YARHP Documentation

Release 0.1

Vahan Ayvazyan <vayvazyan@ludia.com>, Hadrien David <hdavid

October 30, 2012

CONTENTS

Yarhp extends the *Pyramid web framework* with the purpose of making it easy to build a RESTful web service by simply declaring your resources in an intuitive and plain way, hiding away the technicality and repetitiveness of otherwise rich pyramid api. It also defines default behaviors for CRUD actions for each resource, keeping it simple yet flexible to make adjustments.

```
user = root.add('user', 'users')
user.add('score', 'scores')
```

By simply adding the code above to your bootstrapping routine you define routes, views and validators for all the CRUD actions (index, show, create, etc) for your `user` resource. The second line not only defines everything you need for `score` resource, but also the relationship with the `user`. Thats a lot of (unittested) code written for you, dude !

CHAPTER

ONE

GETTING STARTED WITH YARHP

1.1 Basic layout

Let's consider a simple web game *tictactoe* that has the following resources:

- /users
- /users/{id}/scores

1. Create a pyramid application using `pcreate -t starter tictactoe`.You will now have basic structure with some files and `__init__.py` among them. Remove all the lines in `main` except the first and last two, leaving only config object, scan and return statement.
2. In your Pyramid project's `__init__.py` add the following:

   ```
   config.include('yarhp')
   root = config.get_root_resource()

   user = root.add('user', 'users', model='yarhp.NoModel')
   user.add('score', 'scores', model='yarhp.NoModel')
   ```

   Obviously you have to have `yarhp` pip'd.
3. Create view files for each resource. By default, `yarhp` expects view file for each resource under `your_project/views` module. In our case we create the following files with following content:

   `tictactoe/views/users.py`

   ```
   from yarhp.view import BaseView


   class UsersView(BaseView):
       def index(self):
           return []
   ```

   `tictactoe/views/user_scores.py`

   ```
   from yarhp.view import BaseView


   class UserScoresView(BaseView):
       def index(self, user_id):
           return []

       def show(self, user_id, id):
           return 'show score %s for user %s ' % (id, user_id)
   ```

Notice the class naming is in plural form (the collection name of the resource) followed by `View` in CamelCase. This would be different for models in the examples bellow. Models use the Capitalized singular form.

4. **Run `pserve development.ini` in the root of the project and point your browser to `http://localhost:6543/us`**
   You should get a json response that looks like `{"total": 0, "users": []}`. Similarly you can navigate to `http://localhost:6543/users/100/scores` and should get the same empty result.

1.2 Introducing Models

In the previous section the application was bare minimum. Now lets have some persistence.

Yarhp expects that each resource has by default its model under `project/model` module. (dont forget to create the `__init__.py` file to make the folder a python module.)

1. **Create models for each resource. Note that in the case of models, names are in singular form, not plural as oppose to the v**
   `tictactoe/model/user.py`

```
import shelve

class User(object):
    db = shelve.open('user')
    def __init__(self, **kw):
        self.__dict__.update(kw)

    def save(self):
        self.db[self.id] = self.__dict__
        return self
```

   `tictactoe/model/user_score.py`

```
import shelve

class UserScore(object):
    db = shelve.open('user_score')
    def __init__(self, **kw):
        self.__dict__.update(kw)

    def save(self):
        #key is "composite"
        self.db['%s_%s' % (self.user['id'], self.id)] = self.__dict__
        return self
```

2. In project's `__init__.py` file remove `model='yarhp.NoModel'`, since we have models now. Add some fixtures in the main like so:

```
#test data
user1 = User(id='1', name='Bob').save()
user2 = User(id='2', name='Alice').save()

UserScore(id='1', user=user1.__dict__, score='100').save()
UserScore(id='2', user=user1.__dict__, score='200').save()
UserScore(id='3', user=user1.__dict__, score='300').save()

UserScore(id='1', user=user1.__dict__, score='1000').save()
UserScore(id='2', user=user2.__dict__, score='2000').save()
```

3. The views now look little different too:

```
from yarhp.view import BaseView
from tictactoe.model.user import User

class UsersView(BaseView):
    def index(self):
        return User.db.values()

    def show(self, id):
        return User.db[id]
```

and

```
from yarhp.view import BaseView
from tictactoe.model.user_score import UserScore

class UserScoresView(BaseView):
    def index(self, user_id):
        return [score for score in UserScore.db.values()
                    if score['user']['id'] == user_id]

    def show(self, user_id, id):
        return UserScore.db["%s_%s" % (user_id, id)]
```

Run the server and check in the browser the resources if they work.

The model files can be manually specified in `__init__.py` file during the declaration of the resource if its not the default location:

```
user.add('score', 'scores', model='tictactoe.custom_model.MyFunkyScoreModel')
```

Same goes for the view files as well. `view` param passed to `add` method will do the trick. In both cases you can pass either dotted path to the model class or the class object itself, obviously importing it first.

1.3 Validators

Basic idea behind validators is to, well, validate. In `yarhp` you can either use built-in validators or write your own to validate inputs coming from the request as well as output going into to response.

Lets assume we want to validate the creation of the `user` resource.

```
from yarhp.wrappers import validator

class UserView(BaseView):
    @validator(name={'type':str, 'required':True},
               age={'type':int, 'required':False},
               gender={'type':str, 'required':False})
    def create(self):
        User(name=self.request.params['name'],
            gender=self.request.params.get('gender'))

        return HTTPCreated()
```

As you can see, `validator` decorator does 2 things. It forces `required` fields to be present in the request and also checks for the types. In the example above, `name` is required field and it must have a type `str`, `age` is an `int` and required, `gender` is a `str` but could be ommited.

1.4 Automagic Validation

In case the model is based on SQL Alchemy (or Elixir for that matter), you have an option of deriving your view from `yarhp.sqla.SQLAView` class, which will do bunch of validations for you and also implement basic behaviours of all actions. You, of course, can define your own action in your view. Validation with `SQLAView` works by checking your sql schema and validating your input against that. For example if you have the following elixir entity for `User` model:

```
class User(Entity):
    using_options(tablename='user', identity='1', inheritance='multi')

    id = Field(BigInteger, primary_key=True, autoincrement=False)
    name = Field(Unicode(128), required=True)
    gender = Field(Unicode(10), required=False)
    age = Field(Integer, required=False)
```

And the following view:

```
class UserView(SQLAView):
    ...

    def create(self):
        User(**self.request.params).save()
        return HTTPCreated()
```

The `create` method will be validated against your model defined in `User` automatically. That validation would be equivalent of the `validator` decorator explained in the pervious section.

Since `user` resource is a simple root resource, you could just skip the whole view all together and `yarhp` would implement all actions and validations for you. In case of nested resources, `yarhp` is still in experimental phase.

CHAPTER

TWO

DESIGN AND CONVENTIONS

2.1 Models

1. Always use common names.
2. Always singular.
3. Use composite names *weak entities*, i.e.: entities that cannot exist on their own (without the parents model). A foreign key that's part of the composite primary key is a hint that we should use a composite name for the model which represents this table.

2.2 URLs

1. Always use the *collection name* of the resource (if any)
2. Avoid using verbs.
3. Minimize the use of query string.

Important: Always use route names using helpers to generate URLs. Hard-coded URLs must not be found in the code whether relative, or absolute, in templates, models or views. To generate route path, you must use the `route_path()` method from the `Request` class. It takes a route name and resource IDs and generates the URL:

```
>>> route_path('user_transactions', user_id=3, id=7)
'/users/3/transactions/7'
```

2.3 Route names

1. Always lower cased, underscore separated.
2. *Collection name* is used for routes trying to access *multiple instances* of a resource, e.g: `users` is the route name of `/users`.
3. *Member name* is used for routes accessing a single instance or the only instance of the resource, e.g.:
   - `user` is the route name of `/users/2`
   - `system` is the route name of `/system`
4. parent resource member name is always a prefix for route names of a *nested resource*, e.g.:

- `order_detail_items` is the route name for `/orders/123/details/12/items/7`

2.4 Views

Follows same conventions as route names, with the following changes:

1. Resource name use `CamelCase`.
2. All views are suffixed with `View`.
3. The *collection name* of the served resource (the leaf) is always used, e.g.:

URL	Route name	View
`/users/12/transactions/14`	`user_transactions`	`UserTransactionsView`
`/users`	`users`	`UsersView`

A view class may define the following methods:

- `index`
- `show`
- `create`
- `update`
- `delete`
- `new`
- `edit`

See *API Documentation* for the relation between HTTP verbs and Python view methods.

Notes on View methods and HTTP methods.

If derived from:

1. BaseView Using include param in resource.add() will tell yarhp which actions must be implemented. AttributeError will be raised if one of the icnluded actions are not defined in your view. Using exclude will tell yarhp that the current resource does not support excluded actions and thus 405 Method Not Allowed will be raised instead of AttributeError.
2. SQLAView default index, show and delete methods are implemented and can be overwritten in your view. Note that the defaults work currently only for top level resources.
3. DynamoDBView default create update index show delete methods are implemented. Works only for top level resources.

   Using include/exlude params in resource.add() will have similar to 1) behavior, that is raise AttributeError if action is in include list but not implemented (except the default methods) and raise 405 if action is in exclude list.

By default all methods listed above are included.

CHAPTER

THREE

GLOSSARY

collection name The plural name of a resource. E.g `users`.

collection resource A resource where several instances of its type can exist and need a unique ID in the system to be identified.

marker A portion of the URL which will be dynamically matched, declared with the syntax `{marker}`.

member name The singular name of a resource. E.g: `user`.

named routes A defined route in your application that has a unique (within your routes map) name you can use to refer to.

nested resource A resource that has parents. Like `bookmarks` in `users/2/bookmarks`.

root resource A top level resource that does not have a parent.

singular resource A singleton resource within its parent scope (does not need to be identified by an ID) E.g `users/3/profile` in case a user has a single profile.

CHAPTER

FOUR

API DOCUMENTATION

4.1 Resources

class yarhp.resource.**Resource**(*config*, *member_name=''*, *collection_name=''*, *parent=None*, *uid=''*, *children=None*, *model=None*)

Class providing the core functionality.

```
m = Resource(config)
pa = m.add('parent', 'parents')
pa.add('child', 'children')
```

add(*member_name*, *collection_name=''*, *parent=None*, *uid=''*, ***kwargs*)

Parameters

- **member_name** – singular name of the resource. It should be the appropriate singular version of the resource given your locale and used with members of the collection.
- **collection_name** – plural name of the resource. It will be used to refer to the resource collection methods and should be a plural version of the member_name argument. Note: if collection_name is empty, it means resource is singular
- **parent** – parent resource name or object.
- **uid** – unique name for the resource
- **kwargs** – view: custom view to overwrite the default one. model: custom model class to override the default one used in default view.

  the rest of the keyward arguments are passed to add_resource call.

Returns ResourceMap object

add_from(*resource*)

add a resource with its all children resources to the current resource

ancestors

Returns the list of ancestor resources.

get_ancestors()

Returns the list of ancestor resources.

yarhp.resource.**add_resource**(*config*, *view*, *member_name*, *collection_name*, ***kwargs*)

view is a dotted name of (or direct reference to) a Python view class, e.g. 'my.package.views.MyView'.

member_name should be the appropriate singular version of the resource given your locale and used with members of the collection.

`collection_name` will be used to refer to the resource collection methods and should be a plural version of the member_name argument.

All keyword arguments are optional.

path_prefix Prepends the URL path for the Route with the path_prefix given. This is most useful for cases where you want to mix resources or relations between resources.

name_prefix Perpends the route names that are generated with the name_prefix given. Combined with the path_prefix option, it's easy to generate route names and paths that represent resources that are in relations.

Example:

```
config.add_resource('myproject.views:CategoryView', 'message', 'messages',
    path_prefix='/category/:category_id',
    name_prefix="category_")

# GET /category/7/messages/1
# has named route "category_message"
```

`yarhp.resource.`**`adjust_actions`**(*kwargs*)
Returns the adjusted list of actions based on `kwargs`.

`yarhp.resource.`**`default_model`**(*resource*)
Returns a dotted path to the default model class.

`yarhp.resource.`**`default_view`**(*resource*)
Returns the dotted path to the default view class.

`yarhp.resource.`**`get_root_resource`**(*config*)
Returns the root resource.

`yarhp.resource.`**`resource_factory`**(*request*)
Permits to associate a request with the concerned resource.

4.2 Views

Yarhp provides 3 view classes to help you to reduce lots of repetitive code in your views.

- BaseView
- SQLAView
- DynamoDBView

Last two are derived from BaseView.

class `yarhp.view.`**`BaseView`**(*context*, *request*)
Base class for yarhp views. This class holds some common objects used by the derived SQLAView and DynamoDBView classes. Those are:

`_default_actions`

`__model_class__`

`__validation_schema__`

`_default_actions` are those for which the derived view can omit the implementation for and it will be handled by the parent class.

```
class MyView(SQLAView):
    def index(self):
        ...
```

In this example we have MyView derived from SQLAView and it defines only the index action. The rest of the actions will by default handled by SQLAView. In case the particular view does not need a certain action, it should declare it so when adding the resource:

```
root.add('user', 'users', include=['index'])
```

Here the resource `user` defines only the `index` action.

`__model_class__` is a class object or dotted path to it which is associated with this view. By default `yarhp` expects the model to be defined in `your_project.model.<resource_member_name>.<Resource>View.py`

There are two ways to override this default:

```
root.add('user', 'users', model='path.to.my.model.ClassView')
root.add('whatever',model='yarhp.NoModel') #redefine it to NoModel
```

or

```
class MyView(BaseView):
    __model_class__ = ResourceView
```

`__validation_schema__` is a dict of dicts in the form of:

```
{field_name1:
    {'type':<python type>,
    'required':<bool>},
field_name2:{...}
    ...}
```

This is bare minimum that is checked by validators to make sure that request.params correspond to the model's requirements.

If you want to supply custom validation schema, you should override the `validate_schema` method of your view, which must return the dict in the format above.

`default_action`(*name*)
: `default_index`, `default_create`, etc must be defined in the child class. See SQLAView or DynamoDBView for examples. also `_default_actions` is by default empty. It must be defined by the derived view class.

`edit`(***kw*)
: by default `edit` is not allowed, unless its redefined in the child view

`new`(***kw*)
: by default `new` is not allowed, unless its redefined in the child view

`scan_action_decorators`()
: Iterates over actions and appends decorated wrappers for before and after calls to the corresponding before or after dict

`setup_default_wrappers`()
: Add all the default before and after calls. `before calls` are validate_types and validate_required which are called with the validation schema returned by `validation_schema` method. This allows any subclass to change validation schema.

after calls are those that modify the result returned by the actions: pager : paginates the results, if the result is a sequence. obj2dict: if result is an python object, it calls `to_dict` on it.

if its a list of objects, it enumerates and calls `to_dict` on each element.

wrap_in_dict: if action result is a list, it wraps it into a dict. add_pagination_links: if the result was paginated, it appends pagination links

validation_schema()
Override this method in your own view to provide custom validation schema

class yarhp.view.**NoModel**
No-op model class to use when defining a resource who doesnt have a model

class yarhp.view.**ViewMapper**(***kwargs*)
Custom mapper class for BaseView

class yarhp.view.**ViewNotImplemented**(**args*, ***kwargs*)
Use this view for the routes that raise HTTPMethodNotAllowed for any action

class yarhp.sqla.**SQLAView**(*context*, *request*)
Yarhp view used to serve SQLAlchemy objects. ‘‘SQLAView implements the following logic:

default behaviour for all actions validation agaisnt the sqla model's schema

default_create(***kw*)
default for create action. Raises HTTPConflict if there is an integrity error in DB. Returns HTTPCreated on success.

default_delete(***kw*)
default for delete action. Raises HTTPNotFound if the row is not found. Returns HTTPOk on success.

default_index(***kw*)
default for index action. Returns sqla query object which is passed to the pager to paginate the results.

default_show(***kw*)
default for show action. Raises HTTPNotFound if the row is not found. Returns the resouce as python dict.

default_update(***kw*)
default for update action. Raises HTTPNotFound if the resource being updated is not found. Returns HTTPOk on success.

validation_schema()
Returns the validation schema dict

yarhp.sqla.**get_model_schema**(*model*)
Returns a validation schema for the `model` the schema is a dict of dicts in the form of:

```
{field_name1:{
    'type':<python type>,
    'required':<bool>,
    'length':<int>}
field_name2:{
    ...
}
...

}
```

class yarhp.dynamo.**DynamoDBView**(*context*, *request*)
Simple REST view used to serve dymamodb-mapper objects at the top level.

For hash_key-indexed classes, subclass this and fill in the __model_class__ attribute with the class you want to serve. That's it! The default __model_class__ value is "your_project.model.resource_name.ResourceName"

Example usage:

```
class UserView(DynamoDBView):
    #__model_class__ = 'your_project.model.user.User'
    pass
```

default_create(***kw*)
POST /items: create a new item.

default_delete(***kw*)
DELETE /items/hash_key: delete a single item.

default_index(***kw*)
GET /items: list all items in the collection.

Warning: Uses the scan operation. Please avoid calling this on large collections.

default_show(***kw*)
GET /items/hash_key: get a single item.

default_update(***kw*)
PUT /items/hash_key: update an existing item.

4.3 Utilities

class yarhp.renderers.**YarhpJsonRendererFactory**(*info*)
Yarhp specific json renderer which will apply all after_calls(filters) to the result.

Default filters are: pager, obj2dict, add_parent_links, wrap_in_dict, add_pagination_links

yarhp.utils.**camel2snake**(*text*)
turn the camel case to snake: CamelSname -> camel_snake

yarhp.utils.**maybe_dotted**(*modul*, *throw=True*)
if `modul` is a dotted string pointing to the modul, imports and returns the modul object.

yarhp.utils.**snake2camel**(*text*)
turn the snake case to camel case: snake_camel -> SnakeCamel

4.4 Wrappers

yarhp.wrappers.**add_self_links**(***kwargs*)
Add links to the result dict to the resouce itself.

class yarhp.wrappers.**callable_base**(***kwargs*)
Base class for all before and after calls. `__eq__` method is overloaded in order to prevent duplicate callables of the same type. For example, you could have a before call `pager` which is called in the base class and also decorate the action with `paginate`. `__eq__` declares all same type callables to be the same.

class yarhp.wrappers.**check_ancestors**
Decorator which adds a wrapper to check the ancestors.

yarhp.wrappers.**obj2dict**(***kwargs*)
converts objects in *result* into dicts

class yarhp.wrappers.**pager**(***kwargs*)
kwargs are passed down to webhelpers.paginate.Page

some of the usefull kw args with defaults are: - items_per_page=20: how many items per page - item_count=None: total count if known, otherwise DB will be queried. - presliced_list=False: if collection was alredy sliced fitting into a page

For more see http://sluggo.scrapping.cc/python/WebHelpers/modules/paginate.html

class yarhp.wrappers.**paginate**(***kwargs*)
Decorator used to paginate the results. Its an after-call

yarhp.wrappers.**update_links**(*result*, *new_links*)
if result has "links", update it with new_links if result has "link", change it to "links" and add values from new_links if not empty if result has neither, create "link" or "links" depending len(new_links)

class yarhp.wrappers.**validate_ancestors**(***kwargs*)
Validates that all ancestor resources of the given resource exist

class yarhp.wrappers.**validate_base**(***kwargs*)
Base class for validation callables.

class yarhp.wrappers.**validate_required**(***kwargs*)
Validates that fields in `request.params` are present according to `kwargs` argument passed to ‘‘__call__. Raises ValidationError in case of the mismatch

class yarhp.wrappers.**validate_types**(***kwargs*)
Validates the field types in `request.params` match the types declared in `kwargs`. Raises ValidationError if there is mismatch.

class yarhp.wrappers.**validator**(***kwargs*)
Decorator that validates the type and required fields in request params against the supplied kwargs

```
class MyView():
    @validator(first_name={'type':int, 'required':True})
    def index(self):
        return response
```

yarhp.wrappers.**wrap_in_dict**(***kwargs*)
if result is a list then wrap it in the dict

class yarhp.wrappers.**wrap_me**(*before=None*, *after=None*)
Base class for decorators used to add before and after calls. The callables are appended to the `before` or `after` lists, which are in turn injected into the method object being decorated. Method is returned without any side effects.

4.5 Validators

Validators are implemented as wrappers that are before-calls. Essentially they ‘wrap’ your action (or any method really) and are called before the action is called.

There are two ways how validators can be involved:

- Decorator
- Subclassing from one of yarhp’s base view classes
- The Chuck Norris way

4.5.1 Decorator

```
@validator(name={'type': str, 'required': True},
           age={'type': int, 'required': True},
           birthdate={'type': date, 'required': False})
def create(self):
    pass
```

Keyword arguments passed to the validator are the validation schema that is used to check request.params validity.

4.5.2 Subclassing

Yarhp provides 2 base classes one could derive from to have automated validation in the views. `SQLAView` and `DynamoDBView`. The essential difference between those two is what kind of information about the underlaying schema yarhp is able to collect. In case of SQL for example it gets the type, required and also length of the field.

```
class MyView(SQLAView):
    def create(self):
        pass
```

In the example above the `create` action is automatically gets validated by yarhp: the type and the required field. So you dont have to do anything. The way it does this magic is pulling the metadata (schema) from SQLA and comparing with request.params. If you wish to override any default behavior, you could do:

```
class MyView(SQLAView):
    @validator(name={'type':str, required=False})
    def create(self):
        ...
```

Here we change the `required` to False, thus overriding the metadata coming from the DB schema where name was required. This is obviously not recommended and should be done with care, understanding the side effects. In this case it might raise DB error since name could be NULL, but DB requires it.

4.5.3 The Chuck Norris way

Now what if you write a common validator and you want to apply it to all of your actions? One way is to decorate them all. But another is "Chuck Norris way" !

Stay tuned for more..

4.6 The `add_resource` method

`config.`**`add_resource`**(*member_name*, *collection_name*, ***kwargs*)

defines a new resource, its named routes:

```
config.add_resource("user", "users")
```

view to find : UsersView

URL	route name	HTTP verb	action name
/users	users	GET	index
/users	users	POST	create
/users/:id	user	GET	show
/users/:id	user	PUT	update
/users/:id	user	DELETE	delete
/users/:id/edit	edit_user	GET	edit
/users/new	new_user	GET	new

These last two are non-standard CRUD actions.

```
config.add_resource("transaction", "transactions", parent="user")
```

view to find: UserTransactionsView model: user_transaction.UserTransaction

URL	route name	HTTP verb	action name
/users/:user_id/transactions	user_transactions	GET	index
/users/:user_id/transactions	user_transactions	POST	create
/users/:user_id/transactions/:id	user_transaction	GET	show
/users/:user_id/transactions/:id	user_transaction	PUT	update
/users/:user_id/transactions/:id	user_transaction	DELETE	delete
/users/1/transactions/edit	user_edit_transaction	GET	edit
/users/1/transactions/new	user_new_transaction	GET	new

These last two are non-standard CRUD actions.

singular resource:

```
config.add_resource('detail', parent='transaction')
```

view to find: UserTransactionDetailView

URL	route name	HTTP verb	action
/users/:user_id/transactions/:transaction_id/detail	user_transaction_detail	GET	show
/users/:user_id/transactions/:transaction_id/detail	user_transaction_detail	POST	create
/users/:user_id/transactions/:transaction_id/detail	user_transaction_detail	PUT	update
/users/:user_id/transactions/:transaction_id/detail	user_transaction_detail	DELETE	delete
/users/:user_id/transactions/:transaction_id/detail/edit	user_transaction_edit_detail	GET	edit
/users/:user_id/transactions/:transaction_id/detail/new	user_transaction_new_detail	GET	new

CHAPTER

FIVE

UPGRADE NOTES

5.1 From 0.1.227

- To use `SQLAView`, you need to add `SQLAlchemy == 0.7.2` to your dependencies and update your code to use `from yarhp.sqla import SQLAView`.
- To use `DynamoDBView`, add a dependency on `dynamodb-mapper >= 1.3.2` and change your code to `from yarhp.dynamo import DynamoDBView`.

CHAPTER

SIX

INDICES AND TABLES

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

PYTHON MODULE INDEX

y