# Kitsune Documentation

***Release master***

**SUMO devs**

August 21, 2017

Contents

CHAPTER 1

# Part 1: Contributor's Guide

## Join this project!

Kitsune is the software that runs SUMO (support.mozilla.org) which provides support for Firefox and other Mozilla software.

Interested in helping out? Here's a bunch of things we need your help with.

### Help with support!

First off, you can help people get the most out of Firefox by joining the awesome support community. This community not only helps people with their Firefox issues, but is also the front line in helping drive Firefox development.

For more information on this, see the quickstart guide on the SUMO site.

### Help with hacking!

First step is to set up Kitsune so you can run it and hack on it. For that, see the *Hacking HOWTO for Contributors*.

If you have problems, please let us know! See *Contact us*.

After you've got it up and running, check out the list of bugs that are marked for mentoring:

https://wiki.mozilla.org/Webdev/GetInvolved#support.mozilla.org_.28SUMO.29

If you want to work on one of these bugs, contact the mentor listed in the whiteboard section.

### Help with making Kitsune easier for hacking on!

We're working on making Kitsune easier to hack on. This entails:

- reducing the steps it takes to get Kitsune running down to a smaller minimal set
- making this documentation better
- providing better resources for people who are interested in helping out
- providing better scripts to automate installing and maintaining Kitsune

Any thoughts you have on making this easier are much appreciated. Further, if you could help us, that'd be valuable to us and all those who follow in your footsteps.

# Hacking HOWTO for Contributors



## Summary

This chapter helps you get a minimal installation of Kitsune up and running to make it easier to contribute.

If you're setting Kitsune up for deployment, instead of development, make sure you read all the way to the end and then read the additional sections.

If you have any problems getting Kitsune running, let us know. See the *Contact us*.

## Operating systems

### Windows

If you're using Windows as your operating system, you'll need to set up a virtual machine and run Kitsune in that. Kitsune won't run in Windows.

If you've never set up a virtual machine before, let us know and we can walk you through it. Having said that, it's not easy to do for people who haven't done it before.

### Mac OSX

Just follow along with the instructions below. Several of us use OSX, so if you run into problems, let us know.

### Linux

We know these work in Debian, Ubuntu, Arch, and will probably work in other distributions. It's likely that you'll encounter some steps that are slightly different. If you run into problems, let us know.

### Vagrant

We also have an option of using a virtual machine with Vagrant for an all-in-one installation. This installs all required dependencies and sets up your environment in such a way that makes it easy to run.

For full instruction about installing kitsune via vagrant, check this *installation-vagrant* article.

Once Vagrant is installed, run `vagrant up` to start and configure your virtual machine and `vagrant ssh` to SSH into the box.

Once inside the virtual machine, you can start the server by running the following commands:

```
source virtualenv/bin/activate
cd kitsune
./manage.py runserver 0.0.0.0:8000
```

Now, just navigate to <*http://localhost:8000*> to see the application!

*Skip to Testing*

## Requirements

These are required for the minimum installation:

- git
- Python 2.7
- pip: https://pip.pypa.io/en/latest/
- virtualenv
- MariaDB 5.5 server and client headers
- Memcached Server
- libxml and headers
- libxslt and headers
- libjpeg and headers
- zlib and headers
- libssl and headers
- libffi and headers

These are optional:

- Redis
- ElasticSearch: *Search*

Installation for these is very system dependent. Using a package manager, like yum, aptitude, or brew, is encouraged.

#### memcached

You need to have memcached running. Otherwise CSRF stuff won't work.

If you are running OSX and using homebrew, you can do something like:

```
$ brew install memcached
```

and launch it:

```
$ memcached
```

If you are running RedHat/CentOS/Fedora, once you have installed memcached you can start it and configure it to run on startup using:

```
$ chkconfig memcached on
$ /etc/init.d/memcached start
$ service memcached start
```

---

**Note:** This should probably be somewhere else, but the easy way to flush your cache is something like this:

```
echo "flush_all" | nc localhost 11211
```

Assuming you have memcache configured to listen to 11211.

---

## Getting the Source

Grab the source from Github using:

```
$ git clone https://github.com/mozilla/kitsune.git
$ cd kitsune
```

## Setting up an Environment

It is strongly recommended to run Kitsune in a virtual environment, which is a tool to isolate Python environments from each other and the system. It makes local development much easier, especially when working on multiple projects.

To create a virtual environment:

```
$ virtualenv venv
```

which creates a virtualenv named "venv" in your current directory (which should be the root of the git repo. Now activate the virtualenv:

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

You'll need to run this command every time you work on Kitsune, in every terminal window you use.

## Installing dependencies

### Python Packages

All the pure-Python requirements are provided in the requirements directory. We use a tool called `peep` to install packages and make sure versions are pinned.

```
$ ./peep.sh install -r requirements/default.txt
```

Additionally, you may install some useful development tools. These are not required, but are helpful:

```
$ ./peep.sh install -r requirements/dev.txt
```

If you intend to run the function UI tests, you will also need to install the appropriate dependencies:

```
$ ./peep.sh install -r requirements/test.txt
```

If you have any issues installing via `peep`, be sure you have the required header files from the packages listed in the requirements section above.

For more information on `peep`, refer to the README on the Github page for the project.

### Node.js Packages

Kitsune relies on some Node.js packages. To get those, you will need to install Node.js and NPM.

Now install the Node.js dependencies with:

```
$ npm install
```

This should create a directory named `node_modules` in your git repo.

### Frontend Packages

Kitsune gets libraries and dependencies for client side code from Bower. Bower is installed as a part of the NPM packages in the last step. To install these front-end dependencies run:

```
$ ./node_modules/.bin/bower install
```

This will download dependencies into `bower_components`.

## Configuration and Setup

### settings_local.py

There is a file called `settings_local.py.dist` in the `kitsune/` directory. This contains a sample set of local settings. Copy the file, name it `settings_local.py`. and edit it, following the instructions within. Don't forget to change `<YOUR_PASSWORD>` to your actual database password.

Note the two settings `TEST_CHARSET` and `TEST_COLLATION`. Without these, the test suite will use MySQL's (moronic) defaults when creating the test database (see below) and lots of tests will fail. Hundreds.

Additionally, you can copy and modify any settings from `kitsune/settings.py` into `kitsune/settings_local.py` and the value will override the default.

### Database

You defined a database connection in `kitsune/settings_local.py`.

Now create the database and grant permissions to the user, based on your database settings. For example, using the settings above:

```
$ mysql -u root -p
mysql> CREATE DATABASE kitsune CHARACTER SET utf8 COLLATE utf8_unicode_ci;
mysql> GRANT ALL ON kitsune.* TO kitsune@localhost IDENTIFIED BY '<YOUR_PASSWORD>';
```

To initialize the database, do:

```
$ ./manage.py migrate
```

Then we create a superuser to log into kitsune:

```
./manage.py createsuperuser
```

You'll now have an up-to-date database!

After logging in, you can create a profile for the user by going to `/users/edit` in your browser.

See also the *important wiki documents* documentation.

### Install Sample Data

We include some sample data to get you started. You can install it by running this command:

```
$ ./manage.py generatedata
```

### Install linting tools

Kitsune uses Yelps Pre-commit for linting. It is installed as a part of the dev dependencies in `requirements/dev.txt`. To install it as a Git pre-commit hook, run it:

```
$ venv/bin/pre-commit install
```

After this, every time you commit, Pre-commit will check your changes for style problems. To run it manually, you can use the command:

```
$ venv/bin/pre-commit run
```

which will run the checks for only your changes, or if you want to run the lint checks for all files:

```
$ venv/bin/pre-commit run --all-files
```

For more details see the Pre-commit docs.

### Product Details Initialization

One of the packages Kitsune uses, `product_details`, needs to fetch JSON files containing historical Firefox version data and write them within its package directory. To set this up, run this command to do the initial fetch:

```
$ ./manage.py update_product_details
```

### Pre-compiling JavaScript Templates

We use nunjucks to render Jinja-style templates for front-end use. These templates get updated from time to time and you will need to pre-compile them to ensure that they render correctly. You have two options here:

- One time pre-compile (use this if you are not modifying the templates):

  ```
  $ ./manage.py nunjucks_precompile
  ```

- Use gulp to watch for changes and pre-compile (use this if you are making changes to the templates):

  ```
  $ ./node_modules/.bin/gulp watch
  ```

## Testing it out

To start the dev server, run `./manage.py runserver`, then open up `http://localhost:8000`.

If everything's working, you should see a somewhat empty version of the SUMO home page!

---

**Note:**

If you see an unstyled site and empty CSS files, you have to remove all empty files having a `.less.css` since they are empty and should be regenerated.

To do this, run the following command on the top directory of your Kitsune clone:

```
$ find . -name "*.less.css" -delete
```

Verify the `LESS_BIN` setting in settings_local.py. Then *hard-refresh* your pages on the browser via *Ctrl* + *Shift* + *R*.

---

### Setting it up

A great way to check that everything really is working is to run the test suite. You'll need to add an extra grant in MySQL for your database user:

```
$ mysql -u root -p
mysql> GRANT ALL ON test_kitsune.* TO kitsune@localhost IDENTIFIED BY '<YOUR_PASSWORD>';
```

The test suite will create and use this database, to keep any data in your development database safe from tests.

### Running the tests

Running the test suite is easy:

```
$ ./manage.py test -s --noinput --logging-clear-handlers
```

This may open a Firefox window, which will close automatically.

For more information, see the *test documentation*.

## Trouble-shooting

#### Error: A csrf_token was used in a template, but the context did not provide the value

If you see this, you likely have CACHES specifying to use memcached in your `kitsune/settings_local.py` file, but you don't have memcached running.

See *memcached*.

### Advanced install

The above covers a minimal install which will let you run most of Kitsune. In order to get everything working, you'll need to install some additional bits.

See the following chapters for installing those additional bits:

- Redis: *Redis*
- RabbitMQ: *Celery and Rabbit*
- Elastic Search: *Search*
- Email: *Email from Kitsune*

If you want to install Kitsune on an Apache server in a mod_wsgi environment, see *Running Kitsune with mod_wsgi*.

## Contact us

### SUMO contributor forums

If you're a SUMO contributor, then consider using the contributor forums. This is the place for SUMO community discussions.

### Kitsune hackers

If you're hacking on the Kitsune code and have questions, ping us on IRC.

We hang out in `#sumodev` on irc.mozilla.org. We're usually around during the day in Eastern Time.

If you ask something and all you get is silence, then it's probably the case that we're not around. Please try pinging us again.

You can also use the dev-sumo mailing list.

Current primary developers:

- Ricky Rosario (r1cky)
- Will Kahn-Greene (willkg)
- Rehan Dalal (rdalal)
- Mike Cooper (mythmon)

CHAPTER 2

# Part 2: Developer's Guide

## Conventions

This document contains coding conventions, and things to watch out for, etc.

### Coding conventions

We follow most of the practices as detailed in the Mozilla webdev bootcamp guide.

It is recommended that you install the linter we provide as a pre-commit hook:

```
$ ./scripts/hooks/lint.pre-commit
```

### Git conventions

#### Git workflow

See *Patching Kitsune* for how we use Git, branches and merging.

#### Git commit messages

Git commit messages should have the following form:

```
[bug xxxxxxx] Short summary

Longer explanation with paragraphs and lists and all that where
each line is under 72 characters.

* bullet 1
* bullet 2

Etc. etc.
```

Summary line should be capitalized, short and shouldn't exceed 50 characters. Why? Because this is a convention many git tools take advantage of.

If the commit relates to a bug, the bug should show up in the summary line in brackets.

There should be a blank line between the summary and the rest of the commit message. Lines shouldn't exceed 72 characters.

See these guidelines for some more explanation.

### Git resources and tools

See Webdev bootcamp guide for:

- helpful resources for learning git
- helpful tools

# Patching Kitsune

Submitting a patch to Kitsune is easy! (Fair warning: writing the patch may not be ;)

We use pull requests to manage patches and code reviews, and Bugzilla to handle actual bug tracking.

Because of our infrastructure and how we do deployments, we've developed a fairly straight-forward workflow in git for submitting patches. This is outlined below.

You should run the tests before submitting a pull request. You can find help for getting set up in the *installation docs* and help for running tests in the *testing docs*.

If you ever find yourself stuck, come look for us in #sumodev on Mozilla's IRC network. We're happy to help!

You'll need a Github account and a Bugzilla account.

## The Quick and Dirty

Very quick, very little explanation. Those with strong git fu may already see some shortcuts. Use them!

First, clone your fork, and then point the master branch to Mozilla's fork. Assuming your Github account is `foobar` and you've already forked Kitsune:

```
git clone https://github.com/foobar/kitsune
cd kitsune
git remote add mozilla https://github.com/mozilla/kitsune.git
git fetch mozilla
git checkout -t mozilla/master -B master
```

If you haven't set up your local git user, please do before committing any code for Kitsune. This way you can take credit for your work:

```
git config user.email your@github.email
git config user.name "Your Name"
```

You should only need to do that once. Here's the bit to do every time:

```
git checkout master
git reset --hard mozilla/master
git checkout -b my-feature-123456

# Make a change and commit it.
$EDITOR path/to/file.py
git add path/to/file.py
git commit -m "[Bug 123456] Fooing and the Barring."
git push --set-upstream origin my-feature
```

```
# Open a pull request, get review.
# Respond to feedback:
$EDITOR path/to/file.py
git add path/to/file.py
git commit -m "Feedback from Barfoo"
git push
```

Eventually you'll get an r+. If you have commit access, now you can go ahead and merge your branch. You may, if you want, rebase your branch to clean up any embarrassing mistakes, but it isn't required. If you don't have commit access the next part will be done by someone who does.

There are two options. The first is to press the Big Green Button in GitHub PRs that says "Merge pull Request". If you would prefer to do it manually (or if there are merge conflicts, you can do this:

```
# r+! Merge
git checkout master
git fetch mozilla
git reset --hard mozilla/master
git merge --no-ff my-feature-123456
git push mozilla master  # Bots will alert everyone!
git push origin master  # Optional but nice.
```

After the pull request is closed:

```
git push origin :my-feature  # Delete the remote branch. Nice to others.
git branch -D my-feature # Delete the local branch, if you're done.
```

## The Details

This is the process in more detail, for a relatively small change that will only need one commit, and doesn't need any special treatment, like landing on special branches.

### Fork and Clone Kitsune

On Github, hit the **Fork** button. You'll want to clone **your** fork of the project, at least initially:

```
git clone git@github.com:<yourname>/kitsune.git
```

To help keep up to date, you should add `mozilla/kitsune` as a remote:

```
cd kitsune
git remote add mozilla https://github.com/mozilla/kitsune.git
```

You should avoid changing your `master` branch, it should track `mozilla/master`. This can help:

```
git fetch mozilla
# Update your master branch to track Mozilla's master branch instead.
git checkout -B master -t mozilla/master # Update your master branch to
```

If you haven't set up your local git user, please do before committing any code for Kitsune. This way you can take credit for your work:

```
git config user.email your@github.email
git config user.name "Your Name"
```

The correct way to keep your local master up to date is:

```
git checkout master
git fetch mozilla
git reset --hard mozilla/master
```

This will forcibly move your local master branch to whatever is on the Mozilla master branch, destroying anything you have committed that wasn't pushed. Remember to always work on a branch that is not master!

### Find a Bug

Step one is to make sure there's a bug in Bugzilla. Obvious "bugs" just need a Bugzilla bug to track the work for all the involved teams. There are a number of open bugs if you want to try your hand at fixing something!

New features or changes to features need bugs to build a consensus of developers, support team members, and community members, before we decide to make the change. If you want to change something like this, be sure to file the bug and get a consensus first. We'd hate to have you spend time on a patch we can't take.

### Take the Bug

To make sure no one else is working on the bug at the same time, assign it to yourself in Bugzilla. If you have the proper permissions there's an easy "take" link next to the Assignee field. Ask in the IRC for details.

You can assign bugs to yourself even if you aren't going to immediately work on them (though make sure you will get to them sooner rather than later). Once you are actively working on a bug, set the bug to the `ASSIGNED` state.

### Fix the Bug on a Branch

**Note:** This describes the process for fixing a relatively small bug in a single-commit. Large features may differ.

All bug fixes, changes, new features, etc, should be done on a "feature branch", which just means "any branch besides `master`." You should make sure your local `master` branch is up to date (see above) before starting a new feature branch. Your feature branch should include the bug number in the branch name, if applicable.

```
git checkout master
git fetch mozilla
git reset --hard upstream/master  # Update local master.
git checkout -b my-feature-branch-123456  # Some logical name.
```

Now you're on a feature branch, go ahead and make your changes. Assuming you haven't added any new files, you can do:

```
git commit -a -m "[Bug 123456] Fix the foo and the bar."
```

If you did add new files, you will have to `git add` them before committing.

Note that the commit message contains the bug number after the word "Bug". This helps us and our IRC bots!

### Open a Pull Request

Once you have the bug fixed locally, you'll need to push the changes up to Github so you can open a pull request.

```
git push --set-upstream origin my-feature-branch
```

Then, in your browser, navigate to `https://github.com/<yourname>/kitsune/compare/my-feature-branch` and hit the **Pull Request** button. If the commit message is clear, the form should be filled out enough for you to submit it right away.

We add an `r?` in the pull request message indicating that this pull request is ready to go and is looking for someone to review it.

Othertimes you may want to open a pull request early that isn't quite ready to merge. This is a great way to share the work that you are doing, and get early feedback. Make it clear that your PR isn't ready by putting `[WIP]` in the title. Also make sure to say when it is ready! The best way to do this is to remove `[WIP]` from the title and make a comment asking for `r?`.

### Respond to Review

It's very rare that pull requests will be checked in immediately. Most of the time they will go through one or more rounds of code review and clean-up.

Code review is usually comments made on the pull request or commits in Github, asking for specific changes to be made. If the requested change isn't clear, or you disagree with it, feel free to ask questions inline. Isn't Github's line-by-line commenting great?

Assuming a few small changes need to be made, make the changes locally on the feature branch, then put them in a *new commit*. This makes it easier from reviewers. For example, if Erik reviewed the pull request and asked for some fixes, you might do this:

```
git checkout my-feature-branch
# Make the changes.
git commit -a -m "Feedback from Erik."
git push origin my-feature-branch
```

Github will automatically add the new commit to the pull request, so we'll see it. Leaving it in a separate commit at this stage helps the reviewer see what changes you've made.

There may be more than one round of feedback, especially for complex bugs. The process is exactly the same after each round: make the changes, add them in yet another new commit, push the changes.

There are also a few bots that might interact with your PR. In particular, our continuous integration service will run tests and style checks on your new code. All PRs must be approved by the CI system before they will be merged, so watch out. They show up as either a red X or a green check mark in the PR.

### Ready to Merge!

Once a pull request has gotten an `r+` ("R-plus", it's from Bugzilla) it's ready to merge in. At this point you can rebase and squash any feedback/fixup commits you want, but this isn't required.

If you don't have commit access, someone who does may do this for you, if they have time. Alternatively, if you have commit access, you can press GitHub's "Merge pull request" button, which does a similar process to below. This is the preferred way to merge PRs when there are no complications.

```
git checkout master
git reset --hard mozilla/master
git merge --no-ff my-feature-branch-123456
# Make sure tests pass.
python manage.py test
git push
```

You're done! Congratulations, soon you'll have code running on one of the biggest sites in the world!

Before pushing to `mozilla/master`, I like to verify that the merge went fine in the logs. For the vast majority of merges, *there should not be a merge commit*.

```
git log --graph --decorate
git push mozilla master             # !!! Pushing code to the primary repo/branch!

# Optionally, you can keep your Github master in sync.
git push origin master              # Not strictly necessary but kinda nice.
git push origin :my-feature-branch  # Nice to clean up.
```

This should automatically close the PR, as GitHub will notice the merge commit.

Once the commit is on `mozilla/master`, copy the commit url to the bug.

Once the commit has been deployed to stage and prod, set the bug to `RESOLVED FIXED`. This tells everyone that the fix is in production.

# Development

This covers loosely how we do big feature changes.

## Changes that involve new Python dependencies

We use peep to install dependencies. That means that all dependencies have an associated hash (or several) that are checked at download time. This ensures malicious code doesn't sneak in through dependencies being hacked, and also makes sure we always get the exact code we developed against. Changes in dependencies, malicious or not, will set of red flags and require human intervention.

A peep requirement stanza looks something like this:

```
# sha256: mmQhHJajJiuyVFrMgq9djz2gF1KZ98fpAeTtRVvpZfs
Django==1.6.7
```

hash lines can be repeated, and other comments can be added. The stanza is delimited by non-comment lines (such as blank lines or other requirements).

To add a new dependency, you need to get a hash of the dependency you are installing. There are several ways you could go about this. If you already have a tarball (or other appropriate installable artifact) you could use `peep hash foo.tar.gz`, which will give the base64 encoded sha256 sum of the artifact, which you can then put into a peep stanza.

If you don't already have an artifact, you can simply add a line to the requirements file without a hash, for example `Django`. Without a version, peep will grab the latest version of the dependency. If that's not what you want, put a version there too, like `Django==1.6.7`.

Now run peep with:

```
./peep.sh install -r requirements/default.txt
```

Peep will download the appropriate artifacts (probably a tarball), hash it, and print out something like:

```
The following packages had no hashes specified in the requirements file, which
leaves them open to tampering. Vet these packages to your satisfaction, then
add these "sha256" lines like so:


# sha256: mmQhHJajJiuyVFrMgq9djz2gF1KZ98fpAeTtRVvpZfs
Django==1.6.7
```

Copy and paste that stanza into the requirements file, replacing the hash-less stanza you had before. Now re-run peep to install the file for real. Look around and make sure nothing horrible went wrong, and that you got the right package. When you are satisfied that you have what you want, commit, push, and rejoice.

## Changes that involve database migrations

Any changes to the database (model fields, model field data, adding permissions, ...) require a migration.

### Running migrations

To run migrations, you do:

```
$ ./manage.py migrate
```

It'll perform any migrations that haven't been performed for all apps.

### Creating a schema migration

To create a new migration the automatic way:

1. make your model changes
2. run:

   ```
   ./manage.py makemigrations <app>
   ```

   where `<app>` is the app name (sumo, wiki, questions, ...).
3. add a module-level docstring to the new migration file specifying what it's doing since we can't easily infer that from the code because the code shows the new state and not the differences between the old state and the new stage
4. run the migration on your machine:

   ```
   ./manage.py migrate
   ```
5. run the tests to make sure everything works
6. add the new migration files to git
7. commit

**See also:**

**https://docs.djangoproject.com/en/1.7/topics/migrations/#adding-migrations-to-apps** Django documentation: Adding migrations to apps

## Creating a data migration

Creating data migrations is pretty straight-forward in most cases.

To create a data migration the automatic way:

1. run:

   ```
   ./manage.py makemigrations --empty <app>
   ```

   where `<app>` is the app name (sumo, wiki, questions, ...).

2. edit the data migration you just created to do what you need it to do
3. make sure to add *reverse_code* arguments to all *RunPython* operations which undoes the changes
4. add a module-level docstring explaining what this migration is doing
5. run the migration forwards and backwards to make sure it works correctly
6. add the new migration file to git
7. commit

**See also:**

**https://docs.djangoproject.com/en/1.7/topics/migrations/#data-migrations** Django documentation: Data Migrations

**See also:**

https://docs.djangoproject.com/en/1.7/ref/migration-operations/#runpython

### Data migrations for data in non-kitsune apps

If you're doing a data migration that adds data to an app that's not part of kitsune, but is instead a library (e.g. django-waffle), then create the data migration in the sumo app and add a dependency to the latest migration in the library app.

For example, this adds a dependency to django-waffle's initial migration:

```
class Migration(migrations.Migration):

    dependencies = [
        ...
        ('waffle', '0001_initial'),
        ...
    ]
```

## Changes that involve reindexing

With Elastic Search, it takes a while to reindex. We need to be able to reindex without taking down search.

This walks through the workflow for making changes to our Elastic Search code that require reindexing.

### Things about non-trivial changes

1. We should roll multiple reindex-requiring changes into megapacks when it makes sense and doesn't add complexity.
2. Developers should test changes with recent sumo dumps.

### Workflow for making the changes

1. work on the changes in a separate branch (just like everything else we do)
2. make a pull request
3. get the pull request reviewed
4. rebase the changes so they're in two commits:

(a) a stage 1 commit that changes `ES_WRITE_INDEXES`, updates the mappings and updates the indexing code

(b) a stage 2 commit that changes `ES_INDEXES`, changes `ES_WRITE_INDEXES`, and changes the search view code

**Avoid cosmetic changes that don't need to be made (e.g. pep-8 fixes, etc.)**

5. push those changes to the same pull request
6. get those two changes reviewed

Once that's ok, then that branch should get updated from master, then pushed to stage to get tested.

That branch should **not** land in master, yet.

### Workflow for reviewing changes

Go through and do a normal review.

After everything looks good, the developer should rebase the changes so they're in a stage 1 commit and a stage 2 commit.

At that point:

1. Verify each commit individually. Make sure the code is correct. Make sure the tests pass. Make sure the site is functional.
2. Verify that the `ES_INDEXES` and `ES_WRITE_INDEXES` settings have the correct values in each commit.

### Workflow for pushing changes to stage

Don't land the changes in master, yet!

If you hit problems, deploy the master branch back to the stage server and go back to coding/fixing.

1. Push the branch you have your changes in to the official mozilla/kitsune remote.
2. Deploy the stage 1 commit to stage.
3. Verify that search still works.
4. Verify that the index settings are correct—look at the `ES_INDEXES` and `ES_WRITE_INDEXES` values.
5. Destructively reindex.
6. Deploy the stage 2 commit to stage.
7. Verify that search still works.
8. Verify that the index settings are correct—look at the `ES_INDEXES` and `ES_WRITE_INDEXES` values.
9. Verify bugs that were fixed with the new search code.

### Workflow for pushing those changes to production

If we're also doing a production push, first push next to production and verify that everything is fine. Then continue.

1. Tell the other sumo devs to hold off on pushing to master branch and deploying. Preferably by email and IRC.
2. Once you've told everyone, land the changes in master.
3. Deploy the stage 1 commit to production.

4. Verify that search works.
5. Destructively reindex to the new write index.
6. When reindexing is done, push the stage 2 commit to production.
7. Verify that search works.
8. Verify bugs that were fixed with the new search code.

Pretty sure this process allows us to back out at any time with minimal downtime.

#### On the next day

If everything is still fine, then merge the special branch into master and delete the old read index.

Announce "STUCK THE LANDING!" after a successful mapping change deployment.

# All about testing

Kitsune has a fairly comprehensive Python test suite. Changes should not break tests—only change a test if there is a good reason to change the expected behavior—and new code should come with tests.

## Running the Test Suite

If you followed the steps in *the installation docs*, then you should be all set setup-wise.

To run the tests, you need to do:

```
./manage.py test
```

That doesn't provide the most sensible defaults for running the tests. Here is a good command to alias to something short:

```
./manage.py test -s --noinput --logging-clear-handlers
```

The `-s` flag is important if you want to be able to drop into PDB from within tests.

Some other helpful flags are:

**`-x`:** Fast fail. Exit immediately on failure. No need to run the whole test suite if you already know something is broken.

**`--pdb`:** Drop into PDB on an uncaught exception. (These show up as `E` or errors in the test results, not `F` or failures.)

**`--pdb-fail`:** Drop into PDB on a test failure. This usually drops you right at the assertion.

**`--no-skip`:** All SkipTests show up as errors. This is handy when things shouldn't be skipping silently with reckless abandon.

### Running a Subset of Tests

You can run part of the test suite by specifying the apps you want to run, like:

```
./manage.py test kitsune/wiki kitsune/search kitsune/kbforums
```

You can also specify modules:

```
./manage.py test kitsune.wiki.tests.test_views
```

You can specify specific tests:

```
./manage.py test kitsune.wiki.tests.test_views:VersionGroupTests.test_version_groups
```

See the output of `./manage.py test --help` for more arguments.

### Running tests without collecting static files

By default the test runner will run `collectstatic` to ensure that all the required assets have been collected to the static folder. If you do not want this default behavior you can run:

```
REUSE_STATIC=1 ./manage.py test
```

### The Test Database

The test suite will create a new database named `test_%s` where `%s` is whatever value you have for `settings.DATABASES['default']['NAME']`.

Make sure the user has `ALL` on the test database as well. This is covered in the installation chapter.

When the schema changes, you may need to drop the test database. You can also run the test suite with `FORCE_DB` once to cause Django to drop and recreate it:

```
FORCE_DB=1 ./manage.py test -s --noinput --logging-clear-handlers
```

## Writing New Tests

Code should be written so it can be tested, and then there should be tests for it.

When adding code to an app, tests should be added in that app that cover the new functionality. All apps have a `tests` module where tests should go. They will be discovered automatically by the test runner as long as the look like a test.

- If you're expecting `reverse` to return locales in the URL, use `LocalizingClient` instead of the default client for the `TestCase` class.
- We use "modelmakers" instead of fixtures. Models should have modelmakers defined in the tests module of the Django app. For example, `forums.tests.document` is the modelmaker for `forums.Models.Document` class.

## Changing Tests

Unless the current behavior, and thus the test that verifies that behavior is correct, is demonstrably wrong, don't change tests. Tests may be refactored as long as its clear that the result is the same.

## Removing Tests

On those rare, wonderful occasions when we get to remove code, we should remove the tests for it, as well.

If we liberate some functionality into a new package, the tests for that functionality should move to that package, too.

## JavaScript Tests

Frontend JavaScript is currently tested with Mocha.

### Running JavaScript Tests

To run tests, make sure you have have the NPM dependencies installed, and then run:

```
$ scripts/mocha.sh
```

### Writing JavaScript Tests

Mocha tests are discovered using the pattern `kitsune/*/static/*/js/tests/**/*.js`. That means that any app can have a *tests* directory in its JavaScript directory, and the files in there will all be considered test files. Files that don't define tests won't cause issues, so it is safe to put testing utilities in these directories as well.

Here are a few tips for writing tests:

- Any HTML required for your test should be added by the tests or a `beforeEach` function in that test suite. React is useful for this.
- You can use *sinon* to mock out parts of libraries or functions under test. This is useful for testing AJAX.
- The tests run in a Node.js environment. A browser environment can be simulated using `jsdom`. Specifically, `mocha-jsdom` is useful to set up and tear down the simulated environment.

## Functional UI Tests

We can do more comprehensive front-end testing with the functional UI tests. They're located in the `tests/functional` directory.

### Installing dependencies

Follow the steps in *the installation docs*, including the test dependencies to make sure you have everything you need to run the tests. If you're running the tests against a deployed environment then there's no need to install anything other than Tox.

### Create test users

Some of the tests require logging in as a administrator, and others require logging in as a user. To run these tests you will need to create accounts in the target environment. If you're running against a local instance of the application you can create these users by running the following script:

```
$ ./manage.py shell < ./scripts/create_user_and_superuser.py
```

If you want to run the tests that require administrator access against a deployed instance, then you will need to ask someone on IRC to upgrade one of your test accounts.

The credentials associated with the test users are stored in a JSON file, which we then pass to the tests via the command line. If you used the above mentioned script, then these users are stored in `/scripts/travis/variables.json`. The variables file needs to be referenced on the command line when running the tests.

The following is an example JSON file with the values missing. You can use this as a template:

```
{
  "users": {
    "default": {
      "username": "",
      "password": "",
      "email": ""},
    "admin": {
      "username": "",
      "password": "",
      "email": ""}
  }
}
```

For the purposes of the examples below, assume you named your copy of the file `my_variables.json`.

### Running the tests

Tests are run using the command line. Below are a couple of examples of running the tests:

To run all of the desktop tests against the default environment:

```
$ PYTEST_ADDOPTS=--variables=my_variables.json
$ tox -e desktop
```

To run against a different environment, pass in a value for `--base-url`, like so:

```
$ PYTEST_ADDOPTS=--base-url=https://support.allizom.org
$ PYTEST_ADDOPTS="${PYTEST_ADDOPTS} --variables=my_variables.json"
$ tox -e desktop
```

To run the mobile tests you will need to target a mobile device or emulator using a tool like Appium. You can create a suitable variables file with the necessary capabilities like so:

```
{
  "capabilities": {
    "platformName": "iOS",
    "platformVersion": "9.2",
    "deviceName": "iPhone 6",
    "browserName": "Safari",
  }
}
```

Then you can run this like so:

```
$ PYTEST_ADDOPTS=--driver=Remote
$ PYTEST_ADDOPTS="${PYTEST_ADDOPTS} --port=4723"
$ PYTEST_ADDOPTS="${PYTEST_ADDOPTS} --variables=capabilities.json"
$ PYTEST_ADDOPTS="${PYTEST_ADDOPTS} --variables=my_variables.json"
$ tox -e mobile
```

Alternatively, if you run the mobile tests in Firefox the user agent will be changed to masquerade as a mobile browser.

The pytest plugin that we use for running tests has a number of advanced command line options available. To see the options available, run `pytest --help`. The full documentation for the plugin can be found here.

# Celery and Rabbit

Kitsune uses Celery to enable offline task processing for long-running jobs like sending email notifications and re-rendering the Knowledge Base.

Though Celery supports multiple message backends, we use, and recommend that you use, RabbitMQ. RabbitMQ is an AMQP message broker written in Erlang.

## When is Celery Appropriate

You can use Celery to do any processing that doesn't need to happen in the current request-response cycle. Examples are generating thumbnails, sending out notification emails, updating content that isn't about to be displayed to the user, and others.

Ask yourself the question: "Is the user going to need this data on the page I'm about to send them?" If not, using a Celery task may be a good choice.

## RabbitMQ

### Installing

RabbitMQ should be installed via your favorite package manager. It can be installed from source but has a number of Erlang dependencies.

### Configuring

RabbitMQ takes very little configuration.

```
# Start the server.
sudo rabbitmq-server -detached

# Set up the permissions.
rabbitmqctl add_user kitsune kitsune
rabbitmqctl add_vhost kitsune
rabbitmqctl set_permissions -p kitsune kitsune ".*" ".*" ".*"
```

That should do it. You may need to use `sudo` for `rabbitmqctl`. It depends on the OS and how Rabbit was installed.

## Celery

### Installing

Celery (and Django-Celery) is part of our dependencies. You shouldn't need to do any manual installation.

### Configuring and Running

We set some reasonable defaults for Celery in `settings.py`. These can be overriden either in `settings_local.py` or via the command line when running `manage.py celeryd`.

In `settings_local.py` you should set at least this, if you want to use Celery:

```
CELERY_ALWAYS_EAGER = False
```

This defaults to `True`, which causes all task processing to be done online. This lets you run Kitsune even if you don't have Rabbit or want to deal with running workers all the time.

You can also configure the log level or concurrency. Here are the defaults:

```
CELERYD_LOG_LEVEL = logging.INFO
CELERYD_CONCURRENCY = 4
```

Then to start the Celery workers, you just need to run:

```
./manage.py celeryd
```

This will start Celery with the default number of worker threads and the default logging level. You can change those with:

```
./manage.py celeryd --log-level=DEBUG -c 10
```

This would start Celery with 10 worker threads and a log level of `DEBUG`.

# Redis

This covers installing Redis.

## Installation

Install Redis on your machine.

There are three `.conf` files in `configs/redis/`. One is for testing and is used in `settings_test.py`. The other two are used for the sections in `REDIS_BACKEND`.

There are two ways to set this up. First is to set it up using the configuration below and run three separate Redis servers. The second is to set it up differently, tweak the settings in `kitsune/settings_local.py` accordingly, and run Redis using just the test configuration.

## Configuration

If you want to run three separate Redis servers, add this to `kitsune/settings_local.py`:

```
REDIS_BACKENDS = {
        'default': 'redis://localhost:6379?socket_timeout=0.5&db=0',
        'helpfulvotes': 'redis://localhost:6379?socket_timeout=0.\
            5&db=1',
    }

REDIS_BACKEND = REDIS_BACKENDS['default']
```

Otherwise adjust the above accordingly.

## Running redis

### Running redis

This script runs all three servers—one for each configuration.

I (Will) put that in a script that creates the needed directories in `/var/redis/` and kicks off the three redis servers:

```
#!/bin/bash

set -e

# Adjust these according to your setup!
REDISBIN=/usr/bin/redis-server
CONFFILE=/path/to/conf/files/

if test ! -e /var/redis/sumo/
then
    echo "creating /var/redis/sumo/"
    mkdir -p /var/redis/sumo/
fi

if test ! -e /var/redis/sumo-test/
then
    echo "creating /var/redis/sumo-test/"
    mkdir -p /var/redis/sumo-test/
fi

if test ! -e /var/redis/sumo-persistent/
then
    echo "creating /var/redis/sumo-persistent/"
    mkdir -p /var/redis/sumo-persistent/
fi

$REDISBIN $CONFFILE/redis-persistent.conf
$REDISBIN $CONFFILE/redis-test.conf
$REDISBIN $CONFFILE/redis-volatile.conf
```

# Running Kitsune with mod_wsgi

## Requirements

- See *the installation docs*.
- Apache HTTP server
- mod_rewrite
- mod_headers
- mod_expires
- mod_wsgi
- **Not** `mod_python`! It is incompatible with `mod_wsgi`.

## Overview

Setting up Kitsune to run as a WSGI application is fairly straightforward. You will need to install the requirements as described in *the installation chapter*.

There are 3 steps once Kitsune is installed:

- Set the document root.
- Set up aliases.
- Some file permissions.
- Set up WSGI itself.

### Apache Modules

Most of the Apache modules are part of a default Apache install, but may need to be activated. If they aren't installed, all of them, including `mod_wsgi` should be installable via your favorite package manager.

### WSGI Configuration

In the Apache config (or `<VirtualHost>`) you will need the following:

*Note that values may be slightly different.*

```
DocumentRoot /path/to/kitsune/webroot/

<Directory "/path/to/kitsune/webroot/">
    Options +FollowSymLinks
</Directory>

Alias /media/ "/path/to/kitsune/media/"
Alias /admin-media/ \
    "/path/to/virtualenv/lib/python<version>/site-packages/django/django/contrib/admin/media/"

WSGISocketPrefix /var/run/wsgi

WSGIDaemonProcess kitsune processes=8 threads=1 \
    maximum-requests=4000
WSGIProcessGroup kitsune

WSGIScriptAlias /k "/path/to/kitsune/wsgi/kitsune.wsgi"
```

**`WSGISocketPrefix`:** May or may not be necessary. It was for me.

**`WSGIDaemonProcess`:** `processes` should be set to the number of cores. `threads` should probably be left at 1. `maximum-requests` is good at between 4000 and 10000.

**`WSGIScriptAlias`:** Will make Kitsune accessible from `http://domain/k`, and we use rewrites in `webroot/.htaccess` to hide the `/k`. This will change soon, and the `.htaccess` file won't be necessary.

The `Alias` directives let Kitsune access its CSS, JS, and images through Apache, reducing the load on Django.

### Configuration

Most of our `kitsune/settings.py` is under version control, but can be overridden in a file called `kitsune/settings_local.py`. You can see example settings in the *Hacking HOWTO for Contributors*.

## File Permissions

To upload files, the webserver needs write access to `media/uploads` and all its subdirectories. The directories we currently use are:

```
media/uploads
media/uploads/avatars
media/uploads/images
media/uploads/images/thumbnails
media/uploads/gallery/images
media/uploads/gallery/images/thumbnails
media/uploads/gallery/videos
media/uploads/gallery/videos/thumbnails
```

`media/uploads` and its subdirectories should never be added to version control, as they are installation-/content-specific.

### Product Details JSON

Some people have issues with `django-mozilla-product-details` and file permissions. The management command `manage.py update_product_details` writes a number of JSON files to disk, and the webserver then needs to read them.

If you get file system errors from `product_details`, make sure the files are readable by the webserver (should be by default) and the directory is readable and executable.

By default, `product_details` stores the JSON files in:

```
path/to/virtualenv/lib/python<version>/site-packages/django-mozilla-product-details/product_details/
```

This is configurable. If you have multiple web servers, they should share this data. You can set the `PROD_DETAILS_DIR` variable in `kitsune/settings_local.py` to a different path, for example on NFS.

## Debugging

Debugging via WSGI is a little more interesting than via the dev server. One key difference is that you **cannot** use `pdb`. Writing to `stdout` is not allowed within the WSGI process, and will result in a Internal Server Error.

There are three relevant cases for debugging via WSGI (by which I mean, where to find stack traces):

### Apache Error Page

So you've got a really bad error and you aren't even seeing the Kitsune error page! This is usually caused by an uncaught exception during the WSGI application start-up. Our WSGI script, located in `wsgi/kitsune.wsgi`, tries to run all the initial validation that the dev server runs, to catch these errors early.

So where *is* the stack trace? You'll need to look in your Apache error logs. Where these are is OS-dependent, but a good place to look is `/var/log/httpd`. If you are using SSL, also check the SSL `VirtualHost`'s logs, for example `/var/log/httpd/ssl_error_log`.

#### With DEBUG=True

With `DEBUG = True` in your `kitsune/settings_local.py`, you will see a stack trace in the browser on error. Problem solved!

#### With DEBUG=False

With `DEBUG = False` in your `kitsune/settings_local.py`, you'll see our Server Error message. You can still get stack traces, though, by setting the `ADMINS` variable in `kitsune/settings_local.py`:

```
ADMINS = (
    ('me', 'my@email.address'),
)
```

Django will email you the stack trace. Provided you've set up *email*.

### Reloading WSGI

WSGI keeps Python and Kitsune running in an isolated process. That means code changes aren't automatically reflected on the server. In most default configurations of `mod_wsgi`, you can simply do this:

```
touch wsgi/kitsune.wsgi
```

That will cause the WSGI process to reload.

## Email from Kitsune

The default settings for Kitsune *do not send email*. However, outgoing email is printed to the the command line. If you want to get email, you should double check one thing first: are there any rows in the `notifications_eventwatch` table? If there are, you may be sending email to **real users**. The script in `scripts/anonymize.sql` will truncate this table. Simply run it against your Kitsune database:

```
mysql -u kitsune -p <YOUR_PASSWORD> < scripts/anonymize.sql
```

### Sending Email

So now you know you aren't emailing real users, but you'd still like to email yourself and test email in general. There are a few settings you'll need to use.

First, set the `EMAIL_BACKEND`. This document assumes you're using the SMTP mail backend.

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
```

If you have `sendmail` installed and working, that should do it. However, you might get caught in spam filters. An easy workaround for spam filters or not having sendmail working is to send email via a Gmail account.

```
EMAIL_USE_TLS = True
EMAIL_PORT = 587
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = '<your gmail address>@gmail.com'
EMAIL_HOST_PASSWORD = '<your gmail password>'
```

Yeah, you need to put your Gmail password in a plain text file on your computer. It's not for everyone. Be **very** careful copying and pasting settings from `kitsune/settings_local.py` if you go this route.

# Localization



Kitsune is localized with gettext. User-facing strings in the code or templates need to be marked for gettext localization.

We use Pontoon to provide an easy interface to localizing these files. Localizers are also free to download the PO files and use whatever tool they are comfortable with.

## Making Strings Localizable

Making strings in templates localizable is exceptionally easy. Making strings in Python localizable is a little more complicated. The short answer, though, is just wrap the string in `_()`.

### Interpolation

A string is often a combination of a fixed string and something changing, for example, `Welcome, James` is a combination of the fixed part `Welcome,`, and the changing part `James`. The naive solution is to localize the first part and the follow it with the name:

```
_('Welcome, ') + username
```

This is **wrong!**

In some locales, the word order may be different. Use Python string formatting to interpolate the changing part into the string:

```
_('Welcome, {name}').format(name=username)
```

Python gives you a lot of ways to interpolate strings. The best way is to use Py3k formatting and kwargs. That's the clearest for localizers.

The worst way is to use `%(label)s`, as localizers seem to have all manner of trouble with it. Options like `%s` and `{0}` are somewhere in the middle, and generally OK if it's clear from context what they will be.

### Localization Comments

Sometimes, it can help localizers to describe where a string comes from, particularly if it can be difficult to find in the interface, or is not very self-descriptive (e.g. very short strings). If you immediately precede the string with a comment that starts `L10n:`, the comment will be added to the PO file, and visible to localizers.

Example:

```
rev_data.append({
            'x': 1000 * int(time.mktime(rdate.timetuple())),
            # L10n: 'R' is the first letter of "Revision".
            'title': _('R', 'revision_heading'),
            'text': unicode(_('Revision %s')) % rev.created
            #'url': 'http://www.google.com/'  # Not supported yet
        })
```

### Adding Context with msgctxt

Strings may be the same in English, but different in other languages. English, for example, has no grammatical gender, and sometimes the noun and verb forms of a word are identical.

To make it possible to localize these correctly, we can add "context" (known in gettext as "msgctxt") to differentiate two otherwise identical strings.

For example, the string "Search" may be a noun or a verb in English. In a heading, it may be considered a noun, but on a button, it may be a verb. It's appropriate to add a context (like "button") to one of them.

Generally, we should only add context if we are sure the strings aren't used in the same way, or if localizers ask us to.

Example:

```
from tower import ugettext as _

...

foo = _('Search', context='text for the search button on the form')
```

### Plurals

"You have 1 new messages" grates on discerning ears. Fortunately, gettext gives us a way to fix that in English *and* other locales, the `ngettext` function:

```
ngettext('singular', 'plural', count)
```

A more realistic example might be:

```
ngettext('Found {count} result.',
         'Found {count} results',
         len(results)).format(count=len(results))
```

This method takes three arguments because English only needs three, i.e., zero is considered "plural" for English. Other locales may have different plural rules, and require different phrases for, say 0, 1, 2-3, 4-10, >10. That's absolutely fine, and gettext makes it possible.

### Strings in HTML Templates

When putting new text into a template, all you need to do is wrap it in a `_()` call:

```
<h1>{{ _('Heading') }}</h1>
```

Adding context is easy, too:

```
<h1>{{ _('Heading', 'context') }}</h1>
```

L10n comments need to be Jinja2 comments:

```
{# L10n: Describes this heading #}
<h1>{{ _('Heading') }}</h1>
```

Note that Jinja2 escapes all content output through `{{ }}` by default. To put HTML in a string, you'll need to add the `|safe` filter:

```
<h1>{{ _('Firefox <span>Help</span>')|safe }}</h1>
```

To interpolate, you should use one of two Jinja2 filters: `|f()` or, in some cases, `|fe()`. `|f()` has exactly the same arguments as `u''.format()`:

```
{{ _('Welcome, {name}!')|f(name=request.user.username) }}
```

The `|fe()` is exactly like the `|f()` filter, but escapes its arguments before interpolating, then returns a "safe" object. Use it when the localized string contains HTML:

```
{{ _('Found <strong>{0}</strong> results.')|fe(num_results) }}
```

Note that you *do not need* to use `|safe` with `|fe()`. Also note that while it may look similar, the following is *not* safe:

```
{{ _('Found <strong>{0}</strong> results.')|f(num_results)|safe }}
```

The `ngettext` function is also available:

```
{{ ngettext('Found {0} result.',
            'Found {0} results.',
            num_results)|f(num_results) }}
```

#### Using `{% trans %}` Blocks for Long Strings

When a string is very long, i.e. long enough to make Github scroll sideways, it should be line-broken and put in a `{% trans %}` block. `{% trans %}` blocks work like other block-level tags in Jinja2, except they cannot have other tags, except strings, inside them.

The only thing that should be inside a `{% trans %}` block is printing a string with `{{ string }}`. These are defined in the opening `{% trans %}` tag:

```
{% trans user=request.user.username %}
    Thanks for registering, {{ user }}! We're so...
    hope that you'll...
{% trans %}
```

You can also provide comments:

```
{# L10n: User is a username #}
{% trans user=request.user.username %}
    Thanks for registering, {{ user }}! We're so...
    hope that you'll...
{% trans %}
```

### Strings in Python

**Note:** Whenever you are adding a string in Python, ask yourself if it really needs to be there, or if it should be in the template. Keep logic and presentation separate!

Strings in Python are more complex for two reasons:

1. We need to make sure we're always using Unicode strings and the Unicode-friendly versions of the functions.
2. If you use the `ugettext` function in the wrong place, the string may end up in the wrong locale!

Here's how you might localize a string in a view:

```
from tower import ugettext as _

def my_view(request):
    if request.user.is_superuser:
        msg = _(u'Oh hi, staff!')
    else:
        msg = _(u'You are not staff!')
```

Interpolation is done through normal Python string formatting:

```
msg = _(u'Oh, hi, {user}').format(user=request.user.username)
```

`ugettext` supports context, too:

```
msg = _('Search', 'context')
```

L10n comments are normal one-line Python comments:

```
# L10n: A message to users.
msg = _(u'Oh, hi there!')
```

If you need to use plurals, import the function `ungettext` from Tower:

```
from tower import ungettext, ugettext as _

n = len(results)
msg = ungettext('Found {0} result', 'Found {0} results', n).format(n)
```

#### Lazily Translated Strings

You can use `ugettext` or `ungettext` only in views or functions called from views. If the function will be evaluated when the module is loaded, then the string may end up in English or the locale of the last request! (We're tracking down that issue.)

Examples include strings in module-level code, arguments to functions in class definitions, strings in functions called from outside the context of a view. To localize these strings, you need to use the `_lazy` versions of the above methods, `ugettext_lazy` and `ungettext_lazy`. The result doesn't get translated until it is evaluated as a string, for example by being output or passed to `unicode()`:

```
from tower import ugettext_lazy as _lazy

PAGE_TITLE = _lazy(u'Page Title')
```

`ugettext_lazy` also supports context.

It is very important to pass Unicode objects to the `_lazy` versions of these functions. Failure to do so results in significant issues when they are evaluated as strings.

If you need to work with a lazily-translated string, you'll first need to convert it to a `unicode` object:

```
from tower import ugettext_lazy as _lazy

WELCOME = _lazy(u'Welcome, %s')

def my_view(request):
    # Fails:
    WELCOME % request.user.username

    # Works:
    unicode(WELCOME) % request.user.username
```

### Strings in the Database

There is some user generated content that needs to be localizable. For example, karma titles can be created in the admin site and need to be localized when displayed to users. A django management command is used for this. The first step to making a model's field localizable is adding it to `DB_LOCALIZE` in `settings.py`:

```
DB_LOCALIZE = {
    'karma': {
        'Title': {
            'attrs': ['name'],
            'comments': ['This is a karma title.'],
        }
    },
    'appname': {
        'ModelName': {
            'attrs': ['field_name'],
            'comments': ['Optional comments for localizers.'],
        }
    }
}
```

Then, all you need to do is run the `extract_db` management command:

```
$ python manage.py extract_db
```

*Be sure to have a recent database from production when running the command.*

By default, this will write all the strings to *kitsune/sumo/db_strings.py* and they will get picked up during the normal string extraction (see below).

### Strings in Email Templates

Currently, email templates are text-based and not in HTML. Because of that you should use this style guide:

1. The entire email should be wrapped in autoescape. e.g.

```
{% autoescape false %}
{% trans %}
The entire email should be wrapped in autoescape.
{% endtrans %}


```

```
...
{% endautoescape %}
```

2. After an `{% endtrans %}`, you need two blank lines (three carriage returns). The first is eaten by the tag. The other two show up in the email. e.g.

```
{% trans %}
To confirm your subscription, stand up, put your hands on
your hips and do the hokey pokey.
{% endtrans %}


{{ _('Thanks!') }}
```

   Produces this:

```
To confirm your subscription, stand up, put your hands on
your hips and do the hokey pokey.

Thanks!
```

3. Putting in line breaks in a `trans` block doesn't have an effect since `trans` blocks get gettexted and whitespace is collapsed.

## Testing localized strings

When we add strings that need to be localized, it can take a couple of weeks for us to get translations of those localized strings. This makes it difficult to find localization issues.

Enter Dennis.

Run:

```
$ ./scripts/test_locales.sh
```

It'll extract all the strings, create a `.pot` file, then create a Pirate translation of all strings. The Pirate strings are available in the xx locale. After running the `test_locales.sh` script, you can access the xx locale with:

> http://localhost:8000/xx/

Strings in the Pirate translation have the following properties:

1. they are longer than the English string: helps us find layout and wrapping issues
2. they have at least one unicode character: helps us find unicode issues
3. they are easily discernable from the English versions: helps us find strings that aren't translated

---

**Note:** The xx locale is only available on your local machine. It is not available on -dev, -stage, or -prod.

---

## Linting localized strings

You can lint localized strings for warnings and errors:

```
$ ./manage.py lint locales/
```

You can see help text:

```
$ ./manage.py lint
```

## Getting the Localizations

Localizations are not stored in this repository, but are in a separate Git repo:

> https://github.com/mozilla-l10n/sumo-l10n

You don't need the localization files for general development. However, if you need them for something, they're pretty easy to get:

```
$ cd kitsune
$ git clone https://github.com/mozilla-l10n/sumo-l10n locale
```

## Updating the Localizations

When strings are added or updated, we need to update the templates and PO files for localizers. Updating strings is pretty easy. Check out the localizations as above, then:

```
$ python manage.py extract
$ python manage.py merge
```

Congratulations! You've now updated the POT and PO files.

Sometimes this can leave a bunch of garbage files with `.po~` extensions. You should delete these, never commit them:

```
$ find . -name "*.po~" -delete
```

### Adding a New Locale

Say you wanted to add `fa-IR`:

```
$ mkdir -p locale/fa-IR/LC_MESSAGES
$ python manage.py merge
```

Then add 'fa-IR' to SUMO_LANGUAGES in settings.py and make sure there is an entry in lib/languages.json (if not, add it).

And finally, add a migration with:

```
INSERT INTO `wiki_locale` (`locale`) VALUES ('fa-IR');
```

Done!

## Compiling MO Files

gettext is so fast for localization because it doesn't parse text files, it reads a binary format. You can easily compile that binary file from the PO files in the repository.

We don't store MO files in the repository because they need to change every time the corresponding PO file changes, so it's silly and not worth it. They are ignored by `.gitignore`, but please make sure you don't forcibly add them to the repository.

There is a shell script to compile the MO files for you:

```
$ ./locale/compile-mo.sh locale
```

Done!

### Reporting errors in .po files

We use Dennis to lint .po files for errors that cause HTTP 500 errors in production. Things like malformed variables, variables in the translated string that aren't in the original and that sort of thing.

When we do a deployment to production, we dump all the Dennis output into:

https://support.mozilla.org/media/postatus.txt

We need to check that periodically and report the errors.

If there are errors in those files, we need to open up a bug in **Mozilla Localizations** -> *locale code* with the specifics.

Product:

> Mozilla Localizations

Component:

> The locale code for the language in question

Bug summary:

> Use the error line

Bug description template:

```
We found errors in the translated strings for Mozilla Support
<https://support.mozilla.org/>. The errors are as follows:


<paste errors here>


Until these errors are fixed, we can't deploy updates to the
strings for this locale to production.

Mozilla Support strings can be fixed in the Support Mozilla project
in Pontoon <https://pontoon.mozilla.org/projects/sumo/>.

If you have any questions, let us know.
```

## Search

Kitsune uses Elasticsearch to power its on-site search facility.

It gives us a number of advantages over MySQL's full-text search or Google's site search.

- Much faster than MySQL.
  - And reduces load on MySQL.
- We have total control over what results look like.
- We can adjust searches with non-visible content.
- We don't rely on Google reindexing the site.

- We can fine-tune the algorithm and scoring.

## Installing Elasticsearch

There's an installation guide on the Elasticsearch site:

https://www.elastic.co/guide/en/elasticsearch/reference/1.3/setup-service.html

We're currently using 1.2.4 in production.

The directory you install Elasticsearch in will hereafter be referred to as `ELASTICDIR`.

You can configure Elasticsearch with the configuration file at `ELASTICDIR/config/elasticsearch.yml`.

Elasticsearch uses several settings in `kitsune/settings.py` that you need to override in `kitsune/settings_local.py`. Here's an example:

```
# Connection information for Elastic
ES_URLS = ['http://127.0.0.1:9200']
ES_INDEXES = {'default': 'sumo_dev'}
ES_WRITE_INDEXES = ES_INDEXES
```

These settings explained:

`ES_URLS`

Defaults to `['http://127.0.0.1:9200']`.

Points to the url for your Elasticsearch instance.

> **Warning:** The url must match the host and port in `ELASTICDIR/config/elasticsearch.yml`. So if you change it in one place, you must also change it in the other.

`ES_INDEXES`

Mapping of `'default'` to the name of the index used for searching.

The index name must be prefixed with the value of `ES_INDEX_PREFIX`.

Examples if `ES_INDEX_PREFIX` is set to `'sumo'`:

```
ES_INDEXES = {'default': 'sumo'}
ES_INDEXES = {'default': 'sumo_20120213'}

ES_INDEXES = {'default': 'tofurkey'}  # WRONG!
```

`ES_WRITE_INDEXES`

Mapping of `'default'` to the name of the index used for indexing.

The index name must be prefixed with the value of `ES_INDEX_PREFIX`.

Examples if `ES_INDEX_PREFIX` is set to `'sumo'`:

```
ES_WRITE_INDEXES = ES_INDEXES
ES_WRITE_INDEXES = {'default': 'sumo'}
ES_WRITE_INDEXES = {'default': 'sumo_20120213'}

ES_WRITE_INDEXES = {'default': 'tofurkey'}  # WRONG!
```

**Note:** The separate roles for indexes allows us to push mapping changes to production. In the first push, we'll push the mapping change and give `ES_WRITE_INDEXES` a different value. Then we reindex into the new index. Then we push a change updating `ES_INDEXES` to equal `ES_WRITE_INDEXES` allowing the search code to use the new index.

If you're a developer, the best thing to do is have your `ES_WRITE_INDEXES` be the same as `ES_INDEXES`. That way you can reindex and search and you don't have to fiddle with settings in between.

There are a few other settings you can set in your `kitsune/settings_local.py` file that override ElasticUtils defaults. See the ElasticUtils docs for details.

Other things you can change:

`ES_INDEX_PREFIX`

Defaults to `'sumo'`.

All indexes for this application must start with the index prefix. Indexes that don't start with the index prefix won't show up in index listings and cannot be deleted through the esdelete subcommand and the search admin.

**Note:** The index names in both `ES_INDEXES` and `ES_WRITE_INDEXES` **must** start with this prefix.

`ES_LIVE_INDEXING`

Defaults to False.

You can also set `ES_LIVE_INDEXING` in your `kitsune/settings_local.py` file. This affects whether Kitsune does Elasticsearch indexing when data changes in the `post_save` and `pre_delete` hooks.

For tests, `ES_LIVE_INDEXING` is set to `False` except for Elasticsearch specific tests so we're not spending a ton of time indexing things we're not using.

`ES_TIMEOUT`

Defaults to 5.

This affects timeouts for search-related requests.

If you're having problems with ES being slow, raising this number might be helpful.

## Using Elasticsearch

### Running

Start Elasticsearch by:

```
$ ELASTICDIR/bin/elasticsearch
```

That launches Elasticsearch in the background.

### Indexing

Do a complete reindexing of everything by:

```
$ ./manage.py esreindex
```

This will delete the existing index specified by `ES_WRITE_INDEXES`, create a new one, and reindex everything in your database. On my machine it takes under an hour.

If you need to get stuff done and don't want to wait for a full indexing, you can index a percentage of things.

For example, this indexes 10% of your data ordered by id:

```
$ ./manage.py esreindex --percent 10
```

This indexes 50% of your data ordered by id:

```
$ ./manage.py esreindex --percent 50
```

I use this when I'm fiddling with mappings and the indexing code.

You can also specify which models to index:

```
$ ./manage.py esreindex --models questions_question,wiki_document
```

See `--help` for more details:

```
$ ./manage.py esreindex --help
```

---

**Note:** Once you've indexed everything, if you have `ES_LIVE_INDEXING` set to `True`, you won't have to do it again unless indexing code changes. The models have `post_save` and `pre_delete` hooks that will update the index as the data changes.

---

---

**Note:** If you kick off indexing with the admin, then indexing gets done in chunks by celery tasks. If you need to halt indexing, you can purge the tasks with:

```
$ ./manage.py celeryctl purge
```

If you do this often, it helps to write a shell script for it.

---

### Health/statistics

You can see Elasticsearch index status with:

```
$ ./manage.py esstatus
```

This lists the indexes, tells you which ones are set to read and write, and tells you how many documents are in the indexes by mapping type.

### Deleting indexes

You can use the search admin to delete the index.

On the command line, you can do:

```
$ ./manage.py esdelete <index-name>
```

### Implementation details

Kitsune uses elasticutils and pyelasticsearch.

Most of our code is in the `search` app in `kitsune/search/`.

Models in Kitsune that are indexable use `SearchMixin` defined in `models.py`.

Utility functions are implemented in `es_utils.py`.

Sub commands for `manage.py` are implemented in `management/commands/`.

## Searching on the site

### Scoring

These are the default weights that apply to all searches:

wiki (aka kb):

```
document_title__match           6
document_content__match         1
document_keywords__match        8
document_summary__match         2
```

questions (aka support forums):

```
question_title__match           4
question_content__match         3
question_answer_content__match  3
```

forums (aka contributor forums):

```
post_title__match               2
post_content__match             1
```

Elasticsearch is built on top of Lucene so the Lucene documentation on scoring covers how a document is scored in regards to the search query and its contents. The weights modify that—they're query-level boosts.

Additionally, this blog post from 2006 is really helpful in terms of providing insight on the implications of the way things are scored.

### Filters

We use a series of filters on document_tag, question_tag, and other properties of documents like *has_helpful*, *is_locked*, *is_archived*, etc.

In ElasticSearch, filters remove items from the result set, but don't affect the scoring.

We cannot apply weights to filtered fields.

### Regular search

A *regular* search is any search that doesn't start from the *Advanced Search* form.

You could start a *regular* search from the front page or from the search form on any article page.

Regular search does the following:

1. searches only kb and support forums
2. (filter) kb articles are tagged with the product (e.g. “desktop”)
3. (filter) kb articles must not be archived
4. (filter) kb articles must be in Troubleshooting (10) and How-to (20) categories
5. (filter) support forum posts tagged with the product (e.g. “desktop”)
6. (filter) support forum posts must have an answer marked as helpful
7. (filter) support forum posts must not be archived

It scores as specified above.

#### Ask A Question search

An *Ask a question* or *AAQ* search is any search that is performed within the AAQ workflow. The only difference to *regular* search is that *AAQ* search shows forum posts that have no answer marked as helpful.

#### Advanced search

The *advanced* search is any search that starts from the *Advanced Search* form.

The advanced search is defined by whatever you specify in the *Advanced Search* form.

For example, if you search for knowledge base articles in the Troubleshooting category, then we add a filter where the result has to be in the Troubleshooting category.

#### Link to the Elasticsearch code

Here’s a link to the search view in the master branch:

https://github.com/mozilla/kitsune/blob/master/kitsune/search/views.py

# Frontend Infrastructure

Frontends assets for Kitsune are managed through Django Pipeline.

## Bundles

To reduce the number of requests per page and increase overall performance, JS and CSS resources are grouped into *bundles*. These are defined in `kitsune/bundles.py`, and are loaded into pages with template tags. Each bundle provides a list of files which will be compiled (if necessary), minified, and concatenated into the final bundle product.

In development, the minification and concatenation steps are skipped. In production, each file is renamed to contain a hash of it’s contents in the name, and files are rewritten to account for the changed names. This is called cache busting, and allows the CDN to be more aggressive in caching these resources, and for clients to get updates faster when we make changes.

## Style Sheets

The styles written for Kitsune is written in Less. Libraries, of course, have styles written in CSS. These are combined into bundles and shipped as minified CSS.

Less files are recognized by an extension of `.less`.

## Javascript

There are a few kinds of Javascript in use in Kitsune.

### Plain JS

Plain JS is not suspect to any compilation step, and is only minified and concatenated. Plain JS files should be written to conform to ES3 standards, for compatibility.

Plain JS files have an extension of `.js`.

### ES6 JavaScript

EcmaScript 6 is the next version JavaScript that has been recently standardized. Because it is very new, it does not have wide spread browser support yet, and so it is compiled using Babel to ES5. Because it is compiled to ES5, and not ES3, it is not suitable for use in user facing parts of the site, which require maximum compatibility.

These files are recognized by the ES6 compiler by an extension of `.es6`, and *should* end in `.js.es6` for clarity. However, see the note about Browserify below.

For more information about ES6 syntax and features, see lukehoban/es6features.

### JSX

JSX is a syntax extension on top of ES6 (and in some places, ES7) which adds support for an XML-like trees. It is used in Kitsune as a way to specify DOM elements in React Component render methods. JSX is compiled using Babel as well, and in fact all ES6 files may contain JSX syntax, since Babel compiles it by default.

These files don't have a specific individual extension, but use the `.es6` extension. For clarity, standalone jsx files should use the extension of `.jsx.es6`.

### Browserify

Files with the extension `.browserify.js` are treated as Browserify entry points. They may include other JS files using ES6 modules syntax. The files included in this way may also make use of the ES6 module system, regardless of their extension.

All files loaded this way are treated as ES6+JSX files. This is generally the only way ES6 and JSX code should be included in a bundle, and so in practice the extensions assigned to those files don't matter to Django Pipeline, and should be named to be clear to the reader.

Browserify has been configured with the babelify and bowerify transformers, to be able to load ES6 files and files from Bower.

### Bower

Frontend dependencies are downloaded using Bower. In a bundle file, they are listed as `package-name/path/in/package.js`. Django Pipeline will find the correct Bower package to pull files from.

Bower is not normally compatible with Browserify. A Browserify transformer called bowerify makes an include request for Bower packages load the primary entry point of the Bower package to make them compatible.

## Army of Awesome

### Setting up the Army of Awesome Twitter Application

#### Create the Twitter application

Go to https://dev.twitter.com/apps/new and fill in the required fields. Also be sure to fill in a Callback URL, it can be any dummy URL as we override this value by passing in an oauth_callback URL. Set up the application for read-write access.

You will need to enter the consumer key and consumer secret in settings_local.py below.

#### Update settings_local.py

Set the following settings in `kitsune/settings_local.py`:

```
TWITTER_CONSUMER_KEY = <consumer key>
TWITTER_CONSUMER_SECRET = <consumer secret>
TWITTER_COOKIE_SECURE = False
```

### Fetch tweets

To fetch tweets, run:

```
$ ./manage.py cron collect_tweets
```

You should now see tweets at /army-of-awesome.

### Common replies

Common replies should be defined in a wiki article with slug 'army-of-awesome-common-replies'. The format for the article content is:

```
=Category 1=

==Reply 1==
Reply goes here http://example.com/kb-article

==Reply 2==
Another reply here

=Category 2=
```

```
==Reply 3==
And another reply
```

Note that replies can't be over 140 characters long, including the @username and #fxhelp hash tag. Therefore, each reply defined here should not go over 125 characters. The reply must be plain text without any HTML or wiki syntax.

## Karma System

TODO: Fill this out

## Important Wiki Documents

There are a number of "important" wiki documents, i.e. documents with well-known names that are used in parts of the interface besides the normal knowledge base.

These documents make it possible to update content on, for example, the home pages without requiring code changes.

In all cases, the title of the English document matters, but the slug, or title of the localized versions, does not.

For a list of the documents, check out the Managing Landing Page Content article on SUMO.

## Other Notes

### Questions

#### Troubleshooter Add-on

When asking a question, users are prompted to install an addon that will return extra information to SUMO from about:support. This is opt-in, and considered generally non-sensitive data, but is not revelaed to all sites because of fingerprinting concerns.

This add-on only provides data to white listed domains. The built in whitelist is:

- https://support.mozilla.org/
- https://support.allizom.org/
- https://support-dev.allizom.org/
- http://localhost:8000/

Note that the protocol and port are significant. If you try and run the site on http://localhost:8900/, the add-on will not provide any data.

The source of the addon is on GitHub, and it is hosted on AMO. The add-on is hosted on AMO instead of SUMO so that AMO will do the heavy lifting of providing automatic updates.

#### about:support API

The about:support API replaces the Troubleshooter Add-on and is available starting in Firefox 35. To test this locally during development, you need to run an ssl server and change some permissions in your browser.

To run the ssl server, first add sslserver to INSTALLED_APPS in settings_local.py:

```
INSTALLED_APPS = list(INSTALLED_APPS) + ['sslserver']
```

Run the ssl server:

```
$ ./manage.py runsslserver
```

Then you need to run the following in the Browser Console:

> Services.perms.add(Services.io.newURI("https://localhost:8000", null, null), "remote-troubleshooting", Services.perms.ALLOW_ACTION);

---

**Note:** You need to Enable chrome debugging in developer console settings, so that you can access the browser console.

See also https://developer.mozilla.org/en-US/docs/Tools/Browser_Console

---

## Licenses

CHAPTER 3

# Part 3: SUMO

# API

SUMO has a series of API endpoints to access data.

**Contents**



## Search suggest API

**Endpoint** `/api/2/search/suggest/`

**Method** `GET`

**Content type** `application/json`

**Response** `application/json`

The search suggest API allows you to get back kb documents and aaq questions that match specified arguments.

Arguments can be specified in the url querystring or in the HTTP request body.

### Required arguments

| argument | type | notes |
|---|---|---|
| q | string | This is the text you're querying for. |

### Optional arguments

| argument | type | notes |
|---|---|---|
| locale | string | default: `settings.WIKI_DEFAULT_LANGUAGE`<br>The locale code to restrict results to.<br>Examples:<br>• `en-US`<br>• `fr` |
| product | string | default: None<br>The product to restrict results to.<br>Example:<br>• `firefox` |
| max_documents | integer | default: 10<br>The maximum number of documents you want back. |
| max_questions | integer | default: 10<br>The maximum number of questions you want back. |

### Responses

All response bodies are in JSON.

#### HTTP 200: Success

With an HTTP 200, you'll get back a set of results in JSON.

```
{
    "documents": [
        {
            "id": ...                   # id of kb article
            "title": ...                # title of kb article
            "url": ...                  # url of kb article
            "slug": ...                 # slug of kb article
            "locale": ...               # locale of the article
            "products": ...             # list of products for the article
            "topics": ...               # list of topics for the article
            "summary": ...              # paragraph summary of kb article (plaintext)
            "html": ...                 # html of the article
        }
        ...
    ],
    "questions": [
        {
            "id": ...                   # integer id of the question
            "answers": ...              # list of answer ids
            "content": ...              # content of question (in html)
            "created": ...              # datetime stamp in iso-8601 format
            "creator": ...              # JSON object describing the creator
            "involved": ...             # list of JSON objects describing everyone who
                                          participated in the question
```

```
            "is_archived": ...          # boolean for whether this question is archived
            "is_locked": ...            # boolean for whether this question is locked
            "is_solved": ...            # boolean for whether this question is solved
            "is_spam": ...              # boolean for whether this question is spam
            "is_taken": ...             # FIXME:
            "last_answer": ...          # id for the last answer
            "num_answers": ...          # total number of answers
            "locale": ...               # the locale for the question
            "metadata": ...             # metadata collected for the question
            "tags": ...                 # tags for the question
            "num_votes_past_week": ... # the number of votes in the last week
            "num_votes": ...            # the total number of votes
            "product": ...              # the product
            "solution": ...             # id of answer marked as a solution if any
            "taken_until": ...          # FIXME:
            "taken_by": ...             # FIXME:
            "title": ...                # title of the question
            "topic": ...                # FIXME:
            "updated_by": ...           # FIXME:
            "updated": ...              # FIXME:
        },
        ...
    ]
}
```

### Examples

Using curl:

```
curl -X GET "http://localhost:8000/api/2/search/suggest/?q=videos"

curl -X GET "http://localhost:8000/api/2/search/suggest/?q=videos&max_documents=3&max_questions=3"

curl -X GET "http://localhost:8000/api/2/search/suggest/" \
     -H "Content-Type: application/json" \
     -d '
{
   "q": "videos",
   "max_documents": 3,
   "max_questions": 0
}'
```

## Locales API

**GET /api/2/locales/**

All locales supported by SUMO.

**Example request**:

```
GET /api/2/locales/ HTTP/1.1
Accept: application/json
```

**Example response**:

```
HTTP/1.0 200 OK
Vary: Accept, X-Mobile, User-Agent
```

```
Allow: OPTIONS, GET
X-Frame-Options: DENY
Content-Type: application/json

{
   "vi": {
      "name": "Vietnamese",
      "localized_name": "Ti\u1ebfng Vi\u1ec7t",
      "aaq_enabled": false
   },
   "el": {
      "name": "Greek",
      "localized_name": "\u0395\u03bb\u03bb\u03b7\u03bd\u03b9\u03ba\u03ac",
      "aaq_enabled": false
   },
   "en-US": {
      "name": "English",
      "localized_name": "English",
      "aaq_enabled": true
   },
   ...
}
```

**Request Headers**

- Accept – application/json

**Response Headers**

- Content-Type – application/json

**Status Codes**

- 200 OK – no error

# IRC Bots

Kitsune developers hang out in the `#sumodev` channel of Mozilla's IRC network. We use a number of IRC bots to help disperse information via this channel. Here are the important ones.

### firebot

`firebot` (sometimes `firewolfbot`) is a common fixture of Mozilla IRC channels. He spits out information about bugs in Bugzilla. Just include the word `bug` followed by a space and a number (e.g. `bug 12345`, `bug 624323`) and `firebot` will post some info about the bug and a link to Bugzilla in the channel.

`firebot` is also an InfoBot and does a lot of generic MozBot-type stuff.

### travis-ci

`travis-ci` is the IRC notifier from Travis. He tells us about the status of our CI builds.

#### kitsunebot

kitsunebot is a jig bot that acts as a bridge between Github and Jenkins. (Yes, Github can trigger Jenkins builds itself. No, it doesn't do it very well.) So far, she is very quiet, but we'd like to make her more useful.

kitsunebot filters commits by branch and triggers the correct Jenkins build. She also announces changes in IRC, but only shows the full list of changes committed for certain branches.

#### qatestbot

qatestbot tells us when the QA test suite fails and also allows us to kick off QA test suite runs.

See *tests-chapter-qa-test-suite* for details.

## Kitsune Deployments

This documents the current development (dev), staging, production (prod), and Continuous Integration (CI) servers for the support.mozilla.com instance of Kitsune.

This document will not detail *how* Kitsune is deployed, only why, when, and where.

### The Source

All of the source code for Kitsune lives in a single Github repo.

### Branches

#### Master

The master branch is our main integration points. All new patches should be based on the latest master (or rebased to it).

Pull requests are created from those branches. Pull requests may be opened at any time, including before any code has been written.

Pull requests get reviewed.

Once reviewed, the branch is merged into master, except in special cases such as changes that require re-indexing. See *Changes that involve reindexing*.

We deploy to production from master.

### Dev

Dev is a small environment that is updated manually. We use it primarily to develop infrastructure changes, like upgrading to Python 2.7.

### Stage

We deploy to stage anything we want to test including deployments themselves. We deploy using the big red button. Typically we deploy to stage from master, but we can deploy from any rev-ish thing.

### Production

We deploy to production from master by specified revisions. We deploy when things are ready to go using the big red button.

## Standard Operating Procedure

### Site is broken because of bad locale msgstr

Occasionally, the localized strings cause the site to break.

For example, this causes the site to break:

```
#: kitsune/questions/templates/questions/includes/answer.html:19
msgid "{num} answers"
msgstr "{0} antwoorden"
```

In this example, the *{0}* is wrong.

How to fix it:

- If Kadir is around:
  1. Tell Kadir. He can fix it in Pontoon.
  2. Once it's fixed, push to production. This will pick up the new strings.
- If someone with commit access to the locales Git repo is around:
  1. Tell that person. She/he can fix it in Git.
  2. Once it's fixed, push to production. This will pick up the new strings.

## Service Level Agreement

This isn't a zero-tolerance policy, but a series of policy points we work towards when making changes to the site.

Measurements are based on what we can see in graphite which means it's all server-side. Also, we use the upper_90 line since that tracks the more extreme side of performance.

This SLA will probably change over time. Here it is now.

1. View performance

   upper_90 for server-side rendering of views for GET requests should be under 800ms.

2. Search availability

   Search should work and return useful results.

   The implication here is that it's not ok to be reindexing into an index that searches are against.

3. Browser support

   See this page in the wiki:

   https://wiki.mozilla.org/Support/Browser_Support

CHAPTER 4

# Part 4: Administration

## Products and Topics

This document explains what Products and Topics are in Kitsune and how to use them.

### Adding Products

#### Visibility

#### Display Order

### Adding Topics

#### Visibility

#### Display Order

## Groups

This document explains what kinds of groups exist in Kitsune, how to add them and how to manage permissions.

### Group Types

#### Locale

#### Contributor Group

### Adding Groups

### Managing Permissions

## Users

This document explains how to manage permissions for users, user roles and removing users.

### User Roles

### Managing Permissions

### Removing Users

## Questions

This document explains what kinds of question states exist in Kitsune, how they are set and their implications.

### Configuring new products

To configure a new product for AAQ you must edit `config.py` within the questions app.

First, ensure the `Product` object exists for this product in the products app.If not create a new `Product`.

Next, Add a new item to the `products` dictionary using something like:

```
('product-slug', {
    'name': _lazy(u'Product Name'),
    'subtitle': _lazy('A brief description'),
    'extra_fields': [],
    'tags': ['tag-slug'],
    'product': 'product-slug',
    'categories': SortedDict([
        ('topic-slug', {
            'name': _lazy(u'Topic name'),
            'topic': 'topic-slug',
            'tags': []
        }),
    ])
}),
```

`'product-slug'` should be the slug of the `Product` object for this product.

You will also need to add a new 96x96 icon to the `img/logos.large.sprite.png` and will need to update the `.logo-sprite` class in `questions.less` to account for the new logo.

The same changes will need to be made to `img/mobile/logos-sprite-2x.png` and a non-retina (half size) version `img/mobile/logos-sprite.png`. Finally update the `.logo-sprite` class in `mobile/main.less`.

### Question States

#### Not yet posted

Questions are created within the *Ask a question* workflow, but they are not shown in question listings until the user is confirmed. With Persona for authentication this status is impossible, since email addresses are confirmed right away.

#### Default

This is the unmarked state of the thread.

Implications:

- Users can reply
- Visible in regular SUMO searches (with at least one helpful reply)
- Visible to external searches
- Visible in the regular questions list
- Visible in the *related threads* section

#### Locked

This is the locked state of a thread. A thread can be locked in two ways:

- By manually locking it via the question UI
- Automatically after 180 days.

Implications:

- Users can't reply
- Moderators can unlock to reply
- If there is an answer, the locked thread is still shown by search engines and our internal search.
- If there is no answer, the locked thread will not be shown by search engines and our internal search.

#### Not indexed

Questions with no answers that are older than 30 days have a meta tag telling search engines not to show them.

## Forums

This document explains what kinds of forums exist in Kitsune, how to add them and how to manage permissions.

### Forum Types

### Adding Forums

### Managing Permissions

## Badges

Kitsune is an Issuer of Open Badges. This document explains how to setup and administrate badges in Kitsune.

### Creating a New Badge

### Badge Image Requirements

There are two requirements for badge images:

- Needs to be a square

- The maximum size is 256kb

Badge images will be automatically resized to a 256x256 PNG image on upload. We recommend creating and uploading the badge image in these dimensions and format. If possible, the image should have a transparent background so it blends in everywhere we display it.

## Awarding Badges Manually

## Automatic Badges

CHAPTER 5

# Indices and tables

- *genindex*
- *modindex*

## /api

```
GET /api/2/locales/,47
```