Divmod Documentation

Release 0.1

Divmod, Inc. & Corbin Simpson

Sep 27, 2017

Contents

CHAPTER 1

Products

These are things that Divmod produced.

Divmod Axiom

Axiom is an object database whose primary goal is to provide an object-oriented layer with what we consider to be the key aspects of OO, i.e. polymorphism and message dispatch, without hindering the power of an RDBMS. It is designed to 'feel Pythonic', without encouraging the typical ORM behavior such as *Potato Programming*.

Axiom provides a full interface to the database, which strongly suggests that you do not write any SQL of your own. Metaprogramming is difficult and dangerous (as many, many SQL injection attacks amply demonstrate). Writing your own SQL is still possible, however, and Axiom does have several methods which return fragments of generated schema if you wish to use them in your own queries.

Axiom currently supports only SQLite and does NOT have any features for dealing with concurrency. We do plan to add some later, and perhaps also support other databases in the future. Take a look at *Concurrency and Scalability* for more information - we'll update this as the community makes progress on these issues.

Performance

How does Axiom perform?

Here are some simple third-party benchmarks.

Download

- Stable: Download the latest release - 0.6.0! (Requires *Divmod Epsilon*) (Release Notes)
- Trunk: svn co http://divmod.org/svn/Divmod/trunk/Axiom Axiom

See Also

- *Axiom Tutorial*: Short, sharp Axiom/Mantissa tips and tricks.
- *Why use Axiom?*: Glyph writes about the advantages of Axiom over other RDBMS and ORM alternatives.
- *Reference*: A reference to the Axiom public API. (Incomplete but evolving!)
- **Development** version of the Axiom API docs
- axiom-examples
- Axiom tickets
- axiomatic
- *Transition mini-HOWTO*: How to transition from pre-`axiom.dependency` code.
- *Powerups*
- WritingUpgradeTests: writing stubloader tests for schema upgrades.
- axiom-files

Index

Concurrency and Scalability

Several questions seem to keep coming relating to these issues; this is an attempt to summarize the discussion around them.

Why is the API synchronous?

Generally database access through the SQLite backend is fast enough that database operations don't block long enough to be of concern.

Are you sure?

Sort of. Current usage seems to indicate that it works out just fine, but you should conduct your own testing to determine whether this model is suitable for your particular usage patterns.

Is it thread-safe, though?

No. Accessing a Store from different threads will break. Accessing different Stores, even backed by the same SQLite database, is fine.

But how do you achieve scalability then?

Use multiple databases. For example, *Divmod Mantissa* has per-user and per-application stores.

What if that isn't good enough?

If you do need finer-grained concurrency, then running transactions in separate threads is one way to go about it. Glyph's blog has more on this subject; also see #537 which covers the DivmodAxiom implementation of this model.

Axiom Tutorial

JP Calderone's occasional series on Axiom and Mantissa usage. Via JP's blog.

- How to create a Mantissa Server
- Configuring HTTP and HTTPS ports, and static content
- Configuring Static Resources on a Mantissa Server
- Redirecting HTTP request logs
- Adding Axiomatic plugins
- Axiom Queries
- Axiom Powerups

Why use Axiom?

Because it's great, of course! More specifically, it's an object-relational database that's ...

Actually Relational

Axiom imposes an **extremely** straightforward mapping between tables and classes: 1 to 1. Your tables get behavior. They can refer to each other. If you're familiar with the relational model but are learning OO and want some of its features, Axiom won't explode your brain with a million new concepts. After learning how a few simple Python expressions work, you can get up and running.

Axiom also respects the relational model and allows you to perform queries on groups of items at once, without loading them first. Support for updates and deletes without loading items will also be implemented before 1.0 is released. You don't need to resort to writing your own 'for' loops or your own SQL code in order to load a joined query. No *Potato Programming*.

Actually Object-Oriented

Tables have behavior. You can have a reference to an arbitrary item. Items can exist outside the database. You can mix in code from utility classes to give your items additional methods, without affecting its schema. Simple database types like strings and integers can be mapped into complex, rich Python values, such as email addresses or a fixed-point decimal class.

Extensible

Axiom's '*Powerups*' mechanism allows you enhance a database, or **any row** within a database, with additional functionality.

Active

Axiom isn't a static database. It is dynamic, it contains behaviors.

You can, for example, install a scheduler which will run tasks at specific times when the database is open.

Twisted-friendly

The aforementioned scheduler is implemented using Twisted's reactor. Axiom was explicitly designed for use with events and integrates nicely into a Twisted-based environment.

Future-Proof

Axiom applications can keep running while they're being upgraded from one schema to another (with a partial dataset available). Upgrades are handled automatically, and the code for upgrading can be kept separate from your application code. You can even write upgraders which involve complex relationships between multiple rows.

Simple

Axiom gives you the power of an SQL database without the hassle of generating code from within code.

Working with Axiom databases is easy. The underlying SQLite engine is pretty simple to use already, but Axiom doesn't add layers of complexity that get in your way. Compare loading a row and manipulating it with a Python class:

```
from axiom.store import Store
s = Store('store.axiom')
x = s.getItemByID(0)
```

```
from pysqlite2 import dbapi2
from mymodule import MyClass
con = dbapi2.connect('store.sqlite')
cur = con.cursor()
cur.execute('select * from my_table where oid = 0')
row = con.fetchall()[0]
x = MyClass(row)
```

In this case the Axiom example is actually quite a lot shorter!

Safe

Axiom keeps your in-memory objects up to date with what's happening in the database, even when transactions have to be rolled back. No need to manage connections of your own. No more forgetting to commit a transaction, or forgetting to roll back in an error handler.

Axiom doesn't expose **any** SQL to the application developer. Internally Axiom is also very careful about SQL generation: it doesn't use string concatenation except where absolutely necessary, and instead uses `?` interpolation. This means that your site will be completely secure against SQL injection attacks.

Axiom also makes sure that your objects in memory correspond to what's in the database. If a transaction fails, Axiom will log the error and revert all the Python objects involved in the transaction to the state they are in after reverting. This brings the safety of database transactions into your Python code.

Powerups

A powerup is a type of Axiom plugin. Zero or more powerups (Axiom items) can be registered to another axiom item (as long as it is in the same store) and can be retrieved according to their interface, either by normal adaption of the subject (in which case the highest priority powerup is returned), or as a group (in order of priority) by using the `axiom.item.Item.powerupsFor` method.

See Also

- class axiom.item.Empowered

Reference

item.Item(store=store.Store, **kw)

A class describing an object in an Axiom Store. Your subclass of `axiom.item.Item` must define the following attributes:

- `typeName`: A string uniquely identifying this class schema in the store.
- `schemaVersion`: An integer denoting the schema version. See *Upgraders*.

You will also want to define other attributes and these must be special Axiom class attributes. A selection of standard types are already defined in the axiom.attributes module.

A simple example:

```
from axiom import item, attributes

class ShopProduct(item.Item):
    typeName = 'ShopProduct'
    schemaVersion = 1

    name = attributes.text(allowNone=False)
    price = attributes.integer(allowNone=False)
    stock = attributes.integer(default=0)

    def __repr__(self):
        return '<ShopProduct name='%s' price='%d' stock='%d'>' % (self.name, self.
↪price, self.stock)
```

Limitations

- Axiom Items only support one level of inheritance. You could not for example write a subclass of the ShopProduct class above. This is by design, and you are encouraged to explore object composition and adaption instead.

store.Store([dbdir=None[, debug=False[, parent=None[, idInParent=None]]]])

A database in which Axiom Items can be stored. An Axiom Store can be instantiated with a dbdir parameter, in which case it will be persisted to the filesystem at the given path. Alternatively, if instantiated without a dbdir parameter, the store will exist inmemory only for the lifetime of the python process.

```
from axiom import store

s = store.Store('/tmp/example.axiom')
s = store.Store() # An inmemory store
```

If `debug=True`, the store will print out all SQL commands as they are issued to the underlying Sqlite database.

Add Items to the Store

```
p = ShopProduct(store=s, name=u'Tea Bags', price=2)
```

That's all there is to it. The returned item can be treated just like any other python object. Changes made to it are automatically persisted to the store.

```
>>> p.name
u'Tea Bags'
>>> p.stock
0
>>> p.stock += 20
>>> p.stock
20
```

If you want to avoid duplicate items you can instead use the findOrCreate method (see below)

Retrieve Items from the Store

- `getItemByID(storeID[, default=_noItem]):`

  Returns the item with the given 'storeID'. If no matching item raises KeyError or 'default' if given. Every item in the store has a unique 'storeID' attribute.

- `findFirst'''(userItemClass[, **attrs]):`

  Returns the first item of class 'userItemClass' or None. The query can be further narrowed by specifying 'attrs', eg

  ```
  >>> s.findFirst(ShopProduct, name=u'Tea Bags')
  <ShopProduct name='Tea Bags' price='2' stock='20'>
  ```

- `findOrCreate(userItemClass[, **attrs]):`

  Returns the first item of class 'userItemClass' or creates it if it doesn't already exist. eg.

  ```
  >>> s.findOrCreate(ShopProduct, name=u'Pot Noodle')
  TypeError: attribute `= integer() <ShopProduct.price>`_ must not be None
  ```

  but we must give all attributes required to create the new item

  ```
  >>> s.findOrCreate(ShopProduct, name=u'Pot Noodle', price=3)
  <ShopProduct name='Pot Noodle' price='3' stock='0'>
  ```

- `query(tableClass[, comparison=None[, limit=None[, offset=None[, sort=None[, justCount=False[, sumAttribute=None]]]]]]):`

  Return generator of items matching class 'tableClass' and 'comparison'. Limited to length 'limit' beyond 'offset'. Sorted by attribute 'sort'. Examples:

```
>>> 'All products'
>>> [x.name for x in s.query(ShopProduct)]
[u'Tea Bags', u'Pot Noodle']
>>> 'Products in stock'
>>> [x.name for x in s.query(ShopProduct, ShopProduct.stock > 0)]
[u'Tea Bags']
>>> 'Products in stock AND which cost less than 5'
>>> from axiom.attributes import AND, OR
>>> [x.name for x in s.query(ShopProduct, AND(ShopProduct.stock > 0, ShopProduct.
↪price < 5))]
[u'Tea Bags']
```

You get the idea. Try turning on store debugging and you will get an idea of what is going on behind the scenes.

```
>>> s.debug = True
>>> [x.name for x in s.query(ShopProduct, sort=ShopProduct.stock.ascending)]
** SELECT item_ShopProduct_v1 .oid, item_ShopProduct_v1 .* FROM item_ShopProduct_
↪v1 ORDER BY item_ShopProduct_v1.`ASC --
********** COMMIT **********
  lastrow: None
  result: [(4, u'Pot Noodle', 3, 0), (3, u'Tea Bags', 2, 20) <stock]>`_
[u'Pot Noodle', u'Tea Bags']
```

Axiom is also capable of constructing more complex queries involving table joins behind the scenes. For more complete examples see axiom-examples.

Substore

A Store that also exists as an Item in a parent Store.

- class axiom.substore.SubStore

Powerups

A powerup is a type of Axiom plugin. Zero or more powerups (Axiom items) can be registered to another axiom item (as long as it is in the same store) and can be retrieved according to their interface, either by normal adaption of the subject (in which case the highest priority powerup is returned), or as a group (in order of priority) by using the axiom.item.Item.powerupsFor method.

- class axiom.item.Empowered

Upgraders

?

Examples : Shop

```
from zope.interface import Interface, implements

from axiom.attributes import AND, OR
from axiom import item, attributes, sequence
from epsilon.extime import Time
```

```
class Person(item.Item):
    '''A person here is not specific to the shopping part of the application.
    It may have been defined elsewhere eg during registration for an enquiries system.
    We can't subclass Person but we can plugin our shop specific attributes and␣
↪methods using
    the Axiom powerup pattern.
    '''

    typeName = 'Person'
    schemaVersion = 1

    name = attributes.text(allowNone=False)
    dob = attributes.timestamp(allowNone=False)

    def __repr__(self):
        return '<Person name='%s' dob='%s'>' % (self.name, self.dob.
↪asISO8601TimeAndDate())

class IShopCustomer(Interface):
    pass

class ShopCustomer(item.Item):
    '''A ShopCustomer is a powerup for person.'''
    implements(IShopCustomer)

    typeName = 'ShopCustomer'
    schemaVersion = 1

    installedOn = attributes.reference()

    def installOn(self, other):
        assert self.installedOn is None, 'cannot install ShopCustomer on more than␣
↪one person'
        self.installedOn = other
        other.powerUp(self, IShopCustomer)

    '''Customer specific methods'''
    def getProductsOrdered(self):
        # An example of an inner join query
        return self.store.query(
            ShopProduct,
            AND(
                ShopOrder.customer == self.installedOn,
                ShopProductOrdered.order==ShopOrder.storeID,
                ShopProductOrdered.product==ShopProduct.storeID
            )
        )

    def getOrders(self):
        return self.store.query(
            ShopOrder,
            ShopOrder.customer == self.installedOn
        )

class ShopProduct(item.Item):
    typeName = 'ShopProduct'
    schemaVersion = 1
    name = attributes.text(allowNone=False)
    price = attributes.integer(allowNone=False)
    stock = attributes.integer(default=0)

    def __repr__(self):
        return '<ShopProduct name='%s' price='%d' stock='%d'>' % (self.name, self.
↪price, self.stock)

class ShopProductOrdered(item.Item):
    '''Links a product and quantity of product to an order.'''
    typeName = 'ShopProductOrdered'
    schemaVersion = 1

    order = attributes.reference(allowNone=False)
    product = attributes.reference(allowNone=False)
    quantity = attributes.integer(default=1)

class ShopOrder(item.Item):
    typeName = 'ShopOrder'
    schemaVersion = 1

    customer = attributes.reference(allowNone=False)
    purchaseDate = attributes.timestamp(allowNone=False)

    def __init__(self, **kw):
        IShopCustomer(kw['customer'])
        super(ShopOrder, self).__init__(**kw)

    def addProduct(self, product, quantity=1):
        po = self.store.findOrCreate(
            ShopProductOrdered,
            order=self,
            product=product)
        po.quantity = quantity

    def getProducts(self):
        return self.store.query(ShopProductOrdered, ShopProductOrdered.order == self)

    def getTotalPrice(self):
        #XXX: Axiom will issue multiple queries here, but it could be done in one SQL␣
↪query. Is there a way to issue such a query?
        total = 0
        for p in self.getProducts():
            total += p.product.price*p.quantity
        return total

    def __repr__(self):
        return '<ShopOrder customer='%s' purchaseDate='%s' items='%s'>' % (self.
↪customer.name, self.purchaseDate.asISO8601TimeAndDate(), self.items)

def populateStore(s):

    customerDetails = [
        (u'Joe Bloggs', '1977-05-08'),
        (u'Jane Doe', '1959-05-22'),
    ]
    for name, dob in customerDetails:
        p = Person(store=s, name=name, dob=Time.fromISO8601TimeAndDate(dob))

        # This is where we powerup the Person with additional ShopCustomer bits
        ShopCustomer(store=s).installOn(p)

    products = [
        ShopProduct(store=s, name=u'Tea Bags', price=2),
        ShopProduct(store=s, name=u'Cornflakes', price=3),
        ShopProduct(store=s, name=u'Lemonade', price=4),
        ShopProduct(store=s, name=u'Peanuts', price=5),
    ]

    quantities = [1,2,4]

    for c in s.query(ShopCustomer):
        o = ShopOrder(store=s, customer=c.installedOn, purchaseDate=Time())
        o.addProduct(random.choice(products), random.choice(quantities))
        o.addProduct(random.choice(products), random.choice(quantities))

if __name__ == '__main__':
    import random
    from axiom import store

    s = store.Store(debug=False)
    populateStore(s)

    '''We only want a Person who is also a ShopCustomer.
    We therefore search for ShopCustomer but grab a reference to the person within␣
↪(installedOn)
    When you want the person reference to behave like a shopcustomer
    adapt it to the IShopCustomer interface'''
    p = s.findFirst(ShopCustomer).installedOn

    print [x.name for x in IShopCustomer(p).getProductsOrdered()]
    print '%s has ordered the following products since registering:' % p.name

    print 'A breakdown of %s's orders' % p.name
    print ['Items: %s, Total: %d'%(['%s X %d'%(y.product.name, y.quantity) for y in x.
↪getProducts()], x.getTotalPrice()) for x in IShopCustomer(p).getOrders()]
```

Transition mini-HOWTO

Notes

A few bullet points with stuff to do transitioning from pre-dependency code to the new API(s).

- `axiom.item.InstallableMixin` has been removed since it is unnecessary, as is the `installedOn` attribute that was on subclasses thereof. `powerup.installedOn()` is now spelled `axiom.dependency.installedOn(powerup)`.
- `powerup.installOn(target)` is now spelled `axiom.dependency.installOn(powerup, target)`. See also `axiom.dependency.uninstallFrom(powerup, target)`.
- Instead of explicitly powering the target up in `installOn`, set the `powerupInterfaces` class attribute to a sequence of interfaces, or of `(interface, priority)` tuples.

- If you are implementing `INavigableElement`, you need something like:

```
privateApplication = dependsOn(PrivateApplication)
```

- Declare other dependencies of your powerups as appropriate.
- Get rid of your `Benefactor` / `BenefactorFactory` classes, and instead pass an `installablePowerups` sequence when constructing your offering. For example:

```
installablePowerups = [
    (u'Operator admin', u'Operator administration', OperatorAdmin),
    (u'Reports', u'Data reporting functionality', Reports),
    ]
```

- TODO: writing upgraders

Example

`xmantissa.webapp` was migrated as part of this change. That serves as a good example, and will be (incompletely) presented below as a demonstration. The examples below elide most of the code and focus just on the changes. Please refer to the different file versions themselves for a complete representation.

Before (reference):

```
class PrivateApplication(Item, PrefixURLMixin):
    ...
    implements(ISiteRootPlugin, IWebTranslator)
    ...
    installedOn = reference()
    ...
    def installOn(self, other):
        super(PrivateApplication, self).installOn(other)
        other.powerUp(self, IWebTranslator)

        def findOrCreate(*a, **k):
            return other.store.findOrCreate(*a, **k)

        findOrCreate(StaticRedirect,
                     sessioned=True,
                     sessionless=False,
                     prefixURL=u'',
                     targetURL=u'/'+self.prefixURL).installOn(other, -1)

        findOrCreate(CustomizedPublicPage).installOn(other)

        findOrCreate(AuthenticationApplication)
        findOrCreate(PreferenceAggregator).installOn(other)
        findOrCreate(DefaultPreferenceCollection).installOn(other)
        findOrCreate(SearchAggregator).installOn(other)
    ...
```

After (reference):

```
class PrivateApplication(Item, PrefixURLMixin):
    ...
    implements(ISiteRootPlugin, IWebTranslator)
    ...
    powerupInterfaces = (IWebTranslator,)
    ...
    customizedPublicPage = dependsOn(CustomizedPublicPage)
    authenticationApplication = dependsOn(AuthenticationApplication)
    preferenceAggregator = dependsOn(PreferenceAggregator)
    defaultPreferenceCollection = dependsOn(DefaultPreferenceCollection)
    searchAggregator = dependsOn(SearchAggregator)
```

Glossary

item.Item(store=store.Store, **kw) An object in an Axiom Store.

Powerups An Axiom Item that acts as a type of Axiom plugin.

store.Store([dbdir=None[, debug=False[, parent=None[, idInParent=None]]]]) A database that Axiom Items can be stored in.

Substore A Store that also exists as an Item in a parent Store.

So You Want To Change Your Schema

This is a very brief overview of how to write upgrade tests for Axiom. I hope someone else will clean it up and add more information. This assumes familiarity with Axiom, Divmod project layout, trial unit tests, Combinator, and probably a dozen other things.

In order to understand the following you will need to look at these two modules: `axiom.test.historic.stub_catalog1to2` and `axiom.test.historic.test_catalog1to2`

Do Stuff

1. write a file, `stub_foo1to2.py` (this should look vaguely like the aforementioned `stub_catalog1to2.py`) in your project's `tests/historic` directory, named appropriately for your object and versions. This is written using the *old* API, the one present in trunk before your branch is merged.
2. the number in the call to saveStub is the revision of `$COMBINATOR_PROJECTS/Divmod/trunk` when you are writing the upgrader.
3.
```
% cd $COMBINATOR_PROJECTS/Divmod/branches/foo-19191919/Foo/foo/test/historic/
% chbranch Divmod trunk
% python stub_foo1to2.py # this will create a file called 'foo1to2.axiom.tbz2'.
% svn add foo1to2.axiom.tbz2
```
4. write `test_foo1to2.py` (this should look vaguely like the aforementioned `test_catalog1to2.py`), and it works like a regular trial test, except for loading your stub database automatically.

Dynamic Typing

Reference to Any

A column in your schema may contain a reference to any row in the database.

Schema Migration: Automatic Upgrades

Axiom defines a comprehensive API for upgrading from one version of a schema to the next, both writing upgraders, and scheduling them to be run.

Transaction Management and Consistency

While it is definitely possible to run in autocommit mode, axiom provides a 'transact' method to manage transactions for you. Transactions are reverted when an exception crosses the transaction boundary, so errors will generally not affect the state of your persistent data. When transactions are reverted, they are reverted **all the way**: objects in memory are restored to the state they were in before the transaction began, both in memory and on disk. Axiom also takes care to keep Python identity consistent with database identity within a process; if you load an object twice, and change one of its attributes, the attribute will therefore be changed in both places, making it safe to share references between different areas of code.

This is a critical feature for financial applications that is missing from other popular Python database tools, such as SQLObject and SQLAlchemy.

Multi-Database

Items stored in an Axiom database keep an explicit reference to the Store that they are a part of, rather than keeping an implicit store on the stack or in a process-global location. 'Explicit is better than implicit'. Again, other popular tools make it difficult to share data between different databases.

Divmod Combinator

Combinator is a tool that Divmod Python programmers use to manage multiple branches of our software. It integrates with Subversion.

It can be used to manage any number of branches with any number of projects. It sets up your Python sys.path to point at the appropriate directories for the set of branches you are currently working on. It also sets up your `PATH` environment variable so that you can access any scripts that come along with those projects.

It is mainly of interest if you are checking code out of SVN: users of installed versions of Divmod software can probably ignore this project (for now).

Note: Combinator does not currently work with SVN version 1.2 due to changes in the way SVN stores its local repository. See #2144 for details.

Rationale

Subversion is a nice development system, but some tasks are unnecessarily difficult. In particular, as we migrated from CVS to SVN, we discovered that there are some operations which were impractically difficult in CVS but simple in SVN, such as creating branches, but while the implementation of managing and merging branches was adequate, there were too much flexibility, and too many ways to subtly incorrectly merge a branch.

As one example of such a problem, in SVN one must always pass the revision where the branch was created on trunk as an argument to the merge command, and determining that number involves reading the output of another long-running command. Some branches cannot be merged in this manner, depending on where the branch was originally created from.

We developed some idioms for avoiding common errors during merge and encoded them in a set of shell scripts. Then we discovered another set of common problems: often developers working on a branch would do a bunch of work, and then find themselves mystified that their changes did not seem to be taking effect, due to a mismatch between the environment of their development tools and the shells where test commands were being run.

Combinator began as a set of idioms and shell scripts and has evolved into a set of Python tools which enforce a simple workflow for using SVN and Python together to work on projects that use branches to gather changes together while eliminating common errors.

Download

Combinator is in the Divmod repository.

If you want to use it without the rest of the Divmod projects, see the CombinatorTutorial.

Use

Start with [source:trunk/Combinator/README.txt README.txt] to get your environment set up.

Note: If you follow the UNIX setup instructions and an exception is raised along the lines of *OSError: [Errno 2] No such file or directory: '/home/YOURNAME/.local/lib/python2.4/site-packages'*, you should update to the latest trunk revision of Combinator - this bug has been fixed!

CombinatorTutorial is a guide to typical Combinator use including setting up an SVN repository to play with.

Reading about [wiki:UltimateQualityDevelopmentSystem our development process] is likely to give you some insight into how it's intended to be used.

chbranch

chbranch is the tool for switching to a different branch. Provide `chbranch` with a *project name* and *branch name* and it will modify all Combinator-enabled environments so that Python imports are satisfied from that branch of the project. If necessary, the branch will be checked out.

mkbranch

mkbranch is the tool for creating new branches. Provide `mkbranch` with a *project name* and *branch name* and it will create a new branch with that name, switch a copy of trunk to it, and do the equivalent of a `chbranch` to the new branch.

unbranch

unbranch is the tool for merging a branch's changes into trunk. First, use `chbranch` to change to the branch to be merged. Then, make sure that the trunk working copy either contains no changes or contains only changes which you want included in the merge (note: it is strongly, strongly recommended that if the merge will be committed that the trunk working copy contain no changes). Finally, run `unbranch` with the *project name* and the changes from the branch will be merged into the trunk working copy. They will not be committed automatically.

Divmod Epsilon

A small utility package that depends on tools too recent for Twisted (like datetime in python2.4) but performs generic enough functions that it can be used in projects that don't want to share Divmod's other projects' large footprint.

Currently included:

- A powerful date/time formatting and import/export class (ExtimeDotTime), for exchanging date and time information between all Python's various ways to interpret objects as times or time deltas.
- Tools for managing concurrent asynchronous processes within Twisted.
- A metaclass which helps you define classes with explicit states.
- A featureful Version class.
- A formal system for application of monkey-patches.

Download

- Stable: [http://divmod.org/trac/attachment/wiki/SoftwareReleases/Epsilon-0.6.0.tar.gz?format=raw Get the most recent release - 0.6.0!] ([source:/tags/releases/Epsilon-0.6.0/NEWS.txt Release Notes])
- Trunk: svn co http://divmod.org/svn/Divmod/trunk/Epsilon Epsilon

See Also

- ''Development'' version of the [http://buildbot.divmod.org/apidocs/epsilon.html Epsilon API docs]

Divmod Mantissa

Mantissa is an application server. It provides integration between the *Divmod Axiom* 'smart' object database, the *Divmod Nevow* web templating/AJAX/COMET framework and the Twisted framework.

Read more about Mantissa's philosophy and motivation: *Mantissa is the Deployment Target*.

The goal of Mantissa is to provide a common platform that open-source contributors can use to help us build customized additions to the Divmod service. Divmod is going to offer a variety of different services, and if you want to write something that hooks into our network, Mantissa is how you do it.

Download

- Stable: [http://divmod.org/trac/attachment/wiki/SoftwareReleases/Mantissa-0.7.0.tar.gz?format=raw Get the most recent release - 0.7.0!] ([source:/tags/releases/Mantissa-0.7.0/NEWS.txt Release Notes])
- Trunk: svn co http://divmod.org/svn/Divmod/trunk/Mantissa Mantissa (See the dependencies here: [source:trunk/Mantissa/DEPS.txt DEPS.txt])

Tutorials

- [wiki:DivmodAxiom/AxiomTutorial Exarkun's Axiom and Mantissa tutorial]: Short, sharp Axiom/Mantissa tips and tricks.
- [wiki:MantissaWikiTutorial Mantissa Wiki Tutorial] A bit short on explanations, but complete

- [wiki:MantissaBlogTutorial Mantissa Blog Tutorial]
- [wiki:MantissaHowTo An example of how to build a Mantissa application (incomplete)]

See Also

- ''Development'' version of the [http://buildbot.divmod.org/apidocs/xmantissa.html Mantissa API docs]
- Several of the Axiom docs on the DivmodAxiom page are also Mantissa docs.
- [wiki:DivmodMantissa/Sharing /Sharing]: description of sharing functionality
- [wiki:DivmodMantissa/Concepts /Concepts]: other concepts with which Mantissa users should be familiar

Index

Mantissa is the Deployment Target

Mantissa is the deployment target, not the Web.

What Does That Mean?

Mantissa is an application server; it can serve an application over several protocols. HTTP is one of those protocols, but it [HttpIsntSpecial isn't the only one].

The web interface that Mantissa provides should be pluggable, but it should ''not'' be so flexible that every Mantissa application provides its own fixed skin and look and feel.

In other words, when you write a Mantissa application, Mantissa is the deployment target. You deploy your application into a Mantissa application container that faces the web, '''not''' directly onto the web. Ideally the Mantissa web container will be able to show components from several different independently-developed applications simultaneously.

This means that you SHOULD NOT be writing your own IResource implementations or full-page templates; applications should be plugging in somewhere on the page. The set of interfaces developed for this right now (INavigableFragment, INavigableElement) are not necessarily the best, but others will arise soon.

This is not gold-plating or over-engineering, as many seem to think; it springs from a specific requirement. Divmod specifically intends to develop something like 10 different applications, and launch each one separately, but make it easy for people to sign up and for our subscribers to activate new services. Each of those applications is pluggable and has integration points with other applications. These applications are all going to share the same web interface and should have a common look and feel, common interface elements, and a shared infrastructure (for example: password management, search).

Examples of Where This Matters

The most trivial example is side-by-side viewing. It would be good if the tab-based navigation system could move towards being one where you click on a tab to 'go to a page', vs. one where you click on a tab to 'launch' a Mantissa object, enabling side-by-side viewing of multiple interactive objects on one LivePage. (Donovan's ill-fated launch-bar demonstration is what I mean by this, for those who saw it before it was deleted.)

Obviously you cannot lay out the entire page in one component if it is going to be displayed on the same page as a separate component.

A better example might be search. There should be one search box for every Mantissa application. When a user searches, each application should be queried (where ‘application’ in this case is equivalent to ‘thing installed on a user’s store, implementing the appropriate search interface’)

The search results page should not be laid out or rendered by any particular application, but instead, be an aggregation of all the search results for a particular term. This means that the search page has to be part of the framework, not part of an application.

Divmod Nevow

[[PageOutline(2-3,Contents)]]

[[Image(http://divmod.org/tracdocs/nevow_whtbck.png, left)]]

Nevow - Pronounced as the French ‘nouveau’, or ‘noo-voh’, Nevow is a web application construction kit written in Python. It is designed to allow the programmer to express as much of the view logic as desired in Python, and includes a pure Python XML expression syntax named stan to facilitate this. However it also provides rich support for designer-edited templates, using a very small XML attribute language to provide bi-directional template manipulation capability.

Nevow also includes *Formless*, a declarative syntax for specifying the types of method parameters and exposing these methods to the web. Forms can be rendered automatically, and form posts will be validated and input coerced, rendering error pages if appropriate. Once a form post has validated successfully, the method will be called with the coerced values.

Athena - Finally, Nevow includes *Divmod Athena*, a two-way bridge between Javascript in a browser and Python on the server. *Divmod Athena* is compatible with Mozilla, Firefox, Windows Internet Explorer 6, Opera 9 and Camino (*The Divmod Fan Club*). Event handlers can be written in pure Python and Javascript implementation details are hidden from the programmer, with Nevow taking care of routing data to and from the server using XmlHttpRequest. Athena supports a widget authoring framework that simplifies the authoring and management of client side widgets that need to communicate with the server. Multiple widgets can be hosted on an Athena page without interfering with each other. Athena supports automatic event binding so that that a DHTML event (onclick,onkeypress,etc) is mapped to the appropriate javascript handler (which in turn may call the server).

Download

- Stable: Latest release - 0.9.31
- Trunk: svn co http://divmod.org/svn/Divmod/trunk/Nevow/ Nevow

Features

- *XHTML templates*: contain no programming logic, only nodes tagged with nevow attributes
- *data/render methods*: simplify the task of separating data from presentation and writing view logic
- *stan*: An s-expression-like syntax for expressing xml in pure python
- *Athena*: Cross-browser JavaScript library for sending client side events to the server and server side events to the client after the page has loaded, without causing the entire page to refresh
- *formless*: (take a look at formal for an alternate form library) For describing the types of objects which may be passed to methods of your classes, validating and coercing string input from either web or command-line sources, and calling your methods automatically once validation passes.

- *webform*: For rendering web forms based on formless type descriptions, accepting form posts and passing them to formless validators, and rendering error forms in the event validation fails

Documentation

- The Nevow Guide An introductory guide covering Nevow basics (Getting Started, Object Traversal, Object Publishing, XML Templates, Deploying Nevow Applications)
- Nevow API
- Meet Stan: An excellent tutorial on the Nevow Document Object Model by Kieran Holland
- Twisted Components: If you are unfamiliar with Interfaces and Adapters then Nevow may not make much sense. This is essential reading.
- *Error Handling*: How to create custom error (404 and 500) pages
- *Form Handling (A summary of Nevow form handling techniques)* * JavaScript WYSIWYG Editors integration with Nevow/formal
- deployment
- emacs
- *Putting Nevow Page under Apache Proxy*
- Using Nevow with Genshi templates: original and dynamic
- *Tutorial: Using Storm with Nevow*
- *Nevow & Athena FAQ*

Bleeding Docs - **SURGEON GENERAL'S WARNING**: Reading the docs listed below pertain to code that has not yet been released and may cause Lung Cancer, Heart Disease, Emphysema, and Pregnancy complications.

- *Context Removal* - Conversion steps for moving from `context`-based Nevow code to `context`-less code.

Examples

To run the examples yourself (Source in [source:trunk/Nevow/examples]):

```
richard@lazar:/tmp$ cd Nevow/examples/
richard@lazar:/tmp/Nevow/examples$ twistd -noy examples.tac
2005/11/02 15:18 GMT [-] Log opened.
2005/11/02 15:18 GMT [-] twistd SVN-Trunk (/usr/bin/python 2.4.2) starting up
2005/11/02 15:18 GMT [-] reactor class: twisted.internet.selectreactor.SelectReactor
2005/11/02 15:18 GMT [-] Loading examples.tac...
2005/11/02 15:18 GMT [-] Loaded.
2005/11/02 15:18 GMT [-] nevow.appserver.NevowSite starting on 8080
2005/11/02 15:18 GMT [-] Starting factory <nevow.appserver.NevowSite instance at␣
↪0xb6c8110c>
```

... visit http://localhost:8080 and you'll begin to appreciate the possibilities!

Help / Support

You will find plenty of experts on the mailing lists and in the chatrooms who will happily help you, but *please* make sure you *read all the documentation*, *study all the examples* and *search the mailing list archives* first. The chances are that your question has already been answered.

- *Mailing list*: The twisted-web and divmod-dev mailing list pages have subscription instructions and links to the web based archives.
- *IRC*: Nevow developers and users can be found on Freenode in #twisted.web
- *Blogs*: dialtone, fzZzy, Tv
- Tickets (More tickets)

Related Projects

- Eocmanage: An alternative to Mailman built with Twisted and Nevow.
- Pollenation's Formal Project: A fresh take on automatic form generation for Nevow, with a simpler interface and more input types than Formless. This project was formerly known as 'forms'.
- *Divmod Mantissa*: An extensible, multi-protocol, multi-user, interactive application server built on top of Axiom and Nevow.
- Stiq: A web news system built using Nevow
- WubWubWub: 'Making Twisted.Web look like Apache since 2002' A fully featured Twisted based webserver for serving multiple twisted.web and Nevow apps.

Index of Nevow documents

Getting started with Divmod Nevow

Note: This is incomplete documentation in progress.

Getting Started: A basic page

Currently the *nevow.rend* module contains the Page class which should be subclassed to create new pages. A page is the added as a child of the root page, or it's instantiation can be defined in a *childFactory* or *child_* special method. *rend.Page* contains the context which is slowly being removed and will soon be replaced by *page.Page* which should be easily adaptable.

We will construct a page that returns 'Hello world!', and propose some structural alternatives.

```
from nevow import rend, loaders, tags

class APage(rend.Page):
    docFactory = loaders.stan(tags.html[
        tags.head[
            tags.title['Hello World Example']
        ],
        tags.body[
            tags.div(id='hello', _class='helloicator')['Hello World!']
        ]
    ])
```

This page uses Stan to construct an object-like representation which is flattened into XHTML.

Rendering can also dispatch methods inside the page class known as render specials.

```python
from nevow import rend, loaders, tags

class APage(rend.Page):
    docFactory = loaders.stan(tags.html[
        tags.head[
            tags.title['Hello World Example']
        ],
        tags.body[
            tags.div(render=tags.directive('hi'))
        ]
    ])

    def render_hi(self, ctx, data):
        return ctx.tag[ tags.div(id='hello', _class='helloicator')['Hello World']]
```

Putting it together

To put it together as a deployable application all we really need is an application servlet.

A compact example of a boiler plate Nevow application could look like this

```python
# Page modules
from nevow import rend, loaders, tags

# Deployment modules
from nevow import appserver
from twisted.application import service, internet

class APage(rend.Page):
    addSlash = True

    docFactory = loaders.stan(tags.html[
        tags.head[
            tags.title['Hello World Example']
        ],
        tags.body[
            tags.div(render=tags.directive('hi'))
        ]
    ])

    def render_hi(self, ctx, data):
        return ctx.tag[ tags.div(id='hello', _class='helloicator')['Hello World']]


siteRoot = APage() # Set our page as the site root
site = appserver.NevowSite(siteRoot)

demo = internet.TCPServer(8080, site)
```

```
application = service.Application('demo')
demo.setServiceParent(application)
```

It's common to encapsulate the specific service in a deployment function as follows

```
# Page modules
from nevow import rend, loaders, tags

# Deployment modules
from nevow import appserver
from twisted.application import service, internet

class APage(rend.Page):
    addSlash = True

    docFactory = loaders.stan(tags.html[
        tags.head[
            tags.title['Hello World Example']
        ],
        tags.body[
            tags.div(render=tags.directive('hi'))
        ]
    ])

    def render_hi(self, ctx, data):
        return ctx.tag[ tags.div(id='hello', _class='helloicator')['Hello World']]

def deployApp():
    siteRoot = APage() # Set our page as the site root
    site = appserver.NevowSite(siteRoot)
    return site

demo = internet.TCPServer(8080, deployApp())

application = service.Application('demo')
demo.setServiceParent(application)
```

The server can be started by issuing the command `twistd -ny simple.py`.

Note: It is possible to attach multiple sites and protocol servers to a single service parent.

Nevow Tutorial: part 4

Using the `children` class attribute

```
######################################################################
# Run using 'twistd -noy file.tac', then point your browser to
# http://localhost:8080
# A very simple Nevow site.
######################################################################

from twisted.application import service, internet

from nevow               import appserver
from nevow                  import rend
from nevow                  import loaders
from nevow                  import tags as T

class SubPage ( rend.Page ):
    docFactory = loaders.stan (
        T.html [ T.head ( title = 'Sub Page' ),
                 T.body [ T.h1 [ 'This is a Sub Page' ],
                          T.p [ 'I lack much of interest.' ],
                          T.p [ 'You might find ',
                                T.a ( href = '/' ) [ 'this' ],
                                ' more interesting' ],
                          ]
                 ]
        )

class MainPage ( rend.Page ):
    docFactory = loaders.stan (
        T.html [ T.head ( title = 'Main Page' ),
                 T.body [ T.h1 [ 'This is the Main Page' ],
                          T.p [ 'Try going to the pages ',
                                T.a ( href = 'foo' ) [ 'foo' ],
                                ' or ',
                                T.a ( href = 'bar' ) [ 'bar' ],
                                ],
                          T.p [ 'Don't try going ',
                                T.a ( href = 'baz' ) [ 'here' ],
                                ' as it doesn't exist.'
                                ]
                          ]
                 ]
        )

    children = {
        'foo' : SubPage(),
        'bar' : SubPage()
        }

######################################################################
# Nevow Boilerplate
######################################################################

application = service.Application ( 'nevowdemo' )
port        = 8080
res         = MainPage()
site        = appserver.NevowSite ( res )
webService  = internet.TCPServer ( port, site )
webService.setServiceParent ( application )
```

Note the following:

- Instead of a `childFactory()` method we have an explicit mapping of child path elements to page objects;
- Two independent child pages, `foo` and `bar` have been created. These are persisted in the dictionary; they are not created on demand;
- An attempt to visit a child page that doesn't have an entry in the `children` dictionary will fail.

Of course, **any** page can have children, not just the main page. Try modifying the example above to add another page below the `SubPage` objects.

Continue on to tutorial-five.

Form Handling (A summary of Nevow form handling techniques)

[[PageOutline(2-3,Contents)]]

There is more than one way to correctly handle forms in Nevow. This short tutorial demonstrates three Nevow form handling techniques. It includes some real world problems and (hopefully) demonstrates the most appropriate form handling recipe to use in each case.

Examples are included on separate wiki pages for clarity and to make it easy for the reader to download and run them. Where example files end in .tac.py, the example should be run as twistd -noy Example.tac.py. Some screenshots are provided so that you can quickly see the output of the examples.

Automatic Form Generation and Handling

Form handling is one of the most tedious parts of developing a web application. Fortunately there are two sophisticated form handling libraries available for Nevow; *Formless* and Pollenation Forms. Both offer automatic form generation, validation and type coercion.

Formless is older and comes bundled with Nevow but can be confusing for beginners and frustrating for more experienced users. That said, it is highly customisable and allows html coder to precisely layout the form while still taking advantage of the automatic validation and type coercion.

Pollenation Forms is both simpler for the beginner and offers powerful features for the experienced user. It is the recommended form handling library for those who do not need precise customisation of the form layout in the document template.

The Simplest Form - Example 1: A News Item Editor Page

[[Image(Example1.0.png,200,float:right;clear:right;margin-bottom:2px;)]] [[Image(Example1.1.png,200,float:right;clear:right;margin-bottom:2px;)]] [[Image(Example1.2.png,200,float:right;clear:right;margin-bottom:2px;)]]

The following example demonstrates and compares the simplest use of Formless / Pollenation Forms. We start with a basic tac file (src) containing everything but the form:

- the rend.Page subclass and docFactory
- a simple list to store news items
- a data_* method to expose the list of stories to the template
- a method to save new items to the database which redirects to a completion page on success
- Formless Recipe (src): To expose the method (saveNewsItem) in our web page with Formless we:
  - import annotate and webform
  - update the template to include form layout css and a placeholder (<n:invisible />) for our form renderer
  - add the Formless standard css file as a static.File child resource * define a corresponding bind_saveNewsItem method whose job is to return a description of the saveNewsItem method arguments
  - define a custom renderer which adds the results of webform.renderForms() to the page output.
- Pollenation Forms Recipe (src): With Pollenation Forms we:
  - import forms

- mixin the forms.ResourceMixin to our rend.Page subclass.
- update the template to include form layout css and a placeholder for our form renderer (forms.ResourceMixin defines its own render_form method so we use that)
- update our saveNewsItem method to accommodate the extra args that forms will pass to it.
- define a special form_* method which builds and returns the form.

A comparison of the two recipes:

- Pollenation Forms API is more transparent (subjective of course, but generally accepted).
- Pollenation Forms generates cleaner html.
- Pollenation Forms form interface is more usable and provides clearer feedback messages. (ie 'Submit' button rather than the Formless 'Call'. Why? Especially when the new Formless bind_* methods make it such a rigmarole to customise the button...but more on that later.)
- Pollenation Forms default rendered form looks prettier (again subjective) though in both cases, the generated html is sprinkled generously with class and id attributes so that by adding your own CSS you can transform the way the form looks.

So that looks like a pretty ringing endoresment of Pollenation Forms, but it must be remembered that this is a comparison of what the two libraries produce by default in very simple case. Equally important is how they can be extended and customised to fit your application, and that's what we start looking at next.

- nevow-form-handling-formless
- nevow-form-handling-pollenation-forms

Manual Form Handling

Sometimes a form is so simple that it is easiest to write the form html by hand and handle the result manually. The following code demonstrates a form with which the user can choose his preferred number of items per page in a datagrid.

The <select> tag has a javascript onchange handler to automatically submit its parent form and is presented inline with the content of its parent <p> tag. To achieve this automatically using webform.renderForms() would have required defining a custom form template with custom patterns etc (see later example).

(It should be noted that Formless or Pollenation Forms could be used here for automatic coercion and validation of the form variables without employing their form rendering machinery. An example of this may be added later.)

Error Handling

Just an example for now.

```
from zope.interface import implements

from twisted.application import service, strports
from twisted.web import http

from nevow import appserver, context, inevow, loaders, rend, static, tags as T, url

# Inspired by http://divmod.org/users/wiki.twistd/nevow/moin.cgi/Custom404Page

class ErrorPage(rend.Page):
    def render_lastLink(self, ctx, data):
        referer = inevow.IRequest(ctx).getHeader('referer')
        if referer:
            return T.p[T.a(href=referer)['This link'],
                       ' will return you to the last page you were on.']
        else:
            return T.p['You seem to be hopelessly lost.']

class The500Page(ErrorPage):
    implements(inevow.ICanHandleException)
    docFactory = loaders.xmlstr('''
<!DOCTYPE html PUBLIC '-//W3C//DTD XHTML 1.0 Strict//EN'
           'http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd'>
<html xmlns:n='http://nevow.com/ns/nevow/0.1'>
    <head>
        <title>500 error</title>
        <style type='text/css'>
        body { border: 6px solid red; padding: 1em; }
        </style>
    </head>
    <body>
        <h1>Ouchie. Server error.</h1>
        <p>The server is down.</p>
        <p><em>Ohhh. The server.</em></p>
        <p n:render='lastLink' />
    </body>
</html>''')

    def renderHTTP_exception(self, ctx, failure):
        request = inevow.IRequest(ctx)
        request.setResponseCode(http.INTERNAL_SERVER_ERROR)
        res = self.renderHTTP(ctx)
        request.finishRequest( False )
        return res

class The404Page(ErrorPage):
    implements(inevow.ICanHandleNotFound)
    docFactory = loaders.xmlstr('''
<!DOCTYPE html PUBLIC '-//W3C//DTD XHTML 1.0 Strict//EN'
           'http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd'>
<html xmlns:n='http://nevow.com/ns/nevow/0.1'>
    <head>
        <title>404'd !!!</title>
        <style type='text/css'>
        body { border: 6px solid orange; padding: 1em;}
        </style>
    </head>
    <body>
        <h1>OW! my browser!</h1>
        <p>Were you just making up names of files or what?</p>
        <p>(apologies to http://homestarrunner.com)</p>
        <p n:render='lastLink' />
    </body>
</html>''')
    def renderHTTP_notFound(self, ctx):
        return self.renderHTTP(ctx)

class OopsPage(rend.Page):
    def exception(self, ctx, data):
        1/0 # cause an exception on purpose
    docFactory = loaders.stan(exception)

class RootPage(rend.Page):
    docFactory = loaders.stan(
            T.html[T.head[T.title['Error Pages']],
                   T.body[T.h1['The Only Real Page'],
                          T.a(href='notherebaby')[
                            'This link goes nowhere, pleasantly.'],
                          T.br,
                          T.a(href='oops')[
                            'This link causes an exception!'],

                            ]]
                              )

    def locateChild(self, ctx, segments):
        ctx.remember(The404Page(), inevow.ICanHandleNotFound)
        ctx.remember(The500Page(), inevow.ICanHandleException)
        return rend.Page.locateChild(self, ctx, segments)

root = RootPage()
root.putChild('oops', OopsPage())

site = appserver.NevowSite(root)
# You should be able to do this instead of in locateChild, but there seems to be a bug
# http://divmod.org/trac/ticket/526
#site.remember(The404Page(), inevow.ICanHandleNotFound)
#site.remember(The500Page(), inevow.ICanHandleException)

application = service.Application('News item editor')
strports.service('8080', site).setServiceParent(application)
```

Formless

A Note About Formless TypedInterfaces

One of the original goals for Formless was that as well as web forms, it should be flexible enough to handle forms for other user interfaces e.g. curses, gtk. The idea was that the developer should create a TypedInterface (an enhanced Zope Interface) describing the signature of a method that might be invoked in response to a form submission. By implementing this interface in a Nevow.rend.Page, the developer could present a web UI. Elsewhere the same TypedInterface could be implemented to provide a gtk UI for example. In practice though, only the web form rendering code was maintained.

- see the Formless section of this 2004 paper by Donovan Preston, the original author of Nevow and Formless

Recently *TypedInterface has been deprecated in favour of the newer and simpler 'bind_*' syntax.* (see Nevow Changelog 2005-07-12) Confusingly for the beginner, most of the Nevow examples still use TypedInterfaces, but hopefully these will be updated in due course.

In the rest of this document we will use only the new bind_* syntax.

More on Standard Form Controls and Input Types

Custom Form Controls and Input Types

Customising Form Layout

TODO

Authentication and Authorisation

Misc Recipes

Using Formless to Render a Guarded Login Page

```
class LoginPage(rend.Page):
    docFactory = loaders.stan(
        T.html[
            T.head[T.title['Login Please']],
            T.body[
                webform.renderForms()
            ]
        ]
    )

    def bind_login(self, ctx):
        return [
            ('ctx', annotate.Context()),
            ('username', annotate.String(required=True)),
            ('password', annotate.PasswordEntry(required=True)),
        ]

    def login(self, ctx, username, password):
        sess = inevow.ISession(ctx)
        req = inevow.IRequest(ctx)
        credentials = UsernamePassword(username, password)

        def errorHandler(error):
            #error.trap(UnauthorizedLogin)
            raise annotate.ValidateError(
                {},
                formErrorMessage='Your username and / or password were not recognised.
↪',
                partialForm={'username':username})

        def successHandler(pageAndRemainingSegments):
            return url.root

        return sess.guard.login(req, sess, credentials, []).
↪addCallback(successHandler).addErrback(errorHandler)
```

Context Removal

Step 1

Change Resource Inheritance

The `Page` class is now imported from `nevow.page` instead of `nevow.rend`. Class declarations change as indicated below.

From:

```
class Root(rend.Page):
...
```

To:

```
class Root(page.Page):
...
```

Step 2

Change ‘‘child_‘‘ Class Attributes

Class `child_*` class attributes change in that the prefix is no longer needed and all `child_`s are stored together in a `dict`.

From:

```
...
  child_foo = RendPageInstance
...
```

To:

```
...
  children = {
    'foo': PagePageInstance,
  ...}
...
```

Step 3

Remove Context from Signatures

The `context` object is no longer passed in the method signatures. Passed parameters changes as indicated below.

From:

```
...
  def render_foo(self, context, data):
  ...
```

To:

```
...
  def render_foo(self, request, tag):
  ...
```

Step 4

Use Decorators

The `method` names no longer need to contain `render_` and `child_`. Method names change as indicated below.

From:

```
...
  def render_foo(self, context, data):
    ...
...
  def child_bar(self, context):
    ...
```

To:

```
...
  def foo(self, request, tag):
    ...
  page.renderer(foo)
...
  def bar(self, request):
    ...
  page.child(bar)
```

Or, for a version of python that supports the decorator syntax:

```
...
  @page.renderer
  def foo(self, request, tag):
    ...
...
  @page.child
  def bar(self, request):
    ...
```

Step 5

Change Fill Slot Calls

The `fillSlots()` calls are still `tag` methods, but `tag` is now passed directly to `render` methods and not accessed as a `context` attribute. Make changes as indicated below.

From:

```
...
  def render_entries(self, ctx, data):
    ctx.tag.fillSlots('author', 'The Humble Author')
    ctx.tag.fillSlots('title', 'The Excellent Title')
    ctx.tag.fillSlots('content', 'The Interesting Content')
    return ctx.tag
```

To:

```
...
  def entries(self, request, tag):
    tag.fillSlots('author', 'The Humble Author')
    tag.fillSlots('title', 'The Excellent Title')
    tag.fillSlots('content', 'The Interesting Content')
    return tag
  page.renderer(entries)
```

Nevow Guard

Nevow Guard provides an authentication wrapper for pages (and their child pages)

Unit Testing

Note: Much of this page documents the current development version of Nevow's testing features, not the current release version. Make sure you are using the right version of the code if you attempt to make use of this documentation.

Divmod uses the stand-alone JavaScript interpreter SpiderMonkey to run unit tests on some portions of our JavaScript sources. You can find several examples in the Divmod repository.

These unit tests are valuable: they reveal simple errors, provide a kind of documentation as to the purpose of various functions, and ensure that the code is exercised in as much isolation as is possible. However, they are not all-encompassing. After all, they aren't even executed by the same runtime that will be used by **any** user to execute them.

`nit` can also be used for unit testing (see below).

Functional Testing

To account for this, Nevow provides `nit`, a runner for tests which are designed to be run by an actual browser. Nit tests are placed in Python source files named with the `livetest_` prefix. There are several examples of this kind of test in the Divmod repository as well.

If you are familiar with XUnit, the API presented by nit should present few surprises. The primary interface of interest to test developers is the `TestCase` class, which is subclassed and extended to define new tests.

Server Side

`nevow.livetrial.testcase.TestCase` is the base class for the server-side implementation of test methods. Subclasses of `TestCase` define the JavaScript class which will implement the client-side portion of test methods as well as the document which will be used to render them. `TestCase` is a `nevow.athena.LiveFragment` subclass, so it can also define methods which are exposed to the client.

A TestCase might be no more complex than the following:

```
from nevow.livetrial.testcase import TestCase
from nevow.athena import expose

class AdditionTestCase(TestCase):
    jsClass = 'Examples.Tests.AdditionTestCase'
```

Client Side

On the JavaScript side, test authors subclass `Nevow.Athena.Test.TestCase` and define actual test methods on it. These methods may return Deferreds if they are testing asynchronous APIs. Test methods which return a Deferred which eventually errbacks are treated as failing; those which return a Deferred which eventually callbacks are treated as succeeding. Tests which return anything other than a Deferred are also treated as succeeding, and tests which throw any error are treated as failing.

The JavaScript half of the above example might look like this:

```
// import Nevow.Athena.Test

Examples.Tests.AdditionTestCase = Nevow.Athena.Test.TestCase.subclass('Examples.Tests.
↪AdditionTestCase');
Examples.Tests.methods(
    function test_integerAddition(self) {
        self.assertEqual(1 + 1, 2);
    },

    function test_stringAddition(self) {
        self.assertEqual('a' + 'a', 'aa');
    });
```

Command Line

Tests are collected into a suite automatically by the nit command line. For example, `nit nevow` will launch a server which runs all of Nevow's nits. The server listens on http://localhost:8080/ by default. To run the tests, visit that URL with a browser and click the `Run Tests` button.

External Links

You might want to look at this walkthrough on how to write tests for Athena.

Putting Nevow Page under Apache Proxy

It is generally not recommended to put the staff that requires many simultaneous connections under apache, unless Event MPM is used. But sometimes that's just fine, e.g for tests.

Imagine you want to hide http://localhost:8080/ behind http://go.site.com/

This is what you put on apache with mod_proxy enabled:

```
<VirtualHost *:80>  # of course, *:80 can be replaced by something

    # external site URL
    ServerName go.site.com

    # internal (proxied) URL, note vhost/<protocol>/<host>/ scheme (*)
    ProxyPass / http://localhost:8080/vhost/http/go.site.com/

</VirtualHost>
```

Usually people also add 'ProxyRequests Off' to their apache conf to ensure that apache won't be used as free proxy.

But it's clearly not enough to setup apache, because it will send requests to Twisted asking http://localhost:8080/.. , while Nevow application should process request as if it were http://go.site.com.

There is a special module nevow.vhost.VHostMonsterResource() (please note nevow.vhost is used, not twisted.web.vhost), which takes url in the form http://localhost:8080/vhost/http/go.site.com/* and fixes request to make Resource think that it were direct go.site.com/* .

Example

Let’s hide a resource MyResource, which works perfectly as http://localhost:8080, under http://go.site.com.

Here is a brief ‘proxy.tac’ example, which can be run by ‘twistd -noy proxy.tac’

```
from twisted.application import service, strports
from nevow import appserver, inevow, rend, loaders, vhost
from zope.interface import implements

# This is the root resource we are hiding behind proxy
# It *was* working as http://localhost:8080, but now it should *become* http://go.
↪site.com
class MyResource(rend.Page):

    def child_(self, ctx):
        return HelloPage()

# A simple root page, *was* http://localhost:8080/,
# will *become* http://go.site.com/
# (or http://localhost:8080/vhost/http/go.site.com/ from proxied url)
class HelloPage(rend.Page):

    docFactory = loaders.xmlstr('''\
<html><body>Hello</body></html>
''')

# thanks Damascene for this wrapper
# it delegates /vhost/* requests to VHostMonsterResource, which fixes request
class VhostFakeRoot:
    '''
    I am a wrapper to be used at site root when you want to combine
    vhost.VHostMonsterResource with nevow.guard. If you are using guard, you
    will pass me a guard.SessionWrapper resource.
    Also can hide generic resources
    '''
    implements(inevow.IResource)
    def __init__(self, wrapped):
        self.wrapped = wrapped

    def renderHTTP(self, ctx):
        return self.wrapped.renderHTTP(ctx)

    def locateChild(self, ctx, segments):
        '''Returns a VHostMonster if the first segment is 'vhost'. Otherwise
        delegates to the wrapped resource.'''
        if segments[0] == 'vhost':
            return vhost.VHostMonsterResource(), segments[1:]
        else:
            return self.wrapped.locateChild(ctx, segments)

# setup/run site
site = appserver.NevowSite(VhostFakeRoot(MyResource()))
application = service.Application('go')

strports.service('8080', site).setServiceParent(application)
```

How that works ?

- Request comes as http://localhost:8080/vhost/http/go.site.com/<**>
- VhostFakeRoot.locateChild is called
- 'vhost' is stripped from path and http/go.site.com/* comes to VHostMonsterResource
- http is stripped by VHostMonsterResource, request is fixed
- go.site.com is stripped by VHostMonsterResource, request is fixed
- MyResource comes into play and does its job with <**>

Notes

1. Headers from Twisted are consumed by apache.. That may be a problem
2. Nevow.livePage stuff had problems working through proxy

Alternative Method

There's an alternative (a lot easier method) to achieve the desired result. Just do it in apache config:

```
<VirtualHost [insert IP or * here]:80>
  ServerName mysub.mydomain.com

  ErrorLog /var/log/apache2/mysub.mydomain.com_error_log
  CustomLog /var/log/apache2/mysub.mydomain.com_access_log combined

  ProxyVia On
  ProxyPass / http://127.0.0.1:8080/
  ProxyPassReverse / http://127.0.0.1:8080/
</VirtualHost>
```

Also see *Reverse Proxy*.

Reverse Proxy

A Reverse Proxy forwards the requests it receives from the internet to one or more slave webservers. Lighttpd and Apache both provide reverse proxy modules. Any absolute urls in the response will contain the scheme (possibly the hostname) and port number on which the slave is running. These must be rewritten before being returned to the client. In Nevow this can be handled by vhost.VHostMonsterResource.

Example 1

You have an existing webserver running on port 80, doing name based virtual hosting of several existing websites and want to run a Nevow based site alongside them on the same IP.

Solution

Forward all requests at this virtual host to the Nevow slave server.

Sample Nevow App

```python
from zope.interface import implements
from twisted.application import service, strports
from nevow import appserver, inevow, loaders, rend, url, vhost

class MyPage(rend.Page):
    '''
    I am a simple resource for demo purposes only. I will return a 'MyPage'
    for any child you ask me to locate. I display the current url as calculated
    nevow.url.
    '''
    addSlash = True
    docFactory = loaders.xmlstr('''
<html xmlns:n='http://nevow.com/ns/nevow/0.1'>
    <head>
        <title n:render='urlpath'></title>
    </head>
    <body>
        <h1 n:render='urlpath'></h1>
    </body>
</html>
''')

    def render_urlpath(self, ctx, data):
        return ctx.tag[url.here]

    def locateChild(self, ctx, segments):
        return MyPage(), segments[1:]

class VhostFakeRoot:
    '''
    I am a wrapper to be used at site root when you want to combine
    vhost.VHostMonsterResource with nevow.guard. If you are using guard, you
    will pass me a guard.SessionWrapper resource.
    '''
    implements(inevow.IResource)
    def __init__(self, wrapped):
        self.wrapped = wrapped

    def renderHTTP(self, ctx):
        return self.wrapped.renderHTTP(ctx)

    def locateChild(self, ctx, segments):
        '''Returns a VHostMonster if the first segment is 'vhost'. Otherwise
        delegates to the wrapped resource.'''
        if segments[0] == 'vhost':
            return vhost.VHostMonsterResource(), segments[1:]
        else:
            return self.wrapped.locateChild(ctx, segments)

siteRoot = VhostFakeRoot(MyPage())
application = service.Application('reverse proxy / vhost example')
strports.service('8080', appserver.NevowSite(siteRoot)).setServiceParent(application)
```

Save as eg sample.tac and run using twistd -noy sample.tac

Sample Lighttpd Config:

```
$HTTP['host'] =~ '^(www.example.com)$' {
        url.rewrite-once = ('^/(.*)' => '/vhost/http/%0/$1')
        # In lighttpd we alter the path manually using rewrite rule. %0
        # refers to the hostname and $1 is the path.
        proxy.server = ( '' =>
                ( (
                'host' => '127.0.0.1',
                'port' => 8080
                ) )
        )
}
```

If you prefer a mixed deployment strategy where static content is served through the faster lighttpd while dynamic content is still served by twisted you can use the following recipe.

```
$HTTP['host'] =~ '^(www.example.org)$' {
        url.rewrite-once = (
                '^/static/.*' => '$0',
                '^/(.*)' => '/vhost/http/%0/$1'
        )
        $HTTP['url'] !~ '^/static/' {
                proxy.server = ( '' =>
                        ( (
                        'host' => '127.0.0.1',
                        'port' => 8080
                        ) )
                )
        }
        server.document-root = '/path/to/your/project/trunk/'
}
```

There are 2 assumptions in this recipe:

- The static content is located at the /static/ subtree of the website.
- The project root contains a 'static' directory that is used to serve static content.

Sample Apache Config (Ref)

```
<VirtualHost www.example.com>
ProxyPass / http://localhost:8080/vhost/http/www.example.com/
ServerName www.example.com
</VirtualHost>
```

Example 2

Nevow is only to be used for part of an existing static site at a non-root url

Sample Lighttpd Config:

TODO

Sample Apache Config

TODO

See Also

- http://lighttpd.net/documentation/proxy.html
- http://httpd.apache.org/docs/2.0/mod/mod_proxy.html#proxypass

Divmod Athena

Athena is a two-way communication channel for Nevow applications. Peruse the examples. Or make this page better.

Nevow & Athena FAQ

History

Athena is the best-of-breed approach to developing interactive javascript (AJAX / Comet) applications with Divmod-Nevow. It should be noted that it *supersedes* previous solutions, such as livepage. These prior solutions should not be used with Athena development. Before using Athena, you may want to check out the *Getting started with Divmod Nevow*.

Development Environment

If you haven't developed with JavaScript before, you may wish to set up your development environment before beginning with Athena. Some tips may be available, depending on your preferred tools:

- nevow-athena-emacs
- Firebug - Firefox based javascript debugger

Simple LiveElement demo

Subclass `nevow.athena.LiveElement` and provide a `docFactory` which uses the `liveElement` renderer:

```python
from nevow import athena, loaders, tags as T

class MyElement(athena.LiveElement):
    docFactory = loaders.stan(T.div(render=T.directive('liveElement')))
```

Put the result onto a `nevow.athena.LivePage` somehow:

```python
class MyPage(athena.LivePage):
    docFactory = loaders.stan(T.html[
        T.head(render=T.directive('liveglue')),
        T.body(render=T.directive('myElement'))])

    def render_myElement(self, ctx, data):
        f = MyElement()
        f.setFragmentParent(self)
        return ctx.tag[f]
    def child_(self, ctx):
        return MyPage()
```

Put the page into a `nevow.appserver.NevowSite` somehow:

```
from nevow import appserver
site = appserver.NevowSite(MyPage())
```

Hook the site up to the internet:

```
from twisted.application import service, internet

application = service.Application('Athena Demo')
webService = internet.TCPServer(8080, site)
webService.setServiceParent(application)
```

Put it all into a `.tac` file and run it:

```
twistd -noy myelement.tac
```

And hit http://localhost:8080/. You now have an extremely simple Athena page.

Customizing Behavior

Add a Twisted plugin which maps your module name onto your JavaScript source file:

```
from nevow import athena

myPackage = athena.JSPackage({
    'MyModule': '/absolute/path/to/mymodule.js',
    })
```

Place this Python source file into `nevow/plugins/` (the Twisted plugin documentation describes where else you can put it, with the exception that Nevow plugins should be placed beneath a `nevow` directory as opposed to a `twisted` directory).

In the JavaScript source file (in this case, `mymodule.js`), import `Nevow.Athena`:

```
// import Nevow.Athena
```

Next, subclass the JavaScript `Nevow.Athena.Widget` class (notice the module name that was defined in the plugin file):

```
MyModule.MyWidget = Nevow.Athena.Widget.subclass('MyModule.MyWidget');
```

Now, add a method to your newly defined class:

```
MyModule.MyWidget.methods(
    function echo(self, argument) {
        alert('Echoing ' + argument);
        return argument;
    });
```

Define the JavaScript class which will correspond to your `LiveElement` subclass:

```
from nevow import athena, loaders, tags as T

class MyElement(athena.LiveElement):
    jsClass = u'MyModule.MyWidget'
    docFactory = loaders.stan(T.div(render=T.directive('liveElement')))
```

Invoking Code in the Browser

Add some kind of event source (in this case, a timer, but this is incidental) which will cause the server to call a method in the browser:

```
from twisted.internet import reactor

from nevow import athena, loaders, tags as T

class MyElement(athena.LiveElement):
    jsClass = u'MyModule.MyWidget'
    docFactory = loaders.stan(T.div(render=T.directive('liveElement')))

    def __init__(self, *a, **kw):
        super(MyElement, self).__init__(*a, **kw)
        reactor.callLater(5, self.myEvent)

    def myEvent(self):
        print 'My Event Firing'
        self.callRemote('echo', 12345)
```

Start up the server again and revisit <http://localhost:8080>.

Invoking Code on the Server

Add an event source (in this case, a user-interface element, but this is incidental) which will cause the browser to call a method on the server:

```
class MyElement(athena.LiveElement):
    docFactory = loaders.stan(T.div(render=T.directive('liveElement'))[
        T.input(type='submit', value='Push me',
            onclick='Nevow.Athena.Widget.get(this).clicked()')])
    ...
```

Update the JavaScript definition of `MyModule.MyWidget` to handle this event and actually call the server method:

```
MyModule.MyWidget.method(
    'clicked',
    function(self) {
        self.callRemote('echo', 'hello, world');
    });
```

Add a method to `MyElement` which the browser will call, and expose it to the browser:

```
class MyElement(athena.LiveElement):
    ...

    def echo(self, argument):
        print 'Echoing', argument
        return argument
    athena.expose(echo)
```

Start up the server again and revisit <http://localhost:8080>.

Download the files for this tutorial:

- myelement.tac
- mymodule.js
- mymodule_pkg.py

Testing

Visit the athena-testing or Test Driven Development with Athena

Implementation

Though Divmod's use of it predates the term by several years, Athena uses what some have come to call Comet. Athena's JavaScript half makes an HTTP request before it actually needs to retrieve information from the server. The server does not respond to this request until it has something to tell the browser. In this way, the server can push events to the browser instantly.

Tickets

See open tickets for Athena here.

Divmod Athena

Adding LiveElements to a LivePage on fly tutorial

Javascript function Nevow.Athena.Widget.addChildWidgetFromWidgetInfo can be used to do that.

All you need on the server is to return a LiveElement instance:

```
from nevow import athena

class WhateverElement(athena.LiveElement):

  @athena.expose
  def getNewLiveElement(self):
    return SomeOtherLiveElement()
```

Note: you should also call `setFragmentParent` on the new LiveElement (see here).

On the client, you need some code & a free XML node to append the new element to:

```
// import Nevow.Athena
// import Divmod.Runtime

What.Ever = Nevow.Athena.Widget.subclass('What.Ever');
What.Ever.methods = (

 function foo(self) {
   d = self.callRemote('getNewLiveElement');

   d.addCallback(

     function liveElementReceived(le) {

       d2 = self.addChildWidgetFromWidgetInfo(le);
       d2.addCallback(
         function childAdded(widget) {

           /* widget is a Nevow.Athena.Widget instance and it
            * represents a newly created widget for the liveelement
            * got from the server */

           /* find a node to attach the widget to: */
           self.nodeById('lastNode').appendChild(widget.node);

           /* you could also use, for example:
            *
            * var node = self.nodeById('lastNode');
            * node.replaceChild(widget.node, node.firstChild);
            */
         });

     });
```

Athena FAQ

17. When I reload a page, the server logs a traceback ending with `exceptions.AssertionError: Cannot render a LivePage more than once`

1. LivePage instances maintain server-side state that corresponds to the connection to the browser. Because of this, each LivePage instance can only be used to serve a single client. When you serve LivePages, make sure that you create a new instance for each render pass.

   Here is an example(from Dominik Neumann) about using LivePage as a main page: Wrap you Index page like:

```
class RootPage(Index):
 '''
 always return a new Index
 '''
 def child_(self, ctx):
         return Index()
```

17. I can't debug athena on Internet explorer. what gives?

1. Athena include the livefragment javascript in the body of the document via a script tag. Visual studio doesn't appear to be happy with this and does not let you set break points. If you load the javascript in the head of the document then you can set a break point in your athena livefragment javascript using visual studio. The best way I found to do this is manually include the javacript in the header and then information athena that the

livefragment javascript is already loaded. One way to do this is to call the 'hidden' _shouldInclude method on the athena livepage instance, e.g self._shouldInclude('yourjsmodule'). This will let athena know that the javascript is already loaded and not to load it twice.

17. Why doesn't Athena support Safari?

1. Athena supports recent versions of Safari.

17. How can I unit-test javascript code using Athena?

1. The same way you unit-test your ordinary Twisted or Divmod software: by using trial. You can find athena-testing. Have a look at nevow.test.test_javascript to see, how tests are prepared and run.

17. How can I add elements to an already rendered LivePage?

1. See Nevow.Athena.Widget.addChildWidgetFromWidgetInfo and the nevow-athena-tutorials-live-elements-on-fly

Nevow & Athena FAQ

17. When I reload a page, the server logs a traceback ending with `exceptions.AssertionError: Cannot render a LivePage more than once`

1. LivePage instances maintain server-side state that corresponds to the connection to the browser. Because of this, each LivePage instance can only be used to serve a single client. When you serve LivePages, make sure that you create a new instance for each render pass.

17. I can't debug athena on Internet explorer. what gives?

1. Athena include the livefragment javascript in the body of the document via a script tag. Visual studio doesn't appear to be happy with this and does not let you set break points. If you load the javascript in the head of the document then you can set a break point in your athena livefragment javascript using visual studio. The best way I found to do this is manually include the javacript in the header and then information athena that the livefragment javascript is already loaded. One way to do this is to call the 'hidden' _shouldInclude method on the athena livepage instance, e.g self._shouldInclude('yourjsmodule'). This will let athena know that the javascript is already loaded and not to load it twice.

17. Why doesn't Athena support Safari?

1. Safari has a broken JS implementation that throws an error when a nested 'named' function is encountered, e.g.

```
methods(function foo(self) {},function bar(self) {});
```

A workaround would be to use anonymous functions instead of named functions. Another issue is that someone needs to implement a custom runtime module in runtime.js to support Safari.

17. How can I unit-test javascript code using Athena?

1. The same way you unit-test your ordinary *Twisted* or *Divmod* software: by using *trial*. You can find athena-testing. Have a look at *nevow.test.test_javascript* to see, how tests are prepared and run.

Demo: news edit

```
from twisted.application import service, strports
from nevow import appserver, loaders, rend, static, url

class NewsEditPage(rend.Page):
    docFactory = loaders.xmlstr('''
<!DOCTYPE html PUBLIC '-//W3C//DTD XHTML 1.0 Strict//EN'
```

```
           'http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd'>
<html xmlns:n='http://nevow.com/ns/nevow/0.1'>
    <head>
        <title>Example 1: A News Item Editor</title>
        <link rel='stylesheet' href='form_css' type='text/css' />
    </head>
    <body>
        <h1>Example 1: A News Item Editor</h1>
        <fieldset>
            <legend>Add / Edit News Item</legend>
            <p>Form Goes Here</p>
        </fieldset>

        <ol n:render='sequence' n:data='newsItems'>
            <li n:pattern='item' n:render='mapping'>
                <strong><n:slot name='title' /></strong>: <n:slot name='description' /
↪>
            </li>
        </ol>
    </body>
</html>
''')

    def __init__(self, *args, **kwargs):
        self.store = kwargs.pop('store')

    def saveNewsItem(self, newsItemData):
        self.store.append(newsItemData)
        return url.here.click('confirmation')

    def data_newsItems(self, ctx, name):
        return self.store

class ConfirmationPage(rend.Page):
    docFactory = loaders.xmlstr('''
<!DOCTYPE html PUBLIC '-//W3C//DTD XHTML 1.0 Strict//EN'
           'http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd'>
<html>
    <body>
        <h1>Your item has been saved</h1>
        <ul>
            <li><a href='./'>Go back</a></li>
        </ul>
    </body>
</html>
''')

# A place to store news items. A list of dicts in this simple case.
store = [dict(title='Lorum Ipsum', description='''
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Sed sed enim mollis
nulla faucibus aliquet. Praesent nec nibh. Nam eget pede. Nam tincidunt purus id
lorem. Vestibulum lectus nisl, molestie vitae, feugiat egestas, sodales et,
tellus. Vivamus eu libero. Nulla facilisi. Nullam nec dolor. Proin ac diam at
neque auctor pulvinar. Maecenas eros nibh, fermentum at, eleifend at, malesuada
eu, nunc. Sed posuere felis eu ipsum. In volutpat. Phasellus viverra. Quisque
dignissim mattis turpis. Maecenas accumsan ipsum vel orci. Cras ac lectus. Sed
nec nisl. Integer iaculis elit scelerisque sapien. Curabitur ac diam.
''')]
```

```
rootResource = NewsEditPage(store=store)
rootResource.putChild('confirmation', ConfirmationPage())

application = service.Application('News item editor')
strports.service('8080', appserver.NevowSite(rootResource)).
↪setServiceParent(application)
```

Demo: results

```
import string

from twisted.application import service, strports
from nevow import appserver, inevow, rend, tags as T, loaders

class ResultsPage(rend.Page):
    docFactory = loaders.xmlstr('''
<!DOCTYPE html PUBLIC '-//W3C//DTD XHTML 1.0 Strict//EN'
           'http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd'>
<html xmlns:n='http://nevow.com/ns/nevow/0.1'>
    <body>
        <form action='.'>
            <table border='1'>
                <thead>
                    <tr>
                        <th>Number</th>
                        <th>Letter</th>
                    </tr>
                </thead>
                <tfoot>
                    <tr>
                        <td colspan='2'>
                            Showing <select
                                        n:render='itemsPerPageOptions'
                                        name='itemsPerPage'
                                        onchange='this.form.submit();' /> items per␣
↪page
                        </td>
                    </tr>
                </tfoot>
                <tbody n:render='sequence' n:data='alphabet'>
                    <tr n:pattern='item' n:render='mapping'>
                        <td><n:slot name='index' /></td>
                        <td><n:slot name='letter' /></td>
                    </tr>
                </tbody>
            </table>
        </form>
    </body>
</html>
''')
    itemsPerPageChoices = range(10, 60, 10)

    def beforeRender(self, ctx):
        sess = inevow.ISession(ctx)
        if not hasattr(sess, 'pagerPrefs'):
            sess.pagerPrefs = dict(itemsPerPage = self.itemsPerPageChoices[0])

        try:
            itemsPerPage = abs(int(ctx.arg('itemsPerPage', 0)))
        except ValueError: #when the submitted value can't be converted to int
            itemsPerPage = 0

        if itemsPerPage > 0 and itemsPerPage in self.itemsPerPageChoices:
            sess.pagerPrefs['itemsPerPage'] = itemsPerPage

    def render_itemsPerPageOptions(self, ctx, data):
        options = [T.option(value=i)[i] for i in self.itemsPerPageChoices]
        default = inevow.ISession(ctx).pagerPrefs.get('itemsPerPage')
        #extract the default option tag and set it's selected attribute
        options[self.itemsPerPageChoices.index(default)](selected='selected')
        return ctx.tag.clear()[options]

    def data_alphabet(self, ctx, name):
        alphabet = string.ascii_lowercase
        #a dummy dataset, in real life this might come from a DB.
        data = [dict(index=i, letter=alphabet[i]) for i in range(len(alphabet))]
        return data[:inevow.ISession(ctx).pagerPrefs.get('itemsPerPage')]

application = service.Application('items per page')
strports.service('8080', appserver.NevowSite(ResultsPage())).
↪setServiceParent(application)
```

Tutorial: Using Storm with Nevow

Note: There is work on integrating Storm with Twisted (storm/twisted-integration). Until then, the best way is probably to use deferToThread.

(Here is *A possible approach with Nevow and Storm/twisted-integration*)

Before a tutorial can be written on how to use storm in Nevow, someone needs to figure out 'best practices for using Storm in asynchronous codebases.' From a chat in #storm on freenode (11 July 2007):

```
[5:13pm] dialtone: storm API is fine, one just needs to see if: a) it's thread
                   safe and b) objects returned can be modified in a different
                   thread
[5:13pm] dialtone: and then committed in another one yet
[5:13pm] radix: dialtone: It is certainly thread-safe, but you should not use
                   the same Store in multiple threads
[5:14pm] dialtone: how do you do with deferToThread then?
[5:14pm] radix: dialtone: I don't know.
[5:14pm] radix: dialtone: Probably you don't load the objects until you get
                   into the other thread.
[5:14pm] dialtone: that's sort of a problem probably, unless there's an
                   alternative solution
[5:15pm] radix: dialtone: Nobody has figured out best practices for using Storm
                   in asynchronous codebases yet.
[5:15pm] dialtone: ouch :(
[5:15pm] radix: dialtone: (That is, assuming your application actually wants to
                   make the database operations asynchronous)
[5:16pm] dialtone: I can accept it being synchronous, but run in a different
                   thread using deferToThread
[5:16pm] radix: dialtone: for example, it would be very reasonable to use Storm
```

```
                    like Axiom: with SQLite, just blocking on database interaction
[5:16pm] dialtone: radix: yes well... but that comes with a lot of restrictions
                    in itself
[5:17pm] radix: dialtone: I even have a Twisted server which uses Storm to talk
                    to postgres in a totally blocking manner
[5:17pm] radix: but it's not a very typical server
[5:17pm] dialtone: sqlite doesn't really scale that well with high concurrency
                    and scaling a shared sqlite instance is pretty hard
[5:17pm] dotz: dialtone: What's wrong with 'ceate a high level API for data
                    manipulation and retrival, use some rpc mechanism, like pb,
                    and make the hi-level database backend another process'?
[5:18pm] dialtone: dotz: that' waaay more effort than what I'm currently doing
[5:18pm] radix: dialtone: I'm sure there are some clever things that can be done
                    to use Storm in asynchronous contexts, and I'm fairly interested
                    in learning about them
[5:18pm] dotz: dialtone: and what are you currently doing?
[5:18pm] dialtone: as I said: I'm using the query builder from sqlalchemy to build
                    and execute queries that return resultsets with a dict-like API
[5:19pm] dialtone: I simply write the queries and use the dict-like APi in nevow
[5:19pm] dialtone: and I run the functions in a separate thread with deferToThread
[5:20pm] dialtone: radix: maybe writing a store class that keeps track of which
                    threads owns which connection and then reuse the connection when
                    needed
[5:20pm] dialtone: this way you could share the store object and run queries
                    independently of the context
[5:20pm] radix: dialtone: maybe. I'm not sure it'd be that simple.
```

A possible approach with Nevow and Storm/twisted-integration

(thanks to these very nice guys in mailing list)

Set up pool

```python
from storm.databases.sqlite import SQLite
from storm.uri import URI
from storm.twisted.store import StorePool

database = SQLite(URI('sqlite:///test.db'))
pool = StorePool(database, 5, 10)

pool.start()
```

Create models

```python
from storm.locals import *
from storm.twisted.wrapper import DeferredReference

class People(object):
    __storm_table__ = 'people'

    id                      =       Int(primary=True)
    email           =       Unicode()
    nickname        =       Unicode()
    passwd          =       Unicode()
    avatar          =       Unicode()


class Topic(object):
    __storm_table__ = 'topic'

    id                      =       Int(primary=True)
    topic_title     =       Unicode()
    keyword         =       Unicode()
    content         =       Unicode()
    people_id       =       Int()
    people          =       DeferredReference(people_id, People.id)


    posted          =       DateTime()
```

and have fun

```
from twisted.internet.defer import inlineCallbacks, returnValue

from db import pool
from db.models import Topic


    def getTopics(self):
            @inlineCallbacks
            def transaction(store):
                    dr = yield store.find(Topic, Topic.topic_title==self.topic, Topic.
↪keyword==self.keyword)

                    dr.order_by(Desc(Topic.posted))
                    items = yield dr.all()

                    for item in items:
                            #do whatever.
                            people = yield item.people
                            #do whatever else.

                    #this can recycle stores in pool
                    yield store.commit()

                    returnValue(    ) #return whatever you want.

            return pool.transact(transaction)
```

Divmod PyFlakes

PyFlakes a Lint-like tool for Python, like PyChecker or PyLint. It is focused on identifying common errors quickly without executing Python code.

Its primary advantage over PyChecker is that it is *fast*. You don't have to sit around for minutes waiting for the checker to run; it runs on most large projects in only a few seconds.

The two primary categories of defects reported by PyFlakes are:

- Names which are used but not defined or used before they are defined
- Names which are redefined without having been used

These can each take many forms. For example, PyFlakes will tell you when you have forgotten an import, mistyped a variable name, defined two functions with the same name, shadowed a variable from another scope, imported a module twice, or two different modules with the same name, and so on.

Download

- 0.5.0 Release (Release Notes) or `pip install pyflakes`
- Trunk: `bzr branch lp:divmod.org && cd divmod.org/PyFlakes` (Browse)

Exits

- PyFlakes on PyPI
- PyFlakes on Launchpad
- PyFlakes on SWiK

Divmod Sine

A voice over IP application server.

Sine provides:

- SIP Registrar
- SIP Proxy
- Third party call control (3PCC)
- Voice-mail
- Through-the-web configuration
- [wiki:DivmodMantissa Divmod Mantissa] integration

Download

- Release: [http://divmod.org/trac/attachment/wiki/SoftwareReleases/Sine-0.3.0.tar.gz?format=raw Download the latest release - 0.3.0!] (Requires [wiki:DivmodMantissa Mantissa]) ([source:/tags/releases/Sine-0.3.0/NEWS.txt Release Notes])
- Bleeding Edge: svn co http://divmod.org/svn/Divmod/trunk/Sine Sine

See also

- ‘’Development” version of the [http://buildbot.divmod.org/apidocs/sine.html Sine API docs]

Divmod Vertex

Vertex is an implementation of the Q2Q protocol (sort of like P2P, but one better). There are a few moving parts in Vertex:

- PTCP: a protocol which is nearly identical to TCP, but which runs over UDP. This lets Q2Q penetrate most NAT configurations.
- JUICE ([JU]ice [I]s [C]oncurrent [E]vents): a very simple but immensely flexible protocol which forms the basis of the high-level aspects of Q2Q
- vertex: a command line tool which exposes a few features useful in many situations (such as registration and authentication)

Q2Q is a very high-level protocol (alternatively, transport) the goal of which is to make communication over the internet a possibility (if you enjoy setting up tunnels or firewall rules whenever you need to transfer a file between two computers, Q2Q may not be for you). Q2Q endpoints aren't hardware addresses or network addresses. They look a lot like email addresses and they act a lot like instant message addresses. You can hook into yours wherever you can access the internet, and you can be online or offline as you choose (Q2Q supports multiple unrelated protocols, so you also might be online for some services but offline for others). Two people with Q2Q addresses can easily communicate without out-of-band negotiation of their physical locations or the topology of their networks. If Alice wants to talk to Bob, Alice will always just open a connection to bob@divmod.com/chat. If Bob is online anywhere at all, the connection will have an opportunity to succeed (Bob might be busy or not want to talk to Alice, but that is another matter ;). The connection is authenticated in both directions, so if it does succeed Alice knows she is talking to the real Bob and vice versa.

The Q2Q network has some decentralized features (there is no one server or company which can control all Q2Q addresses) and features of centralization (addresses beneath a particular domain are issued by a server for that domain; once issued, some activities require the server to be contacted again, while others do not). Vertex includes an identity server capable of hosting Q2Q addresses. Once you've installed the 0.1 release, you can run it like this:

```
exarkun@boson:~$ cat > q2q-standalone.tac
from vertex.q2qstandalone import defaultConfig
application = defaultConfig()
exarkun@boson:~$ twistd -noy q2q-standalone.tac
2005/10/15 00:12 EDT [-] Log opened.
2005/10/15 00:12 EDT [-] twistd 2.0.1 (/usr/bin/python2.4 2.4.1) starting up
2005/10/15 00:12 EDT [-] reactor class: twisted.internet.selectreactor.SelectReactor
2005/10/15 00:12 EDT [-] Loading q2q-standalone.tac...
2005/10/15 00:12 EDT [-] Loaded.
2005/10/15 00:12 EDT [-] vertex.q2q.Q2QService starting on 8788
2005/10/15 00:12 EDT [-] Starting factory <Q2QService 'service'@-488b6a34>
2005/10/15 00:12 EDT [-] vertex.ptcp.PTCP starting on 8788
2005/10/15 00:12 EDT [-] Starting protocol <vertex.ptcp.PTCP instance at 0xb777884c>
2005/10/15 00:12 EDT [-] Binding PTCP/UDP 8788=8788
2005/10/15 00:12 EDT [-] vertex.q2q.Q2QBootstrapFactory starting on 8789
2005/10/15 00:12 EDT [-] Starting factory <vertex.q2q.Q2QBootstrapFactory instance at␣
↪0xb77787cc>
```

You can acquire a new Q2Q address using the vertex command line tool:

```
exarkun@boson:~$ vertex register exarkun@boson password
```

boson is a name on my local network, making this address rather useless. On the other hand, no one will be able to pretend to be me by cracking my hopelessly weak password ;) If you set up a Vertex server on a public host, you will be able to register a real, honest-to-goodness Q2Q address beneath its domain (careful - so will anyone else).

vertex also offers a tool for requesting a signed certificate from the server. These certificates can be used to prove ones identity to foreign domains without involving ones home server. Another feature vertex provides is a toy file transfer application. Bob can issue a vertex receive while Alice issues a vertex send pointed at him, and the file will be transferred.

Much of the real power of Q2Q is exposed to developers using two methods: listenQ2Q and connectQ2Q. These work in roughly the same way Twisted's listenTCP and connectTCP work: they offer support for writing servers and clients that operate on the Q2Q network.

Exits

- [http://divmod.org/projects/vertex Old Divmod project page for Vertex]
- [http://www.swik.net/vertex Vertex on swik]
- [wiki:UsingVertex Using Vertex], a work-in-progress page about using Vertex in an application
- ''Development" version of [http://buildbot.divmod.org/apidocs/vertex.html Vertex API docs]

Status

- Vertex does ''‘not'‘' currently pass its test suite on Windows or Mac OS X.
- This [query:?group=status&component=Vertex&order=priority custom Vertex ticket query] will show you all the tickets (bugs and feature requests) filed against Vertex.
- Vertex is likely usable as a Q2Q server on Linux and Windows.
- Vertex is likely usable as a Q2Q client on Linux.

Download

- Release: [http://divmod.org/trac/attachment/wiki/SoftwareReleases/Vertex-0.3.0.tar.gz?format=raw Latest release: 0.3.0] ([source:/tags/releases/Vertex-0.3.0/NEWS.txt Release Notes])
- Trunk: svn co http://divmod.org/svn/Divmod/trunk/Vertex Vertex

Imaginary

Imaginary numbers are so named not because they are fictitious, but because they are along a different axis from real numbers.

Divmod Imaginary is a simulationist's take on the realm of role playing, interactive fiction, and multiplayer dungeons. It incorporates gameplay features from each area while attempting to provide a richer environment than is generally available from existing systems.

Status: Imaginary is currently on the back of some peoples' minds.

- Project Home Page
- Browse the source
- View the tickets

See also instructions for starting an Imaginary server.

CHAPTER 2

Retired Projects

This is the rest home for Divmod projects that didn't quite make it up the evolutionary ladder. These projects are not being developed or maintained.

Lupy

The Lupy project has been RETIRED! For full-text indexing and search using Python and Lucene we recommend [http://pylucene.osafoundation.org/ OSAF's PyLucene] instead.

Lupy is a is a full-text indexer and search engine written in Python. It is a port of Jakarta Lucene 1.2 to Python. Specifically, it reads and writes indexes in Lucene binary format. Like Lucene, it is sophisticated and scalable. Lucene is a polished and mature project and you are encouraged to read the documentation found at the Lucene home page.

Lupy requires Python 2.3 or greater.

Pyndex

The Pyndex project has been RETIRED! For full-text indexing and search we recommend [wiki:DivmodXapwrap Divmod Xapwrap] or [http://pylucene.osafoundation.org/ PyLucene] instead.

Pyndex is a simple and fast full-text indexer (aka search engine) implemented in Python. It uses Metakit as its storage back-end. It works well for quickly adding search to an application, and is also well suited to in-memory indexing and search. It performs best in applications involving a few thousand documents. Pyndex can handle AND, OR and phrase queries.

'Old' Quotient

Quotient's name lives on but the original ATOP-based Quotient is now known as Old Quotient. The latest version of [DivmodQuotient Quotient] is based on [DivmodAxiom Axiom]: our 'smart' database.

The short version is: Quotient has been ported to Axiom and (the new) Mantissa. We have been supporting and will continue to support our existing customer base that uses 'Old' Quotient – never fear :-) We are also investigating potential integration with our [http://blendix.com Blendix] service, so stay tuned!

Divmod Xapwrap

This project is no longer supported. It is not compatible with the latest release of Xapian.

Enhancements to the Python bindings for the [http://www.xapian.org Xapian] full-text indexing library.

While Xapian comes with bindings, they are hard to work with, and being SWIG'd, a literal translation of a C++ API. Xapwrap simplifies working with Xapian databases in Python.

This is implemented as a second layer of wrapping, so Xapian's bindings are still required.

[http://divmod.org/trac/attachment/wiki/SoftwareReleases/Xapwrap-0.3.1.tar.gz?format=raw Get release 0.3.1]

CHAPTER 3

Philosophy

Many writings of various Divmod members are archived here.

Potato Programming

One potato, two potato, three potato, four....

Potato programming is the sort of programming that encourages you to write your own 'for' loops and build up / tear down data structures, rather than passing vectors or iterators down through an interface in such a way that would allow a smarter version of that interface to be more efficient.

In other words, this is the potato-programming way to add a file containing lines of numbers:

```
f = file('test.numbers')
accum = 0.
for line in f:
    accum += float(line)
print accum
```

Contrast this with:

```
f = file('test.numbers')
print sum(map(float, f))
```

This term was coined by R0ml Lefkowitz.

See also: [PotatoProgrammingExplained Potato Programming Explained]

CHAPTER 4

Biographies

Some background information on the people behind Divmod.

Glyph Lefkowitz

Although scientists agree he was a danger even before his days at [http://origin.ea.com/ Origin Systems] working on the never-released sequel to Ultima Online, popular concern about the threat posed by Glyph Lefkowitz began when it became clear he was the author of the [http://www.twistedmatrix.com/ Twisted Framework]. Widespread panic has escalated since he began publishing subversive thoughts through the channel known cryptically as [http://www.livejournal.com/users/glyf/ 'Handwriting on the Sky'].

Allegedly raised by a [http://r0ml.net/blog/ 'computer programmer'], rumours have circulated that he was completely removed from human society as a child, and there is some speculation that [http://yellow5.com/pokey/archive/index52.html he is not human himself]. Sources who have asked not to be named claim he is referred to by relatives as an 'alien mutant'.

Since he can make up any title he wants and write it here, Glyph is Divmod's Chief Architect. That means he is responsible for coordinating the technology that glues the various parts of Divmod's applications and infrastructure together, such as [wiki:DivmodAxiom Axiom] and [wiki:DivmodMantissa Mantissa].

Jean-Paul

Medium-sized mammal of the family Hominidae in the order Primates, found almost exclusively in the eastern portions of North America. Jean-Paul is non-migratory, but at times can be found several hundred miles from his native habitat. Jean-Paul's diet consists primarily of nuts and berries and is supplemented by any of a number of caffeine-rich foods, such as coffee beans and mountain dew.

||,,Jean-Paul in his native habitat.,,|| ||[[Image(http://divmod.org/users/exarkun/bearsalmon.jpg)]]|| ||,,© Kim Heacox/Accent Alaska,,||

- [http://jcalderone.livejournal.com/ Writings by Jean-Paul]

- [http://twistedmatrix.com/projects/mail Other] [http://twistedmatrix.com/projects/names Projects] [http://twistedmatrix.com/projects/news by] [http://twistedmatrix.com/projects/words Jean-Paul]

Moe Aboulkheir

I write computer programs for Divmod, from a third-floor apartment in Cape Town, South Africa. I like building things and seeing them come together; I tend to gravitate towards the implementation of tangible, user-facing features. Just about all of the interesting things about me don't have a place in a corporate personnel blurb, so I'll instead say something about enjoying reading books and listening to music. I don't watch anime or read web comics, although I do enjoy playing computer games, but don't maintain a blog.

Duncan McGreggor (oubiwann)

Brief Bio

Duncan started his hacking career at the ripe old age of 11 in the early 80s. From his adventures in rewriting games on Kaypro's luggable CP/M machine to the world of open source, programming has been his passion. When Duncan wasn't programming, he: was an Army MI linguist; worked his way up to sous chef in a fancy restaurant; studied quantum mechanics and mathematics as a physics major; lived with Tibetan monks, learning meditation and esoteric philosophy; and started his own software consulting company.

After several happy years of consulting, Duncan was lured into the Divmod fold by the legendary reputations of its developers and the opportunity to work with a team that has been coding on the cutting edge for over four years. In addition to his work with Divmod, Duncan actively participates in many open source projects and leads a few of his own.

At Divmod, Duncan is responsible for directing and managing operations such that the architectural visions of Glyph and the developers meets the executive goals of Amir and the Board of Directors.

Sites

- http://oubiwann.blogspot.com/
- http://blendix.com/users/oubiwann/
- http://schrobox.blogspot.com/

Projects

Most everything is linked here: http://www.ohloh.net/accounts/9254

Here are some direct links:

- http://pymon.sourceforge.net/
- http://code.google.com/p/twisted-jsonrpc/
- http://code.google.com/p/testgen4web-python/
- http://code.google.com/p/pyrtf-ng/
- http://code.google.com/p/coymon/

Wiki Notes

- [wiki:People/DuncanMcGreggor/MantissaPluginNotes Mantissa Plugin Notes]
- [wiki:People/DuncanMcGreggor/MantissaAxiomaticProject Mantissa's 'axiomatic project' Subcommand Notes]

Draft Pages

- [wiki:Drafts/FrontPage Front Page]
- [wiki:Download]
- [wiki:DivmodCommunity]

CHAPTER 5

Miscellaneous

Random pages that might not really be needed anymore but shouldn't be discarded.

Python

A pretty good programming language.

If you don't know it, learn it.

http://www.python.org/

Q2Q

Q2q is a protocol for:

- opening authenticated connections, even through NAT
- allowing a user to reliably demonstrate their identity (for distributed authentication)
- receiving real-time data directly from other users

Q2Q provides a mechanism for a user to decide whether they want to expose their IP address to a third party *before* accepting a peer-to-peer connection.

It is byte-stream oriented and application-agnostic. Any peer-to-peer application can use Q2Q to open connections and deliver messages.

Divmod Vertex is the Divmod implemention of Q2Q.

Exits

[http://swik.net/q2q Q2Q tag on swik]

SQLite

SQLite is an embedded relational database.

You can find a copy at http://sqlite.org/, and Python bindings at http://pysqlite.org/ or http://www.rogerbinns.com/apsw.html.

Divmod Axiom currently uses SQLite exclusively as its backend.

The Divmod Fan Club

[[Image(http://divmod.org/tracdocs/fanclub_whtbck.png, right)]]

Do you use Divmod's code?

Do you have a poster of Exarkun on your wall?

Do you love our giant-killing chutzpah?

Do you have a friend or family member on the Divmod team?

Has work you've done using a Divmod project made you fantastically wealthy, such that you don't know what to do with all your extra disposable income?

Now you know.

Joining Up

Join the club!

We offer a few different levels of membership.

- Bronze Membership - $10 per month - **Buy Glyph enough caffeine to get through one average day.**
- Silver Membership - $25 per month - **Take JP out to a restaurant.**
- Gold Membership - $50 per month - **Pay for Allen's cell phone calls to Divmod HQ.**
- Platinum Membership - $100 per month - **Buy Moe's groceries for a month.**
- Diamond Membership - $250 per month - **Buy an iPod for a tireless contributor.**
- fan-club-mithril - starting at $1000 per month (email to sales@divmod.com) - **Buy Amir a pony.**

Huh What?!

Our developer community has approached us to express their appreciation of our work, and to influence us to work on particular aspects of our product for the their benefit.

We've created the Divmod Fan Club, or the 'DFC', to give those users a way to pay back, and a way for us to very publicly say 'thank you!'.

It's Awesome - Why should I sign up?

Conscience. Influence. Status.

Conscience

Simple: you give because you take.

Influence

The club has monthly meetings over IRC where Divmod will propose a group of open-source features to be prioritized. Club members will be allowed to vote on the order those features will be implemented in. We are currently tallying the results of our first meeting - watch this space for our club-designated priorities!

Members will be allowed a number of votes corresponding to the number of dollars per month that their membership costs, except Mithril members, who receive a maximum of 250 votes each, or can instead choose to apply their membership fees directly to the expense of implementing features of their choice.

Status

Members of the club who sign up for one of our Divmod commercial services (such as the soon-to-be-re-launched Divmod Mail) will occasionally get special treats, early access to new features, and extra goodies not available to the general public.

Eventually they will also receive a 'badge' displayed next to their user-icon in public areas.

Additionally, since we are currently using this club to beta-test our billing system, members who sign up during the beta period (prior to the first day of !PyCon) will receive an 'arcanite alloy' bonus, and be allowed 5 additional votes for their membership level during the first 3 meetings. Members who are currently subscribers to Divmod Mail will additionally receive this bonus indefinitely.

Where does the money go?

To pay employees, particularly for time spent on community work that is not essential to Divmod's core business.

To hire or buy gifts for some of our open source contributors.

To pay for hosting.

Once that's done, we may send some of this money to open-source contributors to our products whom we do not employ, and then, if we have hojillions of dollars left over, start paying for development of features in software Divmod's code depends upon, such as SQLite and Python.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search