bricks Documentation

Release 0.2.1

F1bio MacMendes

Apr 29, 2017

Contents

Warning: Beta software You are using a software that has not reached a stable version yet. Please beware that interfaces might change, APIs might disappear and general breakage can occur before *1.0*.

If you plan to use this software for something important, please read the roadmap, and the issue tracker in Github. If you are unsure about the future of this project, please talk to the developers, or (better yet) get involved with the development of Django Bricks!

Django Bricks is a library for creating web components for Django. A Brick is a reusable object with well defined server-side and client-side behaviors and interfaces. Django Bricks allow us to build reusable pieces of functionality in Django while avoiding a little bit of HTML and Javascript ;)

CHAPTER 1

Overview

Django web components

Django-brick is a library that implements server-side web components for your Django application. The goal is to reuse code by building simple pluggable pieces. Think of Lego bricks for the web.

alt https://upload.wikimedia.org/wikipedia/commons/thumb/3/32/Lego_Color_Bricks.jpg/1024px-Lego_Color_Bricks.jpg

Client-side programming has plenty responses for this task: React, Polymer, Vue.js, X-tag etc. Django Bricks provides a server-side alternative that can free you from writing some JavaScript and HTML ;).

Enter the brick

A brick is a Python component with a a well defined interface to present itself for the client. Usually this means that it can render itself as HTML5 (but sometimes we may need more complicated behaviors). Pehaps the most simple brick that you can use is just a HTML5 tag. Django-bricks implement these building blocks in the `bricks.html5` module. The most important action a `bricks.Tag` brick can make is to render itself as HTML:

```
>>> from bricks.html5 import p
>>> elem = p("Hello World!", class_='hello')
```

This can be converted to HTML by calling `str()` on the element:

```
>>> print(str(elem))
<p class="hello">Hello World!</p>
```

Python and HTML have very different semantics. HTML's syntax gravitates around tag attributes + children nodes and does not have a very natural counterpart in most programming languages. Of course we can build a tag in a imperative style, but the end result often feels awkward. We introduce a mini-language to declare HTML fragments in a more natural way:

```
>>> from bricks.html5 import div, p, h1
>>> fragment = \
...     div(class_="alert-box")[
...         h1('Hello Python'),
...         p('Now you can write HTML in Python!'),
...     ]
```

By default, bricks convert it to a very compact HTML; we insert no indentation and only a minimum whitespace. We can pretty print the fragment using the `.pretty` method:

```
>>> print(fragment.pretty())
<div class="alert-box">
  <h1>Hello Python</h1>
  <p>Now you can write HTML in Python!</p>
</div>
```

This is useful for debugging but, it is recommend to never output prettified HTML in production. This just stresses the rendering engine and produces larger files for no real gain for our end users and developers.

CHAPTER 2

Installation instructions

Django Bricks can be installed using pip:

```
$ python -m pip install django-bricks
```

This command will fetch the archive and its dependencies from the internet and install them.

If you've downloaded the tarball, unpack it, and execute:

```
$ python setup.py install --user
```

You might prefer to install it system-wide. In this case, skip the `--user` option and execute as superuser by prepending the command with `sudo`.

Troubleshoot

Windows users may find that these command will only works if typed from Python's installation directory.

Some Linux distributions (e.g. Ubuntu) install Python without installing pip. Please install it before. If you don't have root privileges, download the get-pip.py script at https://bootstrap.pypa.io/get-pip.py and execute it as `python get-pip.py --user`.

CHAPTER 3

Django configuration

The first step is to add Bricks to your installed apps:

```
# settings.py
# ...

INSTALLED_APPS = [
    'bricks.app',
]
```

You're done :)

(Actually we have to improve this section of our documentation. Come later to check it out. But seriously, you can use bricks already!)

CHAPTER 4

Templating

The goal of `bricks.html5` is to replace your template engine by Python code that generates HTML fragments. This approach removes the constraints imposed by the template language and makes integration with surrounding Python code trivial.

I know what you are thinking: *"it is a really bad idea to mix template with logic"*. Bricks obviously doesn't prevent you from shooting yourself on the foot and you can make really messy code if you want. However, things can be very smooth if you stick to focused and simple components that adopt a more functional style.

Our advice is: *break your code in small pieces and compose these pieces in simple and predictable ways*. Incidentally, this is a good advice for any form of code ;).

The fact is that our good old friend *"a function"* is probably simpler to use and composes much better than anything a templating engine has come up with.

Let us dive in!

We want to implement a little Bootstrap element that shows a menu with actions (this is a random example taken from Bootstrap website).

```
<div class="btn-group">
  <button type="button"
          class="btn btn-default dropdown-toggle"
          data-toggle="dropdown"
          aria-haspopup="true"
          aria-expanded="false">
    Action <span class="caret"></span>
  </button>
  <ul class="dropdown-menu">
    <li><a href="#">Action</a></li>
    <li><a href="#">Another action</a></li>
    <li><a href="#">Something else here</a></li>
    <li role="separator" class="divider"></li>
    <li><a href="#">Separated link</a></li>
  </ul>
</div>
```

Of course we could translate this directly into bricks code by calling the corresponding `div()`'s, `button()`'s, etc. But first, let us break up this mess into smaller pieces.

```
from bricks.html5 import button, div, p, ul, li, span

def menu_button(name, caret=True):
    return \
        button(type='button',
               class_='btn btn-default dropdown-toggle',
               data_toggle="dropdown",
               aria_haspopup="true",
               aria-expanded="false")[
            name,
            span(class_='caret') if caret else None,  # Nones are ignored
        ]
```

Ok, it looks like it's a lot of trouble for a simple component. But now we can reuse this piece and easily make as many buttons as we like: `menu_button('File'), menu_button('Edit'), ...`. The next step is to create a function that takes a list of strings and return the corresponding menu (in the real world we might also want to control the href attribute). We are also going to be clever and use the Ellipsis (`...`) as a menu separator.

```
def menu_data(values):
    def do_item(x):
        if x is ...:
            return li(role='separator', class='divider')
        else:
            # This could parse the href from string, or take a tuple
            # input, or whatever you like. The bricks.helpers.link function
            # can be handy here.
            return li[a(href='#')[x]]

    return \
        ul(class_='dropdown-menu')[
            map(do_item, values)
        ]
```

We glue both together...

```
def menu(name, values, caret=True):
    return \
        div(class_='btn-group')[
            menu_button(name, caret=True),
            menu_data(values),
        ]
```

... and create as many new menu buttons as we like:

```
menubar = \
    div(id='menubar')[
        menu('File', ['New', 'Open', ..., 'Exit']),
        menu('Edit', ['Copy', 'Paste', ..., 'Preferences']),
        menu('Help', ['Manual', 'Topics', ..., 'About']),
    ]
```

Look how nice it is now :)

The with statement

If you have more complex logic the "with" syntax can be handy.

```
>>> with div(class_='card') as fragment:
...     +h1('Multi-hello')
...     for i in range(1, 4):
...         +p('hello %s' % i)
>>> print(fragment.pretty())
<div class="card">
  <h1>Multi-hello</h1>
  <p>hello 1</p>
  <p>hello 2</p>
  <p>hello 3</p>
</div>
```

The unary + operator says *"add me to the node defined in the last with statement"*. Nested `with` statements are also supported.

How does it work?

Bricks HTML syntax is obviously just regular Python wrapped in a HTML-wannabe DSL. How does it work?

Take the example:

```
element = \
    div(class_="contact-card")[
        span("john", class_="contact-name"),
        span("555-1234", class_="contact-phone"),
    ]
```

The first positional argument is a single child element or a list of children. Keyword arguments are interpreted as tag attributes. Notice we did not use `class` as an argument name because it is a reserved keyword in Python. Bricks, however, ignores all trailing underscores and converts underscores in the middle of the argument to dashes.

If your tag uses underscore in any attribute name or if you happen to have the attributes to values stored in a dictionary, just use the `attrs` argument of a tag constructor.

```
# <div my_attr="1" attrs="2" data-attr="3">foo</div>

div('foo', attrs={'my_attr': 1, 'attrs': 2}, data_attr=3)
```

Imperative API

The contact-card element above could have been created in a more regular imperative fashion:

```
element = div(class_="contact-card")
span1 = span("john", class_="contact-name")
span2 = span("555-1234", class_="contact-phone")
element.children.extend([span1, span2])
```

This is not as expressive as the first case and forces us to think *imperative* instead of thinking in *declarative markup*. This is not very natural for HTML and also tends to be more verbose. The "square bracket syntax" is just regular Python indexing syntax abused to call `.children.extend` to insert child elements into the tag's children attribute.

More specifically, the `tag[args]` creates a copy of the original tag, flatten all list and tuple arguments, insert them into the copied object, and return it. The same hack is applied to the metaclass and this allow us to call tags that do not define any attribute like this:

```
element = \
    div[
        span('Foo'),
        span('Bar'),
    ]
```

And since lists, tuples, mappings, and generators are flattened, we can also define a tag's children with list comprehensions and maps:

```
element = \
    div[
        [span(x) for x in words],
        map(lambda x, y: a(x, href=b), words, hyperlinks),
    ]
```

Since square brackets were already taken to define the children elements of a tag, we cannot use them to directly access the children elements of a tag. Instead, this must be done explicitly using the `tag.children` interface. It behaves just as a regular list so you can do things as

```
>>> elem = div('foo', class_='elem')
>>> elem.children.append('Hello world')
>>> first = elem.children.pop(0)
>>> print(elem)
<div class="elem">Hello world</div>
```

Similarly to children, attributes are also exposed in a special attribute named *attrs* that behaves like a dictionary:

```
>>> elem = div('foo', class_='elem')
>>> elem.attrs['data-answer'] = 42
>>> elem.attrs.keys()
dict_keys(['class', 'data-answer'])
```

The attrs dictionary also exposes the `id` and `class` elements as read-only values. `id` is also exposed as an attribute and `class` is constructed from the list of classes in the `tag.classes` attribute.

```
>>> elem = div('foo', class_='class', id='id')
>>> elem.id, elem.classes
('id', ['class'])
>>> elem.id = 'new-id'
>>> print(elem)
<div id="new-id" class="class">foo</div>
```

CHAPTER 5

Javascript API

Basic API

bricks(*api_name*, *{args}*)

This function call a registered function in the server and return its result.

It can be called either with a pure positional or pure named arguments form.

bricks('api-name', {arg1: value1, arg2: value2, ...}): The most common form of remote call requires named arguments.

bricks('api-name*', arg1, arg2, arg3, ...): This will call the remote api function with the given arguments and return the result. An asterisk in the end of the api function name tells that it expect positional arguments only. This is required if you want to pass a single positional argument that is an object in order to avoid bricks RPC to interpret it as a dictionary of named arguments.

This function returns a jQuery promise and callbacks can be attached to it using the `.then()`, `.done()`, `.fail()`, etc methods:

```
bricks('get-user-data', 'user123'))
    .then(function(result) {
        // do something with the result
    })
    .then(function(result) {
        // do something else
    });
```

In Django, api functions are registered using the `@bricks.rpc.api` decorator to a function. These functions always receive a request as the first argument, followed by the arguments passed from javascript. The return value is transmitted back to the client and returned to the caller.

```
import bricks

@bricks.rpc.api
def function(request, arg1, arg2, arg3, ...):
```

```
    ...
    return value
```

All exceptions raised in python-land are transmitted to javascript, adapted, and re-raised there.

Remember that all communication is done through JSON streams, hence all input arguments and the resulting value must be JSON-encodable.

See Also: *bricks.sync()* - Synchronous call (for debug purposes)

bricks.**sync**(*api_name*, *{args}*)

This function accepts the same signature but immediately returns the result. Synchronous AJAX functions should never be used in production since they lock the client until the request is completed, degrading user experience. You won't notice it testing locally since the latency is so low, but surely a client in a slow internet connection with think your site is broken.

This function exists for debug purposes only.

bricks.**call**(*api_name*, *{args}*)

Like the regular bricks function, but will not run any program returned by the server.

```
import bricks

@bricks.api
def crazy_function(client, arg1, arg2, ...):
    client.alert("The server is crazy!")
    client.jquery('div').hide()
    return 42
```

Using `bricks.call()` prevents the client code from executing.

```
bricks.call('crazy-function')
    .then(function(result) {
        console.log('the answer is ' + result)
    })
```

It will not hide any div or show any javascript alert.

bricks.**js**(*api_name*, *{args}*)

Consumes an API entry point that simply returns some javascript code and immediately execute it.

In Django, functions those entry points are registered using the `@bricks.rpc.js` decorator:

```
import bricks

@bricks.rpc.js
def js_maker(request, arg1, arg2, arg3, ...):
    return string_of_javascript_code()
```

bricks.**rpc**(*api_name*, *options*)

The workhorse behind *bricks()*, *bricks.call()*, *bricks.js()* and `bricks.html()` functions. It receives a single object argument that understands the following parameters

Args:

api: Api name of the called function/program

params: List of positional arguments to be passed to the calling function.

kwargs: An object with all the named arguments.

server: Override the default server root. Usually bricks will open the URL at http://<localdomain>/bricks/api-function-name.

async: If true, returns a promise. Otherwise, it blocks execution and returns the result of the function call.

method: Can be any of 'api', 'program', 'js', or 'html'.

program: If true (default), execute any received programmatic instructions.

error: If true (default), it will raise any exceptions raised by the remote call.

result: If given, will determine the result value of the function call.

timeout: Maximum amount of time (in seconds) to wait for a server response. Default to 30.0.

converter: A function that process the resulting JSON result and convert it to the desired value.

The `bricks.json` module

The `bricks.json` module defines a few functions for handling the bricks flavored JSON. The API was modeled after Python's json module rather than Javascript.

Supported types

Besides regular JSON types, the js-client for bricks also implement a few additional data types.

Type name (@)	Python	Javascript	Notes
datetime	datetime.datetime	Date	

`bricks.json.`**`encode`**(*obj*)
Encode object into a bricks-flavored JSON-compatible structure.

`bricks.json.`**`decode`**(*obj*)
Return the Javascript object equivalent to the given bricks-flavored JSON-compatible structure.

`bricks.json.`**`dumps`**(*obj*)
Stringfy javascript object to a bricks-flavored JSON stream.

`bricks.json.`**`loads`**(*String data*)
Load javascript object from a bricks-flavored JSON encoded string.

CHAPTER 6

Bricks flavored JSON

Bricks uses JSON as a data serialization format for server/client and P2P communication. JSON is obviously constrained to just a few primitive data types. In order to serialize more complex data, it is usually necessary to transform it to a JSON-friendly format such as a dictionary or list (a.k.a Objects and Arrays, in JavaScript parlance).

This approach is fine if the receiving end of communication knows exactly which kind of data to expect and how to construct the desired object from JSON. Bricks defines a simple protocol to handle extension types: all non-primitive types must define an '@' key mapping to the extension type name.

```
{
   '@': 'datetime',
   'year': 1982,
   'month': 1,
   'day': 6,
   'hour': 12,
   'minute': 52,
   'second': 0,
   'microsecond': 0,
}
```

Of course both ends of communication must still agree on how to serialize/deserialize this data type. Bricks makes no attempt to define an schema or any kind of formal description of a data type and its validation. These are orthogonal concerns that could be handled by third party libraries.

Bricks implements transformation for a few common Python types (such as the datetime example given bellow) and **bricks.js** handles them in Javascript when feasible.

The `bricks.json` module

The *`bricks.json`* provides basic functions for manipulating JSON data.

Conversion from/to JSON is handled by the *`bricks.json.decode()`* and `brick.json.encode()` functions:

```
>>> from bricks.json import encode, decode
>>> encode({1, 2, 3})
{
    '@': 'set',
    'data': [1, 2, 3],
}
>>> decode({'@': 'set', 'data': [1, 2, 3]})
{1, 2, 3}
```

Custom types

Support for custom types is given by the

CHAPTER 7

Remote Procedure Calls (RPC)

Bricks provides a module for simple RPC based on the *JSON-RPC 2.0 spec <http://www.jsonrpc.org/specification>*. The module implements views that can be used either by the frontend through our JavaScript library or by other Python programs to call Bricks RPC endpoints. The later can be useful in a distributed server architecture, where one server can call endpoints defined by other.

CHAPTER 8

API Reference

API documentation for the Django Bricks module.

Components

Helper functions

Helper functions generate safe HTML code fragments for several useful situations.

Rendering

bricks.helpers.**render**(*obj*, *request=None*, ***kwargs*)

Renders object as a safe HTML string.

This function uses single dispatch to make it extensible for user defined types:

```
@render.register(int)
def _(x, **kwargs):
    if x == 42:
        return safe('the answer')
    else:
        return safe(x)
```

A very common pattern is to render object from a template. This has specific support:

```
render.register_template(UserProfile, 'myapp/user_profile.jinja2')
```

By default, it populates the context dictionary with a snake_case version of the class name, in this case, `{'user_profile': x}`. The user may pass a `context` keyword argument to include additional context data.

If you want to personalize how this is done, it is possible to use register_template to register a context factory function. The function should receive the object, a request and kwargs:

```
@render.register_template(UserProfile, 'myapp/user_profile.jinja2')
def context(profile, request=None, context=None, **kwargs):
    context = context or {}
    context.update(kwargs, request=request, profile=profile)
    return context
```

Notes

All implementations must receive the user-defined object as first argument and accept arbitrary keyword arguments.

bricks.helpers.**render_tag**(*tag*, *data=None*, *attrs=None*, *children_kwargs=None*, *request=None*, ***attrs_kwargs*)

Renders HTML tag.

Parameters

- **tag** – Tag name.
- **data** – Children elements for the given tag. Each element is rendered with the render_html() function.
- **attrs** – A dictionary of attributes.
- **request** – A request object that is passed to the render function when it is applied to children.
- ****attr_kwargs** – Keyword arguments are converted to additional attributes.

Examples

```
>>> render_tag('a', 'Click me!', href='www.python.org')
'<a href="www.python.org">Click me!</a>
```

bricks.helpers.**markdown**(*text*, ***, *output_format='html5'*, ***kwargs*)

Renders Markdown content as HTML and return as a safe string.

Escaping

bricks.helpers.**safe**(*x*)

Convert string object to a safe Markup instance.

bricks.helpers.**escape**(*s*) → markup

Convert the characters &, <, >, ‘, and ” in string s to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. Marks return value as markup string.

bricks.helpers.**escape_silent**(*s*) → markup

Like escape but converts None to an empty string.

bricks.helpers.**unescape**(*s*)

Convert all named and numeric character references (e.g. >, >, &x3e;) in the string s to the corresponding unicode characters. This function uses the rules defined by the HTML 5 standard for both valid and invalid character references, and the list of HTML 5 named character references defined in html.entities.html5.

bricks.helpers.**sanitize**(*data*, ***kwargs*)
Sanitize HTML and return as a safe string.

Utilities

bricks.helpers.**attr**(*x*, ***kwargs*)
Renders object as an HTML attribute value.

It define the following dispatch rules:

str: Quotations and & are escaped, any other content, including <, >, is allowed.

numeric types: Are simply converted to strings.

lists and mappings: Are converted to JSON and returned as safe strings. This is used in some modern javascript frameworks reads JSON from tag attributes.

bricks.helpers.**attrs**(*x*, ***kwargs*)
Convert object into a list of key-value HTML attributes.

Parameters

- **uses multiple dispatch, so the behaviour might differ a little bit** (*It*) –
- **o the first argument.** (*depending*) –
- **mappings** – Renders key-values into the corresponding HTML results.
- **sequences** – Any non-string sequence is treated as sequence of (key, value) pairs. If any repeated keys are found, it keeps only the last value.
- **protocol** (**attrs**) – Any object that define an `attrs` attribute that can be either a mapping or a sequence of pairs.
- **all cases, attrs takes arbitrary keyword attributes that are** (*In*) –
- **as additional attributes. PyML converts all underscores** (*interpreted*) –
- **in the attribute names to dashes since this is the most common** (*present*) –
- **in HTML.** (*convention*) –

bricks.helpers.**hyperlink**(*x*, *href=None*, *attrs=None*, ***kwargs*)
Creates an hyperlink string from object and renders it as an <a> tag.

It implements some common use cases:

str: Renders string as content inside the <a>... tags. Additional options including href can be passed as keyword arguments. If no href is given, it tries to parse a string of "Value <link>" and uses href='#' if no link is found.

dict or mapping: Most keys are interpreted as attributes. The visible content of the link must be stored in the 'content' key:

```
>>> hyperlink({'href': 'www.python.com', 'content': 'Python'})
<a href="www.python.com">Python</a>
```

django User: You must monkey-patch to define get_absolute_url() function. This function uses this result as the href field.

The full name of the user is used as the hyperlink content.

```
>>> hyperlink(User(first_name='Joe', username='joe123'))
<a href="/users/joe123">Joe</a>
```

In order to support other types, use the lazy_singledispatch mechanism:

```
@hyperlink.register(MyFancyType)
def _(x, **kwargs):
    return safe(render_object_as_safe_html(x))
```

See also:

pyml.helpers.attrs(): See this function for an exact explanation of how keyword arguments are translated into HTML attributes.

Client and JavaScript emulation

Bricks flavored JSON

Serialize/deserialize Bricks flavored JSON (see *Bricks flavored JSON*).

Functions

bricks.json.**encode**(*data*)
Encode some arbitrary Python data into a JSON-compatible structure.

This naive implementation does not handle recursive structures. This might change in the future.

This function encode subclasses of registered types as if they belong to the base class. This is convenient, but is potentially fragile and make the operation non-invertible.

bricks.json.**decode**(*data*)
Decode a JSON-like structure into the corresponding Python data.

bricks.json.**dumps**(*obj*)
Return a JSON string dump of a Python object.

bricks.json.**loads**(*data*)
Load a string of JSON-encoded data and return the corresponding Python object.

bricks.json.**register**(*cls*, *name=None*, *encode=None*, *decode=None*)
Register encode/decode pair of functions for the given Python type. Registration extends Bricks flavored JSON to handle arbitrary python objects.

Parameters

- **cls** – python data type
- **name** – name associated with the '@' when encoded to JSON.
- **encode** – the encode function; convert object to JSON. The resulting JSON can have non-valid JSON types as long as they can be also converted to JSON using the *bricks.json.encode()* function.

- **decode** – decode function; converts JSON back to Python. The decode function might assume that all elements were already converted to their most Pythonic forms (i.e., all dictionaries with an '@' key were already decoded to their Python forms).

See also:

Custom types

Errors

class bricks.json.**JSONDecodeError**
Error raised when decoding a JSON structure to a Python object.

class bricks.json.**JSONEncodeError**
Error raised when encoding a Python object to JSON.

Remote procedure calls (RPC)

JSON-RPC 2.0 utilities.

Functions and decorators

bricks.rpc.**jsonrpc_endpoint**(*login_required=False*, *perms_required=None*)
Decorator that converts a function into a JSON-RPC enabled view.

After using this decorator, the function is not usable as a regular function anymore.

```
@jsonrpc_endpoint(login_required=True)
def add_at_server(request, x=1, y=2):
    return x + y
```

Generic views

class bricks.rpc.**RPCView**(*function*, *action='api'*, *login_required=False*, *perms_required=None*, *request_argument=True*, ***kwds*)
Wraps a Bricks RPC end point into a view.

Parameters

- **function** – (required) The function that implements the given API.
- **login_required** – If True, the API will only be available to logged in users.
- **perms_required** – The list of permissions a user can use in order gain access to the API. A non-empty list implies login_required.

check_credentials(*request*)
Assure that user has the correct credentials to the process.

Must raise a BadResponseError if credentials are not valid.

execute(*request*, *data*)
Execute the API function and return a dictionary with the results.

get_content_type()
Content type of the resulting message.

For JSON, it returns 'application/json'.

get_data(*request*)
Decode and return data sent by the client.

get_raw_response(*request*, *data*)
Return the payload that will be sent back to the client.

The default implementation simply converts data to JSON.

post(*request*, **args*, ***kwargs*)
Process the given request, call handler and return result.

wrap_error(*ex*, *tb=None*, *wrap_permission_errors=False*)
Wraps an exception raised during the execution of an API function.

Routing

Contrib modules

CHAPTER 9

Frequently asked questions

Usage

Why is this file empty?

Because this project is in its infancy ;)

We don't know which questions are the most "frequent" yet. If you have doubts go to our issue tracker in github and post a question. If we have to answer it twice, it might end up in this section.

CHAPTER 10

License

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

Index