and-you' %}

{% block page_title %}{{ newsletter_title }}{% endblock %}

{% block newsletter_content %}
  <div id="main-feature">
    <h2>Subscribe to <span>about:mobile</span>!</h2>
    <p>Our about:mobile newsletter brings you the latest and greatest news
        from the Mozilla contributor community.
    </p>
  </div>
{% endblock %}
```

Then add a url to `newsletter/urls.py`:

```
# "about:mobile"
page('newsletter/about_mobile', 'newsletter/mobile.html'),
```

Tabzilla

The latest version of Tabzilla is no longer part of the bedrock repository, and can now be included directly as part of any Mozilla site. Please see the Tabzilla repository for more information:

https://github.com/mozilla/tabzilla/

Translation and Update bar

The Translation and Update Bar functionality that used to be part of Tabzilla can now also be found in its own repository:

https://github.com/mozilla/mozilla-infobar/

Mozilla.UITour

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 and highlighting the position of buttons in the toolbar. It is supported in Firefox 29 onward.

For security reasons `Mozilla.UITour` will only work on white-listed domains and over a secure connection. The white-listed domains are https://www.mozilla.org and https://support.mozilla.org and the special about:home page.

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.

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.

Tracking Protection UITour

In order to test the Tracking Protection tour on a local server or domain other than www.mozilla.org, you must first set an additional preference in `about:config` in addition to white listing UITour for the domain.

- `privacy.trackingprotection.introURL` value e.g. `http://127.0.0.1:8000/%LOCALE%/firefox/%VERSION%/tracking-protection/start/`

Once this preference has been set, the tour can be accessed by opening a new Private Window and then by clicking the "See how this works" CTA button.

JavaScript API

registerPageID(pageId)

Register an ID for use in Telemetry. `pageId` must be a string unique to the page:

```
var pageId = 'firstrun-page-firefox-29';

Mozilla.UITour.registerPageID(pageId);
```

showHighlight(target, effect)

Highlight a button in the browser chrome. `target` is the string ID for the button and `effect` is the animation type:

```
Mozilla.UITour.showHighlight('appMenu', 'wobble');
```

Target types:

- `'accountStatus'`
- `'addons'`
- `'appMenu'`
- `'backForward'`
- `'bookmarks'`
- `'customize'`
- `'help'`
- `'home'`
- `'quit'`
- `'search'`
- `'searchIcon'` (Firefox 34 and above)
- `'urlbar'`
- `'loop'`
- `'forget'`
- `'privateWindow'`
- `'trackingProtection'` (Firefox 42 and above)
- `'controlCenter-trackingUnblock'` (Firefox 42 and above)

- `'controlCenter-trackingBlock'` (Firefox 42 and above)

Effect types:

- `'random'`
- `'wobble'`
- `'zoom'`
- `'color'`
- `'none'` (default)

hideHighlight()

Hides the currently visible highlight:

```
Mozilla.UITour.hideHighlight();
```

showInfo(target, title, text, icon, buttons, options)

Displays a customizable information panel pointing to a given target:

```
var buttons = [
    {
        label: 'Cancel',
        style: 'link',
        callback: cancelBtnCallback
    },
    {
        label: 'Confirm',
        style: 'primary',
        callback: confirmBtnCallback
    }
];

var icon = '//mozorg.cdn.mozilla.net/media/img/firefox/australis/logo.png';

var options = {
    closeButtonCallback: closeBtnCallback
};

Mozilla.UITour.showInfo('appMenu', 'my title', 'my text', icon, buttons, options);
```

Available targets:

Any target that can be highlighted can have an information panel attached.

Additional parameters:

- `title` panel title (string).
- `text` panel description (string).
- `icon` panel icon absolute url (string). Icon should be 48px x 48px.
- `buttons` array of buttons (object)
- `options` (object)

`buttons` array items can have the following properties:

- `label` button text (string)
- `icon` button icon url (string)
- `style` button style can be either *primary* or *link* (string)
- `callback` to be excecuted when the button is clicked (function)
- `options` (object)

`options` can have the following properties:

- `closeButtonCallback` to be excecuted when the (x) close button is clicked (function)

hideInfo()

Hides the currently visible info panel:

```
Mozilla.UITour.hideInfo();
```

showMenu(target, callback)

Opens a targeted menu in the browser chrome.

```
Mozilla.UITour.showMenu('appMenu', function() {
    console.log('menu was opened');
});
```

Available targets:

- `'appMenu'`
- `'bookmarks'`
- `'loop'` (Firefox 35 and above)
- `'controlCenter'` (Firefox 42 and above)

Optional parameters:

- `callback` function to be called when the menu was sucessfully opened.

hideMenu(target)

```
Mozilla.UITour.hideMenu('appMenu');
```

Closes a menu panel.

previewTheme(theme)

Previews a Firefox theme. `theme` should be a JSON literal:

```
var theme = {
    "category":     "Firefox",
    "iconURL":      "https://addons.mozilla.org/_files/18066/preview_small.jpg?
↪1241572934",
    "headerURL":    "https://addons.mozilla.org/_files/18066/1232849758499.jpg?
↪1241572934",
    "name":         "Dark Fox",
    "author":       "randomaster",
    "footer":       "https://addons.mozilla.org/_files/18066/1232849758500.jpg?
↪1241572934",
    "previewURL":   "https://addons.mozilla.org/_files/18066/preview.jpg?1241572934",
    "updateURL":    "https://versioncheck.addons.mozilla.org/en-US/themes/update-
↪check/18066",
    "accentcolor":  "#000000",
    "header":       "https://addons.mozilla.org/_files/18066/1232849758499.jpg?
↪1241572934",
    "version":      "1.0",
    "footerURL":    "https://addons.mozilla.org/_files/18066/1232849758500.jpg?
↪1241572934",
    "detailURL":    "https://addons.mozilla.org/en-US/firefox/addon/dark-fox-18066/",
    "textcolor":    "#ffffff",
    "id":           "18066",
    "description":  "My dark version of the Firefox logo."
};

Mozilla.UITour.previewTheme(theme);
```

resetTheme()

Removes the previewed theme and resets back to default:

```
Mozilla.UITour.resetTheme();
```

cycleThemes(themes, delay, callback)

Cycles through an array of themes at a set interval with a callback on each step:

```
var themes = [
    ...
];

var myCallback = function () {
    ...
};

Mozilla.UITour.cycleThemes(themes, 5000, myCallback);
```

- `themes` (array)
- `delay` in milliseconds (number)
- `callback` to excecute at each step (function)

getConfiguration(type, callback)

Queries the current browser configuration so the web page can make informed decisions on available highlight targets.

Available `type` values:

- `'sync'`
- `'availableTargets'`
- `'appinfo'`
- `'selectedSearchEngine'`
- `'search'`
- `'canReset'`

Other parameters:

- `callback` function to execute and return with the queried data

Specific use cases:

sync

If `'sync'` is queried the object returned can be used to determine if the user has Sync enabled, and also metrics on the number and types of devices used.

```
Mozilla.UITour.getConfiguration('sync', function (config) {
    console.log(config) // { setup: true, desktopDevices: 2, mobileDevices: 1,␣
↪totalDevices: 3 }
});
```

Important: Sync device count metrics only available in Firefox 50 onwards.

availableTargets

If `'availableTargets'` is queried the object returned by the callback contain array called `targets`. This can be used to determine what highlight targets are currently available in the browser chrome:

```
Mozilla.UITour.getConfiguration('availableTargets', function (config) {
    console.dir(config.targets);
});
```

appinfo

If `'appinfo'` is queried the object returned gives information on the users current Firefox version.

```
Mozilla.UITour.getConfiguration('appinfo', function (config) {
    console.dir(config); //{defaultUpdateChannel: "aurora", version: "48.0a2",␣
↪distribution: "default", defaultBrowser: true}
});
```

The `defaultUpdateChannel` key has many possible values, the most important being:

- `'release'`
- `'beta'`
- `'aurora'`
- `'nightly'`

- `'default'` (self-build or automated testing builds)

The `distribution` key holds the value for the Firefox distributionId property. This value will be `default` in most cases but can differ for repack or funnelcake builds.

Important: `appinfo` is only available in Firefox 35 onward. The `defaultBrowser` property will only be returned on Firefox 40 or later. The `distribution` property will only be returned on Firefox 48 or later.

---

selectedSearchEngine

If `'selectedSearchEngine'` is queried the object returned gives the currently selected default search provider.

```
Mozilla.UITour.getConfiguration('selectedSearchEngine', function (data) {
    console.log(data.searchEngineIdentifier); // 'google'
});
```

Important: `selectedSearchEngine` is only available in Firefox 34 onward.

---

search

This is an alias to `'selectedSearchEngine'` that also returns an array of available search engines.

```
Mozilla.UITour.getConfiguration('search', function (data) {
    console.log(data); // { searchEngineIdentifier: "google", engines: Array[8] }
});

.. Important::

    ``search`` is only available in Firefox 43 onward.
```

canReset

If `'canReset'` is queried the callback returns a boolean value to indicate if a user can refresh their Firefox profile via `resetFirefox()`

```
Mozilla.UITour.getConfiguration('canReset', function (canReset) {
    console.log(canReset); // true
});
```

Important: `canReset` is only available in Firefox 48 onward.

---

setConfiguration(name, value);

Sets a specific browser preference using a given key value pair.

Available key names:

- `'defaultBrowser'`

Specific use cases:

defaultBrowser

Passing `defaultBrowser` will set Firefox as the default web browser.

```
Mozilla.UITour.setConfiguration('defaultBrowser');
```

Important: `setConfiguration('defaultBrowser')` is only available in Firefox 40 onward.

showFirefoxAccounts(extraURLCampaignParams);

Allows a web page to navigate directly to `about:accounts?action=signup&entrypoint=uitour`. In Firefox 47 and beyond, optionally accepts an object of `utm_*` key/values, which will be encoded and appended to the `about:accounts` querystring.

Important: All keys in `extraURLCampaignParams` must begin with `utm_`. If an invalid key is present, the call to `showFirefoxAccounts` will fail.

```
// no extra utm_ campaign params. will open
// about:accounts?action=signup&entrypoint=uitour
Mozilla.UITour.showFirefoxAccounts();

// with extra utm_ campaign params. will open
// about:accounts?action=signup&entrypoint=uitour&utm_foo=bar&utm_bar=baz
Mozilla.UITour.showFirefoxAccounts({
    'utm_foo': 'bar',
    'utm_bar': 'baz'
});
```

Important: `showFirefoxAccounts()` is only available in Firefox 31 onward. `extraURLCampaignParams` parameter only functional in Firefox 47 onward.

Note: A convenience method named `utmParamsFxA` exists in `js/base/search-params.js` that pulls all `utm_` params from the current page's URL and places them in an object (along with pre-defined defaults) ready to pass to `showFirefoxAccounts`.

resetFirefox();

Opens the Firefox reset panel, allowing users to choose to reomve add-ons and customizations, as well as restore browser defaults.

```
Mozilla.UITour.resetFirefox();
```

Important: `resetFirefox()` should be called only in Firefox 48 onwards, and only after first calling `getConfiguration('canReset')` to determine if the user profile is eligible.

addNavBarWidget(target, callback);

Adds an icon to the users toolbar

- `target` can be an highlight target e.g. `forget` (string)
- `callback` to excecute once icon added successfully (function)

```
Mozilla.UITour.addNavBarWidget('forget', function (config) {
    console.log('forget button added to toolbar');
});
```

Important: Only available in Firefox 33.1 onward.

setDefaultSearchEngine(id);

Sets the browser default search engine provider.

- `id` string identifier e.g. 'yahoo' or 'google'.

```
Mozilla.UITour.setDefaultSearchEngine('yahoo');
```

- Identifiers for en-US builds: https://mxr.mozilla.org/mozilla-release/source/browser/locales/en-US/searchplugins/list.txt
- Identifiers for other locales: https://mxr.mozilla.org/l10n-mozilla-release/find?string=browser%2Fsearchplugins%2Flist.txt

Important: Only available in Firefox 34 onward.

setSearchTerm(string);

Populates the search UI with a given search term.

- `string` search term e.g. 'Firefox'

```
Mozilla.UITour.setSearchTerm('Firefox');
```

Important: Only available in Firefox 34 onward.

openSearchPanel(callback);

Opens the search UI drop down panel.

- `callback` function to excecute once the search panel has opened

```
Mozilla.UITour.openSearchPanel(function() {
    console.log('search panel opened');
});
```

Important: Only available in Firefox 34 onward.

setTreatmentTag(name, value);

Sets a key value pair as a treatment tag for recording in FHR.

- `name` tag name for the treatment
- `value` tag value for the treatment

```
Mozilla.UITour.setTreatmentTag('srch-chg-action', 'Switch');
```

Important: Only available in Firefox 34 onward.

getTreatmentTag(name, callback);

Retrieved the value for a set FHR treatment tag.

- `name` tag name to be retrieved
- `callback` function to execute once the data has been retrieved

```
Mozilla.UITour.getTreatmentTag('srch-chg-action', function(value) {
    console.log(value);
});
```

Important: Only available in Firefox 34 onward.

ping(callback);

Pings Firefox to register that the page is using UiTour API.

- `callback` function to execute when Firefox has acknowledged the ping.

```
Mozilla.UITour.ping(function() {
    console.log('UiTour is working!');
});
```

Important: Only available in Firefox 35 onward.

openPreferences(id);

Opens the Firefox Preferences tab at a specified section. Accepts one of the following options to be passed as an *id*:

- `'general'`
- `'search'`
- `'content'`
- `'applications'`
- `'privacy'`

- `'security'`
- `'sync'`
- `'advanced'`

```
Mozilla.UITour.openPreferences('privacy');
```

Important: Only available in Firefox 42 onward.

closeTab();

Closes the current tab.

```
Mozilla.UITour.closeTab();
```

Important: This function will do nothing when called from the last browser window when it contains only one tab. You may need to provide a work around for this edge case in your code. This function is also only available in Firefox 46 onward.

showNewTab();

Opens about:newtab in the same tab.

```
Mozilla.UITour.showNewTab();
```

Important: This function is only available in Firefox 51 onward.

Send to Device widget

The *Send to Device* widget is a single macro form which facilitates the sending of a download link for either Firefox for iOS, Firefox for Android, or both. The form allows sending via SMS or Email, although the SMS copy & messaging is shown only to those in the US. Geo-location is handled in JavaScript using GeoDude. 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.

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/base/send-to-device.less'`
- `'js/base/send-to-device.js'`

4. Include the macro in your page template:

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

The macro defaults to sending links for both Android and iOS apps. You can also pass a 'platform' to specify different configuration options:

```
{{ send_to_device(platform='select') }}
```

- `select` shows a drop down so the user can choose their platform.
- `ios` sends the user a link to Firefox for iOS only.
- `android` sends the user a link to Firefox for Android only.

If the page requires a custom title for the widget, you can also pass an optional heading:

```
{{ send_to_device(title_text='Foo Bar') }}
```

If you do not want to show a title, you can pass `include_title=False`:

```
{{ send_to_device(include_title=False) }}
```

To add a logo and rounded corners to the widget for display in a modal:

```
{{ send_to_device(include_logo=True) }}
```

If you need a customized App Store URL (e.g. including page-specific parameters), you can pass `ios_link`:

```
{{ send_to_device(ios_link=firefox_ios_url('mozorg-ios_page-appstore-button_
↪sd') }}
```

5. Initialize the widget:

In your page JS, initialize the widget using:

```
var form = new Mozilla.SendToDevice();
form.init();
```

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);
}
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.

Example

You can view a simple example by navigating to `/styleguide/docs/send-to-device/` in your local development environment (not available in production).

Micro embedded form

A micro embedded version of the send to device form is also available when targeting a single platform (e.g. `platform='android'` or `platform='ios'`).

The styles can be applied by using the following LESS file (instead of the regular stylesheet):

- `'css/base/send-to-device-micro.less'`

Firefox Accounts Signup Form

Introduction

Certain bedrock pages such as `/firefox/accounts` and `/firefox/firstrun` feature a Firefox Accounts signup form using an embedded `iframe`. To test the signup flow on a non-production environment requires some additional Firefox profile configuration.

Local Development

1. Add the following config settings to your `.env` file:

   ```
   FXA_IFRAME_SRC=https://stomlinson.dev.lcip.org/
   CSP_EXTRA_FRAME_SRC=https://stomlinson.dev.lcip.org/
   ```

2. Set your local development server to run on `127.0.0.1:8111`, e.g. `./manage.py runserver 8111`.
3. Quit Firefox.
4. Create a new profile for testing called `FxA Test Local` by following the instructions here.
5. In the new profile folder, create a file called `user.js` and paste in the following content:

```
user_pref("services.sync.log.appender.file.logOnSuccess", true);
user_pref("identity.fxaccounts.auth.uri", "https://stomlinson.dev.lcip.
↪org/auth/v1");
user_pref("identity.fxaccounts.remote.force_auth.uri", "https://
↪stomlinson.dev.lcip.org/force_auth?service=sync&context=fx_desktop_v1");
user_pref("identity.fxaccounts.remote.signin.uri", "https://stomlinson.
↪dev.lcip.org/signin?service=sync&context=fx_desktop_v1");
user_pref("identity.fxaccounts.remote.signup.uri", "https://stomlinson.
↪dev.lcip.org/signup?service=sync&context=fx_desktop_v1");
user_pref("identity.fxaccounts.settings.uri", "https://stomlinson.dev.
↪lcip.org/settings");
user_pref("identity.fxaccounts.remote.webchannel.uri", "https://
↪stomlinson.dev.lcip.org/");
user_pref("services.sync.tokenServerURI", "https://stomlinson.dev.lcip.
↪org/syncserver/token/1.0/sync/1.5");
user_pref("webchannel.allowObject.urlWhitelist", "https://accounts.
↪firefox.com https://accounts.stage.mozaws.net https://content.cdn.
↪mozilla.net https://input.mozilla.org https://support.mozilla.org␣
↪https://install.mozilla.org https://stomlinson.dev.lcip.org/");

user_pref("general.warnOnAboutConfig", false);
user_pref("devtools.chrome.enabled", true);
user_pref("devtools.debugger.remote-enabled", true);
```

6. Start Firefox using the new profile created in step 3.
7. Verify the FxA settings look correct by opening `about:config` and searching for `identity.fxaccounts`.
8. Navigate to the web page containing the form and test signing up.

Demo Server Testing

1. Quit Firefox.
2. Create a new profile for testing called `FxA Test Demo` by following the instructions here.
3. In the new profile folder, create a file called `user.js` and paste in the following content:

```
user_pref("services.sync.log.appender.file.logOnSuccess", true);
user_pref("identity.fxaccounts.auth.uri", "https://api-accounts.stage.
↪mozaws.net/v1");
user_pref("identity.fxaccounts.remote.force_auth.uri", "https://accounts.
↪stage.mozaws.net/force_auth?service=sync&context=fx_desktop_v1");
user_pref("identity.fxaccounts.remote.signin.uri", "https://accounts.
↪stage.mozaws.net/signin?service=sync&context=fx_desktop_v1");
user_pref("identity.fxaccounts.remote.signup.uri", "https://accounts.
↪stage.mozaws.net/signup?service=sync&context=fx_desktop_v1");
user_pref("identity.fxaccounts.settings.uri", "https://accounts.stage.
↪mozaws.net/settings");
user_pref("identity.fxaccounts.remote.webchannel.uri", "https://accounts.
↪stage.mozaws.net/");
user_pref("services.sync.tokenServerURI", "https://token.stage.mozaws.net/
↪1.0/sync/1.5");
user_pref("webchannel.allowObject.urlWhitelist", "https://accounts.
↪firefox.com https://accounts.stage.mozaws.net https://content.cdn.
↪mozilla.net https://input.mozilla.org https://support.mozilla.org␣
↪https://install.mozilla.org https://stomlinson.dev.lcip.org/");
```

```
user_pref("general.warnOnAboutConfig", false);
user_pref("devtools.chrome.enabled", true);
user_pref("devtools.debugger.remote-enabled", true);
```

4. Start Firefox using the new profile you created in step 2.
5. Verify the FxA settings look correct by opening `about:config` and searching for `identity.fxaccounts`.
6. Navigate to the web page containing the form and test signing up.

Clearing the iframe cache

To clear browser cache while testing multiple accounts, append `/clear` to the iframe's source URL, e.g. `https://accounts.stage.mozaws.net/clear`

Embedding on a page

To embed the Firefox Accounts iframe on a page:

1. **Add the FxA JavaScript & Less files to the page's bundles:**
   - `media/js/base/mozilla-fxa-iframe.js`
   - `media/css/base/mozilla-fxa-iframe.less`
2. Add the following attributes and values to any element on the page (the parent element of the `<iframe>` is a good option):

   ```
   id="fxa-iframe-config" data-host="{{ settings.FXA_IFRAME_SRC }}"
   data-mozillaonline-host="{{ settings.FXA_IFRAME_SRC_MOZILLAONLINE
   }}"
   ```

3. **Add the `<iframe>` to the page with the following attributes and values:**

   ```
   <iframe
   id="fxa" scrolling="no" data-src="{{ settings.FXA_IFRAME_SRC }}?
   utm_campaign=fxa-embedded-form&amp;utm_content=fx-{{ version
   }}&amp;service=sync&amp;context=iframe&amp;style=chromeless&amp;
   haltAfterSignIn=true"></iframe>
   ```

   Note: Note that each implementation of the `<iframe>` may require unique URL parameters in the `data-src` attribute for some or all of the following:
   - `utm_medium`
   - `utm_source`
   - `entrypoint`

Analytics

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.

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.

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.

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.

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 nav, footer, and CTA/button link elements, add these data attributes:

Data Attribute	Expected Value
data-link-type	'nav', 'footer', or 'button'
data-link-name	name or text of the link

For all button elements, add this data attribute:

Data Attribute	Expected Value
data-button-name	name or text of the link

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'