playdoh Documentation

Release 1.0

Mozilla

Sep 27, 2017

Contents

Mozilla's Playdoh is a web application **template** based on Django.

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Playdoh is simply a pre-configured Django project that adds some essential modules and middleware to meet the following goals:

- Enhance the **security** of your application and its data
- Achieve optimal **performance** in the face of high traffic
- **Localize** content in multiple languages using Mozilla's L10n standards
- Use the best tools and best practices to make development **fun and easy**

This is an open source project; patches are welcome! Feel free to fork and contribute to this project on Github.

All new web development projects at Mozilla (and maybe elsewhere?) are created using Playdoh. However, Playdoh is not a framework itself. All of its components can be installed and configured individually without using the Playdoh template.

CHAPTER 1

Contents

Getting started

Installation

Requirements

You need Python 2.6 or 2.7, MySQL, git, virtualenv, and a Unix like OS.

Starting a project based on playdoh

The secret to what makes playdoh fun is the funfactory module. When you install funfactory from PyPI or from source you get a command line script that you can use to start a new playdoh project. Install funfactory with a package manager like pip:

```
pip install funfactory
```

You'll now have the Playdoh installer script. Read through `funfactory --help` or start a new project like this:

```
funfactory --python=python2.6 --pkg=yourapp
```

The automatic install process goes like this:

1. Clone the Playdoh git repository
2. Create a custom `yourapp` package
3. Create a virtualenv named `yourapp` (if not already in a virtualenv)
4. Install/compile all requirements
5. Create a local settings file in `yourapp/settings/local.py` and fill in some settings.

Note: If a virtualenv needs to be created and you have `virtualenvwrapper` installed, the created virtualenv will go in your `WORKON_HOME` directory. Otherwise the virtualenv will be installed in `.virtualenv`.

See also:

Installing everything automatically in a Vagrant VM

The Playdoh project layout uses a vendor library, i.e. a subdirectory `vendor` that contains all pure Python libraries required. In addition, a few C based libraries (such as MySQL-python, bcrypt, etc) get built by the installer. For more information on vendor libraries, read *pip and friends: Packaging*.

Configuration

By default the funfactory installer configures your app to use a MySQL database named `playdoh_app`. You'll need to create the database manually:

```
mysql -u root -e 'create database playdoh_app;'
```

If you need to adjust any settings for the database connection, edit `yourproject/settings/local.py`. Synchronize tables and initial data:

```
./manage.py syncdb
```

Start the development server:

```
./manage.py runserver 0.0.0.0:8000
```

You can now view the dev server at http://localhost:8000/ – hooray!

If you start adding pieces that should go back into playdoh, you will probably want to patch funfactory, which is the core of Playdoh.

If your app's configuration requires you to add or remove apps, middleware, or template context processors from the default funfactory configuration, you can use the `get_apps`, `get_middleware`, and `get_context_processors` functions. See the *settings management API* documentation.

Installing a project by hand

The installer script automates everything you'd need to do by hand but it's simple to do it yourself. Here's what you would do:

1. Clone the Playdoh git repository into `customproject`.
2. cd into that directory and rename `project` to `customproject` (this is the actual Python module).
3. Edit setup.py so the module name is `customproject`
4. Copy `customproject/settings/local.py-dist` to `customproject/settings/local.py`.
5. Edit `local.py`:
    - Fill in your DB credentials
    - Enter a secret key
    - Enter an HMAC key for bcrypt password hashing
6. Create a virtualenv (if not already in one)

7. Run `pip install -r requirements/compiled.txt`

Then you should be ready to run syncdb and start up the server:

```
./manage.py syncdb
./manage.py runserver 0.0.0.0:8000
```

Upgrading

There is a *whole section* on that!

Upgrading Playdoh

Upgrading your app to the latest version of playdoh means you'll share the latest features. Depending on the state of your app you'll want to choose one of the following upgrade paths:

Funfactorifying any Project

Applying Playdoh to an existing Django app is a little different than *starting an app from scratch*. Here's how it works:

1. Add funfactory to the *vendor* library. If you don't have a vendor submodule, take a look at how the *installer* sets that up. If you like, you can simply install funfactory from PyPI into a virtualenv.
2. Funfactory expects you to have a top level Python module named `yourapp` or whatever. This is a directory with an `__init__.py` file and all its submodules will be your various Django apps.
3. Change manage.py to match playdohs manage.py. Specifically, it needs to do some setup and calls `funfactory.manage.setup_environ(...)` for that.
4. In `yourapp/settings/base.py` add to the top:

   ```
   from funfactory.settings_base import *
   ```

5. In `yourapp/settings/base.py` change:

   ```
   INSTALLED_APPS = (...)
   ```

   to:

   ```
   INSTALLED_APPS = get_apps(append=(...))
   ```

   Do the same for any other lists which have been customized. This will ensure that you inherit the default funfactory settings.

See the *settings management API* for information about funfactory's built-in settings management functions, including `get_apps`.

6. You can remove any redundant settings in `settings.py` if they appear in `funfactory.settings_base`.

Mind those monkey patches

As of April 2012, every playdoh project needs to invoke the collected monkey patches in funfactory. By doing this explicitly in each project (as opposed to relying on `funfactory/model.py` to do it automatically) we make sure monkey patches are safely executed both when running `runserver` and when running as a WSGI app.

See the *Monkey patches* documentation for the lines you need to add to your root urls.py if you don't already have them.

Manual upgrade

All of Playdohs core is encapsulated by the funfactory module which is a git submodule in the vendor directory or a python package installed in a virtualenv if you choose. You can upgrade your app by upgrading that module.

Recently forked app

If you have a recent fork of playdoh, you can probably safely merge changes from the master branch of playdoh.

Note: "Recently" is subjective. Generally if you've made any commits, you're probably better off doing a *Manual upgrade*.

First, make sure you have a reference to the main playdoh repo:

```
git remote add playdoh git@github.com:mozilla/playdoh.git
```

1. pull and merge with your master branch:

```
git checkout master
git pull playdoh master
```

2. Recursively update the vendor submodules to pull in any new or updated third party Python modules:

```
git submodule update --init
pushd vendor
git submodule sync
git submodule update --init
popd
```

3. Take a look at `project/settings/local.py-dist` to see if there are new settings you need in your own `yourapp/settings/local.py`
4. Run `pip install -r requirements/compiled.txt` in case there are new requirements.

Upgrading an old Playdoh fork

Note: Thank you for being an early adopter! Muhuhahaha.

The Playdoh apps layout was majorly refactored in Jan 2012 as part of Pull 67. Instead of having a directory called `apps` that contains separate Python modules there is now one top level package called `project` or whatever you choose to name it. For each individual Django app therein, you'll now refer to it as a submodule, like `project.users`, `project.payments`, etc. It's also no longer possible to run your root directory as a Python module. That is, the `__init__.py` file was removed.

Upgrading from jingo-minify to django-compressor

django-compressor is the new default and recommended tool for managing static assets. The old used to be jingo-minify and the difference is pretty big. funfactory attempts to set all the ideal settings for using django-compressor for you but you still have to significantly change how you reference things.

With *jingo-minify* you would do:

```
# in settings/base.py
MINIFY_BUNDLES = {
  'css': {
    'common': ('css/main.less', 'css/plugins.css'),
    ...
  'js': {
    'myapp.home': ('js/libs/jquery.js', 'js/home.js'),
    ...

# in base.html
<html>
<head>
{{ css('common') }}

# in myapp/templates/myapp/html.html
{{ js('myapp.home') }}
</body>
</html>
```

Now, with *django-compressor* instead you do this:

```
# in settings/base.py
# Nothing!

# in base.html
{% compress css %}
<link type="text/less" href="{{ static("css/main.less") }}">
<link href="{{ static("css/plugins.css") }}">
{% endcompress %}

# in myapp/templates/myapp/html.html
{% compress js %}
<script src="{{ static("js/libs/jquery.js") }}"></script>
<script src="{{ static("myapp/js/home.js") }}"></script>
{% endcompress %}
```

Since we're now using *django.contrib.staticfiles* you can place app specific static assets together with apps. So instead of:

```
media/js/myapp/home.js
```

it's now:

```
project/myapp/static/myapp/js/home.js
```

During development you don't want to rely on having to run *collectstatic* every time you edit a static file. So what you do is you add this to your root *urls.py* file:

```
from django.contrib.staticfiles.urls import staticfiles_urlpatterns
if settings.DEBUG:
    urlpatterns += staticfiles_urlpatterns()
```

Most playdoh based projects have something similar in their root *urls.py* but now that needs to change to this.

Note: Since *collectstatic* is already part of the upgrade scripts in playdoh, you now don't need to run *compress_assets* any more.

A note about less. *django-compressor* automatically recognizes *<link>* tags with *type="text/less"* and it will try to convert these by executing the *lessc* command and assuming it's on your *PATH*. To override this see the django-compressor settings documentation

Vagrant (Virtualization, not emulation!)

playdoh comes with out-of-the-box support for running your web app in a virtual machine. Basically, this means that you can easily get started developing a playdoh app on your local machine without having to futz with virtualenvs, dependencies, compiling things, or lots of other annoying things that make it a pain to hit the ground running when you start a new app.

How Does it Work?

playdoh uses VirtualBox, Vagrant, and puppet to do its magic. Vagrant allows for us to easily customize and access our virtual machines; puppet lets us run custom commands (like installing MySQL or setting up databases); and VirtualBox runs our created VMs.

The virtual machine mounts your app's directory inside its home directory so the changes you make on your host machine are reflected instantly inside your VM without any weird FTPing or other such barbaric nonsense.

It just works!

How Do I Get Started?

You'll need VirtualBox and vagrant to get started. Head over to the VirtualBox downloads page and get the latest version of VirtualBox for your host operating system (if you run Mac OS X, select the Mac version). Don't worry about guest operating system; vagrant take care of that for you.

Next, install vagrant. Go to the Vagrant downloads page and get the latest version of vagrant for your operating system.

That's it! You're ready to boot the playdoh VM and access your webapp from inside it.

How Do I Boot and Access My VM?

If you don't yet have a project directory, you'll want to recursively clone the playdoh repository:

```
git clone https://github.com/mozilla/playdoh.git my_project_directory --recursive
```

Then, enter the directory and tell vagrant to spin up a VM:

```
cd ~/my_project_directory
vagrant up
```

Done! That's all it takes for your VM to be created and boot! If you're running this for the first time, it will take awhile for your base VM image to download and for puppet to install all the necessary packages, so it might be wise to get a coffee or watch a YouTube video.

Once your VM is booted, just run `vagrant ssh` to ssh into your VM. `cd` into your projects directly and run whatever manage.py commands you like.

playdoh's vagrant setup takes the liberty of forwarding port 8000 (the usual Django development port), so if you want to access your web app, do the following (after using `vagrant ssh` to get into your VM):

```
cd project
./manage.py runserver 0.0.0.0:8000
```

Note that you'll need to explicitly set the host and port for runserver to be accessible from outside the VM.

Learn more about Vagrant by checking out its docs and enjoy not caring about which libraries are installed on your system anymore!

User Guide

Features

This is a **work in progress** feature list of Playdoh.

For a list of useful libraries (bundled with playdoh or not), check out *libraries*.

The base: Django

At the time of writing, Playdoh is based on Django 1.4.5.

Enhancements:

- jinja2 instead of Django's built-in templating system
- some helper utils called jingo to tie it into Django.

Scalability

Playdoh's enhancements to raise django apps' scalability:

- jingo-minify for bundling and minifying CSS and JS assets.

Security

"Secure by default" policy. Security enhancements applied:

- `X-Frame-Options: Deny` (part of commonware) set on all responses unless opted out per response.

- Stronger password hashing for Django's built-in auth system. Default: *sha512*. Recommendation: *bcrypt + HMAC*.
- `secure=True` and `httponly=True` *enabled by default* on all cookies set through django's cookie facility, opt-out possible by cookie. (part of commonware).
- Greatly reduced the need for the use of *|safe* in templates, to minimize opportunities for XSS vulnerabilities. The `|fe()` helper is part of jingo, and django_safeforms is a nugget.
- bleach library bundled for secure-by-default, but heavily customizable HTML sanitization of user input.
- Used django-session-csrf to replace Django's built-in, cookie-based CSRF method with a common, session-based method. This mitigates the risk of cookie forging attacks.

Localization

Advanced Localization (L10n) tool chain, focusing on localizable web apps by default.

Tools and enhancements:

- *jinja2*'s integrated L10n extension based on Babel.
- Enhanced string extraction tools and template tags through tower.
- LocaleURLMiddleware, detecting user's preferred content locale and sticking it into the URL: example.com/*en-US*/stuff.

Testing

Django's built-in test framework. Enhancements:

- django-nose, a test runner that uses nose.

Libraries

There are a number of libraries that either come bundled with playdoh or are otherwise useful for it. The list is incomplete.

For a full list of enhancements enabled by default, check out the *feature list*.

Note: Libraries marked with an *asterisk* are bundled with playdoh by default.

Python

Database

- django-multidb-router*: Round-robin master/slave db router for Django 1.2.
- South: Database migration tool.

Deferred Execution (cron, message queues)

- django-celery*: Celery integration for Django.
- django-cronjobs*: A simple cron-running management command for Django.
- django-gearman: A convenience wrapper for Gearman clients and workers in Django/Python.

Deployment

- django-waffle: A feature flipper for Django.
- django-arecibo*: Track the errors on your website in a database.

Internationalization (i18n) and Localization (L10n)

- Babel*: A collection of tools for internationalizing Python applications.
- pytz: World Timezone Definitions for Python.
- tower*: A library that builds on Babel and Jinja2 to make extracting strings easy and uniform.
- The Translate Toolkit*: Tools for working between translation formats.

Middleware

- commonware*: Middlewares and other stuff we share.
- django-mobility*: Middleware and decorators for directing users to your mobile site.

Mozilla

- django-moz-header: Common header/footer templates and CSS for Django-based Mozilla sites.
- django-mozilla-product-details*: Pulls Mozilla product details library data from SVN and makes it available as Python dictionaries.

Security and Data Sanitization

- bleach*: An easy, HTML5, whitelisting HTML sanitizer.
- django-sha2*: Monkey-patches strong password hashing support into Django.
- happyforms: Extension to Django Forms that strips spaces.
- django-session-csrf*: Replaces Django's cookie-based CSRF method with a session-based one, in order to mitigate certain cookie-forging attacks.

Templates and Caching

- django-cache-machine: Automatic caching and invalidation for Django models through the ORM.
- jingo*: An adapter for using Jinja2 templates with Django.
- jingo-minify*: Concatenate and minify JS and CSS for Jinja2 + Jingo + Django.
- hera: Client for Zeus Traffic Manager SOAP API.

Testing

- django-fixture-magic: Utilities to extract and manipulate Django fixtures.
- django-nose*: Django test runner using *nose*.
- nose*: *nose* extends unittest to make testing easier.
- test-utils*: A grab-bag of testing utilities that are pretty specific to our Django, Jinja2, and nose setup.

Various

- nuggets*: Little utilities that don't deserve a package.

JavaScript

- jQuery*: The jQuery JavaScript library.

Asynchronous Tasks (Celery)

One goal in developing web projects is to minimize request times. Long request times lead to users leaving your website and ultimately a loss of revenue and mindshare. Smaller request times lead to huger bags of money that you can take to the bank.

If you can separate things that need to happen immediately (e.g. sending a message, setting a privacy setting) versus things that can take some time (e.g. deleting a tag, updating a search index, hitting an external API) you can take advantage of a queue.

Playdoh includes Celery for managing asynchronous task queues. Celery is one of the more popular task queue systems for Django.

In production Celery uses RabbitMQ (or some other backend) to store tasks that you want to do later.

While developing locally, Playdoh defaults to:

```
CELERY_ALWAYS_EAGER = True
```

This causes your Celery tasks to happen during the request-response cycle, which is great for development.

You can place your tasks in `tasks.py` and decorate them like so:

```
from celery.task import task


@task
def my_awesome_task():
    time.sleep(100000)
```

You can get more advanced with the decorator so read the docs.

In your code (often in a view or in the model) you can call the task by doing:

```
my_awesome_task.delay(arg1, arg2, kwarg1="x", kwarg2="y")
```

`task.apply_async` can also be used, but has a more complicated syntax. See Executing Tasks.

See also:

- Celery Docs
- Django Celery Docs
- Queue everything and delight everyone
- Example settings/local.py

Database Migrations

Please use South.

The documentation is pretty good. Before you start using South, make sure to read through the South tutorial.

Localization (L10n)

Playdoh comes with all the libraries and tools you will need in production or development for localization.

Localization works by searching for localizable text in your source code and extracting (or merging them) into *.po* files in *locale/*.

Note: All new locales must be whitelisted before they show up in production. See *Creating New Locales* for details.

Requirements

The one set of low level tools not provided is `gettext`.

```
# for ubuntu/debian
aptitude install gettext

# for mac
brew install gettext
```

Testing It Out, Right Now

Playdoh comes with a sample app called `examples` which is also localized into French. This is so that you can get the hang of how things work out of the box. To get started, simply follow these quick steps:

1. `./bin/compile-mo.sh locale/`
2. Set `DEV=True` in `settings/local.py`
3. `./manage.py runserver`
4. Point your browser to `http://127.0.0.1:8000/fr`

Installation

If you don't have any localization you need to do the following:

1. `./manage.py extract -c`
2. `./manage.py merge -c`

This will populate your `locale/` directory.

Usage

Once your directories are set up you'll need to tend to your strings and their localization.

Strings

You can localize strings both in python code as well as Jinja templates.

In python:

```
from tower import ugettext as _, ugettext_lazy as _lazy

yodawg = _lazy('The Internet')

def myview(request):
    return render('template.html', {msg=_('Hello World'), msg2=yodawg})
```

`_lazy` is used when we are not in scope of a request. This lets us evaluate a string, such as `yodawg`, at the last possible second when we finally can draw upon the request's context. E.g. in a template:

```
{{ msg2 }}
```

Will be evaluated to whatever `The Internet` is in the requester's locale.

In Jinja we can use the following syntax for localized strings:

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

{% trans link='http://mozilla.org' %}
<p>Go to this <a href="{{ link }}">site</a>.</p>
{% endtrans %}
```

Updating our strings

If you make changes to your strings you can update them:

```
./manage.py extract
./manage.py merge
```

Note: you do not need `-c` for either command if you've created the required directories.

If you want to see these changes in your project:

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

Creating New Locales

In your project's `settings.py` add the new locale to the `PROD_LANGUAGES` tuple. During development (when `settings.DEV` is True) all locales will show up if their po files exist but in production, locales will only show up if they exist in `PROD_LANGUAGES` *and* their po files exist.

For example, make Polish and Brazilian Portuguese show up in production when each locale is fully ready to go live like this:

```
PROD_LANGUAGES = (
    'en-US',
    'pl',
    'pt-BR',
)
```

Then run `./manage.py merge -c` which will create directories for any locales that are missing in `/locale`.

If you want your visible locales in development to match that of production, you can set dev to use the same whitelist, like this in settings:

```
DEV_LANGUAGES = PROD_LANGUAGES
```

Good Practices

Let's say you have some template:

```
<h1>Hello</h1>

<p>Is it <a href="http://about.me/lionel.richie">me</a> you're looking for?</p>
```

Let's say you are told to translate this. You could do the following:

```
{% trans %}
<h1>Hello</h1>

<p>Is it <a href="http://about.me/yo">me</a> you're looking for?</p>
{% endtrans %}
```

This has a few problems, however:

- It forces every localizer to mimic your HTML, potentially breaking it.
- If you decide to change the HTML, you need to either update your `.po` files or buy all your localizers a nice gift because of all the pain you're inflicting upon them.
- If the URL changes, your localizer has to update everything.

Here's an alternative:

```
<h1>_('Hello')</h1>

<p>
  {% trans about_url='http://about.me/lionel.richie' %}
    Is it <a href="{{ about_url }}">me</a> you're looking for?
  {% endtrans %}
</p>
```

or if you have multiple paragraphs:

```
<h1>_('Hello')</h1>

{% trans about_url='http://about.me/lionel.richie' %}
  <p>
    Is it <a href="{{ about_url }}">me</a> you're looking for?
  </p>
  <p>
    I can see it in your eyes.
  </p>
{% endtrans %}
```

Here are the advantages:

- Localizers have to do minimal HTML.
- The links and even structure of the document can change, but the localizations can stay put.

Be mindful of work that localizers will have to do.

Support for Mobile Devices

Playdoh enables you to create Django apps that run on mobile devices using the django-mobility middleware which negotiates between user agents. In your templates, you can check if you're in mobile like this:

```
{% if request.MOBILE %}
  {{ _('This text is only visible on mobile devices.') }}
{% endif %}
```

In your views module you can use the mobility decorator to toggle between templates:

```
from mobility.decorators import mobile_template

@mobile_template('examples/{mobile/}home.html')
def home(request, template=None):
    # ...
    return jingo.render(request, template, data)
```

See the django-mobility documentation for more details about setting up your backend.

For frontend mobile strategies like media queries and other alternatives, check out Approaches to Mobile Web Development followed by Part 2 (Separate Sites) and Part 3 (Responsive Design).

To develop mobile sites you can simulate various headers using the User Agent Switcher for Firefox.

Logging

Logging is a great way to track what is going on with your app.

This makes it easier for other developers to debug what's going on without resorting to `print` statements or `pdb` calls. It also makes it easier to diagnose failures in production.

Playdoh supports logging out of the box.

Note: This is similar to Django Logging, but we make it easier to setup loggers and handlers.

How to log

Logging is provided via commonware.

Within our webapps we typically use a small namespace prefix for logging:

```
m: mozillians
k: kitsune
i: input
z: zamboni
```

When we log in our app we can do the following:

```
import commonware.log

log = commonware.log.getLogger('i.myapp')


def my_view(request):
    log.info('%s got here' % request.user)
    pass
```

Anytime someone goes to *myapp.views.my_view* a log entry at level *INFO* will be made.

In development this shows up as standard output.

Silence Logging

Sometimes logging can be noisy:

```
elasticsearch: DEBUG: Search disabled for <function add_to_index at 0x102e58848>.
```

Commonware as configured in playdoh, allows you to configure logging via a `dict` in your *settings* (preferrably *settings/local.py*):

```
error = dict(level=logging.ERROR)
info = dict(level=logging.INFO)

LOGGING = {
    'loggers': {
        'product_details': error,
        'nose.plugins.manager': error,
        'django.db.backends': error,
        'elasticsearch': info,
    },
}
```

This configuration says:

- Only tell me if there are errors with any logs in the *product_details*, *nose.plugins.manager* and *django.db.backends* namespace
- Only tell me if there are *INFO*, *WARNING* or *ERROR* messages with ElasticSearch.

Template Context

Playdoh adds some context processors that add variables to all of your templates. These are defined in funfactory. Other third party libraries might also add some context processors; see *each library*'s docs for more info.

`{{ LANGUAGES }}` Dictionary of all available languages, such as `{'en-us': 'English (US)'}`.

`{{ LANG }}` The currently active language code, such as `en-us`.

{{ DIR }} `ltr` if the activate language read lefts to right otherwise `rtl` (for example: Arabic, Hebrew, etc are right to left).

{{ request }} The request object that was passed to the view function.

{{ settings }} The settings object for the current application.

Errors

Mailing errors

By default `log_settings.py` mails out errors to anybody listed in `settings.ADMINS`.

Arecibo

Optionally, you can use Arecibo to record your errors. This can help stop an email flood, gives you a URL for a traceback and statistics. To add-in Arecibo, add the destination server to your *settings/local.py*.

If are developing a site for Mozilla, you can use an existing Arecibo server, otherwise you'll need to set one up.

For example in *settings/local.py*:

```
ARECIBO_SERVER_URL = 'http://amckay-arecibo.khan.mozilla.org'
```

Finally, please ensure *settings_test.py* is set to the default:

```
ARECIBO_SERVER_URL = ''
```

`log_settings.py` will use `funfactory.log.AreciboHandler` if Celery is setup to ping Arecibo.

See also:

Further information on django_arecibo.

Settings Management API

Funfactory's default settings module provides some useful helper functions that make managing your playdoh app's settings easier.

- `get_apps(exclude=(), append=())`

  The `get_apps` function returns the current INSTALLED_APPS tuple, modified based on the `exclude` and `append` parameters, which can be tuples or lists. INSTALLED_APPS originates in funfactory's `settings_base.py` file, containing the default suite of funfactory apps, but can be modified using this function to add or remove apps as necessary for your project.

  The state of the INSTALLED_APPS tuple returned by this function persists so that you can modify it as necessary for your app in `settings/base.py` and then further modify it for your local install in `settings/local.py`.

  Example:

```
INSTALLED_APPS = get_apps(
    exclude=('cronjobs', 'djcelery'),
    append=('debug_toolbar',)
)
```

- `get_middleware(exclude=(), append=())`

  Similar to `get_apps`, this function returns the current MIDDLEWARE_CLASSES tuple and persists its state through the various settings files.

- `get_template_context_processors(exclude=(), append=())`

  Again, similar to `get_apps` but returns the current TEMPLATE_CONTEXT_PROCESSORS tuple and persists its state through the various settings files.

Maintaining playdoh, playdoh-lib, playdoh-docs and funfactory

This section of the docs covers maintaining all the pieces that make up playdoh and funfactory. This is for contributors to these code bases.

Maintenance process

General process for how things change:

1. write up an issue in the Playdoh issue tracker
2. discuss the issue until a consensus for solution is arrived at
3. do the work—make sure to add tests where possible
4. create a pull request adding a `r?` at the end of the description indicating the pull request is ready for review
5. review happens

   This can take a while and go back and forth and involve multiple people. At some point the pull request is approved.
6. the reviewer who `r+` the pull request presses the Merge button
7. for playdoh-lib changes and any non-trivial changes for the other repos, an email should go out to dev-webdev mailing list. See *Getting help and contacting playdoh devs* for details on the mailing list.
8. Victory dance!

Maintaining libs in playdoh-lib

Adding a new library via git submodules

To add a library via git submodules, you must first know the git url to clone. Once you know that, then you do:

1. `git submodule add <GIT-URL> src/<REPO>`
2. Add a line to `vendor.pth`. Note that this file is sorted alphabetically.
3. `git commit vendor.pth .gitmodules src/<REPO>`

Updating a library via git submodules

To update a library via git submodules:

1. `cd src/<REPO>`
2. `git fetch --all -p`
3. `git checkout <REVISH>`
4. `cd ..`
5. Do a `git log --oneline <FROM>..<TO>` for the library and copy/paste that output into the commit message so that playdoh-lib users can see from the playdoh-lib log what's changed.
6. `git commit -a`

Adding a new library via pip

From the playdoh-lib root:

```
pip install -I --install-option="--home=`pwd`" <LIB>
```

where `LIB` is any pip-happy library spec.

After you do this, make sure that any scripts that get added to `bin/` have a header like this:

```
#!/usr/bin/env python
```

Then `git add` all the new files and commit.

Updating a library via pip

First, delete the library files:

```
cd lib/python/
git rm -rf <LIB>
```

Note the `git` part. This removes all the files from the git repository.

After you do that, follow the instructions in **Adding a library via pip** to add the update.

Dealing with binary dependencies

Binary dependencies can't be checked into playdoh-lib repository. Instead, they require a change in the funfactory repository.

Add a line to `funfactory/funfactory/requirements/compiled.txt`.

Dealing with source dependencies

When you install/upgrade a library via git submodules, you have to make sure the minimum required versions of dependencies are all accounted for in either git submodules or `lib/`.

When you install/upgrade a library via pip, it'll install the dependencies for you into `lib/`.

Testing

Testing playdoh-lib changes

Will tests playdoh-lib changes by pushing the updates to his github clone, then adding that github clone to one of his projects, fetching the changes and running the tests in his project.

Testing funfactory changes

funfactory has two sides:

1. **funfactory the library**

   You can add your funfactory clone as a remote to your project, get the changes and run your project's tests.

2. **funfactory the playdoh project builder**

   You can run the tests. funfactory uses tox and the instructions are in `README.rst` in the funfactory repository.

Depending on the changes you're making it probably makes sense to do both of these.

Testing playdoh changes

To test template changes, you want to run the funfactory tests with the `FF_PLAYDOH_REMOTE` and `FF_PLAYDOH_BRANCH` environment settings set.

For more details, see the `README.rst` in the funfactory repository.

You will also probably want to create a playdoh project with your changes and make sure they work correctly.

pip and friends: Packaging

(largely borrowed from Zamboni)

Your app will depend on lots of tasty open source Python libraries. The list of all your dependencies should exist in several places:

1. requirements/prod.txt
2. As a submodule of vendor or vendor-local

Ultimately your app code will run against the libraries under vendor/vendor-local via mod_wsgi.

Why requirements? For development, you can use virtualenvs and pip to have a tidy self-contained environment. If run from the Django runserver command, you don't even need a web server.

The vendor library

The /vendor library is supposed to contain all packages and repositories. It enables the project to be deployed as one package onto many machines, without relying on PyPI-based installations on each target machine.

By default, the vendor lib is checked out as a *git submodule* under vendor/. If you *do* need to check it out separately, do:

```
git clone --recursive git://github.com/mozilla/playdoh-lib.git ./vendor
```

Once the playdoh-lib repo has been downloaded to /vendor, you only need to install the **compiled** packages (as defined in requirements/compiled.txt). These can come from your system package manager or from:

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

The vendor-local library

The /vendor-local directory is laid out exactly like vendor and works the same way. All of your custom pure Python dependencies should go here so that you can freely merge with playdoh's vendor directory if necessary. See the section on adding new packages below.

Global vs. local library

Playdoh provides its default library in the vendor/ directory. You *may* fork and change it, but that will make it hard to pull updates from the upstream library later.

If you want to make only a few **local additions** or override some of the libs in vendor/, make those changes to the directory vendor-local/ instead, which (in manage.py) is given precedence over playdoh's vendor dir.

compiled.txt vs prod.txt

If a Python library requires compilation, it should be recorded in compiled.txt. These aren't as portable and cannot be shipped in the vendor library. For local development, it's nice to pip install these into a virtualenv. A common practise is to use virtualenv **only** for compiled libraries and vendor for the rest of your dependencies.

Adding new packages

If we wanted to add a new dependency called cheeseballs to playdoh, you would add it to requirements/prod.txt. If your library isn't used in production, then put it in requirements/dev.txt. This makes it available to users installing into virtualenvs.

We also need to add the new package to the vendor-local lib, since that is what runs in production...

First, we need to add the source. There are two ways, depending on how this project is hosted:

Non-git based repos (hg, CVS, tarball)

For such repos or for packages coming from PyPI, do:

```
pip install -I --install-option="--home=`pwd`/vendor-local" cheeseballs
cd vendor-local
git add lib/python/cheeseballs
git commit
```

Optionally, if the package installs executables add them too. For example:

```
cd vendor-local
git add bin/cheeseballer
git commit
```

For hg repos that are not on PyPI, they can be installed with pip too but omit the `--home` option and use the `--src` instead. For example:

```
pip install -I --src='vendor-local/src' \
-e hg+http://bitbucket.org/jespern/django-piston@default#egg=django-piston
cd vendor-local
git add src/django-piston
git commit
```

Note: Installed source packages need to be appended to `vendor-local/vendor.pth`. See note below. For example:

```
echo src/django-piston >> vendor-local/vendor.pth
```

git-based repositories

For a git-based package, add it as a git submodule:

```
cd vendor-local
git submodule add git://github.com/mozilla/cheeseballs.git src/cheeseballs
git commit vendor.pth .gitmodules src/cheeseballs
```

Further, you then need to update `vendor-local/vendor.pth`. Python uses `.pth` files to dynamically add directories to `sys.path` (docs).

The file format is simple. Consult `vendor/vendor.pth` for reference.

Some packages (like `html5lib` and `selenium`) are troublesome, because their source lives inside an extra subdirectory `src/` inside their checkout. So they need to be sourced with `src/html5lib/src`, for example. Hopefully you won't hit any snags like that.

Done. Try `./manage.py shell` and then `import cheeseballs` to make sure it worked.

Testing Your Vendor Change

It's critical that you test your app running under mod_wsgi. Although you may use runserver day to day, go ahead and run some code through WSGI to prove vendor is setup properly. (throw an import into your view, etc)

Advanced Topics

TODO [automate these instructions](<https://github.com/mozilla/playdoh/issues/30)

Initial creation of the vendor library

The vendor repo was seeded with

```
pip install -I --install-option="--home=`pwd`/vendor" --src='vendor/src' -r␣
↪requirements/dev.txt

# ..delete some junk from vendor/lib/python...

# Create the .pth file so Python can find our src libs.
find src -type d -depth 1 >> vendor.pth

# Add all the submodules.
for f in src/*; do
    pushd $f >/dev/null && REPO=$(git config remote.origin.url) && popd > /dev/null &&
↪ git submodule add $REPO $f
done
git add .
```

Adding lots of git submodules

As noted in *Adding new packages*, git-based packages are *git submodules* inside the vendor library. To set up the first batch of submodules, something like the following happened:

```
for f in src/*
    pushd $f && REPO=$(git config remote.origin.url) && popd && git submodule add
↪$REPO $f
```

For reference: pip

The classical method of installing is using pip. We have our packages separated into three files:

requirements/compiled.txt All packages that require (or go faster with) compilation. These can't be distributed cross-platform, so they need to be installed through your system's package manager or pip.

requirements/prod.txt The minimal set of packages you need to run zamboni in production. You also need to get `requirements/compiled.txt`.

requirements/dev.txt All the packages needed for running tests and development servers. This automatically includes `requirements/prod.txt`.

With pip, you can get a development environment with:

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

Operations

Care and feeding of a Playdoh-based app.

Web Server

Apps are typically run under Apache and mod_wsgi in production. Entry point:

```
wsgi/playdoh.wsgi
```

(or whatever you rename it to...)

Developers can set that up or run in stand-alone mode:

```
./manage.py runserver 0.0.0.0:8000
```

It is critical that developers run the app at least once via mod_wsgi before certifying an app as **stage ready**.

Apache

This is a typical virtualhost directive being used in production:

```
<VirtualHost *:80>
    ServerName %HOSTNAME%

    Alias /media %APP_PATH/media

    WSGIScriptAlias / %APP_PATH/wsgi/playdoh.wsgi
    WSGIDaemonProcess playdoh processes=16 threads=1 display-name=playdoh
    WSGIProcessGroup playdoh
</VirtualHost>
```

gunicorn

Totally optional and only for the cool kids.

A lighter weight method of testing your mod_wsgi setup is by using gunicorn.

One time setup:

```
pip install gunicorn
ln -s wsgi/playdoh.wsgi wsgi/playdoh.py
```

Each Time:

```
touch wsgi/__init__.py
gunicorn wsgi/playdoh:application
```

Middleware Caching

TODO (memcache, redis)

Frontend Caching

Apps are typically run behind a Zeus load balancer.

Database

Apps typically use a MySQL database.

Message Queue

Playdoh comes packaged with celery and works well with RabbitMQ.

Updating your Environment

You can run `update_site.py` to keep your app current. It does the following:

- Updates source
- Updates vendor libraries
- Runs Database Migrations
- Builds JS and CSS

```
./bin/update_site.py -e dev
./bin/update_site.py -e stage
./bin/update_site.py -e prod
```

You may pass a `-v` and update_site will explain what commands as it runs them.

If there is an error on any step, the script stops.

IT will typically put `bin/update_site.py` into a cron for auto-deployment to stage environments.

Edit your copy to customize your branching and/or release practices.

Best Practices

This page lists several best practices for writing secure web applications with playdoh.

`|safe` considered harmful

Using something like `mystring|safe` in a template will prevent Jinja2 from auto-escaping it. Sadly, this requires us to be really sure that `mystring` is not raw, user-entered data. Otherwise we introduce an XSS vulnerability.

We have therefore eliminated the need for `|safe` for localized strings. This works:

```
{{ _('Hello <strong>world</strong>!') }}
```

String interpolation

When you *interpolate* data into such a string, however, the resulting output will lose its "safeness" and be escaped again. To mark the *localized part* of an interpolated string as safe, do not use `|f(...)|safe`. The data could be unsafe. Instead, use the helper `|fe(...)`. It will escape all its arguments before doing string interpolation, then return HTML that's safe to use:

```
{{ _('Welcome back, <strong>{username}</strong>!')|fe(username=user.display_name) }}
```

`|f(...)|safe` is to be considered unsafe and **should not pass code review**.

If you interpolate into a base string that does *not contain HTML*, you may keep on using `|f(...)` without `|safe`, of course, as the auto-escaping won't harm anything:

```
{{ _('Author name: {author}')|f(author=user.display_name) }}
```

Form fields

Jinja2, unlike Django templates, by default does not consider Django forms "safe" to display. Thus, you'd use something like `{{ form.myfield|safe }}`.

In order to minimize the use of `|safe` (and thus possible unsafe uses of it), playdoh monkey-patches the Django forms framework so that form fields' HTML representations are considered safe by Jinja2 as well. Therefore, the following works as expected:

```
{{ form.myfield }}
```

Mmmmh, Cookies

Django's default way of setting a cookie is set_cookie on the HTTP response. Unfortunately, both **secure** cookies (i.e., HTTPS-only) and **httponly** (i.e., cookies not readable by JavaScript, if the browser supports it) are disabled by default.

To be secure by default, we use commonware's `cookies` app. It makes secure and httponly cookies the default, unless specifically requested otherwise.

To disable either of these patches, set `COOKIES_SECURE = False` or `COOKIES_HTTPONLY = False` in `settings.py`.

You can exempt any cookie by passing `secure=False` or `httponly=False` to the `set_cookie` call, respectively:

```
response.set_cookie('hello', value='world', secure=False, httponly=False)
```

Content Security Policy (CSP) compliance

Content Security Policy is a web-security-related proposal put forward by Mozilla Security. It is currently a W3C working draft.

Its primary goal is to **mitigate cross-site scripting (XSS) risk** by enforcing certain policies on a web site.

While the proposal is still subject to change there are certain best practices that should be implemented today, because they prepare web applications for CSP compliance and should already be considered standard practices, even if a project does not enforce CSP yet:

Avoid inline CSS and JS

Avoid the use of inline CSS and JavaScript inside your HTML documents. This includes:

Replace the following methods of writing **CSS** with a separate file:

- `<style>` elements
- `style` attributes on HTML elements

Replace the following methods of writing **JavaScript** with a separate file:

- `<script>` elements that wrap inline code

- `javascript:` URIs
- event-handling HTML attributes (like `onclick`)

Do not create code from strings

In JavaScript, never create **code from strings**, including calls to:

- `eval()`
- `new Function()` constructor
- `setTimeout()` called with a non-callable argument
- `setInterval()` called with a non-callable argument

CSRF-protect your forms

Django comes with a built-in, cookie-based CSRF protection facility. Sadly, the integrity of cookies can be compromised under certain circumstances (through Flash, or across subdomains on the same domain), so we replaced the CSRF method with a session-based method (as is common across web frameworks).

To CSRF-protect a form for logged-in users, just add this to your template, inside the `<form>` tag:

```
{{ csrf() }}
```

To make this work for anonymous users, with a light-weight session stored in Django's cache, decorate a view with `@anonymous_csrf`:

```
from session_csrf import anonymous_csrf

@anonymous_csrf
def login(request):
    ...
```

If a form is supposed to be CSRF-protected for logged-in users, but not for anonymous users, use the `@anonymous_csrf_exempt` decorator:

```
from session_csrf import anonymous_csrf_exempt

@anonymous_csrf_exempt
def protected_in_another_way(request):
    ...
```

Finally, to disable CSRF protection on a form altogether (if you know what you're doing!), Django's `csrf_exempt` decorator still works as expected.

To learn more about this method, refer to the django-session-csrf README.

CEF (Common Event Format) logging

Playdoh is set up for Common Event Format (CEF) logging. CEF is a unified logging format for security-relevant events and can be used by ArcSight and similar applications.

For example, to log a user resetting their password, you would do something like this:

```
import logging
from funfactory.log import log_cef

def pw_reset(request, user):
    log_cef('Password Reset', logging.INFO, request, username=user.username,
            signature='PASSWORDRESET', msg='User requested password reset')
```

For more information about logging and suggestions on what kinds of events to log, refer to the Mozilla Security Wiki.

Monkey patches

Unfortunately we have to rely on a set of monkey patches. These are loaded from funfactory.

This is still something you have to invoke from your playdoh project. The best place to do this is from the root `urls.py` file. The two lines you need are:

```
from funfactory.monkeypatches import patch
patch()
```

This should ideally happen before the `urlpatterns` is set up and it's demonstrated in the default project urls.py.

Note: We used to rely on `funfactory.models` to automatically load and run `funfactory.monkeypatches.patch()` but this is not reliable when running as a WSGI app.

Ideally all monkey patches will one day go away. Either by individual apps getting smarter or Django getting smarter.

The reason the monkey patches are in funfactory is because it is assumed that all playdoh instances will need the monkey patches and therefore it's a convenience to put this into funfactory. Most of the monkey patches are unobtrusive meaning they are not applied unless the relevant app is in your `INSTALLED_APPS`.

Note: Because monkey patching is evil and can cause untoward side effects, we log a `debug` message. Hopefully this will remind you when you get an odd traceback, to see if perhaps this monkey patching was to blame.

Troubleshooting

Python namespace collisions

When naming your top-level project (i.e., the directory you're checking your code out into), keep in mind that this will **become a Python module**, so name it accordingly and don't use spaces or funny characters.

Make sure that your project directory is called **differently than**:

- any Python builtin (`code`, `email`, ...)
- any module on your Python path (such as `nose`, `celery`, `redis`)
- any app inside your `apps` directory.

It is a good idea to give your project a concise code name that's somewhat unique to avoid such namespace problems.

Sessions not persisting after authentication?

Be sure to double check your settings file(s) for anything that might not work with localhost (but will work in production). For example, make sure that if you require `SESSION_COOKIE_SECURE` to be set to True in production, set it to False in development (localhost). Or else your sessions will never work properly.

CSRF issues?

A common problem is that `{{ csrf() }}` returns an empty string in your templates.

By default playdoh uses `session_csrf` instead of Django's default CSRF framework. One common mistake when going from standard Django CSRF to `session_csrf` is that you now need to decorate all your view functions that expect anonymous users to use it (e.g. the login view). For example:

```
from session_csrf import anonymous_csrf
...

@anonymous_csrf
def login(request):
    ...
```

Note that `session_csrf` is switched on by default in `funfactory`. Therefore, you need to make sure your settings are set up correctly. These are the things you need to have in your settings:

- A working cache backend (e.g. `CACHES` also known as `CACHE_BACKEND`)
- As part of a working cache backend, make sure memcached (or whatever you are using for caching) is running on your machine
- In `TEMPLATE_CONTEXT_PROCESSORS` make sure it contains `session_csrf.context_processor`
- In `MIDDLEWARE_CLASSES` make sure it contains `session_csrf.CsrfMiddleware`

See also:

- https://github.com/mozilla/django-session-csrf/blob/master/session_csrf/__init__.py
- https://github.com/django/django/blob/master/django/template/defaulttags.py#L40

No logging?

If you're sure you set up loggers and handlers in `settings.LOGGING` but you see errors like "no handlers for the.log.name" in your live site, it could mean that your app is not importing `monkeypatches`. Your `project/app/urls.py` should have this line:

```
from funfactory.monkeypatches import patch
patch()
```

This is included with the playdoh skeleton so you'd probably only run into this if you were trying to port an existing app to playdoh or do an upgrade.

If you still don't see logging messages, check that your local settings didn't override `settings.LOGGING`. Also make sure that all loggers have at least `{'handlers': ['console']}`.

Yes, logging will make you feel a world of pain.

Getting help and contacting playdoh devs

IRC:

We hang out on `#webdev` on `irc.mozilla.org`. That's a synchronous chat, so if someone doesn't answer your questions, then it's possible no one is around or the people who are around don't know the answer.

Mailing list:

All security notices and playdoh project updates are sent to the dev-webdev mailing list.

If you use playdoh, it behooves you greatly to join this list. At the time of this writing, the list is pretty low traffic.

Writing up issues:

If you find yourself stuck, write up an issue in github.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search