Spade Documentation

Release 0.1

Sam Liu

Sep 27, 2017

Contents

Overview: Spade is at its core a metrics tool that allows quick visualization of what kinds of CSS properties websites are using. It allows input of a list of websites to scrape, and it crawls to 1 level on each website whilst submitting a variety of user agent strings in order to ascertain different kinds of markup returned as a result. It tries to detect UA sniffing by commparing the returned markup structure of each site.

All the information is recorded into a database after the crawl, and is accessible from a web interface. For more information on installation and use of the tool, please continue through the documentation.

Contents:

CHAPTER 1

Installation

1. Install MySQL.
2. Install the dependencies in `requirements/compiled.txt` either via system-level package manager, or with `pip install -r requirements/compiled.txt` (preferably into a virtualenv in the latter case).
3. If you will need to run the tests or work on Spade development, install the development-only dependencies into your virtualenv with `pip install -r requirements/dev.txt`.
3. Copy `spade/settings/local.sample.py` to `spade/settings/local.py` and modify the settings as appropriate for your installation.
4. Run `./manage.py syncdb` to create the database tables.

Vagrant Setup

1. Run `vagrant up` in a terminal. This will create a new VM that will have Spade running on it. It will run the necessary Puppet scripts
2. add `127.0.0.1 dev.spade.org` to /etc/hosts
3. Navigate to http://dev.spade.org:8000 in your browser

CHAPTER 2

Scraper

Spade comes with a built-in scraper to crawl websites. It crawls all urls given by a text file via command line args, as well as 1-level-deep links within each site. It saves html, css, and javascript from the pages using whatever user agents are specified in the database.

Using the scraper

1. Add user agent strings that you would like to crawl with by running the management command:

```
python manage.py useragents --add "Firefox / 15.0" --desktop
python manage.py useragents --add "Fennec / 15.0" --primary
python manage.py useragents --add "Android / WebKit"
```

   Detecting UA-sniffing issues requires at least three user-agents to be added: a desktop user-agent to be used as baseline, a "primary" mobile user agent (the one we want to make sure sites are sniffing, if they sniff mobile UAs at all), and at least one other mobile UA to check against. A "UA sniffing issue" will be reported for a URL if that URL returns markedly different content for any non-primary mobile UA (compared to the desktop UA content), but returns the desktop content for the primary mobile UA.

2. Call the scrape command, giving it a text file of URLs to parse:

```
python manage.py scrape [newline delimited text file of URLS]
```

CHAPTER 3

Architecture

The primary components are listed below (by Python module path) and described:

spade.controller.management.commands

Contains the `scrape` and `useragents` management commands.

spade.model.models

Contains the database models:

A `UserAgent` stores a user-agent string that will be used to scrape sites the next time the `scrape` management command is run.

A `Batch` represents a single run of the `scrape` management command.

A `BatchUserAgent` stores a user-agent string that actually was used when scraping a particular batch. This is copied from a `UserAgent` when `scrape` is run; the separation prevents future changes to the user-agent list from modifying or corrupting data from past runs.

A `SiteScan` object is created for each top-level URL in the list of URLs given to the `scrape` management command.

A `URLScan` object is created for each URL scanned; this includes the initial top-level URLs, and all linked pages one level deep.

A `URLContent` object stores the scraped contents of a single URL for a particular user agent. In other words, for every `URLScan` there will be N `URLContent` objects, if there are N `UserAgent` records at the time the scrape is initiated.

A `LinkedCSS` contains information about a single linked CSS file. Every CSS file at a distinct URL has only one `LinkedCSS` record, even if it was linked from multiple scraped HTML pages (thus `LinkedCSS` has a many-to-many relationship with `URLContent`).

Similarly, a `LinkedJS` contains information about a single linked JS file.

When the contents of a `LinkedCSS` file are parsed by `spade.utils.cssparser.CSSParser`, a `CSSRule` object is created for every CSS rule in the file, and a `CSSProperty` object for every property in every rule.

The various `*Data` models contain aggregated data about issues detected in the scan.

spade.scraper

A Scrapy scraper that *scrapes a list of given URLs* with all user-agent strings listed in the database, following links one level deep, and saving all response contents (including linked JS and CSS) in the database.

spade.settings

Contains the Django project settings.

spade.tests

Contains the tests.

spade.utils.data_aggregator

Contains a `DataAggregator` class that populates the `BatchData`, `SiteScanData`, `URLScanData`, `URLContentData` and `LinkedCSSData` models with summary aggregate data about the scan.

spade.utils.css_parser

Contains a `CSSParser` class that can take raw CSS, parse it, and store it into the `CSSRule` and `CSSProperty` database models.

spade.utils.html_diff

Contains a `HTMLDiff` class that can compare the tag structure of two chunks of HTML, ignoring differences in tag content and attributes, and return a measure of their similarity (0.0 if they have nothing in common, 1.0 if they are identical).

spade.view.urls

The URL configuration for the site.

Run `python manage.py runserver` to fire up a development web server and view the app in your browser at `http://localhost:8000/`.

spade.view.views

Contains the Django view functions.

CHAPTER 4

Development

Developing spade requires installing the dev-only dependencies:

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

Tests

To run the Python tests, run `./runtests.py`.

Dependencies

To add or change a pure-Python production dependency, add or modify the appropriate line in `requirements/pure.txt`, then run `bin/generate-vendor-lib.py`. You should see the actual code changes in the dependency reflected in `vendor/` if you `git diff`. Commit both the change to `requirements/pure.txt` and the changes in `vendor/`.

To add or change a non-pure-Python production dependency, simply add or modify the appropriate line in `requirements/compiled.txt`.

To add or change a development-only dependency, simply add or modify the appropriate line in `requirements/dev.txt`.

CHAPTER 5

TODO

Add support for detecting CSS prefixing issues

The `DataAggregator` attempts to detect UA-sniffing issues (by comparing markup structures returned from the same URL for different UAs), but it does not attempt to detect prefixed-CSS issues. The data needed for this detection is all present in the database in the `CSSRule` and `CSSProperty` models, but there is no code yet to iterate over those models and look for cases where a non-mozilla prefixed property is used without the moz-prefixed or unprefixed equivalent.

Evaluate adequacy of UA-sniffing-detection method

The scraper follows this algorithm when scraping:

1. Given a top-level URL from the URLs file, issue a request to that URL with each configured user-agent string.
2. From that point on, each user agent effectively crawls the site separately, following the links found in the pages delivered to that user agent.

This gives an accurate picture of the site as each user agent would really see it (which is good for the CSS prefix checking), but in case of redirection to separate mobile sites, it means that there may be very few (or no) URLs on the site that are scraped in common by all user agents. The current form of UA-sniffing detection (looking at markup returned to different UAs for the same URL) is only effective if a site has at least one URL that returned actual content to all user agents. It may be necessary to add more sophisticated UA-sniffing detection code that accounts for different redirects received by different user agents as well.

Integrate South for schema and data migrations

At the moment, since Spade (including the database schema) is still under heavy development, it's often easiest after a model change to simply drop and recreate the database and run syncdb again, rather than worrying about how to structure a migration for existing data.

At some point, Spade will be deployed into production and begin collecting non-throwaway data. Before that happens, South should be integrated so that future model changes can incorporate migrations to alter the schema and migrate data as needed.

Complete the UI

The views in `spade/view/views.py` and the Django templates in `spade/view/templates` are incomplete, and need to be finished.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search