1 Development

2 Getting started
   2.1 Installation via Docker .................................................... 5
   2.2 Development .................................................................. 9
   2.3 Troubleshooting .............................................................. 17
   2.4 The Kuma test suite ......................................................... 19
   2.5 Client-side testing ........................................................... 22
   2.6 Performance testing .......................................................... 25
   2.7 Feature toggles ................................................................. 25
   2.8 Celery and async tasks ...................................................... 27
   2.9 Search .......................................................................... 29
   2.10 Localization .................................................................. 30
   2.11 CKEditor ...................................................................... 32
   2.12 Documentation ............................................................... 33
   2.13 Installation via Vagrant .................................................... 35
Kuma is the platform that powers MDN (developer.mozilla.org)
Development

**Code**  https://github.com/mozilla/kuma

**Issues**  http://mzl.la/mdn_backlog (Product)

  Pull Request Queues

**Dev Docs**  https://kuma.readthedocs.io/en/latest/installation.html

**CI Server**  https://travis-ci.org/mozilla/kuma

**Mailing list**  https://lists.mozilla.org/listinfo/dev-mdn

**IRC**  irc://irc.mozilla.org/mdndev

  http://logs.glob.uno/?c=mozilla%23mdndev (logs)

**Servers**  What’s Deployed on MDN?

  https://developer.allizom.org/ (stage)

  https://developer.mozilla.org/ (prod)
Want to help make MDN great? Our contribution guide lists some good first projects and offers direction on submitting code.

Contents:

## Installation via Docker

Starting in 2016, we support using Docker for local development, and we are transitioning to Docker for integration testing and deployment as well.

### Current Status:

- Kuma developers are using Docker for daily development and maintenance tasks. Staff developers primarily use Docker for Mac. Other staff members and contributors use Docker’s Ubuntu packages.
- When the master branch is updated, the kuma_base image is refreshed and published to quay.io. This image contains system packages and third-party libraries.
- The Docker development environment is evolving rapidly, to address issues found during development and to move toward a containerized design. You may need to regularly reset your environment to get the current changes.
- The Docker environment doesn’t yet support everything from the Vagrant environment, such as local SSL development and automatic asset compiling.
- We are documenting tips and tricks on the Troubleshooting page.

### Docker setup

1. Install the Docker platform, following Docker’s instructions for your operating system, such as Docker for Mac for MacOS, or for your Linux distribution. Linux users will also want to install Docker Compose and follow post-install instructions to confirm that the development user can run Docker commands.

   To confirm that Docker is installed correctly, run:

   ```bash
docker run hello-world
   ```

2. Clone the kuma Git repository, if you haven’t already:

   ```bash
git clone --recursive git@github.com:mozilla/kuma.git
   ```

3. Ensure you are in the existing or newly cloned kuma working copy:
cd kuma

4. Initialize and customize .env. Linux users should set the UID parameter in .env, to avoid issues when mixing docker-compose and docker commands:

   cp .env-dist.dev .env
   vim .env  # Or your favorite editor

5. Pull the Docker images and build the containers:

   docker-compose pull
   docker-compose build

6. Start the containers in the background:

   docker-compose up -d

The following instructions assume that you are running from a folder named kuma, so that the containers created are named kuma_web_1, kuma_worker_1, etc. If you run from another folder, like mdn, the containers will be named mdn_web_1, mdn_worker_1, etc. Adjust the command accordingly, or force the kuma name with:

   docker-compose up -d -p kuma

Load the Sample Database

Download and load the sample database:

   wget -N https://mdn-downloads.s3-us-west-2.amazonaws.com/mdn_sample_db.sql.gz
   docker exec -i kuma_web_1 bash -c "zcat | ./manage.py dbshell" < mdn_sample_db.sql.gz

It takes a few seconds to load, with this expected output:

   mysql: [Warning] Using a password on the command line interface can be insecure.

This command can be adjusted to restore from an uncompressed database, or directly from a mysqldump command.

Compile locales

Localized string databases are included in their source form, and need to be compiled to their binary form:

   docker exec -i kuma_web_1 make localecompile

Dozens of lines of warnings will be printed:

   cd locale; ./compile-mo.sh . ; cd --
   ./af/LC_MESSAGES/django.po:2: warning: header field 'PO-Revision-Date' still has the initial default value
   ./af/LC_MESSAGES/django.po:2: warning: header field 'Last-Translator' still has the initial default value
   ...
   ./zu/LC_MESSAGES/promote-mdn.po:4: warning: header field 'PO-Revision-Date' still has the initial default value
   ./zu/LC_MESSAGES/promote-mdn.po:4: warning: header field 'Last-Translator' still has the initial default value

Warnings are OK, and will be fixed as translators update the strings on Pontoon. If there is an error, the output will end with the error, such as:

   ./az/LC_MESSAGES/django.po:263: 'msgid' and 'msgstr' entries do not both end with '\n'
   msgfmt: found 1 fatal error

Chapter 2. Getting started
These need to be fixed by a Kuma developer. Notify them in the #mdndev IRC channel or open a bug. You can continue with installation, but non-English locales will not be localized.

**Generate static assets**

Static assets such as CSS and JS are included in source form, and need to be compiled to their final form:

```bash
docker exec -i kuma_web_1 make build-static
```

A few thousand lines will be printed, like:

```bash
## Generating JavaScript translation catalogs ##
processing language en_US
processing language af
processing language ar
...
## Compiling (Sass), collecting, and building static files ##
Copying '/app/build/locale/jsi18n/af/javascript.js'
Copying '/app/build/locale/jsi18n/ar/javascript.js'
Copying '/app/build/locale/jsi18n/az/javascript.js'
...
Post-processed 'build/styles/wiki.css' as 'build/styles/wiki.css'
Post-processed 'build/styles/error-404.css' as 'build/styles/error-404.css'
Post-processed 'build/styles/mdn.css' as 'build/styles/mdn.css'
...
1687 static files copied to '/app/static', 1773 post-processed
```

**Visit the Homepage**


**Prepare for Front-end Development**

When doing front-end development on your local machine, you’ll probably want to run **gulp**, to rebuild front-end assets as they edited, rather than running `make build-static` after each change.

First, install Node.js v6, using the [install instructions](#) for your OS.

Next, from the root directory of your Kuma repository, install **gulp** and dependencies:

```bash
npm install
```

Now, you can run **gulp** (probably from its own shell):

```bash
node_modules/.bin/gulp
```

Alternatively, you can install **gulp** globally:

```bash
sudo npm install -g
```

And then run **gulp** more simply:

```bash
gulp
```

### 2.1. Installation via Docker
Create an admin user

Many Kuma settings require access to the Django admin, including configuring social login. It is useful to create an admin account with password access for local development.

If you want to create a new admin account, use `createsuperuser`:

```
docker exec -it kuma_web_1 ./manage.py createsuperuser
```

This will prompt you for a username, email address (a fake address like `admin@example.com` will work), and a password.

If your database has an existing account that you want to use, run the management command. Replace `YOUR_USERNAME` with your username and `YOUR_PASSWORD` with your password:

```
docker-compose run --rm web ./manage.py ihavepower YOUR_USERNAME \  
--password YOUR_PASSWORD
```

With a password-enabled admin account, you can log into Django admin at [http://localhost:8000/admin/login/](http://localhost:8000/admin/login/)

Enable GitHub Auth (optional)

To enable GitHub authentication, you’ll need to register an OAuth application on GitHub, with settings like:

- **Application name**: MDN Development for (<username>).
- **Application description**: My own GitHub app for MDN!

As an admin user, add a django-allauth social app for GitHub:

- **Provider**: GitHub.
- **Name**: MDN Development.
- **Client id**: `<your GitHub App Client ID>`.
- **Secret key**: `<your GitHub App Client Secret>`.
- **Sites**: Move `example.com` from “Available sites” to “Chosen sites”.

Now you can sign in with GitHub.

To associate your password-only admin account with GitHub:

3. Click your username at the top to view your profile.
4. Click Edit to edit your profile.
5. Under My Profiles, click Use your GitHub account to sign in.

To create a new account with GitHub, use the regular “Sign in” widget at the top of any page.

With social accounts are enabled, you can disable the admin password in the Django shell:
Interact with the Docker containers

The current directory is mounted as the /app folder in the web and worker containers (kuma_web_1 and kuma_worker_1). Changes made to your local directory are usually reflected in the running containers. To force the issue, the container can be restarted:

docker restart kuma_web_1 kuma_worker_1

You can connect to a running container to run commands. For example, you can open an interactive shell in the web container:

docker exec -it kuma_web_1 /bin/bash
make bash # Same command, less typing

To view the logs generated by a container:

docker logs kuma_web_1

To continuously view logs from all containers:

docker-compose logs -f

To stop the containers:

docker-compose stop

For further information, see the Docker documentation, such as the Docker Overview and the documentation for your operating system. You can try Docker’s guided tutorials, and apply what you’ve learned on the Kuma Docker environment.

Development

Pick an environment

There are two development environments, and you need to install at least one of them first.

- The Docker containerized environment is the preferred development environment. The Docker images are already provisioned, so setup is faster. It is a better model for the production environment, and after the planned rehost will be almost exactly the same. It does not yet have all of the features of the Vagrant environment, and it is currently slower for many development tasks.
- The Vagrant-managed VM is the mature but deprecated development environment. It can be tricky to provision. It is often a poor model for the production environment, and can not be used to test infrastructure changes.

Docker and Vagrant can be used at the same time on the same “host machine” (your laptop or desktop computer).
Basic Docker usage

Edit files as usual on your host machine; the current directory is mounted via Docker host mounting at /app within the kuma_web_1 and other containers. Useful docker sub-commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>docker exec -it kuma_web_1 bash</code></td>
<td># Start an interactive shell</td>
</tr>
<tr>
<td><code>docker logs kuma_web_1</code></td>
<td># View logs from the web container</td>
</tr>
<tr>
<td><code>docker-compose logs -f</code></td>
<td># Continuously view logs from all containers</td>
</tr>
<tr>
<td><code>docker restart kuma_web_1</code></td>
<td># Force a container to reload</td>
</tr>
<tr>
<td><code>docker-compose stop</code></td>
<td># Shutdown the containers</td>
</tr>
<tr>
<td><code>docker-compose up -d</code></td>
<td># Start the containers</td>
</tr>
<tr>
<td><code>docker-compose rm</code></td>
<td># Destroy the containers</td>
</tr>
</tbody>
</table>

There are `make` shortcuts on the host for frequent commands, such as:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>make up</code></td>
<td># docker-compose up -d</td>
</tr>
<tr>
<td><code>make bash</code></td>
<td># docker exec -it kuma_web_1 bash</td>
</tr>
<tr>
<td><code>make shell_plus</code></td>
<td># docker exec -it kuma_web_1 ./manage.py shell_plus</td>
</tr>
</tbody>
</table>

Run all commands in this doc in the kuma_web_1 container after `make bash`.

Basic Vagrant usage

Edit files as usual on your host machine; the current directory is mounted via NFS at /home/vagrant/src within the VM. Updates should be reflected without any action on your part. Useful vagrant sub-commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vagrant ssh</code></td>
<td># Connect to the VM via ssh</td>
</tr>
<tr>
<td><code>vagrant suspend</code></td>
<td># Sleep the VM, saving state</td>
</tr>
<tr>
<td><code>vagrant halt</code></td>
<td># Shutdown the VM</td>
</tr>
<tr>
<td><code>vagrant up</code></td>
<td># Boot up the VM</td>
</tr>
<tr>
<td><code>vagrant destroy</code></td>
<td># Destroy the VM</td>
</tr>
</tbody>
</table>

Run all commands in this doc on the VM after `vagrant ssh`.

Running Kuma

When the Docker container environment is started (`make up` or similar), all of the services are also started. The development instance is available at http://localhost:8000.

The Vagrant environment runs everything in a single VM. It runs MySQL, ElasticSearch, Apache, and other “backend” services whenever the VM is running. There are additional Kuma-specific services that are configured in Procfile, and are run with:

`foreman start`

The Vagrant development instance is then available at https://developer-local.allizom.org.

Running the tests

One way to confirm that everything is working, or to pinpoint what is broken, is to run the test suite.

Django tests

Run the Django test suite:
make test

For more information, see the test documentation.

Front-end tests

To run the front-end (selenium) tests, see Client-side Testing.

Kumascript tests

If you’re changing Kumascript, be sure to run its tests too. See https://github.com/mozilla/kumascript.

Front-end Development and Compiling Sass files

Sass files need to be compiled for changes to take effect, but don’t worry, with DEBUG=True (which is the default for local development), the compilation can be done automatically by Gulp.

When doing front-end development on your local machine, run the following in its own shell from the root directory of your local Kuma repository:

```bash
gulp
```

This `gulp` command will do two things. First, it will watch all files under ./kuma/static, and any changed file that is not a Sass file (.scss or .sass) under ./kuma/static/styles, will be copied to ./static as is (no compilation will be done).

Second, it will watch all files with a .scss extension under ./kuma/static/styles, and any change will trigger a stylelint of the changed file, as well as a recompile of all top-level .scss files. All of the resulting compiled files will then be copied to ./static, and immediately available to your local server.

**Note:** It is currently faster for local development to compile Sass using gulp-sass instead of Django Pipeline. This may change in the future.

If you’d like to manually run stylelint locally on all .scss files under ./kuma/static/styles, do this:

```bash
gulp css:lint
```

If you haven’t already installed Node.js and gulp on your local machine, see Prepare for Front-end Development.

By default DEBUG=True in docker-compose.yml, and in that mode, as mentioned above, source files are compiled on-demand. If for some reason you want to run with DEBUG = False, just remember that source files will no longer be compiled on-demand. Instead, after every change to one or more source files, you’ll have to do the following:

```bash
docker-compose exec web ./manage.py collectstatic
docker-compose restart web
```

in order for your changes to be visible.

Database migrations

Apps are migrated using Django’s migration system. To run the migrations:
manage.py migrate

If your changes include schema modifications, see the Django documentation for the migration workflow.

Coding conventions

See CONTRIBUTING.md for details of the coding style on Kuma.

New code is expected to have test coverage. See the Test Suite docs for tips on writing tests.

Managing dependencies

Python dependencies

Kuma tracks its Python dependencies with pip. See the README in the requirements folder for details.

Front-end dependencies

Front-end dependencies are managed by Bower and checked into the repository. Follow these steps to add or upgrade a dependency:

1. On the host, update bower.json.
2. (Docker only) In the container, install git (apt-get install -y git).
3. (Docker only) In the container, install bower-installer (npm install -g bower-installer).
4. In the VM or container, install the dependency (bower-installer).
5. On the host, prepare the dependency to be committed (git add path/to/dependency).

Front-end dependencies that are not already managed by Bower should begin using this approach the next time they’re upgraded.

Advanced configuration

Environment variables are used to change the way different components work. There are a few ways to change an environment variables:

- Exporting in the shell, such as:

  ```
  export DEBUG=True;
  ./manage.py runserver
  ```

- A one-time override, such as:

  ```
  DEBUG=True ./manage.py runserver
  ```

- Changing the environment list in docker-compose.yml.
- Creating a .env file in the repository root directory.

One variable you may wish to alter for local development is DEBUG_TOOLBAR, which, when set to True, will enable the Django Debug Toolbar:

```
DEBUG_TOOLBAR=True
```

Note that enabling the Debug Toolbar can severely impact response time, adding around 4 seconds to page load time.
The Docker environment

Running docker-compose will create and run several containers, and each container's environment and settings are configured in docker-compose.yml. The settings are “baked” into the containers created by docker-compose up.

To override a container's settings for development, use a local override file. For example, the web service runs in container kuma_web_1 with the default command “gunicorn -w 4 --bind 0.0.0.0:8000 --timeout=120 kuma.wsgi:application”. A useful alternative for debugging is to run a single-threaded process that loads the Werkzeug debugger on exceptions (see docs for runserver_plus), and that allows for stepping through the code with a debugger. To use this alternative, create an override file docker-compose.dev.yml:

```yaml
version: "2"
services:
  web:
    command: ./manage.py runserver_plus 0.0.0.0:8000
    stdin_open: true
    tty: true
```

This is similar to “docker run -it <image> ./manage.py runserver_plus”, using all the other configuration items in docker-compose.yml. Apply the custom setting with:

```
docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d
```

You can then add pdb breakpoints to the code (import pdb; pdb.set_trace) and connect to the debugger with:

```
docker attach kuma_web_1
```

To always include the override compose file, add it to your .env file:

```
COMPOSE_FILE=docker-compose.yml:docker-compose.dev.yml
```

A similar method can be used to override environment variables in containers, run additional services, or make other changes. See the docker-compose documentation for more ideas on customizing the Docker environment.

The Vagrant environment

It is easiest to configure Vagrant with a .env file, so that overrides are used when vagrant up is called. A sample .env could contain:

```
VAGRANT_MEMORY_SIZE=4096
VAGRANT_CPU_CORES=4
# Comments are OK, for documentation and to disable settings
# VAGRANT_ANSIBLE_VERBOSE=true
```

Configuration variables that are available for Vagrant:

- **VAGRANT_NFS**
  
  Default: true (Windows: false) Whether or not to use NFS for the synced folder.

- **VAGRANT_MEMORY_SIZE**
  
  The size of the Virtualbox VM memory in MB. Default: 2048.

- **VAGRANT_CPU_CORES**
  
  The number of virtual CPU core the Virtualbox VM should have. Default: 2.
Kuma Documentation, Release latest

- **VAGRANT_IP**
  The static IP the Virtualbox VM should be assigned to. Default: 192.168.10.55.

- **VAGRANT_GUI**
  Whether the Virtualbox VM should boot with a GUI. Default: false.

- **VAGRANT_ANSIBLE_VERBOSE**
  Whether the Ansible provisioner should print verbose output. Default: false.

- **VAGRANT_CACHIER**
  Whether to use the `vagrant-cachier` plugin to cache system packages between installs. Default: true.

**The database**

The database connection is defined by the environment variable `DATABASE_URL`, with these defaults:

```bash
DATABASE_URL=mysql://kuma:kuma@localhost:3306/kuma  # Vagrant
DATABASE_URL=mysql://root:kuma@mysql:3306/developer_mozilla_org  # Docker
```

The format is defined by the `dj-database-url` project:

```bash
DATABASE_URL=mysql://user:password@host:port/database
```

If you configure a new database, override `DATABASE_URL` to connect to it. To add an empty schema to a freshly created database:

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

To connect to the database specified in `DATABASE_URL`, use:

```
./manage.py dbshell
```

**Asset generation**

Kuma will automatically run in debug mode, with the `DEBUG` setting turned to `True`. Setting `DEBUG=False` will put you in production mode and generate/use minified (compressed) and versioned (hashed) assets. To emulate production, and test compressed and hashed assets locally:

1. Set the environment variable `DEBUG=false`.
2. Start (`docker-compose up -d`) or restart (`docker-compose restart`) your Docker services.
4. Restart the web process using `docker-compose restart web`.

**Secure cookies**

To prevent error messages like “Forbidden (CSRF cookie not set.)”; set the environment variable:

```bash
CSRF_COOKIE_SECURE = false
```

This is the default in Docker, which does not support local development with HTTPS.
Deis Workflow Demo instances

You can deploy a hosted demo instance of Kuma by following these steps:

1. Create a new branch, you cannot create a demo from the master branch.
2. from the Kuma project root directory, run the following command:

   ```
   make create-demo
   ```

3. Your demo will be accessible within about 10 minutes at:

   ```
   https://mdn-demo-<your_branch_name>.virginia.moz.works
   ```

4. Mozilla SRE’s will periodically remove old instances

5. Connecting to the demo database instance

   If you have access to Kubernetes, you can run the following command to connect to the MySQL instance:

   ```
   MY_GIT_BRANCH=$(git rev-parse --abbrev-ref HEAD)
   DEMO_MYSQL_POD=$(kubectl -n "mdn-demo-${MY_GIT_BRANCH}" get pods | grep "^mysql" | awk '{ print $1 }')
   kubectl -n "mdn-demo-${MY_GIT_BRANCH}" exec -it ${DEMO_MYSQL_POD} bash
   mysql -p developer_mozilla_org
   ```

   **Note:** if you copy and paste the code above into a bash terminal and are wondering why the commands don’t appear in your bash history, it’s because there’s whitespace at the beginning of the line.

Maintenance Mode

Maintenance mode is a special configuration for running Kuma in read-only mode, where all operations that would write to the database are blocked. As the name suggests, it’s intended for those times when we’d like to continue to serve documents from a read-only copy of the database, while performing maintenance on the master database.

For local Docker-based development in maintenance mode:

1. If you haven’t already, create a read-only user for your local MySQL database:

   ```
   docker-compose up -d
   docker-compose exec web mysql -h mysql -u root -p
   (when prompted for the password, enter "kuma")
   mysql> source ./scripts/create_read_only_user.sql
   mysql> quit
   ```

2. Create a `.env` file in the repository root directory, and add these settings:

   ```
   MAINTENANCE_MODE=True
   DATABASE_USER=kuma_ro
   ```

   Using a read-only database user is not required in maintenance mode. You can run in maintenance mode just fine with only this setting:

   ```
   MAINTENANCE_MODE=True
   ```

   and going with a database user that has write privileges. The read-only database user simply provides a level of safety as well as notification (for example, an exception will be raised if an attempt to write the database slips through).

3. Update your local Docker instance:
4. You may need to recompile your static assets and then restart:

   docker-compose exec web make build-static
   docker-compose restart web

You should be good to go!

There is a set of integration tests for maintenance mode. If you’d like to run them against your local Docker instance, first do the following:

1. Load the latest sample database (see Load the Sample Database).

2. Ensure that the test document “en-US/docs/User:anonymous:uitest” has been rendered (all of its macros have been executed). You can check this by browsing to http://localhost:8000/en-US/docs/User:anonymous:uitest. If there is no message about un-rendered content, you are good to go. If there is a message about un-rendered content, you will have to put your local Docker instance back into non-maintenance mode, and render the document:

   • Configure your .env file for non-maintenance mode:
     
     MAINTENANCE_MODE=False
     DATABASE_USER=root

   • docker-compose up -d


   and then put your local Docker instance back in maintenance mode:

   • Configure your .env file for maintenance mode:

     MAINTENANCE_MODE=True
     DATABASE_USER=kuma_ro

3. Configure your environment with DEBUG=False because the maintenance-mode integration tests check for the non-debug version of the not-found page:

     DEBUG=False
     MAINTENANCE_MODE=True
     DATABASE_USER=kuma_ro

This, in turn, will also require you to recompile your static assets:

   docker-compose up -d
   docker-compose exec web ./manage.py compilejsi18n
   docker-compose exec web ./manage.py collectstatic
   docker-compose restart web

Now you should be ready for a successful test run:

   py.test --maintenance-mode -m "not search" tests/functional --base-url http://localhost:8000 --driver Chrome --driver-path /path/to/chromedriver

Note that the “search” tests are excluded. This is because the tests marked “search” are not currently designed to run against the sample database.
Troubleshooting

Kuma has many components. Even core developers need reminders of how to keep them all working together. This doc outlines some problems and potential solutions running Kuma.

Fixing Docker issues

The Docker development environment is evolving rapidly. You may need to reset your containers with each change. As we gain experience with Docker, we’ll have more and targeted advice for issues with this development environment.

Kuma “Reset”

These commands will reset your environment to a “fresh” version, while retaining the database:

```
cd /path/to/kuma
docker-compose down
make clean
git submodule sync --recursive && git submodule update --init --recursive
docker-compose pull
docker-compose build --pull
docker-compose up
```

Reset a corrupt database

The Kuma database can become corrupted if the system runs out of disk space, or is unexpectedly shutdown. MySQL can repair some issues, but sometimes you have to start from scratch. When this happens, pass an extra argument --volumes to down:

```
cd /path/to/kuma
docker-compose down --volumes
make clean
git submodule sync --recursive && git submodule update --init --recursive
docker-compose pull
docker-compose build --pull
docker-compose up -d mysql
sleep 20  # Wait for MySQL to initialize. See notes below
docker-compose up
```

The --volumes flag will remove the named MySQL database volume, which will be recreated when you run docker-compose up.

The mysql container will take longer to start up as it recreates an empty database, and the kuma container will fail until the mysql container is ready for connections. The 20 second sleep should be sufficient, but if it is not, you may need to cancel and run docker-compose up again.

Once mysql is ready for connections, follow the installation instructions, starting at Provisioning a database, to configure your now empty database.

Run alternate services

Docker services run as containers. To change the commands or environments of services, it is easiest to add an override configuration file, as documented in The Docker environment.
Linux file permissions

On Linux, it is common that files created inside a Docker container are owned by the root user on the host system. This can cause problems when trying to work with them after creation. We are investigating solutions to create files as the developer’s user.

In some cases, you can specify the host user ID when running commands:

```
docker-compose run --rm --user $(id -u) web ./manage.py collectstatic
```

In other cases, the command requires root permissions inside the container, and this trick can’t be used.

Another option is to allow the files to be created as root, and then change them to your user on the host system:

```
find . -user root -exec sudo chown $(id -u):$(id -g) \{\} \;
```

Fixing Vagrant issues

Kuma “Reset”

These commands will attempt to fix the most common problems in a Vagrant development environment:

```
cd /path/to/kuma
vagrant halt
make clean
git submodule sync --recursive && git submodule update --init --recursive
vagrant up && vagrant provision
```

Running individual processes

It is usually easier to see and debug problems if you run MDN processes individually, instead of running them via foreman. You can run each process exactly as it is listed in Procfile

- web - runs the Django development server.
- worker - runs the celery worker process for tasks.
- camera - stores a snapshot of celery tasks to display in admin site.
- kumascript - runs the node.js process for KumaScript macros.

An alternative is to run most processes via foreman, and override one or more with a custom command. Open two sessions with vagrant ssh. In the first session, run all but the target process (such as web):

```
foreman run all=1,web=0
```

Then you can run your own alternate for web, or the default, in the second session:

```
gunicorn kuma.wsgi -w 2 -b 0.0.0.0:8000
./manage.py runserver_plus 0.0.0.0:8000
./manage.py runserver_plus --print-sql 0.0.0.0:8000
./manage.py runserver_plus --threaded 0.0.0.0:8000
```

Errors after switching branches

- You should occasionally re-run the VM setup, especially after updating code with major changes. This will ensure that the VM environment stays up to date with configuration changes and installation of additional services.
On the Host run:

```
vagrant provision
```

- If you see `ImportError: errors`, you may need to update your git submodules and/or clean out your `*.pyc` files to make sure python has all the latest files it needs:

  ```
k git submodule update --init
make clean
```

- If you see `DatabaseError: (1146, "Table ‘...’ doesn’t exist")` errors, you probably need to run database migrations:

  ```
python manage.py migrate
```

Errors with KumaScript

KumaScript is a very intensive process. If you are only working on python code or front-end code that doesn’t affect live site content, you can usually avoid running it. (See Running individual processes).

- If you see lots of KumaScript timeout errors and you’re running a VM, try increasing the memory allocated to the VM.

  Update the `.env` file in the `/home/vagrant/src` folder:

  ```
MEMORY_SIZE=4096
```

- If you see `Kumascript service failed unexpectedly: HTTPConnectionPool` make sure you enabled KumaScript.

- If changes to Sass stylesheets do not have any effect and you’ve got `DEBUG=True` (debug mode, which is the default), then make sure you’re also running (perhaps in a separate shell):

  ```
gulp
```

- If changes to Sass stylesheets do not have any effect and you’ve got `DEBUG=False` (production mode), then you’ll have to remember to do the following after any change:

  ```
docker-compose exec web make collectstatic
docker-compose restart web
```

Getting more help

If you have more problems running Kuma, please:

1. Paste errors to pastebin.
2. Email the dev-mdn list.
3. After you email dev-mdn, you can also ask in IRC.

The Kuma test suite

Kuma 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. New code should come with tests.

Commands should be run inside the development environment, after `make bash` with Docker, or `vagrant ssh` with Vagrant.
Setup

Before you run the tests, you have to build assets:

```
make build-static
```

Running the test suite

If you followed the steps in the installation docs, then all you should need to do to run the test suite is:

```
make test
```

The default options for running the test are in `pytest.ini`. This is a good set of defaults.

If you ever need to change the defaults, you can do so at the command line by running what the Make task does behind the scenes:

```
py.test kuma
```

Some helpful command line arguments to `py.test` (won’t work on `make test`):

- **--pdb**: Drop into pdb on test failure.
- **--create-db**: Create a new test database.
- **--showlocals**: Shows local variables in tracebacks on errors.
- **--exitfirst**: Exits on the first failure.

See `py.test --help` for more arguments.

Running subsets of tests and specific tests

There are a bunch of ways to specify a subset of tests to run:

- only tests marked with the ‘spam’ marker:
  
  ```
  py.test -m spam
  ```

- all the tests but those marked with the ‘spam’ marker:
  
  ```
  py.test -m "not spam"
  ```

- all the tests but the ones in `kuma/core`:
  
  ```
  py.test --ignore kuma/core
  ```

- all the tests that have “foobar” in their names:
  
  ```
  py.test -k foobar
  ```

- all the tests that don’t have “foobar” in their names:
  
  ```
  py.test -k "not foobar"
  ```

- tests in a certain directory:
  
  ```
  py.test kuma/wiki/
  ```

- specific test:
See http://pytest.org/latest/usage.html for more examples.

### Showing test coverage

While running the tests you can record which part of the code base is covered by test cases. To show the results at the end of the test run use this command:

```
make coverage test
```

To generate an HTML coverage report, use:

```
make coverage test html
```

### 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.

### Markers

See:

```
py.test --markers
```

for the list of available markers.

To add a marker, add it to the `pytest.ini` file.

To use a marker, add a decorator to the class or function. Examples:

```python
import pytest

@pytest.mark.spam
class SpamTests(TestCase):
    ...

class OtherSpamTests(TestCase):
    @pytest.mark.spam
    def test_something(self):
        ...
```

### Adding tests

Code should be written so that 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 `/en-US/docs/Mozilla versus /docs/Mozilla`, use `LocalizingClient` instead of the default client for the `TestCase` 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 it’s 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.

Client-side testing

Kuma has a suite of functional tests using Selenium and pytest. This allows us to emulate users interacting with a real
browser. All these test suites live in the /tests/ directory.
The tests directory comprises of:

- /functional contains pytest tests.
- /pages contains Python PyPOM page objects.
- /utils contains helper functions.

Setting up test virtual environment

1. In the terminal go to the directory where you have downloaded kuma.
2. Create a virtual environment for the tests to run in (the example uses the name mdntests):
   
   ```
   $ virtualenv mdntests
   ```

3. Activate the virtual environment:

   ```
   $ source mdntests/bin/activate
   ```

4. Make sure the version of pip in the virtual environment is high enough to support hashes:

   ```
   $ pip install "pip>=8"
   ```

5. Install the requirements the tests need to run:

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

You may want to add your virtual environment folder to your local .gitignore file.

Running the tests

Before running the tests you will need to download the driver for the browser you want to test in. Drivers are listed
and linked to in the pytest-selenium documentation.
The minimum amount of information necessary to run the tests against the staging server is the directory the tests are
in, which driver to use, and the subset of tests to run (by specifying a marker or marker expression like `\-m "not
login"`, for more information see Markers). Currently, all of the tests except the tests marked “login” should run
successfully against the staging server.
In the virtual environment run:

\$ py.test -m "not login" tests/functional/ --driver Chrome --driver-path /path/to/chromedriver

You will be prompted “Do you want the application ‘python’ to accept incoming network connections?” The tests seem to run fine no matter how you answer.

This basic command can be modified to use different browsers, to run the tests against a local environment, or to run tests concurrently.

**Only running tests in one file**

Add the name of the file to the test location:

\$ py.test -m "not login" tests/functional/test_search.py --driver Chrome --driver-path /path/to/chromedriver

**Run the tests against a different url**

By default the tests will run against the staging server. If you’d like to run the tests against a different URL (e.g., your local environment) pass the desired URL to the command with --base-url:

\$ py.test -m "not login" tests/functional/ --base-url http://localhost:8000 --driver Chrome --driver-path /path/to/chromedriver

**Run the tests in parallel**

By default the tests will run one after the other but you can run several at the same time by specifying a value for -n:

\$ py.test -m "not login" tests/functional/ -n auto --driver Chrome --driver-path /path/to/chromedriver

**Run the tests against a server configured in maintenance mode**

By default the tests will run against a server assumed to be configured normally. If you’d like to run them against a server configured in maintenance mode, simply add --maintenance-mode to the py.test command line. For example, if you’ve configured your local environment to run in maintenance mode:

\$ py.test --maintenance-mode -m "not search" tests/functional/ --base-url http://localhost:8000 --driver Chrome --driver-path /path/to/chromedriver

Note that the tests marked “search” were excluded assuming you’ve loaded the sample database. If you’ve loaded a full copy of the production database, you can drop the -m "not search".

The --maintenance-mode command-line switch does two things. It will skip any tests that don’t make sense in maintenance mode (e.g., making sure the signin link works), and include the tests that only make sense in maintenance mode (e.g., making sure that endpoints related to editing redirect to the maintenance-mode page).

**Run tests on SauceLabs**

Running the tests on SauceLabs will allow you to test browsers not on your host machine.

1. Signup for an account.
2. Log in and obtain your Remote Access Key from user settings.
3. Run a test specifying SauceLabs as your driver, and pass your credentials and the browser to test:
Alternatively you can save your credentials in a configuration file so you don’t have to type them each time.

**Run tests on MDN’s Continuous Integration (CI) infrastructure**

If you have commit rights on the mozilla/kuma GitHub repository you can run the UI tests using the MDN CI Infrastructure. Just force push to mozilla/kuma@stage-integration-tests to run the tests against https://developer.allizom.org.

You can check the status, progress, and logs of the test runs at MDN’s Jenkins-based multi-branch pipeline.

**MDN CI Infrastructure**

The MDN CI infrastructure is a Jenkins-based, multi-branch pipeline. The pipelines for all branches are defined by the Jenkinsfile and the files under the Jenkinsfiles directory. The basic idea is that every branch may have its own custom pipeline steps and configuration.

Jenkins will auto-discover the steps and configuration by checking within the Jenkinsfiles directory for a Groovy (`.groovy`) and/or YAML (`.yml`) file with the same name as the branch. For example, the “stage-integration-tests” branch has a Jenkinsfiles/stage-integration-tests.yml file which will be loaded as configuration and used to determine what to do next (load and run the Groovy script specified by its `pipeline.script` setting - Jenkinsfiles/integration-tests.groovy - and the script, in turn, will use the dictionary of values provided by the `job` setting defined within the configuration).

Note that the YAML files for the integration-test branches provide settings for configuring things like the version of Selenium to use, the number of Selenium nodes to spin-up, the Dockerfile to use to build the container, the URL to test against, and which subset of integration tests to run (via a pytest marker expression, see **Markers**).

The integration-test Groovy files use a number of global Jenkins functions that were developed to make the building, running and pushing of Docker containers seamless (as well as other cool stuff, see mozmar/jenkins-pipeline). They allow us to better handle situations that have been painful in the past, like the stopping of background Docker containers.

The “prod-integration-tests” branch also has its own Jenkinsfiles/prod-integration-tests.yml file. It’s identical to the YAML file for the “stage-integration-tests” branch except that it specifies that the tests should be run against the production server rather than the staging server (via its `job.base_url` setting).

Similarly, the “master” branch has it’s own pipeline, but instead of being configured by a YAML file, the entire pipeline is defined within its Jenkinsfiles/master.groovy file.

The pipeline for any other branch which does not provide its own Groovy and/or YAML file will follow that defined by the Jenkinsfiles/default.groovy file.

You can check the status, progress, and logs of any pipeline runs via MDN’s Jenkins-based multi-branch pipeline.

**Markers**

- **nondestructive**
  Tests are considered destructive unless otherwise indicated. Tests that create, modify, or delete data are considered destructive and should not be run in production.

- **smoke**
  These tests should be the critical baseline functional tests.
New instances of kuma have empty databases so only a subset of tests can be run against them. These tests are marked with `nodata`.

- **login**
  These tests require the testing accounts to exist on the target site. For security reasons these accounts will not be on production. Exclude these tests with `-m "not login"`

### Guidelines for writing tests

See Bedrock and the Web QA Style Guide.

### Performance testing

In 2015, as part of a “measure and improve performance” initiative, MDN staff configured Locust for performance testing of MDN. The locust test files simulate read traffic to MDN, fetching the top pages in roughly the same ratios as they were fetched on MDN, and can be run against test environments to measure changes in code, caching and settings.

These tests are not maintained or run during the current development process, and the instructions have not been updated for the Docker development environment.

### Running tests

**Note**: DO NOT RUN LOCUST TESTS AGAINST PRODUCTION

1. Start locust from the development VM:

```
vagrant ssh
make locust
```


   - number of users to simulate
   - users spawned per second

See Start locust for more.

### Adding a test suite

See Writing a locustfile for more.

### Feature toggles

MDN uses feature toggles to integrate un-finished feature changes as early as possible, and to control the behavior of finished features.
Some site features are controlled using `django-waffle`. You control these features in the django admin site’s `waffle` section.

Some site features are controlled using `con stance`. You control these features in the django admin site’s `con stance` section.

**Waffle features**

### Switches

Waffle switches are simple booleans - they are either on or off.

- **application_ACAO** - Enable Access-Control-Allow-Origin=0 header.
- **dumb_doc_urls** - (deprecated) Disable the render-time changing of /docs/ URLs to the equivalent Zone URLs (see PR 3331 for reasoning).
- **enable_optimizely** - Enable the Optimizely JavaScript.
- **store_revision_ips** - Save request data, including the IP address, to enable marking revisions as spam.
- **welcome_email** - Send welcome email to new user registrations.
- **wiki_error_on_delete** - Throw an error if a user tries to delete a page.
- **wiki_force_immediate_rendering** - Force wiki pages to render immediately in the same http request in which they are saved (not in a background process).
- **newsletter** - Show newsletter sign-up site wide.
- **newsletter_article** - Show newsletter sign-up in article footer (must be used in combination with newsletter).
- **foundation_callout** - Show Foundation donation tile on homepage.

### Flags

Waffle flags control behavior by specific users, groups, percentages, and other advanced criteria.

- **kumabanned** - (deprecated) added to users to mark them as banned.
- **kumaediting** - Enable/disable wiki editing.
- **page_move** - (deprecated) enable/disable page move feature.
- **registration_disabled** - Enable/disable new user registration.
- **search_doc_navigator** - Show the search doc navigator feature.
- **search_suggestions** - Show the advanced search filter suggestions interface.
- **section_edit** - Show section edit buttons.
- **sg_task_completion** - Enable the Survey Gizmo pop-up.
- **spam_admin_override** - Tell Akismet that edits are never spam.
- **spam_spammer_override** - Tell Akismet that edits are always spam.
- **spam_testing_mode** - Tell Akismet that edits are tests, not real content.
- **spam_checks_enabled** - Toggle spam checks site wide.
- **spam_submissions_enabled** - Toggle Akismet spam/spam submission ability.
• wiki_samples - Add button to open samples in Codepen or jsFiddle.
• wiki_spam_exempted - Exempt users and user groups from checking submissions for spam.
• wiki_spam_training - Call Akismet to check submissions, but don’t block due to detected spam or Akismet errors.

Constance features

Constance configs let us set operational *values* for certain features in the database - so we can control them without changing code. They are all listed and documented in the admin site’s constance section.

Celery and async tasks

Kuma uses Celery to enable asynchronous task processing for long-running jobs and to speed up the request-response cycle.

When is Celery appropriate

You can use Celery to do any processing that doesn’t need to happen in the current request-response cycle. 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.

Some examples where Celery is used:

• **Sending notification emails on page changes** - If this was done in the page change request, then pages with many watchers would be slower to edit, and errors in email sending would result in a “500 Internal Server Error” for the editor.

• **Populating the contributor bar** - Gathering and sorting the contributor list can be a slow process that would delay rendering of stale pages. Instead, viewers quickly see the content, possibly without the contributor bar. Once the async task has populated the cache, future viewers get the content and the contributor bar.

• **Generating the spam moderator dashboard** - Calculating spam statistics is a potentially long process, and if it takes more than 30 seconds the viewer will get a “502 Bad Gateway” error. The async task can take as long as needed, and the spam moderator will see the dashboard when the data is calculated and cached.

• **Rebuilding sitemaps** - Search engines need recent site URL data, and it can’t be assembled quickly in the request for the sitemap. An async request repopulates the cached sitemap data, keeping this data fast and up-to-date.

There are some downsides to async tasks that should also be considered:

• Message passing, serialization, de-serialization, and data reloading increase the load on the entire system.

• Async tasks require different testing strategies. It is easy to write a passing test that doesn’t reflect how the task is called in production.

• Runtime errors are more difficult to troubleshoot with async tasks, due to missing context on how they were called.

In general, it is better to get an algorithm right in the request loop, and only move it to an asynchronous task when it is identified as a performance issue.
Celery services

A working Celery installation requires several services.

**Worker**

Celery processes tasks with one or more workers. In Kuma, the workers and web processes share a code base, so that Django models, functions, and settings are available to async tasks, and web code can easily schedule async tasks.

In Docker, the worker process runs in the `worker` service/container.

In Vagrant, the worker process is started with the `foreman` command.

**Broker**

Celery requires a message broker for task communication. There are two stable, production-ready alternatives:

- **Redis** is an in-memory data structure store, used as database, cache and message broker. Many projects use it for multiple roles in the same deployment. With Celery 3.1, it is now recommended as a stable broker, with some caveats. It is used in the Docker environment.

  - **RabbitMQ** is an AMQP message broker written in Erlang. The Celery team has recommended it for a long time, and the docs describe it as “feature-complete, stable, durable and easy to install”. It is used in the Vagrant and production environments.

**Result store**

When a task completes, it returns processed data and task states to a results store. Kuma doesn’t use returned data, but it does use returned task state to coordinate multi-step tasks.

Kuma uses a database-backed task store provided by django-celery, a deprecated integration project for Django and Celery. The work to replace this is tracked in bug 1268257.

**Periodic tasks scheduler**

Periodic tasks are scheduled with `celery beat`, which adds tasks to the task queue when they become due. It should only be run once in a deployment, or tasks may be scheduled multiple times.

It is run in the Docker and Vagrant environments by running the single celery worker process with `--beat`. In production, there are several task workers, and the `celery beat` process is run directly on just one worker.

The schedules themselves are configured using the deprecated django-celery database backend. The work to replace this is tracked in bug 1268256.

**Monitoring**

Celery task workers generate events to communicate task and worker health. The `celerycam` service, provided by the deprecated django-celery project, captures these events and stores them in the database. Switching to a supported project, like Flower, is tracked in bug 1268281.

It is not part of the default Docker services, but can be started inside the `worker` service container with:

```bash
./manage.py celerycam --freq=2.0
```
In the Vagrant environment, celerycam is started with other Kuma services with foreman. For more options, see the Monitoring and Management Guide in the Celery documentation.

Configuring and running Celery

We set some reasonable defaults for Celery in kuma/settings/common.py. These can be overridden by the environment variables, including:

- **CELERY_ALWAYS_EAGER**
  
  Default: false (Docker), true (Vagrant, tests).
  
  When true, tasks are executed immediately, instead of being scheduled and executed by a worker, skipping the broker, results store, etc. In theory, tasks should act the same whether executed eagerly or normally. In practice, there are some tasks that fail or have different results in the two modes, mostly due to database transactions.

- **CELERYD_CONCURRENCY**
  
  Default: 4.
  
  Each worker will execute this number of tasks at the same time. 1 is a good value for debugging, and the number of CPUs in your environment is good for performance.

The worker can also be adjusted at the command line. For example, this could run inside the worker service container:

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

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

Search

Kuma uses Elasticsearch to power its on-site search.

Installed version

Elasticsearch releases new versions often, and Kuma is slow to upgrade. The Docker environment uses 2.4, and production uses 1.7, with a planned update to 2.4.

The Kuma search tests use the configured Elasticsearch, and can be run inside the web Docker container to confirm the engine works:

```bash
git test kuma/search/
```

Indexing documents

To see search results, you will need to index the documents.

Using the Admin

This process works both in a development environment and in production:

- Open the Django admin search index list view. On Docker, this is located at http://localhost:8000/admin/search/index/.
• Add a search index by clicking on the “Add index” button in the top right corner. Optionally name it, or leave it blank to generate a valid name based on the time. Spaces are not allowed in the name. Click the “SAVE” button in the lower right corner.

• On the search index list, select the checkbox next to the newly created index (the top most) and select “Populate selected search index via Celery” from the Action dropdown menu. Click “Go” to start indexing asynchronously.

• In the development environment, you can watch the indexing process with:

```
docker-compose logs -f elasticsearch worker
```

• Refresh the search index list until the “populated” field changes to a green checkbox image. In production, you will also get an email notifying you when the index is populated.

• Select the checkbox next to the populated index, then choose “Promote selected search index to current index” in the actions dropdown. Click “Go” to promote the index.

• All three fields will be green checkboxes (promoted, populates, and “is current index?”). The index is live, and will be used for site search.

Similarly you can also demote a search index and it will automatically fall back to the previously created index (by created date). That helps to figure out issues in production and should allow for a smooth deployment of search index changes. It’s recommended to keep a few search indexes around just in case.

If no index is created in the admin UI the fallback “main_index” index will be used instead.

When you delete a search index in the admin interface, it is deleted on Elasticsearch as well.

**Using the shell**

Inside the web Docker container:

```
./manage.py reindex
```

This will populate and activate the fallback index “main_index”. It will be overwritten when the search tests run.

**Localization**

Kuma 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. Pontoon allows translators to use the web UI as well as download the PO files, use whatever tool the translator feels comfortable with and upload it back to Pontoon.

**Note:** We do not accept pull requests for updating translated strings. Please use Pontoon instead.

See the Django documentation on Translations for how to make strings marked for translation in Python and templates.

Unless otherwise noted, run all the commands in this document inside the development environment.

For Docker, enter the environment with `docker-compose run --rm --user $(id -u) web bash`, to ensure that created files are owned by your development user.

For Vagrant, enter the environment with `vagrant ssh`.
Getting the localizations

Localizations are found in this repository under the `locale` folder.

The gettext portable object (`.po`) files need to be compiled into the gettext machine object (`.mo`) files before translations will appear. This is done once during initial setup and provisioning, but will be out of date when the kuma locales are updated.

To refresh the translations, enter the development environment, then:

1. Compile the `.po` files:
   ```bash
   make localecompile
   ```

2. Update the static JavaScript translation catalogs:
   ```bash
   make compilejsi18n
   ```

3. Collect the built files so they are served with requests:
   ```bash
   make collectstatic
   ```

Updating the localizations

When localizable strings are added, changed, or removed in the code, they need to be gathered into `.po` files for translation.

**Note:** This work is done only during the preparation to push to production. You do not need to do this for your PR.

To update the localizations:

1. Inside the development environment, extract and rebuild the translations:
   ```bash
   make localerefresh
   ```

2. On the host system, review the changes to source English strings:
   ```bash
   git diff locale/templates/LC_MESSAGES
   ```

3. Finally, commit the files:
   ```bash
   git add --all locale
   git commit -m "MDN string update YYYY-MM-DD"
   ```

Adding a new locale (UI strings)

The process for getting a new locale on MDN is documented at [Starting a new MDN localization](#). One step is to enable translation of the UI strings.

This example shows adding a Bulgarian (bg) locale. Change `bg` to the locale code of the language you are adding.

1. **Update the Localizations** as above, so that your commit will be limited to the new locale.

2. Add the locale to `CANDIDATE_LANGUAGES` in `kuma/settings/common.py`.

3. Download the latest `languages.json` from `https://product-details.mozilla.org/1.0/languages.json` and place it at `kuma/settings/languages.json`. 

2.10. Localization 31
4. Add the locale to `translate_locales.html` and the `locale/` folder:

   ```
   make locale LOCALE=bg
   ```

5. Generate the compiled files for all the locales, including the new one:

   ```
   make localerefresh
   ```

6. Commit the changes to `locale`, `jinja2/includes/translate_locales.html`, and `kuma/settings`. The other locales should include a new string representing the new language.

When the change is merged to master, enable the language in Pontoon as well, and notify the language community to start UI translation.

**Adding a New Locale (Page Translations)**

Once the new translation community has completed the rest of the process for starting a new MDN localization, it is time to enable the language for page translations:

1. Move the locale from `CANDIDATE_LANGUAGES` to `MDN_LANGUAGES` in `kuma/settings/common.py`.
2. Restart the web server and verify that Django loads the new locale without errors by visiting the locale’s home page, for example `http://localhost:8000/bg/` (https://developer-local.allizom.org/bg/ if you are using Vagrant).
3. Commit the change to `kuma/settings/common.py`.

When the change is merged and deployed, inform the localization lead and the community that they can begin translating content.

**CKEditor**

The Mozilla Developer Network uses a WYSIWYG editor called CKEditor. CKEditor is an open source utility which brings the power of rich text editing to the web. This document details how to update CKEditor within the MDN codebase.

**Building CKEditor**

To rebuild CKEditor, run this on the host system:

```bash
cd kuma/static/js/libs/ckeditor/
./docker-build.sh  # Creates a Java container, builds CKEditor
docker-compose exec web make build-static
```

This builds CKEditor within a Docker VM, using the Java `ckbuilder` tool from CKSource, then builds static resources so that the updated editor is installed where it belongs.

To rebuild CKEditor from a `vagrant` ssh shell (or locally if you have Java installed):

```bash
cd kuma/static/js/libs/ckeditor/source/
./build.sh
```

Portions of the build process will take a few minutes so don’t expect an immediate result.
Updating CKEditor

To update the CKEditor version, you'll need to edit `build.sh` and change the value of `CKEDITOR_VERSION`. Updating is important to keep MDN in sync with CKEditor's functional and security updates.

Updating CKEditor plugins

Some plugins are maintained by MDN staff in the Kuma repo. Others are updated to the tag or commit number specified in `build.sh`:

- `descriptionlist`
- `scayt`
- `wsc`

Change to CKEditor plugins which are bundled into Kuma should be made in the directory `/kuma/static/js/libs/ckeditor/source/ckeditor/plugins/`.

Once you’ve made changes to a plugin, be sure to build the editor and the static resources again, as described in `Building CKEditor`.

Documentation

This documentation is generated and published at Read the Docs whenever the master branch is updated.

GitHub can render our `.rst` documents as ReStructuredText, which is close enough to Sphinx for most code reviews, without features like links between documents.

It is occasionally necessary to generate the documentation locally. It is easiest to do this with a virtualenv on the host system, using Docker only to regenerate the MDN Sphinx template. If you are not comfortable with that style of development, it can be done entirely in Docker using `docker-compose`.

Generating documentation

Sphinx uses a Makefile in the `docs` subfolder to build documentation in several formats. MDN only uses the HTML format, and the generated document index is at `docs/_build/html/index.html`.

To generate the documentation in a virtualenv on the host machine, first install the requirements:

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

Then switch to the `docs` folder to use the Makefile:

```
cd docs
make html
python -m webbrowser file://${PWD}/_build/html/index.html
```

To generate the documentation with Docker:

```
```
A virtualenv is required, to avoid a pip bug when changing the version of a system-installed package.

**Using the MDN Sphinx Theme project**

The documentation uses a Sphinx theme generated from the MDN templates, hosted at https://github.com/mdn/sphinx-theme. This theme is used by MDN and some other groups at Mozilla. If you are going to update the theme, you should switch from the published version specified in docs.txt to a local working copy.

When working with a virtualenv on the host machine, the sphinx-theme project can be checked out anywhere, and installed in the virtualenv:

```
pip install -e /path/to/sphinx-theme
```

Now, when the documentation is built, it will use the current theme in the working copy.

Using Docker, the sphinx-theme project must be checked out inside the Kuma repo:

```
cd /path/to/kuma
git clone git@github.com:mdn/sphinx-theme.git
```

This will make the sphinx-theme available in Docker at the path /app/sphinx-theme. When generating documentation, adjust the command to install the theme from the working copy:

```
docker-compose run --rm --user $(id -u) web sh -c "
    virtualenv /tmp/.venvs/docs && 
    . /tmp/.venvs/docs/bin/activate && 
    pip install -r /app/requirements/docs.txt && 
    pip install -e /app/sphinx-theme && 
    cd /app/docs && 
    make html"
python -m webbrowser file://${PWD}/docs/_build/html/index.html
```

**Regenerate the MDN Sphinx Theme template**

When regenerating the template, the Kuma version should be the one deployed to production. The assets are loaded from the MDN CDN, and if a non-deployed Kuma commit is used to generate the template, files will be unavailable, which may or may not be obvious when you view the documentation locally.

The MDN Sphinx theme template can be regenerated by Kuma:

```
docker-compose run --rm -T \
    -e DEBUG=False \n    -e SITE_URL=https://developer.mozilla.org \n    web \n    /app/manage.py generate_sphinx_template \n    > sphinx-theme/mdn_theme/mdn/layout.html
```

If you are using a virtualenv on the host system, you will need to modify the last line to be the path to your working copy.

In either case, you’ll want to regenerate the documentation to confirm the new layout, and use the browser’s inspector to look for 404s on static assets.

**Publish the new MDN Sphinx Theme**

To use the new theme in the public documentation:
1. Commit the new template to `sphinx-theme`, open a pull request.
2. Remove the clone of the `sphinx-theme` from your Kuma repo (Docker method).
3. After review, merge the PR to the master branch of `sphinx-theme`.
4. Tag and publish a new version of `mdn-sphinx-theme` to PyPI.
5. Update `requirements/docs.txt` in `kuma`, merge to master.

### Installation via Vagrant

Since 2011, staff developers ran a model of the entire MDN stack in a Vagrant-managed virtual machine. This included backing services like a database, caching server, and search engine, as well as the application services like Django, KumaScript, and Celery.

Vagrant started out closely aligned with production, but over the years the Vagrant environment evolved to make development easier, and the production configuration was maintained by the operations team. For example, Vagrant switched the operating system from CentOS to Ubuntu, and the provisioning system from Puppet to Ansible. The Puppet scripts used to maintain the production infrastructure have evolved, and can not be published or reused for development due to configuration secrets. Both systems work, but it is less likely that the development environment can be used to reproduce production issues, or that the development environment can be used as a testbed for infrastructure changes.

Starting in 2011, Kuma is moving to Docker for development, testing, and deployment. Docker is now the primary development environment, and the Docker setup instructions should be used by new developers.

Until the transition is complete, we cover development with both the Docker and the Vagrant environments. We do not plan to keep feature parity between the environments, and will only fix critical errors in Vagrant provisioning.

There are common issues encountered when provisioning Vagrant, or maintaining a long-lived installation. See the *Errors* section for tips and tricks, as well as the Troubleshooting page.

### Install and run everything

1. Install VirtualBox >= 5.0.x from [http://www.virtualbox.org/](http://www.virtualbox.org/)

   **Note:** (Windows) After installing VirtualBox you need to set `PATH=C:\Program Files\Oracle\VirtualBox\VBoxManage.exe;`

2. Install Vagrant >= 1.7 using the installer from vagrantup.com.
3. Install Ansible >= 1.9.x on your machine so that Vagrant is able to set up the VM the way we need it.
   
   See the Ansible Installation docs for which way to use on your computer’s platform.
   
   The most common platforms:
   
   **Mac OS X:**
   
   `brew install ansible`
   
   or if you have a globally installed pip:
   
   `sudo pip install ansible`
   
   **Ubuntu:**
$ sudo apt-get install software-properties-common
$ sudo apt-add-repository ppa:ansible/ansible
$ sudo apt-get update
$ sudo apt-get install ansible

Fedora / RPM-based distribution:
$ sudo dnf install ansible.noarch

For previous versions based on yum, use:
$ sudo yum install ansible.noarch

Windows:
Installation on Windows is complicated but we strive to make it easier in the future. Until then see this blog post for how to Run Vagrant with Ansible Provisioning on Windows.

4. Fork the project. (See GitHub)

5. Clone your fork of Kuma and update submodules:
   ```
git clone git@github.com:<your_username>/kuma.git
cd kuma
git submodule update --init --recursive
   ```

6. Start the VM and install everything. (approx. 15 minutes on a fast net connection):
   ```
vagrant up
   ```

   **Note:** VirtualBox creates VMs in your system drive. Kuma’s VM is approx. 2GB. If it won’t fit on your system drive, you will need to change that directory to another drive.

At the end, you should see something like:

```
PLAY RECAP ********************************************************************
developer-local : ok=147 changed=90 unreachable=0 failed=0
===> developer-local: Configuring cache buckets...
```

If the above process fails with an error, check *Errors*.

7. Log into the VM with ssh:
   ```
vagrant ssh
   ```

8. Use *foreman* inside the VM to start all site services:
   ```
foreman start
   ```

   You should see output like:

```
20:32:59 web.1 | started with pid 2244
20:32:59 celery.1 | started with pid 2245
20:32:59 kumascript.1 | started with pid 2246
...```


10. Visit the homepage at https://developer-local.allizom.org
Create an admin user

You will want to make yourself an admin user to enable important site features.

1. Sign up/in with Persona.
2. After you sign in, SSH into the VM and make yourself an admin (exchange << YOUR_USERNAME >> with the username you used when signing up for Persona):
   ```
   vagrant ssh
   python manage.py ihavepower "<< YOUR_USERNAME >>"
   ```

You should see:

Done!

Enable the wiki

By default, the wiki is disabled with a feature toggle. So, you need to create an admin user, sign in, and then use the Django admin site to enable the wiki so you can create pages.

1. As the admin user you just created, visit the waffle section of the admin site.
2. Click “Add flag”.
3. Enter “kumaediting” for the Name.
4. Set “Everyone” to “Yes”.
5. Click “Save”.

You can now visit https://developer-local.allizom.org/docs/new to create new wiki pages as needed.

Many core MDN contributors create a personal User:<username> page as a testing sandbox.

(Advanced) Enable KumaScript

By default, KumaScript is also disabled with a feature toggle. To enable KumaScript:

1. Sign in as the admin user.
2. Visit the constance config admin panel.
3. Change KUMASCRIPT_TIMEOUT to 600.
4. Click “Save” at the bottom.
5. Import the KumaScript auto-loaded modules:
   ```
   vagrant ssh
   python manage.py import_kumascript_modules
   ```

Note: You must create a user to import kumascript modules.
(Advanced) Enable GitHub Auth

To enable GitHub authentication, register your own OAuth application on GitHub:

- Application name: MDN (<username>).
- Homepage url: https://developer-local.allizom.org/docs/MDN/Contribute/Howto/Create_an_MDN_account.
- Application description: My own GitHub app for MDN!

As the admin user, add a django-allauth social app for GitHub:

- Provider: GitHub.
- Name: developer-local.allizom.org.
- Client id: <your GitHub App Client ID>.
- Secret key: <your GitHub App Client Secret>.
- Sites: example.com -> Chosen sites.

Now you can sign in with GitHub at https://developer-local.allizom.org/

Errors during Installation

vagrant up starts the virtual machine. The first time you run `vagrant up` it also provisions the VM - i.e., it automatically installs and configures Kuma software in the VM. We provision the VM with Ansible roles in the provisioning directory.

Sometimes we put Ansible roles in the wrong order. Which means some errors can be fixed by simply provisioning the VM again:

vagrant provision

In some rare occasions you might need to run this multiple times. If you find an error that is fixed by running `vagrant provision` again, please email us the error at dev-mdn@lists.mozilla.org and we’ll see if we can fix it.

If you see the same error over and over, please ask for more help.

Django database migrations

If you see errors that have “Django database migrations” in their title try to manually run them in the VM to see more about them. To do so:

vagrant ssh
python manage.py migrate

If you get an error, please ask for more help.

Ubuntu

On Ubuntu, `vagrant up` might fail after being unable to mount NFS shared folders. First, make sure you have the nfs-common and nfs-server packages installed and also note that you can’t export anything via NFS inside an encrypted volume or home dir. On Windows NFS won’t be used ever by the way.
If `vagrant up` works but you get the error `IOError: [Errno 37] No locks available`, that indicates that the host machine isn’t running rpc.statd or statd. This has been seen to affect Ubuntu >= 15.04 (running systemd). To enable it, run the following commands:

```
vagrant halt
sudo systemctl start rpc-statd.service
sudo systemctl enable rpc-statd.service
vagrant up
```

If that doesn’t help you can disable NFS by setting the `VAGRANT_NFS` configuration value in a `.env` file. See the `Vagrant configuration` options for more info.

If you have other problems during `vagrant up`, please check `Troubleshooting`.