lazr.batchnavigator Documentation

Release 2.0.0

LAZR Developers <lazr-developers@lists.launchpad.net>'

Jun 20, 2023

CONTENTS

Batch navigation provides a way to navigate batch results in a web page by providing URL links to the next, previous and numbered pages of results.

It uses four query/POST arguments to control the batching:

- memo: A record of the underlying storage index pointer for the position of the batch.
- direction: Indicates whether the memo is at the start or end of the batch.
- start: Cosmetic - used to calculate the apparent location (but note that due to the concurrent nature of repeated visits to batches that the true offset may differ - however the collection won't skip or show items twice. For compatibility with saved URLs, if memo and direction are both missing then start is used to do list slicing into the collection.
- batch: Controls the amount of items we are showing per batch. It will only appear if it's different from the default value set when the batch is created.

These values can be overriden in the request, unless you also pass force_start=True, which will make the start argument (again, defaulting to 0) always chosen.

Imports:

```
>>> from lazr.batchnavigator import BatchNavigator, ListRangeFactory
>>> from zope.publisher.browser import TestRequest
>>> from zope.publisher.http import HTTPCharsets
>>> from zope.component import getSiteManager
>>> sm = getSiteManager()
>>> sm.registerAdapter(HTTPCharsets)
```

```
>>> def build_request(query_string_args=None, method='GET'):
...   if query_string_args is None:
...       query_string = ''
...   else:
...       if getattr(query_string_args, 'items', None) is not None:
...           query_string_args = query_string_args.items()
...       query_string = "&".join(
...           ["%s=%s" % (k,v) for k,v in query_string_args])
...   request = TestRequest(SERVER_URL='http://www.example.com/foo',
...                                 method=method,
...                                 environ={'QUERY_STRING': query_string})
...   request.processInputs()
...   return request
```

A sample request object:

Some sample data.

```
>>> reindeer = ['Dasher', 'Dancer', 'Prancer', 'Vixen', 'Comet',
...             'Cupid', 'Donner', 'Blitzen', 'Rudolph']
```

Because slicing large collections can be very expensive, BatchNavigator offers a non-slice protocol for determining the edge of batches. The range_factory supplies an object implementing IRangeFactory and manages this protocol. ListRangeFactory is a simple included implementation which BatchNavigator will use if no range_factory is supplied.

```
>>> _ = BatchNavigator(reindeer, build_request(),
...     range_factory=ListRangeFactory(reindeer))
```

For the examples in the documentation we let BatchNavigator construct a range_factory implicitly:

```
>>> safe_reindeer = reindeer
>>> safe_reindeer_batch_navigator = BatchNavigator(
...     safe_reindeer, build_request(), size=3)
```

An important feature of lazr.batchnavigator is its reluctance to invoke len() on an underlying data set. len() can be an expensive operation that provides little benefit, so this library tries hard to avoid calling len() unless it's absolutely necessary. To show this off, we'll define a subclass of Python's list type that explodes when len() is invoked on it.

```
>>> class ListWithExplosiveLen(list):
...     """A list subclass that doesn't like its len() being called."""
...     def __len__(self):
...         raise RuntimeError
```

Unless otherwise stated, we will use this list exclusively throughout this test, to verify that len() is never called unless we want it to be.

```
>>> explosive_reindeer = ListWithExplosiveLen(reindeer)
>>> reindeer_batch_navigator = BatchNavigator(
...     explosive_reindeer, build_request(), size=3)
```

The BatchNavigator implements IBatchNavigator. We need to use the 'safe' batch navigator here, because verifyObject probes all methods of the object it's passed, including __len__.

```
>>> from zope.interface.verify import verifyObject
>>> from lazr.batchnavigator.interfaces import IBatchNavigator
```

```
>>> verifyObject(IBatchNavigator, safe_reindeer_batch_navigator)
True
```

The BatchNavigator class provides IBatchNavigatorFactory. This can be used to register a batch navigator factory as a utility, for instance.

```
>>> from lazr.batchnavigator.interfaces import IBatchNavigatorFactory
```

```
>>> verifyObject(IBatchNavigatorFactory, BatchNavigator)
True
```

You can ask the navigator for the chunk of results currently being shown (e.g. to iterate over them for rendering in ZPT):

```
>>> list(reindeer_batch_navigator.currentBatch())
['Dasher', 'Dancer', 'Prancer']
```

You can ask for the first, previous, next and last results' links:

```
>>> reindeer_batch_navigator.firstBatchURL()
''
>>> reindeer_batch_navigator.prevBatchURL()
''
>>> reindeer_batch_navigator.nextBatchURL()
'http://www.example.com/foo?memo=3&start=3'
```

There's no way to get the URL to the final batch without knowing the length of the entire list, so we'll use the safe batch navigator to demonstrate lastBatchURL():

```
>>> safe_reindeer_batch_navigator.lastBatchURL()
'http://www.example.com/foo?direction=backwards&start=6'
```

The next link will be empty when there are no further results:

```
>>> request = build_request({"start": "3", "batch": "20"})
>>> last_reindeer_batch_navigator = BatchNavigator(reindeer, request=request)
>>> last_reindeer_batch_navigator.nextBatchURL()
''
```

The first and previous link should appear even when we start at a point between 0 and the batch size:

```
>>> request = build_request({"start": "2", "batch": "3"})
>>> last_reindeer_batch_navigator = BatchNavigator(reindeer, request=request)
```

Here, we can see too that the batch argument appears as part of the URL. That's because the request asked for a different size than the default one when we create the Batch object, by default, it's 5.

```
>>> last_reindeer_batch_navigator.firstBatchURL()
'http://www.example.com/foo?batch=3'
```

```
>>> last_reindeer_batch_navigator.prevBatchURL()
'http://www.example.com/foo?batch=3&direction=backwards&memo=2'
```

This all works with other values in the query string, too:

```
>>> request = build_request({'fnorb': 'bar',
...                          'start': '3',
...                          'batch': '3'})
>>> reindeer_batch_navigator_with_qs = BatchNavigator(
...     reindeer, request, size=3)
>>> safe_reindeer_batch_navigator_with_qs = BatchNavigator(
...     safe_reindeer, request, size=3)
```

In this case, we created the BatchNavigator with a default size of '3' and the request is asking exactly that number of items per batch, and thus, we don't need to show 'batch' as part of the URL.

```
>>> reindeer_batch_navigator_with_qs.firstBatchURL()
'http://www.example.com/foo?fnorb=bar'
>>> reindeer_batch_navigator_with_qs.prevBatchURL()
'http://www.example.com/foo?fnorb=bar&direction=backwards&memo=3'
>>> reindeer_batch_navigator_with_qs.nextBatchURL()
'http://www.example.com/foo?fnorb=bar&memo=6&start=6'
```

(Again, there's no way to get the last batch without knowing the size of the entire list.)

```
>>> safe_reindeer_batch_navigator_with_qs.lastBatchURL()
'http://www.example.com/foo?fnorb=bar&direction=backwards&start=6'
```

The `force_start` argument allows you to ignore the start value in the request. This can be useful when, for instance, a filter has changed, and the desired behavior is to restart at 0.

```
>>> reindeer_batch_navigator_with_qs = BatchNavigator(
...     reindeer, request, size=3, force_start=True)
```

(continues on next page)

(continued from previous page)

```
>>> reindeer_batch_navigator_with_qs.currentBatch().start
0
>>> reindeer_batch_navigator_with_qs.nextBatchURL()
'http://www.example.com/foo?fnorb=bar&memo=3&start=3'
>>> reindeer[:3] == list(reindeer_batch_navigator_with_qs.currentBatch())
True
```

We ensure that batch arguments supplied in the URL are observed for POST operations too:

```
>>> request = build_request({'fnorb': 'bar',
...                          'start': '3',
...                          'batch': '3'}, method='POST')
>>> reindeer_batch_navigator_post_with_qs = BatchNavigator(
...     reindeer, request)
```

```
>>> reindeer_batch_navigator_post_with_qs.start
3
>>> reindeer_batch_navigator_post_with_qs.nextBatchURL()
'http://www.example.com/foo?fnorb=bar&batch=3&memo=6&start=6'
```

We ensure that multiple size and batch arguments supplied in the URL don't blow up the application. The first one is preferred.

```
>>> request = build_request(
...     [('batch', '1'), ('batch', '7'), ('start', '2'), ('start', '10')])
>>> navigator = BatchNavigator(reindeer, request=request)
>>> navigator.nextBatchURL()
'http://www.example.com/foo?batch=1&memo=3&start=3'
```

The batch argument must be positive. Other numbers are ignored, and the default batch size is used instead.

```
>>> from six.moves.urllib.parse import parse_qs
>>> request = build_request({'batch': '0'})
>>> navigator = BatchNavigator(range(99), request=request)
>>> print('batch' in parse_qs(navigator.nextBatchURL()))
False
```

```
>>> request = build_request({'batch': '-1'})
>>> navigator = BatchNavigator(range(99), request=request)
>>> print('batch' in parse_qs(navigator.nextBatchURL()))
False
```

CHAPTER

ONE

EMPTY BATCHES

You can also create an empty batch that will not have any items:

```
>>> null_batch_navigator = BatchNavigator(
...     None, build_request(), size=3)
>>> null_batch_navigator.firstBatchURL()
''
>>> null_batch_navigator.nextBatchURL()
''
>>> null_batch_navigator.prevBatchURL()
''
>>> null_batch_navigator.lastBatchURL()
''
```

```
>>> null_batch_navigator = BatchNavigator(
...     [], build_request(), size=3)
>>> null_batch_navigator.firstBatchURL()
''
>>> null_batch_navigator.nextBatchURL()
''
>>> null_batch_navigator.prevBatchURL()
''
>>> null_batch_navigator.lastBatchURL()
''
```

TODO:

- blowing up when start is beyond end
- orphans
- overlap

CHAPTER

TWO

SUPPORTING RESULTS WITHOUT A __LEN__

Some result objects do not implement __len__ because generally Python code assumes that __len__ is cheap. SQLObject and Storm result sets both have this behavior, for instance, so that it is cleat that getting the length is a non- trivial operation.

To support these objects, the batch looks for __len__ on the result set. If it does not exist, it adapts the result to zope.interface.common.sequence.IFiniteSequence and uses that __len__.

```
>>> class ExampleResultSet(object):
...     def __init__(self, results):
...         self.stub_results = results
...     def count(self):
...         # imagine this actually returned
...         return len(self.stub_results)
...     def __getitem__(self, ix):
...         return self.stub_results[ix] # also works with slices
...     def __iter__(self):
...         return iter(self.stub_results)
...
>>> from zope.interface import implementer
>>> from zope.component import adapter, getSiteManager
>>> from zope.interface.common.sequence import IFiniteSequence
>>> @adapter(ExampleResultSet)
... @implementer(IFiniteSequence)
... class ExampleAdapter(ExampleResultSet):
...     def __len__(self):
...         return self.stub_results.count()
...
>>> sm = getSiteManager()
>>> sm.registerAdapter(ExampleAdapter)
>>> example = ExampleResultSet(safe_reindeer)
>>> example_batch_navigator = BatchNavigator(
...     example, build_request(), size=3)
>>> example_batch_navigator.currentBatch().total()
9
```

CHAPTER

THREE

ONLY GETS WHAT IS NEEDED

It's also important for performance of batching large result sets that the batch only gets a slice of the results, rather than accessing the entirety.

```
>>> class ExampleResultSet(ExampleResultSet):
...     def __init__(self, results):
...         super(ExampleResultSet, self).__init__(results)
...         self.getitem_history = []
...     def __getitem__(self, ix):
...         self.getitem_history.append(ix)
...         return super(ExampleResultSet, self).__getitem__(ix)
...
```

```
>>> example = ExampleResultSet(reindeer)
>>> example_batch_navigator = BatchNavigator(
...     example, build_request(), size=3)
>>> reindeer[:3] == list(example_batch_navigator.currentBatch())
True
>>> example.getitem_history
[slice(0, 4, None)]
```

Note that although the batch is of the size requested, the underlying list contains one more item than is necessary. This is to make it easy to determine whether a given batch is the final one in the list, without having to explicitly look up the length of the list (potentially an expensive operation).

CHAPTER

FOUR

ADDING CALLBACK FUNCTIONS

Sometimes it is useful to have a function called with the batched values once they have been determined. This is the case when there are subsequent queries that are needed to be executed for each batch, and it is undesirable or overly expensive to execute the query for every value in the entire result set.

The callback function must define two parameters. The first is the batch navigator object itself, and the second it the current batch. The callback function is called once and only once when the BatchNavigator is constructed, and the current batch is determined.

```
>>> def print_callback(context, batch):
...     for item in batch:
...         print(item)
```

```
>>> reindeer_batch_navigator = BatchNavigator(
...     reindeer, build_request(), size=3, callback=print_callback)
Dasher
Dancer
Prancer
```

```
>>> request = build_request({"start": "3", "batch": "20"})
>>> last_reindeer_batch_navigator = BatchNavigator(
...     reindeer, request=request, callback=print_callback)
Vixen
Comet
Cupid
Donner
Blitzen
Rudolph
```

Most likely, the callback function will be bound to a view class. By providing the batch navigator itself as the context for the callback allows the addition of extra member variables. This is useful as the BatchNavigator becomes the context in page templates that are batched.

```
>>> class ReindeerView:
...     def constructReindeerFromAtoms(self, context, batch):
...         # some significantly slow process
...         view.built_reindeer = list(batch)
...     def batchedReindeer(self):
...         return BatchNavigator(
...            reindeer, build_request(), size=3,
...            callback=self.constructReindeerFromAtoms)
```

```
>>> view = ReindeerView()
>>> batch_navigator = view.batchedReindeer()
>>> print(view.built_reindeer)
['Dasher', 'Dancer', 'Prancer']
>>> print(list(batch_navigator.currentBatch()))
['Dasher', 'Dancer', 'Prancer']
```

CHAPTER

FIVE

MAXIMUM BATCH SIZE

Since the batch size is exposed in the URL, it's possible for users to tweak the batch parameter to retrieve more results. Since that may potentially exhaust server resources, an upper limit is put on the batch size. If the requested batch parameter is higher than this, an InvalidBatchSizeError is raised.

```
>>> from lazr.batchnavigator.interfaces import InvalidBatchSizeError
```

```
>>> class DemoBatchNavigator(BatchNavigator):
...     max_batch_size = 5
...
>>> request = build_request({"start": "0", "batch": "20"})
>>> test_raises(
...     InvalidBatchSizeError, DemoBatchNavigator,
...     reindeer, request=request)
Maximum for "batch" parameter is 5.
```

CHAPTER

SIX

URL PARAMETERS

Normally, any parameters passed in the current page's URL are reproduced in the batch navigator's links. A "transient" parameter is one that was only relevant for the current page request and shouldn't be passed on to subsequent ones.

In this next batch navigator, two parameters occur in the page's URL: "noisy" and "quiet."

```
>>> request_parameters = {
...     'quiet': 'ssht',
...     'noisy': 'HELLO',
...     }
```

```
>>> request_with_parameters = build_request(request_parameters)
```

One parameter, "quiet," is transient. There is another transient parameter called "absent," but it's not passed in our ongoing page request.

```
>>> def build_navigator(list):
...     return BatchNavigator(
...         list, request_with_parameters, size=3,
...         transient_parameters=['quiet', 'absent'])
>>> navigator_with_parameters = build_navigator(reindeer)
>>> safe_navigator_with_parameters = build_navigator(safe_reindeer)
```

Of these three parameters, only "noisy" recurs in the links produced by the batch navigator.

```
>>> navigator_with_parameters.nextBatchURL()
'http://www.example.com/foo?noisy=HELLO&memo=3&start=3'
>>> safe_navigator_with_parameters.lastBatchURL()
'http://www.example.com/foo?noisy=HELLO&direction=backwards&start=6'
```

The transient parameter is omitted, and the one that was never passed in in the first place does not magically appear.

CHAPTER

SEVEN

BATCH HEADINGS

The batched values are usually one kind of object such as bugs. The BatchNavigator's heading property contains a description of the objects for display.

```
>>> safe_reindeer_batch_navigator.heading
'results'
```

There is a special case for when there is only one item in the batch, the singular version of the heading is returned.

```
>>> navigator = BatchNavigator(['only-one'], request=request)
>>> navigator.heading
'result'
```

(Accessing .heading causes len() to be called on the underlying list, which is why we have to use the safe batch navigator. In theory, this could be optimized, but there's no real point, since the heading is invariably preceded by the actual length of the underlying list, eg. "10 results". Since len() is called anyway, and its value is cached, a second len() won't hurt performance.)

The heading can be set by passing a singular and a plural version of the heading. The batch navigation will return the appropriate header based on the total items in the batch.

```
>>> navigator = BatchNavigator(safe_reindeer, request=request)
>>> navigator.setHeadings('bug', 'bugs')
>>> navigator.heading
'bugs'
```

```
>>> navigator = BatchNavigator(['only-one'], request=request)
>>> navigator.setHeadings('bug', 'bugs')
>>> navigator.heading
'bug'
```

(Cleanup)

```
>>> sm.unregisterAdapter(HTTPCharsets)
True
>>> sm.unregisterAdapter(ExampleAdapter)
True
```

7.1 Contributing

To run this project's tests, use tox.

To update the project's documentation you need to trigger a manual build on the project's dashboard on https://readthedocs.org.

7.1.1 Getting help

If you find bugs in this package, you can report them here:

https://launchpad.net/lazr.batchnavigator

If you want to discuss this package, join the team and mailing list here:

https://launchpad.net/~lazr-users

or send a message to:

lazr-users@lists.launchpad.net

7.2 NEWS for lazr.batchnavigator

7.2.1 2.0.0 (2022-11-30)

- Drop support for Python 2.7.
- Declare support for Python 3.9, 3.10 and 3.11.
- Add `pre-commit` configuration.
- Publish documentation on Read the Docs.
- Apply `black`, `isort` and `flake8` via `pre-commit`.
- Apply inclusive naming via the woke pre-commit hook.

7.2.2 1.3.1 (2021-09-13)

- Adjust versioning strategy to avoid importing pkg_resources, which is slow in large environments.

7.2.3 1.3.0 (2019-11-04)

- Switch from buildout to tox.
- Add Python 3 support.

7.2.4 1.2.11 (2015-04-09)

- Save a query if the slice is of the form [x:x].

7.2.5 1.2.10 (2011-09-14)

- delegate the calculation of the rough length of a result set to IRangeFactory.

7.2.6 1.2.9 (2011-08-25)

- When a backwards batch is at first too short and when another chunk from the result set is added, _Batch,sliced_list() does no longer use the memo value for the already retrived chunk.
- don't use the parameter start to determine if a previous/next batch exists; don't rely on len(resultset) and to determine the real size of a batch.
- Avoid negative start index on empty result sets.

7.2.7 1.2.7 (2011-07-18)

- retrieve slices of the result set in class _Batch only via methods of the range factory.

7.2.8 1.2.6 (2011-07-28)

- fixed an error in handling backwards batches which return less elements than expected.
- URL-encode all query parameters in BatchNavigator.generateBatchURL()

7.2.9 1.2.5 (2011-07-13)

- Permit changing all variable names with a single prefix.

7.2.10 1.2.4 (2011-04-11)

- Permit overriding determineSize to control how the batch default and concrete sizes are determined in subclasses.
- Listify (once we have sliced) rather than assuming batched slices will honour the complete list protocol.

7.2.11 1.2.3 (2011-04-06)

- Add IRangeFactory and the ability to use backend database hints for efficient retrieval of pages.
- Remove terrible-scaling getBatchURLs method.

7.2.12 1.2.2 (2010-08-19)

- Make len() cheap to call when the current batch is the last (or only) batch.
- Avoid calling len() when generating navigator URLs.

7.2.13 1.2.1 (2010-08-12)

- fix a bug in the len() of a batch when the batch had previously been iterated over

7.2.14 1.2.0 (2010-08-05)

- avoid calling len() on the underlying sequence when possible
- return None for endNumber when the batch is out of range

7.2.15 1.1.1 (2010-05-10)

- Ignore negative batch sizes

7.2.16 1.1 (2009-08-31)

- Remove build dependencies on bzr and egg_info
- remove sys.path hack in setup.py for __version__

7.2.17 1.0 (2009-03-24)

- Initial release on PyPI