prestans Documentation

Release 1.1

Eternity Technologies

Sep 27, 2017

Contents

prestans is a WSGI (PEP-333) complaint micro-framework that allows you to rapidly build quality REST services by introducing a pattern of models, parsers and handlers and in turn taking care of boilerplate code. prestans is aimed towards turly Ajax applications where the client side is completely written in JavaScript using toolkits like Google Closure.

prestans is currently hosted on Github and distributed under the terms defined by the New BSD license. A list of current downloads is available here. We highly recommend using PyPI to install prestans.

CHAPTER 1

Getting Started

prestans is a WSGI compliant REST server framework, best suited for use with applications where the entire interface is written using JavaScript (using frameworks like Google Closure) or a bespoke Mobile client. Although prestans is a standalone framework, it provides hooks (called Providers) to integrate with your application's authentication, caching and other such core services.

We have battle tested prestans under Apache (using mod_wsgi) and Google's AppEngine platform.

Code samples used throughout our documentation is available as a Google AppEngine project, we highly recommend you grab a copy so you can see how it all fits in.

Note: You will require a copy of Google's AppEngine Python SDK (v1.7.0+) to run the sample project.

Our philosophy is **"take as much or as little of the project as you like"**, prestans was designed ground up to sit nicely along side other Python frameworks. Needless to say that a dynamic language such as Python lends itself extremely well to writing frameworks such as prestans, and highly scaleable Web applications.

And incase you are still wondering prestans is a latin word meaning *"excellent, distinguished, imminent"*.

prestans is distributed under the terms and conditions of the New BSD license and is hosted on Github.

Features

- Validation or incoming and outgoing using strongly defined *Models*
- Pluggable architecture allowing prestans to plug into any authentication, caching and serialization requirements.
- A custom URL dispatcher that allows you to re-use handlers for multiple output formats.
- Data Adapters, that allows you to translate persistent objects into REST resources, with a single line of code.
- Validation of URL parameters using strong defined Parameter Sets.
- Dynamically filtering response fields when writing responses to reduce payload sizes.

- Auto generate API documentation using Blueprint (*Utilities*)

We also maintain a set of tools that leverages prestans's `Model` definition schema to generate boiler plate client side parsing of REST resources.

Installation

We recommend installing prestans via PyPI:

```
$ pip install prestans
```

this will build and install prestans for your default Python interpreter.

Alternatively you can download and build prestans using distutils:

```
$ tar -zxvf prestans-1.1.tgz
$ cd prestans-1.1
$ python setup.py install
```

Environments like Google's AppEngine require you to include custom packages as part of your source. Things to consider when distributing prestans with your application:

- Make sure you target a particular release of prestans, distributing our development branch is not recommended.
- If you prefer reference prestans as a Subversion external, ensure you use reference one of the `tags`, it is not recommended to reference `trunk`
- If your server environment has hard limits on number of files, consider using zipimport.

Software Requirements

The server side requires a WSGI compliant environment:

- Python 2.6+, *2.7 recommended*
- WSGI compliant server environment (Apache + mod_wsgi, or Google AppEngine, etc).
- Python Paste components (e.g WebOb)

Client side code is written for Google Closure.

We mostly test on latest releases of Ubuntu Server, and Google's AppEngine.

Concepts

Before you begin building REST services with prestans, it's important that you understand it's key concepts.

Serializers

Serializers are pluggable components that pack and unpack REST data in a seriazable format. For performance reasons most of them are wrappers on existing Python libraries, there's nothing stopping you from implementing one purely in Python.

You should never have to serialize or unserialize data when writing prestans apps, this is soley a job for the serializers. If serialization or unserialization fails, exceptions are raised and prestans sends out a canned error message to the client.

- JSON
- YAML

Note: We are working on XML support, and might settle for AtomPub.

REST Application

REST Application is our router, an instance of REST Application is responsible for mapping URLs to handlers. It's also responsible managing the API call lifecycle and humanising error messages for the client.

REST Application can not be used directly, you must use a sub class that's been paired with a Serializer. Out of the box prestans provides the following REST Application routers:

- `prestans.rest.JSONRESTApplication`
- `prestans.rest.YAMLRESTApplication`

It's possible to write your serializer and REST Application, you should only have to do this if you want to use a format not supported by prestans.

Handlers

Handlers are end points where an API request URL maps to. It's here that your business logic should live and how prestans knows where to hand over to your code. A handler maps to a URL pattern. Handlers should define an instance method for each HTTP method that you want to support.

Regex matched patterns are passed to your handler functions as parameters. Handlers can choose to use RequestParsers to validate incoming requests.

Models

Models are a set of rules that can be used by a prestans parser to validate the body of the request. Models are also use to validate and even auto generate responses from persistent data models.

prestans Models descriptions are quite similar to Django or Google AppEngine models.

Attributes can be of the following types, these are in accordance with popular serialization formats for REST APIs:

- String
- Integer
- Float
- Boolean
- Date Time
- Date
- Time
- Arrays

Each attribute provides a set rules configured by you, that prestans uses to validate incoming and outgoing data.

Request Parsers

Request Parsers allow you to define a set of rules that a request handler can use to validate incoming and outgoing data. Rules are define per HTTP method each handler corresponds supports and allows you to:

- validate sets of parmaeters in the URL
- the body of the request (for `POST`, `PUT`, `PATCH` and `DELETE` methods) by defining *Models*
- a response attribute list template which allows clients to request partially formed responses, the template directly corresponds to the definition of the handler's response format
- a definition of acceptable partially formed requests (based on models)

Complimentary to Request Parsers are `ParameterSet` which allow you defined patterns of acceptable groups of parameters in the URL and `AttributeFilter` which allow you to make exceptions to the rules defined by Models.

Data Adapters

Data Adapters are a set of extensions that allow you to quickly turn persistent data objects into instances of your REST models. prestans allows serialization of prestans managed Data Types, see *Models*. Data Adapters are backend specific (we currently support *SQLAlchemy <http://www.sqlalchemy.org>*_, AppEngine *NDB <https://developers.google.com/appengine/docs/python/ndb>*_).

These Adapters function map persistent models against prestans Models using a registry, allowing prestans to perform the translation to construct your REST handler's response.

Providers

prestans was designed ground up to live along side other Python Web development frameworks, and work under any WSGI compliant environment. This presents us with a challenge of fitting into services that may already be in use by your application or environment.

Providers are wrappers that present prestans with an standardised way to talk to these environment specific services. The provider implements specific code to return the status that prestans expects.

We provide extensive documentation on writing your own providers for environments we don't support out of the box.

These services include:

- Authentication
- Caching
- Throttling

CHAPTER 2

Routing & Handling Requests

First order of business is mapping URLs back to your code. prestans comes with an inbuilt router to help you achieve this, the router is paired with a serializer and manages the lifecycle of the REST API request. Currently prestans provides support for:

- JSON provided by `prestans.rest.JSONRESTApplication`
- YAML provided by `prestans.rest.YAMLRESTApplication`

We plan to support other formats as we need them. You can also write your own for formats you wish to support in your application. Read the section on *Serializers* to learn more about how serailziers work and how you can write your own. Our examples assume the use of JSON as the serialization format.

Each `RESTApplication` sub class paired with a serialzier is used to route URLs to handlers.

> **Warning:** Do not attempt to use an instance of `prestans.rest.RESTApplication` directly.

Regex & URL design primer

URL patterns are described using Regular expression, this section provides a quick reference to handy regex patterns for writing REST services. If you are fluent Regex speaker, feel free to skip this chapter.

Most URL patters either refer to collections or entities, consider the following URL scheme requirements:

- `/api/album/` - refers to a collection of type album
- `/api/album/{id}` - refers to a specific album entity

Notice no trailing slashes at the end of the entity URL. Collection URLs may or may not have a URL slash. The above patterns can would be represented in like Regex as:

- `/api/album/*` - For collection of albums
- `/api/album/([0-9]+)` - For a specific album

If you have entities that exclusively belong to a parent object, e.g. Albums have Tracks, we suggest prefixing their URLs with a parent entity id. This will ensure your handler has access to the {id} of the parent object, easing operations like:

- Does referenced parent object exists?
- When creating a new child object, which parent object would you like to add it to?
- Does the child belong to the intended parent (Works particularly well with ORM layers like SQLAlchemy)

A Regex example of these URL patterns would look like:

- `/api/album/([0-9]+)/track/*`
- `/api/album/([0-9]+)/track/([0-9]+)`

Defining your REST Application

You must use a `RESTApplication` subclass (one that's paired with a serializer) to create map URLs to REST Handlers. A REST application accepts the following optional parameters:

- `url_map` a list of regex to REST handler maps
- `application_name` optional name for your API, this will show up in the logs.
- `debug` set to `True` by default, turn this off in production. This status is made available as `self.request.debug`

url_map a non-optional parameter, requires pairs of URL patterns and REST Handler end points. The following example accepts two numeric IDs which are passed on to the handlers:

```
(r'/api/band/([0-9]+)/album/([0-9]+)/track', pdemo.rest.handlers.track.Collection)
```

prestans would map this URL to the `Collection` class defined in the package `pdemo.rest.handlers.track`, if you were to define a GET method which returned all the tracks for a band's album, it would look like:

```
class Collection(prestans.handlers.RESTRequestHandler):

    def get(self, band_id, album_id):
        ... return all tracks for band_id and album_id
```

If your handler does not support an particuar HTTP method for a URL, simply ignore implementing the appropriate method. An application API definition would be a collection of these URL to Handler pairs. The following is an extract from our demo application:

```
import prestans.rest

import pdemo.handlers
import pdemo.rest.handlers.album
import pdemo.rest.handlers.band
import pdemo.rest.handlers.track

api = prestans.rest.JSONRESTApplication(url_handler_map=[
    (r'/api/band', pdemo.rest.handlers.band.Collection),
    (r'/api/band/([0-9]+)', pdemo.rest.handlers.band.Entity),
    (r'/api/band/([0-9]+)/album', pdemo.rest.handlers.album.Collection),
    (r'/api/band/([0-9]+)/album/([0-9]+)/track', pdemo.rest.handlers.track.Collection)
], application_name="prestans-demo", debug=False)
```

Configuring your WSGI environment

Your WSGI environment has to be made aware of your declared prestans application. A Google AppEngine, app.yaml entry would look like:

```
- url: /api/.*
  script: entry.api
  # Where the package entry contains an attribute called api
```

a corresponding `entry.py` would look like:

```
#!/usr/bin/env/python

import prestans.rest
... along with other imports

api = prestans.rest.JSONRESTApplication(url_handler_map=[
    ... rules go here
], application_name="prestans-demo", debug=False)
```

Under Apache with mod_wsgi it a .wsgi file would look like (note that mod_wsgi requires the application attribute in the entry .wsgi script, best described in their Quick Configuration Guide):

```
#!/usr/bin/env/python

import prestans.rest
... along with other imports

application = prestans.rest.JSONRESTApplication(url_handler_map=[
    ... rules go here
], application_name="prestans-demo", debug=False)
```

API Request Lifecycle

From the outset prestans will handle all trivial cases of validation, non matching URLs, authentication and convey an appropriate error message to the client. It's important that you understand the life cycle of a prestans API request, you can use predefined Exceptions to automatically convey appropriate status codes to the client:

- URL Routers checks for a handler mapping
- Router checks to see if the handler implements the requested method (`GET`, `PUT`, `POST`, `PATCH`, `DELETE`)
- If required checks to see if the user is allowed to access
- Unserializes input from the client
- Runs validation on URL parameters, body models and makes them available via the request object
- Runs pre-hook methods for handlers (use this for establishing DB connections, environment setup)
- **Runs your handler implementation, where you place your API logic**
- Runs post-hook methods for handlers (use this to perform your tear down)
- Serializes your output

To put it in perspective of your handler code, prestans will execute the following:

- prestans runs checks through constraints defined by Parameters Sets and Models

- If your handler overrides the pre run hook, prestans runs `handler_will_run`
- prestans calls the method (i.e `get`, `post`, `put`, `patch`, `delete`), that corresponds to the requested HTTP verb.
- If your handler overrides the pre run hook, prestans runs `handler_did_run`

```
class Collection(prestans.handlers.RESTRequestHandler):

    def handler_will_run(self):
        ... do your setup stuff here

    def get(self, band_id, album_id):
        ... return all tracks for band_id and album_id

    def handler_did_run(self):
        ... do your tear down stuff here
```

Note: Consider defining a Base handler class in your application to perform common operations like establishing database connections in the pre and post hook methods.

Accessing incoming parameters

Handlers can accept input as parts of the URL, or the query string, or in the acceptable serialized format in the body of the request (not available for GET requests):

- Patterns matched using Regular Expression are passed via as part of the function call. They are positionally passed. Default behaviour passes all parameters as strings.
- Query parameters are available as key / value pairs, accessible in a handler as `self.request.get('param_name')`
- Serializers attempt to parse the request body and make the end results available at `self.request.parsed_body_model`

prestans defines a rich API to parse Query Strings, parts of the URL and the raw serialized body:

- Router that calls each handler passing parts of the URL extracted using `regex` to the appropriate handler method.
- Use of Parameter Sets to parse set of acceptable queries, so your handlder doesn't have to worry about if the parameters in the query string are acceptable.
- Use of *Models* and defined types to parse the body of requests, once again releaving you of checking the validity of the body.

This is a signature feature of our framework, and we have dedicated an entire chapter to discuss *Validating Requests*.

Writing Responses

Each handler method in your prestans REST application must return either a:

- Python serializable type, these include basic types are iterables
- Instances of `prestans.types.DataType` or subclasses

To write a response you must:

- Set a proper HTTP response code, by setting `self.response.status_code` to a constant in `prestans.rest.STATUS`
- Populating the body of the response

By default the response is set to a dictionary. Remember that at the end of the REST request lifecycle the response data is sent to the serializer. If your handler is sending arbitary data back to the client, it's suggested you use a key / value scheme to form your response.

`prestans.rest.Response` provides the `set_body_attribute` method, which takes a string key and serializable value:

```
import prestans.rest

class AlbumEntityHandler(prestans.handlers.RESTRequestHandler):

    def get(self, band_id, album_id):

        # Set the handler status code to 200
        self.response.http_status = prestans.rest.STATUS.OK

        # Add new attribute
        self.response.set_body_attribute("name", "Dark side of the moon")
```

prestans provides a well defined API to defined models for your REST API layer. These models are views on your persistent data and perform strong validation relfecting your business logic.

It's highly recommended to use *Models* to form strongly validated responses. In addition prestans provides a set of *Extensions* that ease translation of persistent models to prestans REST models.

Pre-defined exceptions

REST applications should use the breath of HTTP status codes to add meaning to the responses. prestans defines and handles a set of common expcetions that can be used by your application to send our standardised error responses. These `Exception` classes are paired with a status code and accept a string message as part of the constructor.

The string message is meant to make the error message more meaningful to the consumer of the API. Imagine the client wants to fetch an album for a band, it calls the album service with a `band_id` and an `album_id`, if the album is not found or does not belong to the band, the service should throw return the status code of `404` Not Found with enough information that the client can act upon it.

It's not important to echo back values they sent as part of the request, as they should already have access to the original request.

A snippet that outlines this example would look as follows:

```
import prestans.rest

class AlbumEntityHandler(prestans.handlers.RESTRequestHandler):

    def get(self, band_id, album_id):

        ... fetch the album that matches band_id and album_id

        # Raise an exception if the album was not found or didn't belong to the band
        if fetched_album is None or not fetched_album.band_id == int(band_id):
            raise prestans.rest.NotFoundException("Album")
```

```
        # Set the handler status code to 200
        self.response.http_status = prestans.rest.STATUS.OK

        ... and return the album serialized in the appropriate format
```

The following are a list of exceptions provided by prestans along with their paired status code and suggestions for use cases:

Class	HTTP status code	Use cases
prestans.rest.ServiceUnavailableException	503 (Service Unavailable)	The REST service or a related backend service is unavailable
prestans.rest.BadRequestException	400 (Bad Request)	Parameters sent as part of the request are not acceptable
prestans.rest.ConflictException	409 (Conflict)	The request conflicts the rules of the system, e.g duplicate users
prestans.rest.NotFoundException	404 (Not Found)	The requested entity does not exists
prestans.rest.UnauthorizedException	401 (Unauthorised)	The request entity can not be accessed by the current client
prestans.rest.ForbiddenException	403 (Forbidden)	The user is not allowed to access the particular resource

It it obviously possible to use the other error codes by manually setting the handler's resposne code and body message.

CHAPTER 3

Serializers

REST applications should be able to respond to clients in more than one format. Sound a theory but practically REST applications speak one major format and make exceptions to the rule, e.g an exportable report for download in CSV, and delivering the same data to a client in JSON for visualization. Serializers may also require the REST application to format their data in very different ways, e.g JSON would be a tree style response where as CSV would be linear.

Many frameworks expect the REST handler to choose how each response should be serialized (based on URL patters), we end up creating more work (not to mention large if else blocks) for the typical scenarios to accomodate the exceptions. pretans takes a slightly different appraoch to this problem and pairs serializers `RESTApplication` implementations. prestans REST handlers return either a prestans or Python type as their response and rely on the serializer to do their work, this enables you to reuse handlers with multiple serializers simply by referring to the same REST handler class from multiple end points.

Note: The gotcha is mixing and matching serializers to REST handlers that follow similar structures. E.g Tree structures opposed to linear structures.

To make exceptions to the rule it's recommended you create a separate URL scheme that fits the serialization format, mapped to appropriate and often exclusive REST handlers. Not all your business logic work should be done in your REST handlers, if they are reusable consider pushing the code into a common class or if you are using ORM layers consider distributing it based on the persistent models you are working with.

Serializers follow the prestans Provider (see *Getting Started*) paragradim. A serializer provides a wrapper on a pair of functions to write and read a data exchange format. It's recommended you use standard Python functions to perform the serialize and unserialize operations for performance. However if you need to you can write a pure Python implementation of your serializer.

Serializers are never used directly, it's always paired with a RESTApplication. We current provide support for the following serialization formats:

- JSON via `prestans.serializers.JSONSerializer`, paired with `prestans.rest.JSONRESTApplication`
- YAML via `prestans.serializers.YAMLSerializer`, paired with `prestans.rest.YAMLRESTApplication`

Writing your own serializer

The Provider paradigm provides a really simple way for you to write your own serializers and pair them with custom `RESTApplication`. This section discusses and demonstrates how simple it is to write your own serializers.

Serializers extend from `prestans.serializers.Serializer` and expect you to implement these three methods (our example discusses the JSON serializer we ship with prestans):

- `loads` is responsible for unserializing input data for your application, it's provided a Python `string` and is expected to return an python data type, usually an iterable.
- `dumps` provided a pure Python object that may be itterable and needs to be serialized. This function must return the serialized data back to the caller. prestans is responsible for writing the data back to the client
- `get_content_type` is expected to send back the mime type of the serialization format in use as a python `string`

prestans provides `prestans.serializers.UnserializationFailedException` and gracefully handles the client response, if the serialization process has problems it's recommended you raise this exception, with a meaningful message.

```
## @brief Provider for JSON based serializer
#
class JSONSerializer(Serializer):

    ## @brief loads method for JSON serializer
    #
    # @param self The object pointer
    # @param input_string
    #
    @classmethod
    def loads(self, input_string):
        import json
        parsed_json = None
        try:
            parsed_json = json.loads(input_string)
        except Exception, exp:
            raise UnserializationFailedException('Input Body data is not valid JSON')

        return parsed_json

    ## @brief dumps method for JSON serializer
    #
    # @param self The object pointer
    # @param serializable_object
    #
    @classmethod
    def dumps(self, serializable_object):
        import json
        return json.dumps(serializable_object)

    @classmethod
    def get_content_type(self):
        return 'application/json'
```

You can instantiate this class and test it works on the Python interactive interface. Once you are confident that your serializer wrapper works, proceed to pairing it with a RESTApplication that you create.

Pairing it with your REST Application

Nearly all of the `RESTApplication` logic and the prestans API lifecycle is encapsulate in `prestans.rest.RESTApplication` (this class is not paired with a serializer and never meant to be used directly). All custom implementations extend from `prestans.rest.RESTApplication` and expect them to construct a `Request` and `Response` to be used by the API lifecycle. These objects expect the serializer `class` they are meant to use.

REST Application implementations are required to override the following class methods:

- `make_request` expected to return an instance of `prestans.rest.Request`, and is passed in a reference to the WSGI environ
- `make_response` expected to return an instance of `prestans.rest.Response`

The following example is of the commonly used `JSONRESTApplication` taken from the `prestans.rest` package:

```
## @brief REST Application Gateway that speaks JSON
#
class JSONRESTApplication(RESTApplication):

    @classmethod
    def make_request(self, environ):
        rest_request = Request(environ,
                               serializer=prestans.serializers.JSONSerializer)
        return rest_request

    @classmethod
    def make_response(self):
        rest_response = Response(serializer=prestans.serializers.JSONSerializer)
        return rest_response
```

Note: While it's possible, it's considered to be against the prestans design principles to pair a serializer with multiple RESTApplication implementations.

CHAPTER 4

Validating Requests

prestans provides a well defined way of parsing out set of Parameters in the URL, and or ensure that the request body is properly formed and conforms to the rules defined by your application. Validation is one of the biggest time savers provided by prestans, you can reliably assume that if your handler method is being called, the data available to it is valid and conforms to the rules defined by you.

Each handler can be assigned an instance of a `RequestParser` subclass. Each HTTP method has a corresponding variable that expects an instance of `ParserRuleSet`.

Each `ParserRuleSet` can take one of four parameters, all parameters are optional:

- `parameter_sets`, this takes an Array of `ParameterSet` objects
- `body_template`, accepts a subclass of `DataCollection` (most commonly used are `Model` subclass or a prestans``Array``), this is used to validate the raw data sent in the body of an HTTP request. GET requests do not have a request body.
- `response_attribute_filter_template`, accepts a subclass of `DataCollection`, this assists in clients asking prestans to omit attributes in it's response.
- `request_attribute_filter`, accepts a subclass of `DataCollection`, this allows you to relax the rules for a Model for a particular handler and method. This is dicussed later in this section and proves extremely handy for PUT, PATCH requests.

```
class MyRequestParser(prestans.parsers.RequestParser):

    GET = prestans.parsers.ParserRuleSet(
        parameter_sets = [
            KeywordSearchParameterSet()
        ]
    )

    POST = prestans.parsers.ParserRuleSet(
        body_template=project.rest.models.MyModel(),
    )

    PUT = prestans.parsers.ParserRuleSet(
```

```
        body_template=project.rest.models.MyModel(),
    )
```

The above parser description, if assigned to a request handler would ensure that POST and PUT requests have a model in the body that matches the rules defined by `MyModel`. By default prestans is **unforgiving** in matching the body of a request, if the validation fails, prestans provides a meaningful error message to the client.

`response_attribute_filter_template` and `request_attribute_filter` can be used to make exceptions to the parsing and serializing rules, this is discussed in *Making exceptions to the rule*.

GET requests however will have option of providing a combination of name, value pairs that match any or none of these sets. Parameter Set matching is forgiving, your GET handler will be executed regardless of the result of the matching process. Parameter Sets work on the *first in, best dressed** rule, the first set that matches the request satisfies the validation process.

Note: All parameters accepted in RequestParsers are instances.

By default all validation rules are set to `None`, this tells prestans to ignore validation.

Attaching a request handler is as simple as assigning an instance of your `RequestParser` to your `RESTRequestHandler`:

```
class MyRESTRequestHandler(prestans.handlers.RESTRequestHandler):

    request_parser = MyRequestParser()
```

Following this prestans will execute the associated method to the request in your handler.

You can reuse your `RequestParser` across multiple handlers. If your handler does not implement a particular method, any defined rules for that method will be ignored by prestans.

Parameter Sets

REST handlers accept defined sets of URL parameters to allow the client to configure that way it responds. A popular use case is accepting values like *offset*, and *limit* which tells the handler the number of results to send down the wire.

Like `Models` prestans provides a well defined pattern to describe sets of Parameters (called ParameterSets), a set of which can be associated to a handler method. prestans evaluates parameters sent as part of the URL and attempts to match them to the provided templates. If a set of parameters match, they are made available as an instance of your ParameterSet subclass.

The request handler can access the parsed parameter set using `self.request.parameter_set`. By default this is set to None.

`ParameterSets` are matched on a first come best dressed principal. If you find that yourself defining sets that with one too many overlapping instances variables, you might want to re-think the design of your API call.

ParameterSets are defined by sub-classing `prestans.parsers.ParameterSet`. Since data provided in a URL are name value pairs, prestans only allows the use of basic types *(String, Integer, Float)* in ParameterSets.

Consider the following two ParameterSet definitions, one of them allows searching by Keywords, the other by an unread flag, both of them have the common parameters `offset` and `limit`.

```
class KeywordSearchParameterSet(prestans.parsers.ParameterSet):

    keyword = prestans.types.String(required=True)
```

```
    offset = prestans.types.Integer(required=False, default=0)
    limit = prestans.types.Integer(required=False, default=10)

class UnreadParameterSet(prestans.parsers.ParameterSet):

    unread = prestans.types.Boolean(required=True, default=False)
    offset = prestans.types.Integer(required=False, default=0)
    limit = prestans.types.Integer(required=False, default=10)
```

Parameter Sets are defined in a handler method's `ParserRuleSet` which in turn is associated to the handler. prestans follows this design principle throughout the framework to ensure you can reuse as many definitions as possible across handlers in your application.

```
class MyRequestParser(prestans.parsers.RequestParser):

    GET = prestans.parsers.ParserRuleSet(
        parameter_sets = [
            KeywordSearchParameterSet(),
            UnreadParameterSet()
        ]
    )
```

If the client was to call the following URL (assuming you are running a local development server):

```
http://localhost/api/myhandler?keyword=something
```

this would result in prestans assigning an instance of `KeywordSearchParameterSet` to the request handler's `self.request.parameter_set` attribute with values from the URL request parsed as the expected types, and likewise for the `UnreadParameterSet` if the parameter unread was passed. Since neither requests provide the `offset` or `limit` parameters the default values would be assigned to the attributes.

If the client provides values that violates the validation rules defined by the ParameterSet, prestans will reject that request.

All raw URL parameters can be access using the `set.request.get(key_name)` method. This would make available any parameter that do not belong to Parameter Sets.

Note: Raw URL parameters are always strings, you will have to explicitly convert types.

Request Body

Clients accessing REST APIs are expected to send messages in an agreed serialization format. prestans supports a range of serialization methods and provides infrastructure for you to write your own. JSON is probably the most popular serialization format for Ajax Web applications.

Note: Our examples assume JSON as the serialization format in use.

Your handler can define strict rules using prestans *Models* for this incoming data. `Models` is one of prestans's major feature and is discussed in great detail in it's own dedicated section. *Models* in a prestans application can be use to parse and serialize strongly validated data.

This section focuses on how you can use *Models* to parse incoming data. Assume you have a very simple Model defined as follows:

```
class Album(prestans.types.Model):

    title = prestans.types.String(required=True)
    release_year = prestans.types.Integer(required=True, min_value=1200, max_
↪value=2012)
    genre = prestans.types.String(required=True, choices=['rock', 'blues', 'pop'])
```

your REST handler can use a ParserRuleSet to indicate that it wishes to use this model as the template for data sent via the request body. Remember that the serializer chosen as part of your URL router definition is responsible for unserializing the input before Model it's parsed. If unserialization fails prestans will reject the request. An example could look like:

```
class MyRequestParser(prestans.parsers.RequestParser):

    POST = prestans.parsers.ParserRuleSet(
        body_template=Album(),
    )
```

If the body is successfully parsed, an instance of the Model class (with values parsed from the request) is assigned to `self.request.parsed_body`. On failing to parse the body prestans will reject the request providing the client meaningful information about the failure.

Making exceptions to the rule

Keeping Request and Response sizes as small as possible is crucial for performance in REST application. Model design should be strict, to ensure the quality of the data accepted and delivered by your REST services. We pointed out earlier, that by default validation for request and response bodies is absolutely unforgiving.

There are times that you need to make an exception to the rule, consider the following scenarios:

- You have full text description in a Model which you do not want included in the default response. The client has to exclusively request the full text description
- In reverse you might want a service that a client can send only the textual description for update.

One of the ways you can handle this is by writing numerous *Models* that each REST service uses, this works at first but for large applications you'll find yourself maintaining a one too many REST models. If you wish to use DataAdapters to build responses, you have to ensure that you register each defined model, and so on.

prestans offers an easy, clearly defined way per handler to make exceptions to the parsing rules while accepting requests or building responses. This is done assigning `AttributeFilter` instances to your `ParserRuleSet` or the handler's response.

`AttributeFilter` objects are a dynamically configurable sets of rules that can be used in prestans. Each attribute can either have a `Boolean` or an instance of `AttributeFilter` as it's value. Assigning instances of `AttributeFilter` to attributes is how you create a sub filter.

```
my_attr_filter = prestans.parsers.AttributeFilter()
my_attr_filter.name = True
my_attr_filter.phone = True
my_attr_filter.notes = False

# Sub filter
my_attr_filter.addresses = prestans.parsers.AttributeFilter()
my_attr_filter.addresses.street_name = True
my_attr_filter.addresses.city = True
my_attr_filter.addresses.state = False
```

In most cases AttributeFilters are reflection of a Model, so AttributeFilter can be created directly from a model. Optionally you can set the default state of each attribute, by default this is set to False, hence all attributes will be hidden unless specified otherwise.

```
# Typical usage
my_attr_filter = prestans.parsers.AttributeFilter.from_model(MyModel())

# Usage if you want to override the default value
my_attr_filter = prestans.parsers.AttributeFilter.from_model(MyModel(), default_
↪value=True)

# You can change the values after instantiation from a model
my_attr_filter.notes = False
```

Once you've created a filter, all you have to do is tell prestans to use it while evaluating inbound requests or building responses. Here's how.

Request Attribute Filter

When sending the data back to the server there are often cases (e.g updating description of a Product using the PATCH HTTP method) where you only need to send part of the REST model to the server.

prestans allows you to use REST models to validate incoming data, and validation is strict by default. Using filters, you can relax the validation rules when accepting requests.

These rules can be defined per HTTP method using a `RequestParser`.

```
my_attr_filter = prestans.parsers.AttributeFilter.from_model(MyModel(), default_
↪value=True)
my_attr_filter.notes = False

class MyRequestParser(prestans.parsers.RequestParser):

    GET = prestans.parsers.ParserRuleSet(
        parameter_sets = [
            KeywordSearchParameterSet(),
            request_attribute_filter=my_attr_filter
        ]
    )
```

Providing a Response Attribute Filter Template

prestans allows clients to make sensible requests to cut down latency. Consider two very different use cases for your API, a business to business client and your traditional Web or Mobile client. They both care for very different sorts of data, one willing to wait longer than the other, process more data than the later.

Clients can ask prestans to modify the response by providing a JSON serialized configuration that an `AttributeFilter`. This is provided as a parameter in the URL with the key `_response_attribute_list`. This key is reserved by prestans and cannot be used by your application.

```
{
  field_name0: true,
  field_name1: false,
  collection_name0: true,
```

```
  collection_name1: false,
  collection_name2: {
      sub_field_name0: true,
      sub_field_name1: false
  }
}
```

Your REST handler must provide a template prestans can match this input, if the JSON provided by the client has keys that are not present in the template, the request is rejected.

```
class MyRequestParser(prestans.parsers.RequestParser):

    GET = prestans.parsers.ParserRuleSet(
        parameter_sets = [
            KeywordSearchParameterSet(),
        ],
        response_attribute_filter_template=prestans.parsers.AttributeFilter.from_
↪model(MyModel())
    )
```

Your handler end point can get access to this `AttributeFilter` at `self.response.attribute_filter`. Responses are filtered while prestans is serializing output. Keys of the object being serialized must match the attribute filter's list. If you are serializing *Models* it's recommended you create your attribute filter using the model.

You can also manually set an `AttributeFilter`, here an example of an `AttributeFilter` that turns the `notes` field off set inside the handler.

```
def get(self):

    ... do other stuff here first to build response

    # Create your attribute filter from your model
    my_attr_filter = prestans.parsers.AttributeFilter.from_model(MyModel())
    my_attr_filter.notes = False

    # Before you return assign it to self.response.attribute_filter
    self.response.attribute_filter = my_attr_filter
```

If an attribute in the filter is set to be hidden, the prestans serializer omits the key in the JSON response. While parsing on the client side, you should check for the existence of the key.

CHAPTER 5

Models

Models allow you to define rules for your API's data. prestans uses these rules to ensure the integrity of the data exchanged between the client and the server. If you've' used the Django or Google AppEngine prestans models will look very familiar. prestans models are *not* persistent.

prestans types are one of the following:

- `prestans.types.DataType` all prestans types are a subclass of `DataType`, this is the most basic DataType in the prestans world.
- `prestans.types.DateStructure` are a subclass of `DataType` but represent complex types like Date Time.
- `prestans.types.DateCollection` are a subclass of `DataType` and represent collections like Arrays or Dictionaries (refered to as Models in the prestans world).

Each type has configurable properties that prestans uses to validate data. It's important to design your models with the strictest case in mind. Use request and response filters to relax the rules for specific cases, refer to our chapter on *Validating Requests*.

This chapter introduces you to writing Models and using it in various parts of your prestans application. It is possible to write custom `DataType`.

All prestans types are wrappers on Pythonic data types, that you get a chance to define strict rules for each attribute. These rules ensure that the data you exchange with a client is sane, ensures the integrity of your business logic and minimizes issues when persisting data. All of this happens even before your handler is even called.

Most importantly it cuts out the need for writing trivial boilerplate code to validate incoming and outgoing data. If your handler is called you can trust the data is sane and safe to use.

prestans types are divided into, *Basic Types*, and *Collections*, currently supported types are:

- `String`, wraps a Python `str`
- `Integer`, wraps a Python number
- `Float`, wraps a Python number
- `Boolean`, wraps a Python `bool`

- `DataURLFile`, supports uploading files via HTML5 FileReader API
- `DateTime`, wraps Python `datetime`
- `Array`, wraps Python `lists`
- `Model`, wraps Python `dict`

The second half of this chapter has a detailed reference of configuration parameters for each prestans `DataType`.

Writing Models

`Models` are defined by extending `prestans.types.Model`. Models contain attributes which either be a basic prestans type (a direct subclass of `prestans.types.DataType`) or a reference to an instance of another Model, or an `Array` of objects.

The REST standard talks about URLs refering to entities, this is often interpreted literally as REST API URLs refer to persistent models. Your REST API is the *business logic* layer of your Web client / server application. Providing direct access to persistently stored data through your REST API is simply replicating XML-RPC and not only is it bad design in the RESTful world but also extremely insecure.

RESTful APIs should serve back REST models. REST models are views of your data, that make sense as a response to the REST request. It's important to understand this so you can define your REST models to be as strict as possible. Like all good business logic layers, a RESTful API should never accept a request it can't comply with, this includes authority to perform the requested tasks on the data.

Consider a scenario where we are trying to model discographies, where a `Band` has `Albums`, has `Tracks`.

Depending on the implementation of this applicaiton it might be easier to send down `Tracks` when a client requests `Albums`, but might only want to send down `Albums` (without `Tracks`) when a list of `Bands` is requested.

Note: Read our section of Design Notes, to learn more about designing better REST APIs.

General convention for prestans apps is to keep all your REST models in a single package. To start creating models, simply define a class that extends from `prestans.types.Model`

```
... amogst other things
import prestans.types

class Track(prestans.types.Model):

    ... next read about attributes here
```

Defining Attributes

All atributes of a `Model` must be an instance of a `prestans.type`, Attributes can also be relationships to instances or collections of Models.

Attributes are defined at a class level, these are the rules used by prestans for each instance attributes of your `Model`. By default prestans is absolutely unforgiving and will ensure that each attribute satifies all it's conditions. Failure results in aborting the creation of an instance.

At the class level define attributes by instantiating prestans types with your rules, ensure they are as strict as possible, the more your define here the less you have to do in your handler. The objective is not to pass through data that your handler can't work with.

```
class Track(prestans.types.Model):

    id = prestans.types.Integer(required=False)
    name = prestans.types.String(required=True, min_length=1)
    duration = prestans.types.Float(required=True)
```

Our *Type Configuration Reference* guide documents in detail configuration validation options provided by each prestans `DataType`.

Note: prestans Models do not provide back references when defining relationships between Models (like many ORM layers), defining cross references in Models can cause an infinite recursion. REST models are views on your persistent data, in most cases cross references might mean re-thinking your API design. You can also use DataAdapters to prevent an infinite recursion.

To One Relationship

One to One relationships are defined assigning an instance of an existing `Model` to an attribute of another.

Validation rules accepted as instantiation values are for the attribute of the container `Model`, they are evaluated the same way as basic prestans `DataTypes`.

```
class Band(prestans.types.Model):

    ... other attributes ...

    created_by = UserProfile(required=True)
```

On success the attribute will refer to an instance of the child `Model`. Failure to validate attributes of the children result in the failure of the parent `Model`.

To Many Relationship (using Arrays)

prestans provides `prestans.types.Array` to provide lists of objects. Because REST end points refer to Entities, Collections in REST responses or requests must have elements of the same data type.

You must provide an instance prestans DataType (e.g Array of Strings for tagging) or defined Model as the `element_template` property of an `Array`. Each instance in the `Array` must comply with the rules defined by the template. Failure to validate any instance in the `Array`, results as a failure to validate the entire `Array`.

```
class Album(prestans.types.Model):

    ... other attributes ...

    tracks = prestans.types.Array(element_template=Track(), min_length=1)
```

Arrays of Models are validated using the rules defined by each attribute. If you are creating an Array of a basic prestans type, the validation rules are defined in the instance provided as the `element_template`:

```
class Album(prestans.types.Model):

    ... other attributes ...

    tags = prestans.types.Array(element_template=prestans.types.String(min_length=1,␣
↪max_length=20))
```

Self References

Self references in prestans Model definition are the same as self referencing Python objects.

```
... amogst other things
import prestans.types

# Define the Model first
class Genre(prestans.types.Model):

    id = prestans.types.Integer(required=False)
    name = prestans.types.String(required=True, min_length=1)
    year_started = prestans.types.Float(required=True)

    ... and other attributes

# Once defined above you can self refer
Genre.parent = Genre(required=False)
```

Use arrays to make a list:

```
Genre.sub_genres = prestans.types.Array(element_template=Genre())
```

Special Types

Apart the usual suspects (`String`, `Integer`, `Float`, `Boolean`) prestans also provides a few complex `DataTypes`. These are wrappers on data types that have extensive libraries both on browsers and the Python runtime, but are serialized as strings or numbers.

DateTime

DateTime wraps around python `datetime`, serialization formats like JSON serialize dates as strings, there are various standard formats for serializing dates as Strings, by default prestans `DateTime` uses **RFC 822** expressed as `%Y-%m-%d %H:%M:%S` format string in Python. This is because Google Closure's Date API conveniently provides `goog.date.fromIsoString` to parse these Strings.

To use another format string, override the `format` parameter when defining `DateTime` attributes.

```
class Album(prestans.types.Model):

    last_updated =  prestans.types.DateTime(default=prestans.types.CONSTANT.DATETIME_
↪NOW)
```

Assigning python `datetime` instances as the default value for prestans `DateTime` attributes works on the server, our problem lies in auto-generating client side stub code. The use of the constant `prestans.types.CONSTANT.DATETIME_NOW` instruct prestans to handle this properly.

DataURLFile

HTML5's FileReader API is well supported by all modern browsers. Traditionally Web applications used multi part mime messages to upload files in a POST request. The `FileReader` API allows JavaScript to get access to local files and makes for a much nicer solution for file uploads via a REST API.

The FileReader API provides `FileReader.readAsDataURL` which reads the file using as Data URL Scheme, which essentially is a Base64 encoded file with meta information.

```
<!-- Use of data URL to embed an image -->
<img src="
AAAFCAYAAACNbyblAAAAHElEQVQI12P4//8/w38GIAXDIBKE0DHxgljNBAAO
9TXL0Y4OHwAAAABJRU5ErkJggg==" alt="Red dot"/>
<!-- Courtesy Wikipedia -->
```

`prestans.types.DataURLFile` decodes the file Data URL Scheme encoded file and give access to the content and meta information. If you are using a traditional Web server like Apache, `DataURLFile` provides a `save` method to write the uploaded contents out, if you are on a Cloud infrastructure e.g Google AppEngine, you can use the `file_contents` property to get the decoded file.

DataURLFile can restrict uploads based on mime types.

```
class Album(prestans.types.Model):

    ... other attributes
    album_art =  prestans.types.DataURLFile(allowed_mime_types=['image/jpeg', 'image/
↪png', 'image/gif'])
```

Using Models to write Responses

REST APIs should validate any data being sent back down to clients. Your application's persistent layer can't always guarantee that stored data meets your business logic rules.

Models are a great way of constructing sound responses. They are also serializable by prestans. Your handlers can simply pass a collection (using Arrays) or instance of a Model and prestans will serialize the results.

```
class AlbumEntityHandler(prestans.handlers.RESTRequestHandler):

    def get(self, band_id, album_id):

        ... environment specific code to get an Album for the Band

        album = pdemo.rest.models.Album()
        album.name = persistent_album_object.name

        ... and so on until you copy all the values across

        self.response.http_status = prestans.rest.STATUS.OK
        self.response.body = album
```

From the above example it's clear that code to convert persistent objects into REST models becomes repetitive, and as a result error prone. prestans provides `DataAdapters`, that automate the conversion of persistent models to REST models. Read about it in the *Extensions* chapter.

If you use Google's Closure Library for client side development, we provide a complete client side implementation of our types library to create and parse, requests and responses. Details available in the *Google Closure Library*

Extensions (incomplete) section.

Type Configuration Reference

Basic prestans types extend from `prestans.types.DataType`, these are the building blocks of all data represented in systems, e.g Strings, Numbers, Booleans, Date and Times.

Collections contain a series of attributes of both Basic and Collection types.

String

Strings are wrappers on Pythonic strings, the rules allow pattern matching and validation.

Note: Extends `prestans.types.DataType`

- `required` flags if this is a mandatory field, accepts `True` or `False` and is set to `True` by default
- `default` specifies the value to be assigned to the attribute if one isn't provided on instantiation, this must be a String.
- `min_length` the minimum acceptable length of the String, if using the `default` parameter ensure it respects the length.
- `max_length` the maximum acceptable length of the String, if using the `default` parameter ensure it respects the length.
- `format` a regular expression for custom validation of the String.
- `choices` a list of Strings that are acceptable values for the attribute.
- `utf_encoding` set to `utf-8` by default is the confiurable UTF encoding setting for the String.

Integer

Integers are wrappers on Python numbers, limited to Integers. We distinguish between Integers and Floats because of formatting requirements.

Note: Extends `prestans.types.DataType`

- `required` flags if this is a mandatory field, accepts `True` or `False` and is set to `True` by default
- `default` specifies the value to be assigned to the attribute if one isn't provided on instantiation, this must be a Integer.
- `minimum` the minimum acceptable value for the Integer, if using default ensure it's greater or equal to than the minimum.
- `maximum` the maximum acceptable value for the Integer, if using default ensure it's less or equal to than the maximum.
- `choices` a list of choices that the Integer value can be set to, if using default ensure the value is set to of the choices.

Float

Floats are wrappers on Python numbers, expanded to Floats.

Note: Extends `prestans.types.DataType`

- `required` flags if this is a mandatory field, accepts `True` or `False` and is set to `True` by default
- `default` specifies the value to be assigned to the attribute if one isn't provided on instantiation, this must be a Float.
- `minimum` the minimum acceptable value for the Float, if using default ensure it's greater or equal to than the minimum.
- `maximum` the maximum acceptable value for the Float, if using default ensure it's less or equal to than the maximum.
- `choices` a list of choices that the Float value can be set to, if using default ensure the value is set to of the choices.

Boolean

Booleans are wrappers on Python `bools`.

Note: Extends `prestans.types.DataType`

- `required` flags if this is a mandatory field, accepts `True` or `False` and is set to `True` by default
- `default` specifies the value to be assigned to the attribute if one isn't provided on instantiation, this must be a Boolean.

DataURLFile

Supports uploading files using the HTML5 FileReader API.

Note: Extends `prestans.types.DataType`

- `required` flags if this is a mandatory field, accepts `True` or `False` and is set to `True` by default
- `allowed_mime_types`

DateTime

Date Time is a complex structure that parses strings to Python `datetime` and vice versa. Default string format is `%Y-%m-%d %H:%M:%S` to assist with parsing on the client side using Google Closure Library provided DateTime.

Note: Extends `prestans.types.DataStructure`

- `required` flags if this is a mandatory field, accepts `True` or `False` and is set to `True` by default

- `default` specifies the value to be assigned to the attribute if one isn't provided on instantiation, this must be a date. prestans provides a constans `prestans.types.CONSTRANT.DATETIME_NOW` if you want to use the date / time of execusion.
- `format` default format `%Y-%m-%d %H:%M:%S`

Collections

Collections are formalised representations to complex itterable data structures. prestans provides two Collections, Arrays and Models (dictionaries).

Array

Arrays are collections of any prestans type. To ensure the integrity of RESTful responses, `Array` elements must always be of the same kind, this is defined by specifying an `element_template`. prestans Arrays are itterable.

Note: Extends `prestans.types.DataCollection`

- `required` flags if this is a mandatory field, accepts `True` or `False` and is set to `True` by default
- `default` a default object of type `prestans.types.Array` to be used if a value is not provided
- `element_template` a instance of a `prestans.types` subclass that's use to validate each element. prestans does not allow arrays of mixed types because it does not form valid URL responses.
- `min_length` minimum length of an array, if using default it must conform to this constraint
- `max_length` maximum length of an array,

Model

Models are wrapper on dictionaries, it provides a list of key, value pairs formalised as a Python `class` made up of any number of prestans `DataType` attributes. Models can have instances of other models or Arrays of Basic or Complex prestans types.

Note: Extends `prestans.types.DataCollection`

- `required` flags if this is a mandatory field, accepts `True` or `False` and is set to `True` by default
- `default` a default model instance, this is useful when defining relationships

The following is a parallel argument:

- `**kwargs` a set of key value arguments, each one of these must be an acceptable value for instance variables, all defined validation rules apply.

CHAPTER 6

Securing your API

Each project has very different requirements for authentication and more importantly each developer likes to implement each scenario differently. The Python Web world is a world of micro frameworks that work toegether in harmony. prestans does not implement any authentication mechanisms, in turn it implements a set of patterns called `Providers` (refer to our *Getting Started* chapter) that assist in making prestans application respect your application's chosen authentication method.

What this means is prestans provides you the opportunity to tell it what your application considers, authenticated and authorized. Your prestans REST handlers use a set of predefine decorators to communicate with your prestans application's authentication provider to secure REST end points.

Security typically has two parts to the problem, Authentication to see if the user is allowed in the system at all, and Authorization to see if the user is allowed to access a particular resource. If you are checking for Authority, prestans assumes that the user is required to be authenticated.

Security is defined per HTTP method of a handler (e.g a User can read a list of resources, but is not allowed to update them), prestans provides a set of decorators that your REST handler methods use to express their authentication/authorization requirements.

Fitting into your environment

An API end point should respond if the user is unauthenticated, obviously with a message to tell them they are unauthenticated. API's are client agnostic, so It's nearly "*never*"" the API end point's responsibility to send the user to a login page. If a uesr is accessing a resource they are not meant to be, prestans will send a properly formed message as the response.

`prestans.auth` provides a stub for the `AuthContextProvider`, this class is never meant to be used directly, your application is expected to provide a `class` that extends from `AuthContextProvider`.

`AuthContextProvider` defines the following method stubs

```
class AuthContextProvider:

    def is_authenticated_user(self, handler_reference):
```

```
        raise Exception("Direct use of AuthContextProvider not allowed")

    def get_current_user(self):
        raise Exception("Direct use of AuthContextProvider not allowed")

    def current_user_has_role(self, role_name):
        raise Exception("Direct use of AuthContextProvider not allowed")
```

Let's discuss these in order of relevance:

- `is_authenticated_user` must return `True` or `False` to indicate if a user is current logged in, the function is additionally provided a reference to the hander. Your application has the opportunity to use any supporting libraries to determine if the user is logged in and return a response.
- `get_current_user` should return a reference to the `user` object for your application. This can be a persistent object, or user identifier, whatever your application would find most useful when persistenting data.
- `current_user_has_role` is provided a set of rolenames that the handle is allowed to use, role_name can be a refernce to a list of strings, constants whatever your app deems relevant. This method will only run after prestans has checked that the user is authenticated. Obviously you can use `self.get_current_user` to get a reference tot he currently logged in user.

Writing your own provider

Writing your own `AuthContextProvider` comprises of overriding the three methods discussed in the previous section. The following example demonstates the use of Beaker Sessions to validate if the user is logged in. Notice that that most of the code is referencee from another package that provides authentication information to pages rendered by handlers.

The `__init__` method is not used by the parent class, so if you need to pass extra references to objects that you need to use to perform the authentication here's the place to do it.

```
class MyAuthContextProvider(prestans.auth.AuthContextProvider):

    ## @brief custom constructor that takes in a reference to the beaker environment␣
↪var
    #
    def __init__(self, environ):
        self._environ = environ

    ## @brief checks to see if a Beaker session is set or not
    #
    # Beaker session reference is passed into the constructor and made available as
    # an instance variable to the auth context provider
    #
    def is_authenticated_user(self):
        return self._environ and self._environ.get(myapp.auth.SESSION_KEY)

    ## @brief returns a user object from myapp.models
    def get_current_user(self):
        remote_user = self._environ.get(myapp.auth.SESSION_KEY)
        return myapp.auth.get_userprofile_by_username(remote_user)
```

Working with Google AppEngine

prestans ships with an inbuilt provider for Google AppEngine. AppEngine is a WSGI environment and has a very fixed authentication lifecycle encapsulated by `prestans.ext.appengine.AppEngineAuthContextProvider`. The AppEngine AuthContextProvider implements support for OAuth and Google account authentication.

Obviously this does not implement the `current_user_has_role`. If you wish to support role based authorization you must extend this class and implement this function.

Attaching AuthContextProvider to Handlers

Like all things prestans, attaching a auth context provider to a handler is as simple as assigning an instance of your `AuthContextProvider` to your `RESTRequestHandler`'s auth_context property:

```
class MyHandler(prestans.handlers.RESTRequestHandler):

    auth_context = myapp.auth.MyAuthContextProvider()
```

This tells your handler which `AuthContextProvider` to use. Remember that authentication configuration is per HTTP method supported by your request handler:

- If your handler method just wants to ensure that a user is logged in, all you need to do is decorate your HTTP method with `@prestans.auth.login_required`.
- If your handler method wants to test final grained roles use the `@prestans.auth.role_required` decorator. This implies that a user is already logged in.

The following example allows any logged in user to get resources, users with role authors to create and update resources, but only users with role admin to delete resources.

```
class MyRESTHandler(prestans.handlers.RESTRequestHandler):

    auth_context = myapp.auth.MyAuthContextProvider()

    @prestans.auth.login_required
    def get(self):
        .... do what you need to here

    @prestans.auth.role_required(role_name=['authors'])
    def post(self):
        .... do what you need to here

    @prestans.auth.role_required(role_name=['authors'])
    def put(self):
        .... do what you need to here

    @prestans.auth.role_required(role_name=['admin'])
    def delete(self):
        .... do what you need to here
```

CHAPTER 7

Extensions

prestans extensions are purpose built extensions that act as bridges between prestans elements and for argument sake persistent backends. Etensions build on the core prestans framework and are heavily dependent on environment specific packages.

Data Adapters

Our *Models* chapter discusses in detail, the use of Models to validate and build responses returned by handlers. Models can use AttributeFilters to make exceptions to the validation rules set out by your Model's original definition.

We identified the scenario and data validation benefits of converting persistently stored data to REST models, and in turn identified that it's a code laborious process.

DataAdapters fills that gap in prestans, it automates the process of converting persistent models into REST models by providing:

- A static registry `prestans.ext.data.adapters.registry`, that maps persistent models to REST models
- QueryResultsIterator, that iterates through collections of persistent results and turns them into REST models. QueryResultsIterator is specific to backends and uses the registry to determine relationships between persistent and REST models.

Note: You can map multiple REST models to the same persistent model.

For our sample code assume that rest models live in the `pdemo.rest.models` and the persistent models live in `pdemo.models`, and is written for AppEngine.

prestans supports `SQLAlchemy` and AppEngine's `ndb` and `datastore`. You can write your DataAdapter to support custom backends.

Pairing REST models to persistent models

The registry allows you to provide a map acceptable translations between persistent and REST models. If a persistent model maps to more than one REST model, DataAdapters try and make the sensible choice unless you explicitly provide the REST model you wish to adapt the data to.

General practice is to register the persistent models along side their definition. An excerpt from `pdemo.models`.

Registering the persistent model is as easy as calling the `register_adapter` method on `prestans.ext.data.adapters.registry`, and providing it an instance of the appropriate `ModelAdapter`.

Consider a REST model defined `prestans.rest.models`:

```
import prestans.types

class Band(prestans.types.Model):

    id = prestans.types.Integer(required=False)
    name = prestans.types.String(required=True, max_length=30)

    albums = prestans.types.Array(element_template=Album(), required=False)
```

And then in your persistent model package, use `prestans.ext.data.adapters.registry` to join the dots. Ensure that all children models are present in the registry (e.g Album):

```
from google.appengine.ext import ndb
from google.appengine.api import users

import prestans.ext.data.adapters
import prestans.ext.data.adapters.ndb

class Band(ndb.Model):

    name = ndb.StringProperty()

    created = ndb.DateTimeProperty(auto_now_add=True)
    last_updated = ndb.DateTimeProperty(auto_now=True)

    @property
    def albums(self):
        return Album.query(ancestor=self.key).order(Album.year)

    @property
    def id(self):
        return self.key.id()

# Register the persistent model to adapt to the Band rest model, also
# ensure that Album is registered for the children models to adapt
prestans.ext.data.adapters.registry.register_adapter(
    prestans.ext.data.adapters.ndb.ModelAdapter(
        rest_model_class=pdemo.rest.models.Band,
        persistent_model_class=Band
    )
)
```

Adapting Models

Once your models have been declared in the adapter registry, your REST handler:

- Query the data that your handler is expected to return
- Set the HTTP status code
- Use the appropriate QueryResultIterator to construct your REST adapted models
- Assign the returned collection to `self.response.body`

```
from google.appengine.ext import ndb

import pdemo.models
import pdemo.rest.handlers
import pdemo.rest.models

import prestans.ext.data.adapters.ndb
import prestans.handlers
import prestans.parsers
import prestans.rest

class CollectionRequestParser(prestans.parsers.RequestParser):

    GET = prestans.parsers.ParserRuleSet(
        response_attribute_filter_template=prestans.parsers.AttributeFilter.from_
↪model(pdemo.rest.models.Band())
    )

class BandCollection(pdemo.rest.handlers.Base):

    request_parser = CollectionRequestParser()

    def get(self):

        bands = pdemo.models.Band().query()

        self.response.http_status = prestans.rest.STATUS.OK
        self.response.body = prestans.ext.data.adapters.ndb.QueryResultIterator(
            collection=bands,
            target_rest_instance=pdemo.rest.models.Band
        )
```

If you are using AttributeFilters (read our chapter on *Validating Requests* to learn how you can make exceptions to Model validation rules) you can pass them onto the QueryResultsIterator which results in the QueryResultsIterator skipping accessing that property all together significantly reducing the load on the Data Layer:

```
class BandCollection(pdemo.rest.handlers.Base):

    request_parser = CollectionRequestParser()

    def get(self):

        bands = pdemo.models.Band().query()

        self.response.http_status = prestans.rest.STATUS.OK
        self.response.body = prestans.ext.data.adapters.ndb.QueryResultIterator(
            collection=bands,
            target_rest_instance=pdemo.rest.models.Band,
            attribute_filter = self.response.attribute_filter
        )
```

CHAPTER 8

Utilities

This section covers a set of utilities shipped with prestans. These features are complimentary to API design but are not required for your application to function.

prestans.util.signature

`@prestans.util.signature` is a decorator that automatically castes each incoming parameter per handler to their right type. `prestans.util.signature` takes positional arguments of Python types that must match your handler signature.

```
@prestans.util.signature(self, int, int)
def get(self, band_id, album_id):
    ... do what you need here
```

We decided to take this approach to variable casting because it's a per handler and environment specific decision. Our solution is designed to assist not assert.

Note: This function is based upon Andrew Lee's blog post Faux function type signatures in Python

API Blueprint

prestans ships with a special built-in handler base that can produce a blueprint for your prestans API. Documentation is a developer's nightmare, mostly because it's difficult to think back and encapsulate all parts of your design (not to mention that it's not the most exciting part of the job). However it's one of the most important ingredients for success. Consumers are most interested in endpoints provided by your API and what each endpoint expects, e.g. data payloads, URL parameters, etc.

prestans ships with an inbuilt handler that inspects all of your application's registered handlers, models, parameter sets, attribute filters and makes available a description in your chosen serialization format.

Presumeably you might to expose the blueprint to the public, and so you can leverage from all the other features of prestans (e.g authentication, throttling, caching) prestans requires you to implement a simple handler that:

- extends from `prestans.handlers.BlueprintHandler`
- implements the method to respond to an `GET` request (all other requests to blueprint handlers are supressed)
- calls the `create_blueprint` method which returns a serializable dictionary
- add the returned dictionary to the response

A sample implementation would look something like this:

```
import prestans.handlers

class APIBlueprintHandler(prestans.handlers.BlueprintHandler):

    def get(self):

        blueprint = self.create_blueprint()

        self.response.http_status = prestans.rest.STATUS.OK
        self.response.set_body_attribute("api", blueprint)
```

then map it as you would any other handler to a URL that you see fit, remember that this handler will be ignored from the API blueprint:

```
import prestans.rest

import pdemo.handlers
import pdemo.rest.handlers.album
import pdemo.rest.handlers.band
import pdemo.rest.handlers.track

api = prestans.rest.JSONRESTApplication(url_handler_map=[

    # Add the blueprint handler to /api/blueprint
    (r'/api/blueprint', pdemo.rest.handlers.APIBlueprintHandler),

    # Application handlers
    (r'/api/band', pdemo.rest.handlers.band.Collection),
    (r'/api/band/([0-9]+)', pdemo.rest.handlers.band.Entity),
    (r'/api/band/([0-9]+)/album', pdemo.rest.handlers.album.Collection),
    (r'/api/band/([0-9]+)/album/([0-9]+)/track', pdemo.rest.handlers.track.Collection)

], application_name="prestans-demo", debug=False)
```

Warning: If you are planning to make blueprints available on your live service, we seriously recommend using a caching mechanism. Blueprints introspect every handler, parameter set, model to produce it's output and could prove to be computationally expensive.

Each auto generated blueprint:

- Is grouped by Python package that contains your handlers, each module is the key in a dictionary.
- Uses Python docstrings (PEP 257) to fetch descriptions on each handler class and method.
- Includes information on supported handler methods, Parameter Sets, Models, Attribute Filters, constraints of each attribute.

CHAPTER 9

Thoughts on API design

prestans was a result of our careful study into the REST standards, frameworks and appraoches that were popular at the time. The following are a few useful lessons we've learnt along the way. Also refer to our extensive list of extremely useful *Reference Material* we found on the Web.

REST resources are *not* persistent models

Reading around the Web, it seems that traditional client/server programmers somehow concluded that REST is basically a HTTP replacement for XML-RPC, SOAP lovers might have had something to do with this as well. This school of thought lead developers to design of REST APIs (like XML-RPC) as a gateway to each persistent object on the server and making the client responsible for dealing with data relationships, integrity etc. Many frameworks took these ideas and implemented pass through REST gateways to RDBMS backends.

This is completely incorrect.

Data presented to clients talking to REST services is very different to the way data is stored, this is particularly true when you are using NoSQL style databases. **Think of REST resources are views of the stored data**. The job of your server side code to do as much meaningful work as possible with the data and present it to the client in form that is immediately useful.

Again, *REST resources are useful views of your persistent data*.

Collections & Entities

URLs should refer to resource or a kind of data that your client can work with. Resources are *not* persistent entities rather a view of them. There generally are two patterns for each resource that you need to address. Consider the following URL patterns

- `/api/product`
- `/api/product/{id}`

Both deal with a resource called product. The first URL deals with collections, so get all products (GET), or create a new product (POST) are the requests it should respond to.

The second would deal with a specific entity of that kind of resource. So get a product (GET), Update a product (PUT, PATCH), or delete a product (DELETE) are the requests it should respond to.

As a design principle we recommend you handle collections and entities in two seaprate handlers.

Response Size does matter

Database, Web Servers, prestans your handlers, servers are generally pretty quick (if you have written most things well). Network latency is still a killer for REST applications.

A general view is that latency is generally caused by services on the server side running slow, althought can be the case, one thing that slips out of the radar is the size of the response that you send down to the client.

One of our latest applications was sending down large amounts of textual data, was never a problem when were building the application but as it was put to the production the size of stored text went out of hand, pushing the size of a 100 record response to 2.5 Megabytes. It wasn't MySQL, wasn't our code, prestans, Apache, or the server it was purely the size of response.

So when writing REST services, **Size really does matter!**

CHAPTER 10

Google Closure Library Extensions (incomplete)

Google Closure is a set of JavaScript tools, that Google uses to build many of their core products. It provides:

- A JavaScript Optimizer to build a distributable version of your application
- A comprehensive JavaScript library
- A templating system for JavaScript
- A JavaScript style checker and style fixer
- An enhanced stylesheet language that works with the optimizer to minifiy CSS.

Each one of these components is agnostic of the other. Closure is at the heart of building products with prestans.

Google Closure is unlike other JavaScript frameworks (e.g jQuery). An extremely central part of Closure tools is it's compiler (which is not just a minifier), the Closure development philosophy is to use the abstractions and components made available by Closure library and allow the compiler to optimise it for production.

Note: It's assumed that you are familiar with developing applications with Google Closure tools.

prestans provides a number of extensions to Closure Library, that ease and automate building rich JavaScript clients that consume your prestans API. Our current line up includes:

- REST Client, provides a pattern to create Xhr requests, manages the life cycle and parsers responses, also supports Attribute Fitlers.
- Types API, a client side replica of the prestans server types package assisting with parsing responses.
- Code generation tools to quickly produce client side stubs from your REST application models.

It's expected that you will use the Google Closure dependency manager to load the prestans namespaces.

Installation

Our client library follows the same development philosophy as Google Closure library, although we make available downloadable versions of the client library it's highly recommended that you reference our repository as an external source.

This allows you to keep up to date with our code base and benefit from the latest patches when you next compile.

Closure library does the same, and we ensure that we are leveraging off their latest developments.

Unit testing

```
/path/to/depswriter.py --root_with_prefix=". ../prestans" > deps.js
```

To run these unit tests you will need to start Google Chrome with `--allow-file-access-from-files` parameter. Example on Mac OS X:

```
spock:docs devraj$ /Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome --
↪allow-file-access-from-files
```

Extending JavaScript namespaces

Models ensure the validity of data sent to and from the server. The application client should be as responsible validate data on the client side, ensuring that you never send an invalid request or you never accept an invalid response. Discussed later in this chapter are tools provided by prestans that auto generate Closure library compatible versions of your server side Models and Attribute Filters, needless to say our JSON client works seamlessly with these auto generated Models and Filters.

Auto generated code is accompanied with the curse of loosing local modifications (e.g adding a helper method or computed property) when you next run the auto generate process.

Consider the following scenario, prestans auto generates a Model class called `User`, this uses the JavaScript namespace `pdemo.data.model.User`, you now wish to write a function to say concatenate a user's first and last name. The obvious approach is to use `goog.inherits` to create a subclass of `pdemo.data.model.User`. However for dynamic operations like parsing server responses maintaining the namespace is crucial.

Thanks to JavaScript's dynamic nature and Closure's excellent dependency management it's quite easy to implement a pattern that closely resembles Objective-C Categories. The idea is to be able to maintain the custom code in a separate file and be able to dynamically merge it with the auto generated code during runtime.

To achieve this for our hypothetical User class, create a file called `UserExtensions.js`, this will provide the namespace `pdemo.data.model.UserExtension` and depend on `pdemo.data.model.User`.

```
goog.provide('pdemo.data.model.UserExtension');
goog.require('pdemo.data.model.User');

# Closure will ensure that the namespace pdemo.data.model.UserExtension
# is available here, feel free to extend it

pdemo.data.model.User.prototype.getFullName = function() {
    return this.getFirstName() + " " +  this.getLastName();
};
```

Now where you want to create an instance of `pdemo.data.model.User`, use the extension as the dependency `pdemo.data.model.UserExtension`. This ensures that both the auto generated namespace and your extensions are available.

```
goog.provide('pdemo.ui.web.Renderer');

# This will make available the pdemo.data.model.User namespace with your extensions
goog.require('pdemo.data.model.UserExtension');
```

Types API

The Types API is a client side implementation of the prestans types API found on the server side. It assists in directly translating validation rules for Web based clients consuming REST services defined using prestans. Later in this chapter we demonstrate a set of tools that cut out the laborious job of creating client side stubs of your prestans models.

- `String`, wraps a string
- `Integer`, wraps a number
- `Float`, wraps a number
- `Boolean`, wraps a boolean
- `DateTime`, wraps a goog.date.DateTime and includes format configuration from the server side definition.
- `Array`, extends goog.iter.Iterator enables you to use `goog.iter.forEach`, we wrap most of the useful methods provided by Closure iterables.
- `Model`, wraps JavaScript `object`
- `Filter` is an configurable filter that you can pass with API calls, this translates back into attribute strings, discussed in *Validating Requests*.

Array

`prestans.types.Array` extends `goog.iter.Iterator`, allowing you to use `goog.iter.forEach` to iterate.

- `isEmpty`
- `isValid`
- `append`
- `insertAt`
- `insertAfter`
- `length`
- `asArray`
- `clone`

Google Closur provides a number of useful methods

- `removeIf`
- `remove`

- sort
- clear
- containsIf
- contains
- objectAtIndex

REST Client

prestans contains a ready made REST Client to allow you to easily make requests and unpack responses from a prestans enabled server API. Our client implementation is specific to be used with Google Closure and only speaks *JSON*.

The client has three important parts:

- Request Manager provided by `prestans.rest.json.Client`, this queues, manages, cancels requests and is responsible for firing callbacks on success and failure. Your application lodges all API call requests with an instance of `prestans.rest.json.Client`. It's designed to be shared by your entire application.
- Request provided by `prestans.rest.json.Request` is a formalised request that can be passed to a Request Manager. The Request constructor accepts a JSON payload with configuration information, this includs partial URL schemes, parameters, optional body and a format for the response. The Request Manager uses the responses format to parse the server response.
- Response provided by `prestans.rest.json.Response` encapsulates a server response. It also contains a parsed copy of the server response expressed using prestans types.

The general idea is:

- To maintain a globally accessible Request Manager
- Formally define each Xhr operation as a Request object
- The Request Manager handles the life cycle of a Xhr call and call an endpoint in your application on success or failure
- Both these callbacks are provided an instance of `Response` containing the appropriate available information

Request Manager

First step is to create a request manager by instantiating `prestans.rest.json.Client`, it takes the following parameters:

- `baseUrl`, to be consistent with the single point of origin constraint, we assume that all your API calls are prefixed with something like `/api`. If you provide a base URL all your requests should provide URLs relative to the base. This also makes for eased maintenance in case you rearrange your application URLs.
- `opt_numRetries` set to 0 by default, causing requests never to be retried. Xhr implementations are capable of retrying to reach the server in case of failure.

There's a fair chance that your application might launch simultaneous Xhr requests, it's also likely that you would want to cancel some requests on events e.g as the user clicks around names of artists to get a list of their albums, you want to cancel any previously unfinished calls if the user has clicked on another artist name.

Our request manager can work this, this is done by using a shared instance of the request manager across your application. The following code sample demonstrates how you might maintain a global Request Manager instance:

```
goog.provide('pdemo');
goog.require('prestans.rest.json.Client');

pdemo.GLOBALS = {
    API_CLIENT: new prestans.rest.json.Client("/api", 0)
};
```

Then use the `makeRequest` method on the Request Manager instance to dispatch API calls, it requires the following parameters:

- `request` is a `prestans.rest.json.Request` object.
- `callbackSuccessMethod` which is a reference to a function the Request Manager calls if the API call succeeds, the method will be passed a response object. Ensure you use `goog.bind` to bind your function to your namespace.
- `callbackFailureMethod` optional reference to a function the Request Manager calls if the API call fails, this method will be passed a response object with failure information.
- `opt_abortPreviousRequests`, asks the Request Manager to cancel all pending requests.

```
# Assume you have a request object
pdemo.GLOBALS.API_CLIENT.makeRequest(
    request,
    goog.bind(this.successCallback_, this),
    goog.bind(this.failureCallback_, this),
    false
);
```

Note: Request objects tell the manager if they are willing to be aborted, this is configurable per request lodged with the manager.

The second method the Request Manager provides is `abortAllPendingRequests`, this accepts no parameters and is responsible for aborting any currently queued connections. The failure callback is not fired when requests are aborted.

Xhr Communication Events

The Request Manager raises the following events. These come in handy if your application requires global UI interactions e.g a Modal popup if network communication fails, or notification messages on success.

- `prestans.rest.json.Client.EventType.RESPONSE`, raised when a round trip succeeds, this would be raised even if your API raised an error code, e.g Bad Request or Service Unavailable.
- `prestans.rest.json.Client.EventType.FAILURE` raised if a round trip fails.

Example of using `goog.events.EventHandler` to listen to the Failure event:

```
goog.require('goog.events.EventHandler');

# and somewhere in one of your functions
this.eventHandler = new goog.events.EventHandler(this);
this.eventHandler_.listen(pdemo.GLOBALS.API_CLIENT, prestans.rest.json.Client.
↪EventType.FAILURE, this.handleFailure_);
```

The `event` object passed to the end points is of type `prestans.rest.json.Client.Event` a subclass of `goog.events.Event`. Call `getResponse` method on the event to get the `Response` object, this will give you access all the information about the request and it's outcome.

Composing a Request

Requests `prestans.rest.Request`

`prestans.rest.json.Request`

- `identifier` unique string identifier for this request type
- `cancelable` boolean value to determine if this request can be canceled
- `httpMethod` a `prestans.net.HttpMethod` constant
- `parameters` an array of key value pairs send as part of the URL
- `requestFilter` optional instance of `prestans.types.Filter`
- `requestModel` optional instance of `prestans.types.Model`, this will be used to parse the response message body
- `responseFilter` optional instance of `prestans.types.Filter`, used to ignore fields in the response
- `responseModel` Used to unpack the returned response
- `arrayElementTemplate` Used if response model is an array
- `responseModelElementTemplates`
- `urlFormat` sprintf like string used internally with goog.string.format
- `urlArgs` a JavaScript array of parameters used with `urlFormat`

`prestans.net.HttpMethod` encapsulate HTTP verbs as constants, currently supported verbs are:

- `prestans.net.HttpMethod.GET`
- `prestans.net.HttpMethod.PUT`
- `prestans.net.HttpMethod.POST`
- `prestans.net.HttpMethod.DELETE`
- `prestans.net.HttpMethod.PATCH`

Reading a Response

- `requestIdentifier` The string identifier for the request type,
- `statusCode` HTTP status code,
- `responseModel` Class used to unpack response body,
- `arrayElementTemplate` prestans.types.Model,
- `responseModelElementTemplates`
- `responseBody` JSON Object (Optional)

Code Generation

CHAPTER 11

Demo Application (incomplete)

Due to our obsession with music, we thought it be fitting to build a music catalgoue as our demonstration application. The application models Artists, Bands, Albums and Tracks to demonstrate the features and techniques to make REST based Web applications.

Sample data features legendary artists and bands like Pink Floyd, Eric Clapton and Metallica purely due to the developers bias in music.

It's complete with a Google Closure based user interface, which shows off the set of handy automation tools that prestans ships to speed up client side development.

Note: Subversion path for our demo app, https://prestans-demo.googlecode.com/svn/trunk/

The demo app ships with it's own copy of prestans, once you've obtained a copy of the demo app, and assuming you have Google's AppEngine Python SDK setup, just run the following command:

```
$ cd prestans-demo/app
$ dev_appsever.py .
```

Navigate to http://localhost:8080/ and voilà we have an app!

CHAPTER 12

Reference Material

We found the following references useful while writing prestans, they cover a variety of advanced Python programming and Web development topics.

It's important that you understand the basic concepts of Python Web Programming. All our documentation and support is based around the assumption that you are familiar with Python Web development using WSGI and are writing Ajax Web apps.

WSGI

- WSGI the way Web servers talk to Python apps.
- ReUsable Web Components with Python and Future Python Web presented by Ben Bangert (YouTube).
- Hosting Python Web Applications presented by Graham P Dumpleton, author of mod_wsgi (YouTube).

Advanced Python

- Python Decorators various prestans utilities are provided as decorators
- Python Types and Objects an excellent article by Shalabh Chaturvedi on how Python sees Objects and Types.
- Python Attributes and Methods another excellent article by Shalabh Chaturvedi providing an indepth understanding of how attributes and methods work.
- Faux function type signatures in Python using a Python decorator to ensure that your functions get values in the right type from WSGI calls. Originall posted as a response on Stackoverflow.
- Inspecting live objects in Python the inspect module provides functions for introspecting on live objects and their source code. This article by Doug Hellmann shows off many really nice features like discovering method signatures, extracting docstrings, etc.

Software

- Google App Engine an extemely easy to work with Cloud platform run by Google.
- mod_wsgi, a connector module allowing your to run WSGI apps with Apache Web server.
- wsgid, Wsgid is a generic WSGI handler for mongrel2 web server. Mongrel2 is a non-blocking web server backed by a high performance queue (0mq). Wsgid plays a gateway role between mongrel2 and your WSGI application, offering a full daemon environment with start/stop/reload functionality.
- MongoDB, MongoDB (from "humongous") is a scalable, high-performance, open source NoSQL database. Written in C++.

Developer Tools

- JSON Lint, a hosted JSON validation service
- JSON View, a in browser JSON prettifier for Chrome and Firefox.
- Postman a Chrome plugin to ease API testing.

CHAPTER 13

Getting Help

We encourge the use of our mailing lists (run on Google Groups) as the primary method of getting help. You can also write the developers through contact information our website.

- Discuss general discussion, help, suggest a new feature.
- Announcements security / release announcements.

Reporting Issues

We prefer the use of our Issue Tracker on Google Code, to triage feature requests, bug reports.

Before you lodge a lodge a ticket:

- Ensure that you ask a question on our list, there might already be answer out there or we might have already acknowledged the issue
- Seek wisdom from our beautifully written documentation
- Google to see that it's not something to do with your server environment (versions of Web server, WSGI connectors, etc)
- Check to ensure that you are *not* lodging a duplicate request.

When reporting issues:

- Include as much detail as you can about your environment (e.g Server OS, Web Server Version, WSGI connector)
- Steps that we can use to replicate the bug
- Share a bit of your application code with us, it goes a long way to replicate issues

Commercial Support

All commercial endeavors around prestans are managed by Eternity Technologies.

- Help with designing high performance REST apps using prestans
- Extending prestans support for backends, serializers, etc.
- Developer training in Python, prestans, Google Closure.

We also offer custom development services for writing high end Ajax and mobile apps, check out our website to see if we can help you create your next Web / Mobile project.

Index