BowerStatic Documentation

Release 0.7

Martijn Faassen

January 14, 2015

Contents

CHAPTER 1

Introduction

BowerStatic is a WSGI component that can serve static resources from front-end packages (JavaScript, CSS) that you install through the Bower package manager.

Features:

- Integrates with any WSGI-based project.
- Easily serve Bower-managed `bower_components` directories.
- Easily say in Python code you want to include a static resource, which are then automatically inserted in any HTML page you generated. It uses the appropriate `<script>` and `<link>` tags.
- Support for Bower `main` end points. End points for dependencies are automatically included too.
- Declare additional dependencies from one resource to others, either in the same package or in others.
- Infinite caching of URLs by the browser and/or HTTP caching server for increased performance.
- Instantly bust the cache when a new version of Bower package is installed, avoiding force reload.
- Local packages with automatic cache busting as soon as you edit code.

CHAPTER 2

Contents

2.1 Integrating BowerStatic

2.1.1 Introduction

WSGI?

WSGI is a Python standard for interoperability between web applications and web servers. It also allows you to plug in "middleware" that sit between web server and web application that adds extra functionality. BowerStatic provides such middleware, which we will see later.
Most Python web frameworks are WSGI based. This means that if you use such a web framework for your application, your application is a WSGI application. Where this documentation says "WSGI application" you can read "your application".

This tutorial explains how to use BowerStatic with a WSGI application. BowerStatic doesn't have a huge API, but your web framework may provide more integration, in which case you may only have to know even less.

2.1.2 The Bower object

To get started with BowerStatic you need a `Bower` instance. Typically you only have one global `Bower` instance in your application.

You create it like this:

```
import bowerstatic

bower = bowerstatic.Bower()
```

2.1.3 Integrating BowerStatic with a WSGI app

For BowerStatic to function, we need to wrap your WSGI application with BowerStatic's middleware. Here's to do this for our `bower` object:

```
app = bower.wrap(my_wsgi_app)
```

Your web framework may have special BowerStatic integration instead that does this for you.

Later on we will go into more details about what happens here (both an injector and publisher get installed).

2.1.4 Declaring Bower Directories

Bower manages a directory in which it installs components (jQuery, React, Ember, etc). This directory is called `bower_components` by default. Bower installs components into this directory as sub-directories. Bower makes sure that the components fit together according to their dependency requirements.

Each `bower_components` directory is an "isolated universe" of components. Components in a `bower_components` directory can depend on each other only – they cannot depend on components in another `bower_components` directory.

You need to let BowerStatic know where a `bower_components` directory is by registering it with the `bower` object:

```
components = bower.components('components', '/path/to/bower_components')
```

Bowerstatic needs an absolute path to the components. With the help of `module_relative_path` you can use a path relative to the calling module:

```
components = bower.components('components',
   bowerstatic.module_relative_path('path/relative/to/calling/module'))
```

You can register multiple `bower_components` directories with the `bower` object. You need to give each a unique name; in the example it is `components`. This name is used in the URL used to serve components in this directory to the web.

The object returned we assign to a variable `components` that we use later.

2.1.5 Including Static Resources in a HTML page

Errors

If you try to include a component that was not installed, you get an `bowerstatic.Error` exception. The `bower_components` directory is read during startup, so if you just installed that component using `bower install`, you need to restart the server.
If you try to refer to a refer to a file in a component that does not exist you also get an `bowerstatic.Error` exception. If that file then gets added (through a `bower upgrade` or if it's in a local component) it will be found without the need for a restart.

Now that we have a `components` object we can start including static resources from these components in a HTML page. BowerStatic provides an easy, automatic way for you to do this from Python.

Using the `components` object we created earlier for a `bower_components` directory, you create a `include` function:

```
include = components.includer(environ)
```

You need to create the `include` function within your WSGI application, typically just before you want to use it. You need to pass in the WSGI `environ` object, as this is where the inclusions are stored. You can create the `include` function as many times as you like for a WSGI environ; the inclusions are shared.

Now that we have `include`, we can use it to include resources:

```
include('jquery/dist/jquery.js')
```

WSGI environ

BowerStatic's includer system needs to interact with the WSGI `environ` object. If your WSGI-based web framework has a `request` object, then a very good bet is to try `request.environ` to get it.
Your web framework may also have special integration with BowerStatic; in that case the integration can offer the `include` function directly and takes care of interacting with the `environ` for you.

This specifies you want to include the `dist/jquery.js` resource from within the installed `jquery` component. This refers to an actual file in the jQuery component; in `bower_components` there is a directory `jquery` with the sub-path `dist/jquery.js` inside. It is an error to refer to a non-existent file.

If you call `include` somewhere in code where also a HTML page is generated, BowerStatic adds the following `<script>` tag to that HTML page automatically:

```
<script
  type="text/javascript"
  src="/bowerstatic/components/jquery/2.1.1/dist/jquery.js">
</script>
```

2.1.6 Supporting additional types of resources

There are all kinds of resource types out there on the web, and BowerStatic does not know how to include all of them on a HTML page. You can tell the bower object how to handle a new resource type like this:

```
def render_foo(url):
    return "<foo>%s</foo>" % url

bower.renderer('.foo', render_foo)
```

If you now include a resource like `example.foo`, that resource gets included on the web page as `<foo>/path/to/example.foo</foo>`.

You can also use `renderer()` to override existing behavior of how a resource with a particular extension is to be included.

If you include a resource with an unrecognized extension, a `bowerstatic.Error` is raised.

2.1.7 URL structure

Let's look at the URLs used by BowerStatic:

```
/bowerstatic/components/jquery/2.1.1/dist/jquery.js
```

`bowerstatic` The BowerStatic signature. You can change the default signature used by passing a `signature` argument to the `Bower` constructor.

`components` The unique name of the `bower_components` directory which you registered with the `bower` object.

`jquery` The name of the installed component as given by the `name` field in `bower.json`.

`2.1.1` The version number of the installed component as given by the `version` field in `bower.json`.

`dist/jquery.js` A relative path to a file within the component.

2.1.8 Caching

Cache busting

Caches in the browser and caching servers such as Varnish like to hold on to static resources, so that the static resources does not to be reloaded all the time.
But when you upgrade an application, or develop an application, you want the browser to request *new* resources from the server where those resources have changed.
Cache busting is a simple technique to make this happen: you serve changed resources under a new URL. BowerStatic does this automatically for you by including a version number or timestamp in the resource URLs.

BowerStatic makes sure that resources are served with caching headers set to cache them forever [1]. This means that after the first time a web browser accesses the browser, it does not have to request them from the server again. This takes load off your web server.

To take more load off your web server, you can install a install a caching proxy like Varnish or Squid in front of your web server, or use Apache's `mod_cache`. With those installed, the WSGI server only has to serve the resource once, and then it is served by cache after that.

Caching forever would not normally be advisable as it would make it hard to upgrade to newer versions of components. You would have to teach your users to issue a shift-reload to get the new version of JavaScript code. But with BowerStatic this is safe, because it busts the cache automatically for you. When a new version of a component is installed, the version number is updated, and new URLs are generated by the include mechanism.

2.1.9 Main endpoint

Bower has a concept of a `main` end-point for a component in its `bower.json`. You can include the main endpoint by including the component with its name without any file path after it:

```
include('jquery')
```

This includes the file listed in the `main` field in `bower.json`. In the case of jQuery, this is the same file as we already included in the earlier examples: `dist/jquery.js`.

A component can also specify an array of files in `main`. In this case only the first endpoint listed in this array is included.

The endpoint system is aware of Bower component dependencies. Suppose you include 'jquery-ui':

```
include('jquery-ui')
```

The `jquery-ui` component specifies in the `dependencies` field in its `bower.json` that it depends on the `jquery` component. When you include the `jquery-ui` endpoint, BowerStatic automatically also include the `jquery` endpoint for you. You therefore get two inclusions in your HTML:

```
<script
  type="text/javascript"
  src="/bowerstatic/static/jquery/2.1.1/dist/jquery.js">
</script>
<script
  type="text/javascript"
  src="/bowerstatic/static/jquery-ui/1.10.4/ui/jquery-ui.js">
</script>
```

If `main` lists a resource with an extension that has no renderer registered for it, that resource is not included.

[1] Well, for 10 years. But that's forever in web time.

2.1.10 WSGI Publisher and Injector

Earlier we described `bower.wrap` to wrap your WSGI application with the BowerStatic functionality. This is enough for many applications. Sometimes you may want to be able to use the static resource publishing and injecting-into-HTML behavior separately from each other, however.

Publisher

BowerStatic uses the publisher WSGI middleware to wrap a WSGI application so it can serve static resources automatically:

```
app = bower.publisher(my_wsgi_app)
```

`app` is now a WSGI application that does everything `my_wsgi_app` does, as well as serve Bower components under the special URL `/bowerstatic`.

Injector

BowerStatic also automates the inclusion of static resources in your HTML page, by inserting the appropriate `<script>` and `<link>` tags. This is done by another WSGI middleware, the injector.

You need to wrap the injector around your WSGI application as well:

```
app = bower.injector(my_wsgi_app)
```

Wrap

Before we saw `bower.wrap`. This wraps both a publisher and an injector around a WSGI application. So this:

```
app = bower.wrap(my_wsgi_app)
```

is equivalent to this:

```
app = bower.publisher(bower.injector(my_wsgi_app))
```

2.1.11 Using the Publisher and Injector with WebOb

The `Injector` and `Publisher` can also directly be used with WebOb request and response objects. This is useful for integration with web frameworks that already use WebOb, such as Morepath and Pyramid:

```
injector = bower.injector(wsgi=None)
publisher = bower.publisher(wsgi=None)

response = publisher.publish(request, injector.inject(request, response))
```

All that is required is a WebOb request and a response.

2.2 Local Components

2.2.1 Introduction

Now we have a way to publish and use Bower components. But you probably also develop your own front-end code: we call these "local components". BowerStatic also helps with that. For this it is important to understand that locally developed code has special caching requirements:

- When you release a local component, you want it to be cached infinitely just like for Bower components.

  When later a new release is made, you want that cache to be invalidated, and not force end-users to do a shift-reload to get their browser to load the new version of the code.

  We can accomplish this behavior by using a version number in the URL, just like for Bower components.

  XXX one way to release a local component would be to release it as a bower component at this point. But this may be cumbersome for code maintained as part of Python package.

- When you *develop* a local component, you want the cache to be invalidated as soon as you make any changes to the code, so you aren't forced to do shift-reload during development. A simple reload should refresh all static resources.

  A way to look at this is that you want the system to make a new version number for each and every edit to the local component.

2.2.2 Usage

To have local components, you first have to create a special local components registry:

```
local = bower.local_components('local', components)
```

You can have more than one local components registry, but typically you only need one per project.

The first argument is the name of the local components registry. It is used in the URL.

The second argument is a components object for a `bower_components` directory, created earlier with `bower.components()`. This makes all those bower components available in the local component registry, so that the local components can depend on them.

Note that the local components registry does not point to a `bower_components` directory itself. Instead we register directories for individual local components manually.

Location of the local component

You can organize your code so that the local component lives inside a Python package that provides a web API for it to use, so that they can be developed together. You can also organize things differently – this is up to you.

Here's how we add a local component:

```
local.component('/path/to/directory/mycode', version='1.1.0')
```

The `/path/to/directory/mycode` directory should have a `bower.json` file. BowerStatic uses `name` and `main` for local components like it uses them for third party Bower components. The name of the component should be unique within the local registry, as well as not conflict with any component in the Bower components registry.

As with `bower.components`, you can use `bowerstatic.module_relative_path` to create a path relative to the calling module:

```
components = local.component(
    bowerstatic.module_relative_path('path/relative/to/calling/module'),
    version='1.1.0')
```

`dependencies` is also picked up from `bower.json`, but unlike for third party components these dependencies are not automatically installed.

The version number is not picked up from `bower.json`. Instead it is passed through to the local component. This will make it possible to support the right caching behavior. We go into detail about this later.

If you have a local component called `mycode`, and there is a file `app.js` in its directory, it is published under this URL:

```
/bowerstatic/local/mycode/1.1.0/app.js
```

To be able to include it, we can create an includer for the `local` registry:

```
include = local.includer(environ)
```

This includer can be used to include local components, but also the third-party components from the registry that the local components registry was initialized with.

You can now include `app.js` in `mycode` like this:

```
include('mycode/app.js')
```

2.2.3 Versioning

Let's consider versioning in more detail.

The version number is passed in when registering the local component. We want it to do the right thing with caching:

- When the application is deployed, we want the version number to be the version number of that application (or sub-package of that application), so that infinite caching can be used but the cache is automatically busted with an application upgrade.
- When the application is under development, we want the version number to change each time you edit the local component's code, so that the cache is busted each time.

2.2.4 Versioning deployed applications

You can use the version of a Python application easily, as long as it is packaged using `setuptools` (`pip`, `easy_install`, `buildout`, etc). You can retrieve its version number like this:

```
import pkg_resources

version = pkg_resources.get_distribution('myproject').version
```

This picks up the version given in `setup.py` of `myproject`.

Using this to obtain the version and passing it into `local.component()` is enough to make sure the cache is busted when you make a release of your application.

2.2.5 Versioning during development

We have to make sure the cache is busted automatically during development as well. For that we have to turn on BowerStatic's development mode. You can do this by passing `None` as the version into `local.component`.

This causes the version to be automatically determined from the code in the package, and be different each time you edit the code. Since the version is included in the URL to the package, this allows you to get the latest version of the code as soon as you reload after editing a file. No shift-reloads needed to reload the code!

2.2.6 Putting it all together

Development mode is relatively expensive, as BowerStatic has to monitor the local directory for any changes so it can update the version number automatically. You should therefore make sure it is only enabled during development, not during deployment. When your application is deployed you need to pass in a real version number, for instance the one you pick up using `pkg_resources` as described before.

If your application has a notion of a development mode that you can somehow inspect during run-time, you can write a version function that automatically returns `None` in development mode and otherwise returns the application's version number. This ensures optimal caching behavior during development and deployment both. Here's what this function could look like:

```
def get_version():
    if is_devmode_enabled():  # app specific API
        return None
    return pkg_resources.get_distribution('myproject').version
```

You can then register the local component like this:

```
local.component('/path/to/directory/mycode', version=get_version())
```

2.3 Dependencies

2.3.1 Introduction

Client-side JavaScript's "Shared Library" Approach

JavaScript has no standard `import` statement like Python does. Instead, there are a many different ways to declare dependencies between JavaScript modules, each with their own advantages and drawbacks. One way to declare dependencies for client-side code is to use `RequireJS`. NodeJS has its way to declare dependencies between modules on the server side, and tools like browserify exist that can help to bring these to the client. EcmaScript 6 is introducing a module syntax of its own which will hopefully bring order to this chaos.
The JavaScript strategy commonly used to deliver a set of modules with dependencies to the client, especially for production use, is different than Python's: it's more like the way shared libraries work. A shared library (`.so` on Unix systems, `.dll` on Windows) is built from many individual source files.
So, instead of shipping a package with a lot of individual `.js` files, a single bundle is built from all the modules in a package. `dist/jquery.js` for instance is a bundled version of individual underlying jQuery modules that are developed in its `src` directory.
Bundling is done because client-side JavaScript does not have a universal module system, and also because it's more efficient for a browser to load a single bundle than to load many individual files.
A bundling module system like this has a drawback: you cannot declare dependencies to individual modules in other packages. Instead such dependencies are on the package level.

A Bower package may specify in its `bower.json` a dependency on other packages. Bower uses this to install the dependent packages automatically. The `jquery-ui` package for instance depends on the `jquery` package, so when you install `jquery-ui`, the `jquery` package is automatically installed as well.

BowerStatic also uses this information. If you include the endpoint of a package (by not specifying the file), the endpoints of the dependencies are also included automatically.

This is different from dependencies between individual static resources. Bower has no information about these, and in fact there is no universal system on the client to determine these.

Like Bower, BowerStatic therefore does not mandate a particular module system. Use whatever system you like, with whatever server-side bundling tools you like. But to help automate some cases, BowerStatic does let you declare dependencies between resources if you want to, either for resources within a single package or between resources in different packages. This works for static resources of any kind; JavaScript but also CSS.

2.3.2 Dependencies

In order to use dependencies you need to specify extra information for resources. This is done using the resource method on the directory object:

```
components = bower.components('components', '/path/to/bower_components')

components.resource(
   'jquery-ui/ui/minified/jquery-ui.min.js',
   dependencies=['jquery/dist/jquery.min.js'])
```

Here we express that the `jquery-ui.min.js` resource depends on the `jquery.min.js` resource.

When you now depend on `jquery-ui/ui/minified/jquery-ui.min.js` using the same `components` object:

```
include = components.includer(environ)
include('jquery-ui/ui/minified/jquery-ui.min.js')
```

an inclusion to the minified jQuery is also generated:

```
<script
  type="text/javascript"
  src="/bowerstatic/static/jquery/2.1.1/dist/jquery.min.js">
</script>
<script
  type="text/javascript"
  src="/bowerstatic/static/jquery-ui/1.10.4/ui/minified/jquery-ui.min.js">
</script>
```

2.3.3 Resource objects

The `.resource` method in fact creates a resource object that you can assign to a variable:

```
jquery_min = components.resource(
   'jquery/dist/jquery.min.js')
```

You can use this resource object in an `include`:

```
include(jquery_min)
```

This has the same effect as referring to the resource directory using a string.

You can also refer to this resource in another resource definition:

```
jquery_ui_min = components.resource(
   'jquery-ui/ui/minified/jquery-ui.min.js',
   dependencies=[jquery_min])
```

Dealing with explicit resource objects can be handy as it saves typing, and Python gives you an error if you refer to a resource object that does not exist, so you can catch typos early.

2.3.4 Component objects

It is sometimes useful to be able to generate the URL for a component itself, for instance when client-side code needs to construct URLs to things inside it, such as templates. To support this case, you can get the URL of a component by writing this:

```
components.get_component('jquery').url()
```

This will generate the appropriate versioned URL to that component.

2.4 Developing BowerStatic

2.4.1 Install BowerStatic for development

First make sure you have virtualenv installed for Python 2.7.

Now create a new virtualenv somewhere for BowerStatic development:

```
$ virtualenv /path/to/ve_bowerstatic
```

You should also be able to recycle an existing virtualenv, but this guarantees a clean one. Note that we skip activating the environment here, as this is just needed to initially bootstrap the BowerStatic buildout.

Clone BowerStatic from github and go to the `bowerstatic` directory:

```
$ git clone git@github.com:faassen/bowerstatic.git
$ cd bowerstatic
```

Now we need to run `bootstrap.py` to set up buildout, using the Python from the virtualenv we've created before:

```
$ python /path/to/ve_bowerstatic/bin/python/bootstrap.py
```

This installs buildout, which can now set up the rest of the development environment:

```
$ bin/buildout
```

This downloads and installs various dependencies and tools. The commands you run in `bin` are all restricted to the virtualenv you set up before. There is therefore no need to refer to the virtualenv once you have the development environment going.

2.4.2 Running the tests

You can run the tests using py.test. Buildout has installed it for you in the `bin` subdirectory of your project:

```
$ bin/py.test bowerstatic
```

To generate test coverage information as HTML do:

```
$ bin/py.test bowerstatic --cov bowerstatic --cov-report html
```

You can then point your web browser to the `htmlcov/index.html` file in the project directory and click on modules to see detailed coverage information.

2.4.3 flake8

The buildout also installs flake8, which is a tool that can do various checks for common Python mistakes using pyflakes and checks for PEP8 style compliance.

To do pyflakes and pep8 checking do:

```
$ bin/flake8 bowerstatic
```

2.4.4 radon

The buildout installs radon. This is a tool that can check various measures of code complexity.

To check for cyclomatic complexity (excluding the tests):

```
$ bin/radon cc bowerstatic -e "bowerstatic/tests*"
```

To filter for anything not ranked `A`:

```
$ bin/radon cc bowerstatic --min B -e "bowerstatic/tests*"
```

And to see the maintainability index:

```
$ bin/radon mi bowerstatic -e "bowerstatic/tests*"
```

2.5 CHANGES

2.5.1 0.7 (2014-11-15)

- The publisher and injector have been refactored into `PublisherTween` and a `Publisher`, and an `InjectorTween` and an `Injector`. The Tween versions are the ones intended for use by web frameworks that already use WebOb (such as Morepath and Pyramid) as an alternative to using WSGI-based integration.

  The WSGI-based integration has remained unchanged; you can still use `bower.wrap` (or `bower.injector` and `bower.publisher`).

2.5.2 0.6 (2014-11-13)

- Added Python 3 compatibility. Fixes issue #25.
- Bower components and local components can be created based on a path relative to the directory in which the components are created using `bowerstatic.module_relative_path`.
- The injector and the publisher can now also be used directly by providing a WebOb request and response. Third-party frameworks that already use WebOb (such as Morepath and Pyramid) can make use of this to integrate with on the level of their own request and response objects, instead of on the WSGI level.

2.5.3 0.5 (2014-09-24)

- On some platforms and filesystems (such as Linux ext3, Mac OS X) `os.path.getmtime()` returns timestamps with the granularity of seconds instead of subseconds such as Linux ext4. We go for second granularity now by default for autoversioning as this should be good enough during development.

  The test for autoversioning was assuming sub-second granularity and this test failed. This test is now skipped on Mac OS X. What didn't help was weird code in BowerStatic that cut off the last bit of the microsecond isoformat – this was removed.

  This fixes bug #20. (Thanks to Michael Howitz for the bug report)

- Display a nicer error messages when a component depends on another one that doesn't exist. Thanks for Michael Howitz for the improvement.

- Internal toposort module was not imported relative to package, which could lead to errors in some circumstances. Thanks TylorS for reporting! Fixes issue #24.

2.5.4 0.4 (2014-09-08)

- There was a bug in the new `component(name)` method to obtain the component, because it wouldn't work for local components. Fixing this properly took a significant refactoring:
  - the ComponentCollection gains its own fallback behavior, much simpler to implement than in LocalComponentCollection.
  - UrlComponent is now gone and Component gains that functionality; it keeps a reference to the collection itself now.
  - add a lot of free-standing functions to the methods.
  - an earlier hack passing the component collection through an argument is now gone.
- The new `.component()` API to get a component from a collection explicitly in order to get its URL is gone again as it conflicted with an earlier API on local component collections. Instead use `collection.get_component(name)`.

2.5.5 0.3 (2014-08-28)

- BowerStatic failed to initialize if a component was discovered without a `main`. These are handled now – if you try to include a path to such a resource, no extra inclusions are generated. Fixes #5.
- If `main` was an array, only the first such resource was loaded. The correct behavior should be to load all these resources. This required a reworking of how resources get created; instead of creating a single resource for a path, a list of resources is created everywhere. Fixes #6 (and was mentioned in #5).
- Introduce a `component(name)` method on the components object. Given a component name it will give an object that has a `url()` method. This can be used to obtain the URL of a component directory, which is sometimes useful when client-side code needs to construct URLs itself, such as for templates. Fixes issue #8.
- You can register a renderer for a particular extension type using, for example, `bower.renderer('.js', render_js)`, where `render_js` takes a URL and should return a string with a HTML snippet to include on the page.

2.5.6 0.2 (2014-07-18)

- Even if the same resource is included multiple times, it will only be included once. Thanks Ying Zhong for the bug report and suggested fix.

2.5.7 0.1 (2014-07-07)

- Initial public release.

2.6 History

2.6.1 hurry.resource

In 2008 I (Martijn Faassen) built a library called hurry.resource. It could automatically insert the required `<script>` and `<include>` tags into HTML. You could describe these resources in Python code. It was aware of resource dependencies, and also had a facility to automatically included minified versions of particular resources, or bundled versions that included a number of files.

I used `hurry.resource` in the context of applications based on the Grok framework. The Grok integration was involved; you had to hook in at the right place to manipulate the HTML, and `hurry.resource` did not serve static resources itself; it left that up to the web framework too. While I had written `hurry.resource` to be web-framework independent, to my knowledge nobody used it outside of Grok/Zope 3.

`hurry.resource` in turn was inspired by a library called zc.resourcelibrary, which did much the same but had a more limited way to describe resources. The resource metadata system was inspired by the system in YUI 2.

2.6.2 Fanstatic

In 2010, I, Jan-Wijbrand Kolman and Jan-Jaap Driessen rewrote `hurry.resource` into a more capable library. We had the realization that by going with WSGI and by making the system *serve* resources as well, we could create a true web framework for static resources. We decided to rebuild `hurry.resource` into Fanstatic.

We were also inspired by the capabilities of z3c.hashedresource, a library for Zope 3 that could generate cache-busting URLs that aid caching and development (see *Caching*). Since Fanstatic controlled both creating inclusions for resources as well as serving them, we could bring cache busting behavior into Fanstatic.

Another clever hack of Fanstatic was to leverage the Python packaging infrastructure (PyPI, setuptools, etc) to distribute static resources and their descriptions. This way we could easily install a variety of client-side libraries, as long as someone had wrapped them using Fanstatic. The community wrapped quite a few libraries.

Unlike `hurry.resource`, Fanstatic is easy to integrate into any WSGI-based web framework. This helped Fanstatic to become a moderately successful open source project. It was adopted not only by Grok users, but also by many others that use WSGI-based web frameworks. We got quite a few contributions, and a range of advanced features were added to Fanstatic beyond that `hurry.resource` already provided.

2.6.3 BowerStatic

A bottleneck of Fanstatic is that someone needs to sit down and write a Python package for each JavaScript project out there. This takes time. To upgrade a package to a newer version can be cumbersome. Fanstatic makes the developer of Python wrapper library the intermediary, and while this intermediary can add value, they can also be an obstacle.

By 2014, a lot had changed in the client-side world. Fanstatic's reliance on the Python packaging infrastructure was turning from an advantage into a drawback. Bower has become the de-facto way for many client-side libraries to be distributed and installed. Faced with the task to wrap a range of JavaScript libraries using Fanstatic and then maintain those wrapping libraries, I decided to give Fanstatic a rethink instead.

Using the Bower package manager, we can install client-side components without having to go through an intermediary.

Fanstatic has another limitation: just like in Python, you can only have one version of a library installed per project. I was facing a use case where this was not desirable: a large platform with multiple sub-projects that might want to use divergent versions of their client-side components.

So I started thinking about what a static web framework might look like that uses Bower as its underlying packaging system, while retaining some important features of Fanstatic, like automating insertion of link and script tags, static resource serving, and caching.

BowerStatic was born.

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*