Django Debug Toolbar Documentation

Release 1.8

Django Debug Toolbar developers and contributors

Aug 24, 2017

Contents

CHAPTER 1

Installation

Getting the code

The recommended way to install the Debug Toolbar is via pip:

```
$ pip install django-debug-toolbar
```

If you aren't familiar with pip, you may also obtain a copy of the `debug_toolbar` directory and add it to your Python path.

To test an upcoming release, you can install the in-development version instead with the following command:

```
$ pip install -e git+https://github.com/jazzband/django-debug-toolbar.git#egg=django-
↪debug-toolbar
```

Prerequisites

Make sure that `'django.contrib.staticfiles'` is set up properly and add `'debug_toolbar'` to your `INSTALLED_APPS` setting:

```
INSTALLED_APPS = [
    # ...
    'django.contrib.staticfiles',
    # ...
    'debug_toolbar',
]

STATIC_URL = '/static/'
```

If you're upgrading from a previous version, you should review the *change log* and look for specific upgrade instructions.

URLconf

Add the Debug Toolbar's URLs to your project's URLconf as follows:

```
from django.conf import settings
from django.conf.urls import include, url

if settings.DEBUG:
    import debug_toolbar
    urlpatterns = [
        url(r'^__debug__/', include(debug_toolbar.urls)),
    ] + urlpatterns
```

This example uses the `__debug__` prefix, but you can use any prefix that doesn't clash with your application's URLs. Note the lack of quotes around `debug_toolbar.urls`.

Middleware

The Debug Toolbar is mostly implemented in a middleware. Enable it in your settings module as follows:

```
MIDDLEWARE = [
    # ...
    'debug_toolbar.middleware.DebugToolbarMiddleware',
    # ...
]
```

Old-style middleware:

```
MIDDLEWARE_CLASSES = [
    # ...
    'debug_toolbar.middleware.DebugToolbarMiddleware',
    # ...
]
```

Warning: The order of `MIDDLEWARE` and `MIDDLEWARE_CLASSES` is important. You should include the Debug Toolbar middleware as early as possible in the list. However, it must come after any other middleware that encodes the response's content, such as `GZipMiddleware`.

Internal IPs

The Debug Toolbar is shown only if your IP is listed in the `INTERNAL_IPS` setting. (You can change this logic with the `SHOW_TOOLBAR_CALLBACK` option.) For local development, you should add `'127.0.0.1'` to `INTERNAL_IPS`.

CHAPTER 2

Configuration

The debug toolbar provides two settings that you can add in your project's settings module to customize its behavior.

Note: Do you really need a customized configuration?

The debug toolbar ships with a default configuration that is considered sane for the vast majority of Django projects. Don't copy-paste blindly the default values shown below into you settings module! It's useless and it'll prevent you from taking advantage of better defaults that may be introduced in future releases.

DEBUG_TOOLBAR_PANELS

This setting specifies the full Python path to each panel that you want included in the toolbar. It works like Django's `MIDDLEWARE_CLASSES` setting. The default value is:

```
DEBUG_TOOLBAR_PANELS = [
    'debug_toolbar.panels.versions.VersionsPanel',
    'debug_toolbar.panels.timer.TimerPanel',
    'debug_toolbar.panels.settings.SettingsPanel',
    'debug_toolbar.panels.headers.HeadersPanel',
    'debug_toolbar.panels.request.RequestPanel',
    'debug_toolbar.panels.sql.SQLPanel',
    'debug_toolbar.panels.staticfiles.StaticFilesPanel',
    'debug_toolbar.panels.templates.TemplatesPanel',
    'debug_toolbar.panels.cache.CachePanel',
    'debug_toolbar.panels.signals.SignalsPanel',
    'debug_toolbar.panels.logging.LoggingPanel',
    'debug_toolbar.panels.redirects.RedirectsPanel',
]
```

This setting allows you to:

- add built-in panels that aren't enabled by default,

- add third-party panels,
- remove built-in panels,
- change the order of panels.

DEBUG_TOOLBAR_CONFIG

This dictionary contains all other configuration options. Some apply to the toolbar itself, others are specific to some panels.

Toolbar options

- `DISABLE_PANELS`

  Default: `{'debug_toolbar.panels.redirects.RedirectsPanel'}`

  This setting is a set of the full Python paths to each panel that you want disabled (but still displayed) by default.

- `INSERT_BEFORE`

  Default: `'</body>'`

  The toolbar searches for this string in the HTML and inserts itself just before.

- `JQUERY_URL`

  Default: `'//ajax.googleapis.com/ajax/libs/jquery/2.2.4/jquery.min.js'`

  URL of the copy of jQuery that will be used by the toolbar. Set it to a locally-hosted version of jQuery for offline development. Make it empty to rely on a version of jQuery that already exists on every page of your site.

- `RENDER_PANELS`

  Default: `None`

  If set to `False`, the debug toolbar will keep the contents of panels in memory on the server and load them on demand. If set to `True`, it will render panels inside every page. This may slow down page rendering but it's required on multi-process servers, for example if you deploy the toolbar in production (which isn't recommended).

  The default value of `None` tells the toolbar to automatically do the right thing depending on whether the WSGI container runs multiple processes. This setting allows you to force a different behavior if needed.

- `RESULTS_CACHE_SIZE`

  Default: `10`

  The toolbar keeps up to this many results in memory.

- `ROOT_TAG_EXTRA_ATTRS`

  Default: `''`

  This setting is injected in the root template div in order to avoid conflicts with client-side frameworks. For example, when using the debug toolbar with Angular.js, set this to `'ng-non-bindable'` or `'class="ng-non-bindable"'`.

- `SHOW_COLLAPSED`

  Default: `False`

  If changed to `True`, the toolbar will be collapsed by default.

- `SHOW_TOOLBAR_CALLBACK`

  Default: 'debug_toolbar.middleware.show_toolbar'

  This is the dotted path to a function used for determining whether the toolbar should show or not. The default checks are that `DEBUG` must be set to `True`, the IP of the request must be in `INTERNAL_IPS`, and the request must not be an AJAX request. You can provide your own function `callback(request)` which returns `True` or `False`.

Panel options

- `EXTRA_SIGNALS`

  Default: `[]`

  Panel: signals

  A list of custom signals that might be in your project, defined as the Python path to the signal.

- `ENABLE_STACKTRACES`

  Default: `True`

  Panels: cache, SQL

  If set to `True`, this will show stacktraces for SQL queries and cache calls. Enabling stacktraces can increase the CPU time used when executing queries.

- `HIDE_IN_STACKTRACES`

  Default: `('socketserver', 'threading', 'wsgiref', 'debug_toolbar', 'django')`. The first value is `socketserver` on Python 3 and `SocketServer` on Python 2.

  Panels: cache, SQL

  Useful for eliminating server-related entries which can result in enormous DOM structures and toolbar rendering delays.

- `PROFILER_MAX_DEPTH`

  Default: `10`

  Panel: profiling

  This setting affects the depth of function calls in the profiler's analysis.

- `SHOW_TEMPLATE_CONTEXT`

  Default: `True`

  Panel: templates

  If set to `True` then a template's context will be included with it in the template debug panel. Turning this off is useful when you have large template contexts, or you have template contexts with lazy datastructures that you don't want to be evaluated.

- `SKIP_TEMPLATE_PREFIXES`

  Default: `('django/forms/widgets/', 'admin/widgets/')`

  Panel: templates.

  Templates starting with those strings are skipped when collecting rendered templates and contexts. Template-based form widgets are skipped by default because the panel HTML can easily grow to hundreds of megabytes with many form fields and many options.

- SQL_WARNING_THRESHOLD

  Default: 500

  Panel: SQL

  The SQL panel highlights queries that took more that this amount of time, in milliseconds, to execute.

Here's what a slightly customized toolbar configuration might look like:

```
# This example is unlikely to be appropriate for your project.
CONFIG_DEFAULTS = {
    # Toolbar options
    'RESULTS_CACHE_SIZE': 3,
    'SHOW_COLLAPSED': True,
    # Panel options
    'SQL_WARNING_THRESHOLD': 100,   # milliseconds
}
```

CHAPTER 3

Tips

The toolbar isn't displayed!

The Debug Toolbar will only display when `DEBUG = True` in your project's settings. It will also only display if the mimetype of the response is either `text/html` or `application/xhtml+xml` and contains a closing `</body>` tag.

Be aware of middleware ordering and other middleware that may intercept requests and return responses. Putting the debug toolbar middleware *after* the Flatpage middleware, for example, means the toolbar will not show up on flatpages.

Middleware isn't working correctly

Using the Debug Toolbar in its default configuration and with the profiling panel will cause middlewares after `debug_toolbar.middleware.DebugToolbarMiddleware` to not execute their `process_view` functions. This can be resolved by disabling the profiling panel or moving the `DebugToolbarMiddleware` to the end of `MIDDLEWARE_CLASSES`. Read more about it at *ProfilingPanel*

Using the toolbar offline

The Debug Toolbar loads the jQuery library from the Google Hosted Libraries CDN. Your browser will keep it in cache, allowing you to use the toolbar even if you disconnect from the Internet temporarily.

If you want to use the Debug Toolbar without an Internet connection at all, or if you refuse to depend on Google's services, look at the `JQUERY_URL` configuration option.

Performance considerations

The Debug Toolbar is designed to introduce as little overhead as possible in the rendering of pages. However, depending on your project, the overhead may become noticeable. In extreme cases, it can make development impractical. Here's a breakdown of the performance issues you can run into and their solutions.

Problems

The Debug Toolbar works in two phases. First, it gathers data while Django handles a request and stores this data in memory. Second, when you open a panel in the browser, it fetches the data on the server and displays it.

If you're seeing excessive CPU or memory consumption while browsing your site, you must optimize the "gathering" phase. If displaying a panel is slow, you must optimize the "rendering" phase.

Culprits

The SQL panel may be the culprit if your view performs many SQL queries. You should attempt to minimize the number of SQL queries, but this isn't always possible, for instance if you're using a CMS and have disabled caching for development.

The cache panel is very similar to the SQL panel, except it isn't always a bad practice to make many cache queries in a view.

The template panel becomes slow if your views or context processors return large contexts and your templates have complex inheritance or inclusion schemes.

Solutions

If the "gathering" phase is too slow, you can disable problematic panels temporarily by deselecting the checkbox at the top right of each panel. That change will apply to the next request. If you don't use some panels at all, you can remove them permanently by customizing the `DEBUG_TOOLBAR_PANELS` setting.

By default, data gathered during the last 10 requests is kept in memory. This allows you to use the toolbar on a page even if you have browsed to a few other pages since you first loaded that page. You can reduce memory consumption by setting the `RESULTS_CACHE_SIZE` configuration option to a lower value. At worst, the toolbar will tell you that the data you're looking for isn't available anymore.

If the "rendering" phase is too slow, refrain from clicking on problematic panels :) Or reduce the amount of data gathered and rendered by these panels by disabling some configuration options that are enabled by default:

- `ENABLE_STACKTRACES` for the SQL and cache panels,
- `SHOW_TEMPLATE_CONTEXT` for the template panel.

Also, check `SKIP_TEMPLATE_PREFIXES` when you're using template-based form widgets.

CHAPTER 4

Panels

The Django Debug Toolbar ships with a series of built-in panels. In addition, several third-party panels are available.

Default built-in panels

The following panels are enabled by default.

Version

Path: `debug_toolbar.panels.versions.VersionsPanel`

Shows versions of Python, Django, and installed apps if possible.

Timer

Path: `debug_toolbar.panels.timer.TimerPanel`

Request timer.

Settings

Path: `debug_toolbar.panels.settings.SettingsPanel`

A list of settings in settings.py.

Headers

Path: `debug_toolbar.panels.headers.HeadersPanel`

This panels shows the HTTP request and response headers, as well as a selection of values from the WSGI environment.

Note that headers set by middleware placed before the debug toolbar middleware in `MIDDLEWARE_CLASSES` won't be visible in the panel. The WSGI server itself may also add response headers such as `Date` and `Server`.

Request

Path: `debug_toolbar.panels.request.RequestPanel`

GET/POST/cookie/session variable display.

SQL

Path: `debug_toolbar.panels.sql.SQLPanel`

SQL queries including time to execute and links to EXPLAIN each query.

Template

Path: `debug_toolbar.panels.templates.TemplatesPanel`

Templates and context used, and their template paths.

Static files

Path: `debug_toolbar.panels.staticfiles.StaticFilesPanel`

Used static files and their locations (via the staticfiles finders).

Cache

Path: `debug_toolbar.panels.cache.CachePanel`

Cache queries. Is incompatible with Django's per-site caching.

Signal

Path: `debug_toolbar.panels.signals.SignalsPanel`

List of signals, their args and receivers.

Logging

Path: `debug_toolbar.panels.logging.LoggingPanel`

Logging output via Python's built-in `logging` module.

Redirects

Path: `debug_toolbar.panels.redirects.RedirectsPanel`

When this panel is enabled, the debug toolbar will show an intermediate page upon redirect so you can view any debug information prior to redirecting. This page will provide a link to the redirect destination you can follow when ready.

Since this behavior is annoying when you aren't debugging a redirect, this panel is included but inactive by default. You can activate it by default with the `DISABLE_PANELS` configuration option.

Non-default built-in panels

The following panels are disabled by default. You must add them to the `DEBUG_TOOLBAR_PANELS` setting to enable them.

Profiling

Path: `debug_toolbar.panels.profiling.ProfilingPanel`

Profiling information for the processing of the request.

If the `debug_toolbar.middleware.DebugToolbarMiddleware` is first in `MIDDLEWARE_CLASSES` then the other middlewares' `process_view` methods will not be executed. This is because `ProfilingPanel.process_view` will return a `HttpResponse` which causes the other middlewares' `process_view` methods to be skipped.

Note that the quick setup creates this situation, as it inserts `DebugToolbarMiddleware` first in `MIDDLEWARE_CLASSES`.

If you run into this issues, then you should either disable the `ProfilingPanel` or move `DebugToolbarMiddleware` to the end of `MIDDLEWARE_CLASSES`. If you do the latter, then the debug toolbar won't track the execution of other middleware.

Third-party panels

Note: Third-party panels aren't officially supported!

The authors of the Django Debug Toolbar maintain a list of third-party panels, but they can't vouch for the quality of each of them. Please report bugs to their authors.

If you'd like to add a panel to this list, please submit a pull request!

Flamegraph

URL: https://github.com/23andMe/djdt-flamegraph

Path: `djdt_flamegraph.FlamegraphPanel`

Generates a flame graph from your current request.

Haystack

URL: https://github.com/streeter/django-haystack-panel

Path: `haystack_panel.panel.HaystackDebugPanel`

See queries made by your Haystack backends.

HTML Tidy/Validator

URL: https://github.com/joymax/django-dtpanel-htmltidy

Path: `debug_toolbar_htmltidy.panels.HTMLTidyDebugPanel`

HTML Tidy or HTML Validator is a custom panel that validates your HTML and displays warnings and errors.

Inspector

URL: https://github.com/santiagobasulto/debug-inspector-panel

Path: `inspector_panel.panels.inspector.InspectorPanel`

Retrieves and displays information you specify using the `debug` statement. Inspector panel also logs to the console by default, but may be instructed not to.

Line Profiler

URL: https://github.com/dmclain/django-debug-toolbar-line-profiler

Path: `debug_toolbar_line_profiler.panel.ProfilingPanel`

This package provides a profiling panel that incorporates output from line_profiler.

Memcache

URL: https://github.com/ross/memcache-debug-panel

Path: `memcache_toolbar.panels.memcache.MemcachePanel` or `memcache_toolbar.panels.pylibmc.PylibmcPanel`

This panel tracks memcached usage. It currently supports both the pylibmc and memcache libraries.

MongoDB

URL: https://github.com/hmarr/django-debug-toolbar-mongo

Path: `debug_toolbar_mongo.panel.MongoDebugPanel`

Adds MongoDB debugging information.

Neo4j

URL: https://github.com/robinedwards/django-debug-toolbar-neo4j-panel

Path: `neo4j_panel.Neo4jPanel`

Trace neo4j rest API calls in your django application, this also works for neo4django and neo4jrestclient, support for py2neo is on its way.

Pympler

URL: https://pythonhosted.org/Pympler/django.html

Path: `pympler.panels.MemoryPanel`

Shows process memory information (virtual size, resident set size) and model instances for the current request.

Request History

URL: https://github.com/djsutho/django-debug-toolbar-request-history

Path: `ddt_request_history.panels.request_history.RequestHistoryPanel`

Switch between requests to view their stats. Also adds support for viewing stats for ajax requests.

Sites

URL: https://github.com/elvard/django-sites-toolbar

Path: `sites_toolbar.panels.SitesDebugPanel`

Browse Sites registered in `django.contrib.sites` and switch between them. Useful to debug project when you use django-dynamicsites which sets SITE_ID dynamically.

Template Profiler

URL: https://github.com/node13h/django-debug-toolbar-template-profiler

Path: `template_profiler_panel.panels.template.TemplateProfilerPanel`

Shows template render call duration and distribution on the timeline. Lightweight. Compatible with WSGI servers which reuse threads for multiple requests (Werkzeug).

Template Timings

URL: https://github.com/orf/django-debug-toolbar-template-timings

Path: `template_timings_panel.panels.TemplateTimings.TemplateTimings`

Displays template rendering times for your Django application.

User

URL: https://github.com/playfire/django-debug-toolbar-user-panel

Path: `debug_toolbar_user_panel.panels.UserPanel`

Easily switch between logged in users, see properties of current user.

VCS Info

URL: https://github.com/giginet/django-debug-toolbar-vcs-info

Path: `vcs_info_panel.panels.GitInfoPanel`

Displays VCS status (revision, branch, latest commit log and more) of your Django application.

uWSGI Stats

URL: https://github.com/unbit/django-uwsgi

Path: `django_uwsgi.panels.UwsgiPanel`

Displays uWSGI stats (workers, applications, spooler jobs and more).

API for third-party panels

Third-party panels must subclass *`Panel`*, according to the public API described below. Unless noted otherwise, all methods are optional.

Panels can ship their own templates, static files and views. All views should be decorated with `debug_toolbar.decorators.require_show_toolbar` to prevent unauthorized access. There is no public CSS API at this time.

class `debug_toolbar.panels.`**`Panel`**(**args*, ***kwargs*)

Base class for panels.

`nav_title`

Title shown in the side bar. Defaults to `title`.

`nav_subtitle`

Subtitle shown in the side bar. Defaults to the empty string.

`has_content`

`True` if the panel can be displayed in full screen, `False` if it's only shown in the side bar. Defaults to `True`.

`title`

Title shown in the panel when it's displayed in full screen.

Mandatory, unless the panel sets `has_content` to `False`.

`template`

Template used to render `content`.

Mandatory, unless the panel sets `has_content` to `False` or overrides *attr*:content‘.

content
Content of the panel when it's displayed in full screen.

By default this renders the template defined by `template`. Statistics stored with `record_stats()` are available in the template's context.

classmethod get_urls()
Return URLpatterns, if the panel has its own views.

enable_instrumentation()
Enable instrumentation to gather data for this panel.

This usually means monkey-patching (!) or registering signal receivers. Any instrumentation with a non-negligible effect on performance should be installed by this method rather than at import time.

Unless the toolbar or this panel is disabled, this method will be called early in `DebugToolbarMiddleware.process_request`. It should be idempotent.

disable_instrumentation()
Disable instrumentation to gather data for this panel.

This is the opposite of `enable_instrumentation()`.

Unless the toolbar or this panel is disabled, this method will be called late in `DebugToolbarMiddleware.process_response`. It should be idempotent.

record_stats(*stats*)
Store data gathered by the panel. `stats` is a `dict`.

Each call to `record_stats` updates the statistics dictionary.

get_stats()
Access data stored by the panel. Returns a `dict`.

process_request(*request*)
Like process_request in Django's middleware.

Write panel logic related to the request there. Save data with `record_stats()`.

process_view(*request*, *view_func*, *view_args*, *view_kwargs*)
Like process_view in Django's middleware.

Write panel logic related to the view there. Save data with `record_stats()`.

process_response(*request*, *response*)
Like process_response in Django's middleware. This is similar to *generate_stats*, but will be executed on every request. It should be used when either the logic needs to be executed on every request or it needs to change the response entirely, such as `RedirectsPanel`.

Write panel logic related to the response there. Post-process data gathered while the view executed. Save data with `record_stats()`.

Return a response to overwrite the existing response.

generate_stats(*request*, *response*)
Similar to *process_response*, but may not be executed on every request. This will only be called if the toolbar will be inserted into the request.

Write panel logic related to the response there. Post-process data gathered while the view executed. Save data with `record_stats()`.

Does not return a value.

JavaScript API

Panel templates should include any JavaScript files they need. There are a few common methods available, as well as the toolbar's version of jQuery.

`djdt.`**`close`**`()`
Triggers the event to close any active panels.

`djdt.cookie.`**`get`**`()`
This is a helper function to fetch values stored in the cookies.

Arguments

- **`key`** (*`string`*) – The key for the value to be fetched.

`djdt.cookie.`**`set`**`()`
This is a helper function to set a value stored in the cookies.

Arguments

- **`key`** (*`string`*) – The key to be used.
- **`value`** (*`string`*) – The value to be set.
- **`options`** (*`Object`*) – The options for the value to be set. It should contain the properties `expires` and `path`.

`djdt.`**`hide_toolbar`**`()`
Closes any panels and hides the toolbar.

`djdt.`**`jQuery`**`()`
This is the toolbar's version of jQuery.

`djdt.`**`show_toolbar`**`()`
Shows the toolbar.

CHAPTER 5

Commands

The Debug Toolbar currently provides one Django management command.

debugsqlshell

This command starts an interactive Python shell, like Django's built-in `shell` management command. In addition, each ORM call that results in a database query will be beautifully output in the shell.

Here's an example:

```
>>> from page.models import Page
>>> ### Lookup and use resulting in an extra query...
>>> p = Page.objects.get(pk=1)
SELECT "page_page"."id",
       "page_page"."number",
       "page_page"."template_id",
       "page_page"."description"
FROM "page_page"
WHERE "page_page"."id" = 1

>>> print p.template.name
SELECT "page_template"."id",
       "page_template"."name",
       "page_template"."description"
FROM "page_template"
WHERE "page_template"."id" = 1

Home
>>> ### Using select_related to avoid 2nd database call...
>>> p = Page.objects.select_related('template').get(pk=1)
SELECT "page_page"."id",
       "page_page"."number",
       "page_page"."template_id",
       "page_page"."description",
```

```
       "page_template"."id",
       "page_template"."name",
       "page_template"."description"
FROM "page_page"
INNER JOIN "page_template" ON ("page_page"."template_id" = "page_template"."id")
WHERE "page_page"."id" = 1

>>> print p.template.name
Home
```

CHAPTER 6

Change log

1.8

This version is compatible with Django 1.11 and requires Django 1.8 or later.

Features

- New decorator `debug_toolbar.decorators.require_show_toolbar` prevents unauthorized access to decorated views by checking `SHOW_TOOLBAR_CALLBACK` every request. Unauthorized access results in a 404.
- The `SKIP_TEMPLATE_PREFIXES` setting allows skipping templates in the templates panel. Template-based form widgets' templates are skipped by default to avoid panel sizes going into hundreds of megabytes of HTML.

Bugfixes

- All views are now decorated with `debug_toolbar.decorators.require_show_toolbar` preventing unauthorized access.
- The templates panel now reuses contexts' pretty printed version which makes the debug toolbar usable again with Django 1.11's template-based forms rendering.
- Long SQL statements are now forcibly wrapped to fit on the screen.

1.7

Bugfixes

- Recursive template extension is now understood.
- Deprecation warnings were fixed.

- The SQL panel uses HMAC instead of simple hashes to verify that SQL statements have not been changed. Also, the handling of bytes and text for hashing has been hardened. Also, a bug with Python's division handling has been fixed for improved Python 3 support.
- An error with django-jinja has been fixed.
- A few CSS classes have been prefixed with `djdt-` to avoid conflicting class names.

1.6

The debug toolbar was adopted by jazzband.

Removed features

- Support for automatic setup has been removed as it was frequently problematic. Installation now requires explicit setup. The `DEBUG_TOOLBAR_PATCH_SETTINGS` setting has also been removed as it is now unused. See the *installation documentation* for details.

Bugfixes

- The `DebugToolbarMiddleware` now also supports Django 1.10's `MIDDLEWARE` setting.

1.5

This version is compatible with Django 1.10 and requires Django 1.8 or later.

Support for Python 3.2 is dropped.

Bugfixes

- Restore compatibility with sqlparse 0.2.0.
- Add compatibility with Bootstrap 4, Pure CSS, MDL, etc.
- Improve compatibility with RequireJS / AMD.
- Improve the UI slightly.
- Fix invalid (X)HTML.

1.4

This version is compatible with Django 1.9 and requires Django 1.7 or later.

New features

- New panel method *debug_toolbar.panels.Panel.generate_stats()* allows panels to only record stats when the toolbar is going to be inserted into the response.

Bugfixes

- Response time for requests of projects with numerous media files has been improved.

1.3

This is the first version compatible with Django 1.8.

New features

- A new panel is available: Template Profiler.
- The `SHOW_TOOLBAR_CALLBACK` accepts a callable.
- The toolbar now provides a *JavaScript API*.

Bugfixes

- The toolbar handle cannot leave the visible area anymore when the toolbar is collapsed.
- The root level logger is preserved.
- The `RESULTS_CACHE_SIZE` setting is taken into account.
- CSS classes are prefixed with `djdt-` to prevent name conflicts.
- The private copy of jQuery no longer registers as an AMD module on sites that load RequireJS.

1.2

New features

- The `JQUERY_URL` setting defines where the toolbar loads jQuery from.

Bugfixes

- The toolbar now always loads a private copy of jQuery in order to avoid using an incompatible version. It no longer attemps to integrate with AMD.

  This private copy is available in `djdt.jQuery`. Third-party panels are encouraged to use it because it should be as stable as the toolbar itself.

1.1

This is the first version compatible with Django 1.7.

New features

- The SQL panel colors queries depending on the stack level.
- The Profiler panel allows configuring the maximum depth.

Bugfixes

- Support languages where lowercase and uppercase strings may have different lengths.
- Allow using cursor as context managers.
- Make the SQL explain more helpful on SQLite.
- Various JavaScript improvements.

Deprecated features

- The `INTERCEPT_REDIRECTS` setting is superseded by the more generic `DISABLE_PANELS`.

1.0

This is the first stable version of the Debug Toolbar!

It includes many new features and performance improvements as well a few backwards-incompatible changes to make the toolbar easier to deploy, use, extend and maintain in the future.

You're strongly encouraged to review the installation and configuration docs and redo the setup in your projects.

Third-party panels will need to be updated to work with this version.

CHAPTER 7

Contributing

This is a Jazzband project. By contributing you agree to abide by the Contributor Code of Conduct and follow the guidelines.

Bug reports and feature requests

You can report bugs and request features in the bug tracker.

Please search the existing database for duplicates before filing an issue.

Code

The code is available on GitHub.

Once you've obtained a checkout, you should create a virtualenv and install the libraries required for working on the Debug Toolbar:

```
$ pip install -r requirements_dev.txt
```

You can run now run the example application:

```
$ DJANGO_SETTINGS_MODULE=example.settings django-admin migrate
$ DJANGO_SETTINGS_MODULE=example.settings django-admin runserver
```

For convenience, there's an alias for the second command:

```
$ make example
```

Look at `example/settings.py` for running the example with another database than SQLite.

Tests

Once you've set up a development environment as explained above, you can run the test suite for the versions of Django and Python installed in that environment:

```
$ make test
```

You can enable coverage measurement during tests:

```
$ make coverage
```

You can also run the test suite on all supported versions of Django and Python:

```
$ tox
```

This is strongly recommended before committing changes to Python code.

The test suite includes frontend tests written with Selenium. Since they're annoyingly slow, they're disabled by default. You can run them as follows:

```
$ make test_selenium
```

or by setting the `DJANGO_SELENIUM_TESTS` environment variable:

```
$ DJANGO_SELENIUM_TESTS=true make test
$ DJANGO_SELENIUM_TESTS=true make coverage
$ DJANGO_SELENIUM_TESTS=true tox
```

At this time, there isn't an easy way to test against databases other than SQLite.

Style

Python code for the Django Debug Toolbar follows PEP8. Line length is limited to 100 characters. You can check for style violations with:

```
$ make flake8
```

Import style is enforce by isort. You can sort import automatically with:

```
$ make isort
```

Patches

Please submit pull requests!

The Debug Toolbar includes a limited but growing test suite. If you fix a bug or add a feature code, please consider adding proper coverage in the test suite, especially if it has a chance for a regression.

Translations

Translation efforts are coordinated on Transifex.

Help translate the Debug Toolbar in your language!

Mailing list

This project doesn't have a mailing list at this time. If you wish to discuss a topic, please open an issue on GitHub.

Making a release

Prior to a release, the English `.po` file must be updated with `make translatable_strings` and pushed to Transifex. Once translators have done their job, `.po` files must be downloaded with `make update_translations`.

The release itself requires the following steps:

1. Bump version numbers in docs/conf.py, README.rst and setup.py and commit.
2. Tag the new version.
3. `python setup.py sdist bdist_wheel upload`.
4. Push the commit and the tag.
5. Change the default version of the docs to point to the latest release: https://readthedocs.org/dashboard/django-debug-toolbar/versions/

Index