django-manifest-loader

Release 1.01

django-manifest-loader

Dec 28, 2020

CONTENTS

Django Manifest Loader reads a manifest file to import your assets into a Django template. Find the URL for a single asset OR find the URLs for multiple assets by using pattern matching against the file names. Path resolution handled using Django's built-in `staticfiles` app. Minimal configuraton, cache-busting, split chunks. Designed for webpack, ready for anything.

CHAPTER ONE

ABOUT

Django Manifest Loader reads a manifest file to import your assets into a Django template. Find the URL for a single asset OR find the URLs for multiple assets by using pattern matching against the file names. Path resolution handled using Django's built-in `staticfiles` app. Minimal configuraton, cache-busting, split chunks. Designed for webpack, ready for anything.

Turns this

```
{% load manifest %}
<script src="{% manifest 'main.js' %}" />
```

Into this

```
<script src="/static/main.8f7705adfa281590b8dd.js" />
```

- For an in-depth tutorial, check out this blog post here
- Quick start blog post

1.1 Additional resources

- What is cache busting?
- The 100% correct way to split your chunks with Webpack

CHAPTER

TWO

INSTALLATION

```
pip install django-manifest-loader
```

2.1 Django Setup

```
# settings.py

INSTALLED_APPS = [
    ...
    'manifest_loader',  # add to installed apps
    ...
]

STATICFILES_DIRS = [
    BASE_DIR / 'dist'  # the directory webpack outputs to
]
```

You must add webpack's output directory to the `STATICFILES_DIRS` list. The above example assumes that your webpack configuration is set up to output all files into a directory `dist/` that is in the `BASE_DIR` of your project.

`BASE_DIR`'s default value, as set by `$ djagno-admin startproject` is `BASE_DIR = Path(__file__).resolve().parent.parent`, in general you shouldn't be modifying it.

Optional settings, default values shown.

```
# settings.py

MANIFEST_LOADER = {
    'output_dir': None,  # where webpack outputs to, if not set, will search in␣
↪STATICFILES_DIRS for the manifest.
    'manifest_file': 'manifest.json',  # name of your manifest file
    'cache': False,  # recommended True for production, requires a server restart to␣
↪pick up new values from the manifest.
    'loader': DefaultLoader  # how the manifest files are interacted with
}
```

2.2 webpack configuration

webpack is not technically required: Django Manifest Loader by default expects a manifest file in the form output by webpack Manifest Plugin. See the section on custom loaders for information on how to use a different type of manifest file.

You must install the `WebpackManifestPlugin`. Optionally, but recommended, is to install the `CleanWebpackPlugin`.

```
npm i --save-dev webpack-manifest-plugin clean-webpack-plugin
```

```
// webpack.config.js

const { CleanWebpackPlugin } = require('clean-webpack-plugin');
const ManifestPlugin = require('webpack-manifest-plugin');

module.exports = {
  ...
  plugins: [
      new CleanWebpackPlugin(),  // removes outdated assets from the output dir
      new ManifestPlugin(),  // generates the required manifest.json file
  ],
  ...
};
```

For a deep dive into a supported webpack configuration, read the blog post introducting this package here

CHAPTER

THREE

EXAMPLE PROJECT STRUCTURE

```
BASE_DIR
├── dist  # webpack's output directory
│   ├── index.f82c02a005f7f383003c.js
│   └── manifest.json
├── frontend  # a django app
│   ├── apps.py
│   ├── src
│   │   └── index.js
│   ├── templates
│   │   └── frontend
│   │       └── index.html
│   └── views.py
├── manage.py
├── package.json
├── project
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
├── requirements.txt
└── webpack.config.js
```

CHAPTER

FOUR

BASIC USAGE

Django Manifest Loader comes with two template tags that house all logic. The `manifest` tag takes a single string input, such as `'main.js'`, looks it up against the webpack manifest, and then outputs the URL to that compiled file. It works just like Django's built it `static` tag, except it's finding the filename according to your manifest file.

The `manifest_match` tag takes two arguments, a string to pattern match filenames against and a string to embed matched file urls into. See the `manifest_match` section for more information.

4.1 Manifest tag

```
{% load manifest %}

<script src="{% manifest 'main.js' %}"></script>
```

turns into

```
<script src="/static/main.8f7705adfa281590b8dd.js"></script>
```

Where the argument to the tag will be the original filename of a file processed by webpack. If in doubt, check your `manifest.json` file generated by webpack to see what files are available.

This is worthwhile because of the content hash after the original filename, which will invalidate the browser cache every time the file is updated, which will ensure that your users always have the latest assets.

4.2 Manifest match tag

```
{% load manifest %}

{% manifest_match '*.js' '<script src="{match}"></script>' %}
```

turns into

```
<script src="/static/vendors~main.3ad032adfa281590f2a21.js"></script>
<script src="/static/main.8f7705adfa281590b8dd.js"></script>
```

This tag takes two arguments, a pattern to match against, according to the python `fnmatch` package rules, and a string to input the file URLs into. The second argument must contain the string `{match}`, as it is replaced with the URLs.

CHAPTER FIVE

ADVANCED USAGE

5.1 Custom Loaders

Custom loaders allow you to implement your own means of extracting data from your manifest file. If your manifest file is not the default structure of webpack manifest plugin, this is how you can tell `django-manifest-loader` how to read it.

First import the loader parent abstract class, and subclass it in your new loader class.

```
from manifest_loader.loaders import LoaderABC

class MyCustomLoader(LoaderABC):
```

Your new loader must have two static methods that each take two required arguments: `get_single_match(manifest, key)` and `get_multi_match(manifest, pattern)`.

```
from manifest_loader.loaders import LoaderABC

class MyCustomLoader(LoaderABC):
    @staticmethod
    def get_single_match(manifest, key):
        pass

    @staticmethod
    def get_multi_match(manifest, pattern):
        pass
```

- `get_single_match` - returns a `String`, finds a single file in your manifest file, according to the `key`
- `get_multi_match` - returns a `List` of files in your manifest, according to the `pattern`
- `manifest` - this is your full manifest file, after being processed by `json.load()`. It will be a dictionary or list depending on which it is in your manifest file.
- `key` - `String`; the argument passed into the `manifest` template tag. e.g.: in the template tag `{% manifest 'index.js' %}`, the string `'index.js'` is sent to `get_single_match` as `key` (without surrounding quotes)
- `pattern` - `String`; the first argument passed into the `manifest_match` template tag. e.g.: in the template tag `{% manifest_match '*.js' '<script src="{match}"></script>' %}`, the string `'*.js'` is sent to `get_multi_match` as `pattern` (without surrounding quotes)

Below is the code for the default loader, which is a good starting point:

```python
import fnmatch
from manifest_loader.loaders import LoaderABC

class DefaultLoader(LoaderABC):
    @staticmethod
    def get_single_match(manifest, key):
        return manifest.get(key, key)

    @staticmethod
    def get_multi_match(manifest, pattern):
        matched_files = [file for file in manifest.keys() if
                         fnmatch.fnmatch(file, pattern)]
        return [manifest.get(file) for file in matched_files]
```

In the above example, `get_single_match` retrieves the value on the `manifest` dictionary that matches the key `key`. If the key does not exist on the dictionary, it instead returns the key.

`get_multi_match` uses the recommended `fnmatch` python standard library to do pattern matching. You could also use regex in it's place. Here, it iterates through all the keys in the manifest file, and builds a list of the keys that match the given `pattern`. It then returns a list of the values associated with those matched keys.

5.1.1 Activating the custom loader

To put the custom loader into use it needs to be registered in your `settings.py`.

```python
# settings.py
from my_app.utils import MyCustomLoader

MANIFEST_LOADER = {
    ...
    'loader': MyCustomLoader
}
```

5.2 URLs in Manifest File

If your manifest file points to full URLs, instead of file names, the full URL will be output instead of pointing to the static file directory in Django.

Example:

```json
{
  "main.js": "http://localhost:8080/main.js"
}
```

```
{% load manifest %}

<script src="{% manifest 'main.js' %}"></script>
```

Will output as:

```html
<script src="http://localhost:8080/main.js"></script>
```

CHAPTER

SIX

TESTS AND CODE COVERAGE

Run unit tests and verify 100% code coverage with:

```
git clone https://github.com/shonin/django-manifest-loader.git
cd django-manifest-loader
pip install -e .

# run tests
python runtests.py

# check code coverage
pip install coverage
coverage run --source=manifest_loader/ runtests.py
coverage report
```

CHAPTER

SEVEN

THE TWO WAYS TO BUILD A FRONT END

There are two fundamental ways to connect a javascript front end to Django: coupled or decoupled. Django Manifest loader is specifically for the coupled option.

A coupled front end and back end means that Django is responsible for the front ends asset files. As a user you point your web browser to the Django app, and the Django app in turn makes sure you get the front end.

A decoupled front and back end means they are hosted separately. Django has no knowledge of front end asset files, and does not serve them. As a user you point your browser at the staticly hosted front end app and that app interacts with Django through an API.

I typically choose the coupled option as

- I don't want to manage multiple repos
- or multiple servers
- Django is powerful

The decoupled option is good for if

- you value the performance gain of using a static file server
- your front end and django app are managed by different teams
- you want micro services

It's a tradeoff. Django Manifest Loader makes the coupled option much easier than it was before.

CHAPTER

EIGHT

API REFERENCE

8.1 Manifest Tag

Returns the manifest tag

```
@register.tag('manifest')
def do_manifest(parser, token):

    return ManifestNode(token)
```

8.2 Manifest Match Tag

Returns manifest_match tag

```
@register.tag('manifest_match')
def do_manifest_match(parser, token):
    return ManifestMatchNode(token)
```

8.3 ManifestNode

Initializes and renders the creation of the manifest tag and

```
 class ManifestNode(template.Node):
    """ Initalizes the creation of the manifest template tag"""
    def __init__(self, token):
        bits = token.split_contents()
        if len(bits) < 2:
            raise template.TemplateSyntaxError(
                "'%s' takes one argument (name of file)" % bits[0])
        self.bits = bits


    def render(self, context):
        """Renders the creation of the manifest tag"""
        manifest_key = get_value(self.bits[1], context)
        manifest = get_manifest()
        manifest_value = manifest.get(manifest_key, manifest_key)
        return make_url(manifest_value, context)
```

8.4 ManifestMatch Node

Initalizes and renders the creation of the manifest match tag

```
class ManifestMatchNode(template.Node):
    """ Initalizes the creation of the manifest match template tag"""
    def __init__(self, token):
        self.bits = token.split_contents()
        if len(self.bits) < 3:
            raise template.TemplateSyntaxError(
                "'%s' takes two arguments (pattern to match and string to "
                "insert into)" % self.bits[0]
            )

    def render(self, context):
        """ Renders the manifest match tag"""
        urls = []
        search_string = get_value(self.bits[1], context)
        output_tag = get_value(self.bits[2], context)

        manifest = get_manifest()

        matched_files = [file for file in manifest.keys() if
                         fnmatch.fnmatch(file, search_string)]
        mapped_files = [manifest.get(file) for file in matched_files]

        for file in mapped_files:
            url = make_url(file, context)
            urls.append(url)
        output_tags = [output_tag.format(match=file) for file in urls]
        return '\n'.join(output_tags)


def get_manifest():
    """ Returns the manifest file from the output directory """
    cached_manifest = cache.get('webpack_manifest')
    if APP_SETTINGS['cache'] and cached_manifest:
        return cached_manifest

    if APP_SETTINGS['output_dir']:
        manifest_path = os.path.join(APP_SETTINGS['output_dir'],
                                     APP_SETTINGS['manifest_file'])
    else:
        manifest_path = find_manifest_path()

    try:
        with open(manifest_path) as manifest_file:
            data = json.load(manifest_file)
    except FileNotFoundError:
        raise WebpackManifestNotFound(manifest_path)

    if APP_SETTINGS['cache']:
        cache.set('webpack_manifest', data)

    return data
```

8.5 Finding the Manifest File

Returns manifest_file

```
def find_manifest_path():
    static_dirs = settings.STATICFILES_DIRS
    if len(static_dirs) == 1:
        return os.path.join(static_dirs[0], APP_SETTINGS['manifest_file'])
    for static_dir in static_dirs:
        manifest_path = os.path.join(static_dir, APP_SETTINGS['manifest_file'])
        if os.path.isfile(manifest_path):
            return manifest_path
    raise WebpackManifestNotFound('settings.STATICFILES_DIRS')
```

8.6 String Validator

Method validates if it's a string

```

def is_quoted_string(string):
    if len(string) < 2:
        return False
    return string[0] == string[-1] and string[0] in ('"', "'")
```

8.7 Value Validator

Method validates the value

```

def get_value(string, context):

    if is_quoted_string(string):
        return string[1:-1]
    return context.get(string, '')
```

8.8 URL Validator

Function validates if it's a URL

```

def is_url(potential_url):


    validate = URLValidator()
    try:
        validate(potential_url)
        return True
    except ValidationError:
        return False
```

8.9 URL Generator

Returns the URL that will be outputed to the static file directory

```python
def make_url(manifest_value, context):


    if is_url(manifest_value):
        url = manifest_value
    else:
        url = StaticNode.handle_simple(manifest_value)
    if context.autoescape:
        url = conditional_escape(url)
    return url
```

CHAPTER

NINE

IMPROVE DOCUMENTATION

Thanks to everyone who has and who will one day contribute to the documentation for this project. Pull requests or issues filed for documentation fixes, clarifications, and restructuring are all welcome. Open a pull request or issue here.

Documentation is developed using Sphinx.

9.1 Installation

In order to install sphinx

```
pip install -U sphinx
```

9.2 Dependencies for installation

To use .md with Sphynx, it requires Recommonmark.

```
pip install recommonmark
```

9.3 How to run

After installation of sphinx and recommonmark, to generate the `_build` directory that has doc trees and html, `cd docs` then `make html`.

CHAPTER

TEN

CONTRIBUTING

Do it. Please feel free to file an issue or open a pull request. The code of conduct is basic human kindness. See the project on Github here

CHAPTER

ELEVEN

LICENSE

Django Manifest Loader is distributed under the 3-clause BSD license. This is an open source license granting broad permissions to modify and redistribute the software.