Prestans Documentation

Release 2.0

Anomaly Software

November 02, 2016

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 sits better with complex Ajax applications written with JavaScript infrastructure like Google Closure, or native mobile apps.

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.

This document assumes that you have a working knowledge of REST and WSGI.

CHAPTER 1

Installation

For a typical server production deployment, we recommend installing Prestans via PyPI:

```
$ sudo pip install prestans
```

this will build and install Prestans for your default Python interpreter. To keep up-to-date; use the `---upgrade` option:

```
$ sudo pip install prestans --upgrade
```

If you need to run multiple versions of Prestans on a server, consider using Virtualenv.

Warning: Prestans 2.0 is backwards incompatible. 1.x is still available for download, upgrading to 2.x will break your 1.x application.

Alternatively you can download and build Prestans using distutils:

```
$ tar -zxvf prestans-2.x.x.tgz
$ cd prestans-2.x.x
$ sudo 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 (unless of course you require a bleeding edge feature).
- If you prefer to include Prestans as a Git submodule, ensure you use reference one of the `tags`.
- If your server environment has hard limits on number of files, consider using zipimport.

As a Git submodule:

```
$ git submodule add https://github.com/anomaly/prestans.git ext/prestans
```

When including Prestans manually ensure that your web server is able to locate the files.

1.1 Software Requirements

The server side requires a WSGI compliant environment:

- Python 2.7
- WSGI compliant server environment (Apache + mod_wsgi, or Google AppEngine, etc).

- WebOb 1.2.3 (if you're using PyPI this should be installed as a dependency, AppEngine already provides WebOb)
- Your choice of a persistent store might require you to install SQLAlchemy or AppEngine Datastore.

Google Closure Library Extensions covers our client side Javascript library for Google Closure projects.

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

1.2 Deployment notes

The following are environment specific deployment gotchas. Things we've learnt the hard way. We'll ensure to constantly keep updating this section as we find more, we also encourage you to keep a eye out on our mailing lists.

1.2.1 Apache + mod_wsgi

Consider the following directory stucture, you might wish to checkout a bleeding edge Prestans into a source controlled directory (in this instance Git):

```
+-- app
+-- conf
+-- static
+-- ext
    +-- prestans
+-- client
+-- conf
project.pth
```

mod_wsgi's WSGIPythonPath directive tells mod_wsgi to add any locations delclared in files with the `.pth` extension to the runtime Python path. This allows you to put Python modules in a directory e.g `ext` and distribute it with your project.

1.2.2 AppEngine

Python projects under AppEngine use a YAML configuration file called app.yaml to specify versions of Python libraries you project requires. Ensure that you have one of the following included in your app.yaml file.

During development:

```
libraries:
- name: webob
  version: latest
```

During deployment:

```
libraries:
- name: webob
  version: "1.2.3"
```

Leaving this directive out loads version 1.1 of WebOb; Prestans 2.0 onwards specifically uses WebOb 1.2.3+.

1.3 Unit testing

Prestans ships a unit testing suite, due to the nature of the project only the following can be reliably unit tested:

- Prestans Data Type validation

to run the testsuite checkout Prestans via Git and use the Python unitest framework:

```
python -m unitest prestans.testsuite
```

CHAPTER 2

Framework Design

Prestans is a developer-to-developer product. Before we start talking about how you can start harnessing it's features, we thought it might be a good idea to introduce you to some of the design decisions and patterns. This chapter highlights some of under the hood design decisions, it's written so you can gain an understanding of why Prestans behaves the way it does.

It covers the use of HTTP headers to control features of prestans, and when Prestans will decide to bail out processing a request and when it will choose to fail gracefully (still responding back to the requesting client).

If you trust that we have made all the right decisions and you'd rather start working on your Prestans powered API, then head straight to Routing & Handling Requests

2.1 Exceptions

Prestans raises two kinds of exceptions. It handles both of them in very different ways:

- The first are inbuilt Python exceptions e.g `TypeError`, `AssertionError`, Prestans does not handle these exceptions, causing your API to fail process that particular request. This is because these are only raised if you have incorrectly configured your Prestans handler, or more importantly your handler is not respecting it's own rules e.g objects returned by an end point don't match your parser configuration.
- The second are Prestans defined exceptions. These are handled by Prestans and the router will return a proper error response to the requesting client. These are raised if the client placed an incorrect request e.g An attribute wasn't the right length or type. These excpetions are defined in `prestans.exceptions`, along with a guide for it's suggested use.

Prestans defined exceptions can also be used by your handler to notify the client of trivial REST usecases e.g requested entity does not exists. We talk about these in Routing & Handling Requests.

2.2 HTTP Header Reference

HTTP headers are components of the message header for both HTTP requests and responses. They define the rules (e.g preferred content, cookies, software version) for each HTTP requests. These assist the server to best respond to the request and ensures that the client is able to consume it properly.

Prestans uses a combination of standard and custom headers to allow the requesting client to control the behavior of the API (without you having to write any extra code). Some of these headers can be specific to an HTTP Verb (e.g GET requests don't have request bodies and shouldn't send headers specifying the mime type of the body).

prestans' client side add-ons for Google Closure wrap these up properly client library calls.

2.2.1 Content Negotiation

Prestans APIs can speak as many serialization formats as they please. We support JSON and Plist XML out of the box. It's possible to write custom serializers and deserializers if you wish to support any other formats.

Each API request uses the following standard HTTP headers to negotiate the content type of the request and response payload:

- `Accept` - which tells the server formats that the client is willing to format, Prestans compares this to what the current API support, or sends back an error message using the default serializer.
- `Content-Type` - which tells the server what format the request body is; this is only relevant for certain HTTP Verbs. If the Prestans API does not support the format it sends back an error message using the default serializer.

Note: Prefixed URLs to separate content types is an *ugly* solution. If you are new to REST, we highly recommend watching Michael Mahemoff's Web Directions Code 2013 presentation on REST, What every developer should know about REST.

Both these headers are part of the HTTP standards, the handler section in this chapter discusses how you pair up serializers and deserializers to these headers.

2.2.2 Custom headers

Prestans allows the requesting the client to control what parts of an entity they, they want returned as part of a response. The Prestans features we are referring to are (these are talked about in detail in other chapters):

- **Attribute Filters**, allows the client to toggle the visibility of attributes in a response, thus controlling the size and in turn the responsiveness of the API call. The typical use case of an attribute filter:
  - The client requests *only* the key and title of all blog posts
  - The UI waits for the user to choose a post they want to view
  - The client requests *only* the body of the blog post
  - The client displays the blog post in full
  - The client chooses to *cache* the body of that particular post by merging it into the partial entity
- **Minification**, Prestans REST models are designed to be descriptive to ensure your code reads well. Minification rewrites the attribute names in a handler response reducing the payload size. This is particularly useful when you are downloading collections (e.g list of blog posts). The response size can shrink by upto 30%.

Each Prestans may choose to override the requesting client's wishes. This is a design choice, and you must have a very ridig usecase for ignoring the client's request configuration.

Clients use a set of custom HTTP headers to configure these options. These are detailed as headers that a client sends to an API, and information the API sends back as HTTP headers.

Your Web server environment will limit the size of a request and how large headers can be.

Note: IETF's RFC6648 deprecates the use the `X-` prefix for custom headers.

Inbound headers:

- `Prestans-Version`, expected minimum version of the Prestans framework you're expecting to find on the server side. This ensures that your reuqest can reliably processed by the server.

- `Prestans-Minification`, expects the string `On` or `Off` to toggle minification. This is set to `Off` by default.
- `Prestans-Response-Attribute-List`, a serialized and nested structure of booleans to toggle the visibility of each attribute that you are expecting in the response. By default it's assumed that you wish to receive all attributes of the entity. If minification is turned on you must use the minified attribute names. Serialization format is controlled by the `Content-Type` and `Accept` header, this must be the same for the payload and the content of this header.

Outbound headers:

- `Prestans-Version`, the version number of the Prestans framework that the server's running. This allows the client code to ensure that it's can process the response.

CHAPTER 3

Routing & Handling Requests

Web Server Gateway Interface or WSGI (**PEP 333**) is the glue between a Web Server and your Python Web application. The responding application simply has to be a Python callable (a python function or a class that implements the `__call__` method). Each *callable* is passed the Web server environment and a `start_response`.

Note: If you are unfamiliar with WSGI we recommend reading Armin Ronarcher's introduction to WSGI. We also have a great collection of Reference Material on Python Web development.

WSGI interfaces will generally handover requests that match a URL pattern to the mapped WSGI callable. From the callable is responsible for dispatching the request to the appropriate handler based on part of the URL, HTTP verb, headers or any other property of an HTTP request or a combination properties. This middle ware code is refered to as a request router and Prestans provides one of it's own.

Prestans makes use of standard HTTP headers for content negotiation. In addition it uses a handful of custom headers that the client can use to control the Prestans based API's behavior (features include Content Minification and Attribute Subsets for requests and responses). We'll first introduce you to the relevant HTTP and how it effects your API requests followed by how you can handle API requests in prestans.

3.1 Serializers & DeSerializers

Serializers and DeSerializers are pluggable Prestans constructs that assist the framework in packing or unpacking data. Serialzier or deserializer handle content of a particular mime type and are generally wrappers to vocabularies already available in Python (although it possible to write a custom serializer entirely in Python). Serializers always write out a parseable version of models.

Prestans application may speak as many vocabularies as they wish; vocabularies can also be local to handlers (as opposed to applicaiton wide). You must also define a default format.

Each request must send an `Accept` header for Prestans to decide the response format. If the registered handler cannot respond in the requeted format Prestans raises an `UnsupportedVocabularyError` exception inturn producing a `501 Not Implemented` response. All Prestans APIs have a set of default formats all handlers accept, each end-point might accept additional formats.

If a request has send a body (e.g `PUT`, `POST`) you must send a `Content-Type` header to declare the format in use. If you do not send a `Content-Type` header Prestans will attempt to use the default deserializer to deserialize the body. If the `Content-Type` is not supported by the API an `UnsupportedContentTypeError` exception is raised inturn producing a `501 Not Implemented` response.

3.2 Routing Requests

Prestans is built right on top of WSGI and ships with it's own WSGI Request Router. The router is responsible for parsing the HTTP request, setting up a HTTP response, setup a logger (if one wasn't provided as part of the configuraiton), finally check to see if the API services the requested URL and hand the request over to the handler.

The handler is responsible for constructing the response and one return the router, asks the response to serialize itself. If an exception is raised (see Framework Design) is raised by the framework (because it couldn't parse the request or response) or the handler, the router where appropriate, writes a detailed error trace back to the client.

Prestans verbosely logs events per API request. Your system logger is responsible for verbosity of the logger.

3.2.1 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:

- `/album/` - refers to a collection of type album
- `/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:

- `/album/(.*)` - For collection of albums
- `/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:

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

3.2.2 Using Request Router

Warning: *since Prestans 2.1, URL's are matched according to the WSGI environment variable* `'PATH_INFO'`*, this allows for the creation of mount-point agnostic paths. Users of Prestans versions < 2.1 who rely on the* `WSGIScriptAliasMatch` *directive should replace them with WSGIScriptAlias and remove pattern matching from the first argument and adjust their URL route regex definitions to truncate the mount-point path*
example upgrade to **Prestans>=2.1**:

```
WSGIScriptAliasMatch ^/api/(.*) /scripts/my_app.wsgi  # replace with...
WSGIScriptAlias /api /scripts/my_app.wsgi
```

```
application = RequestRouter([
    (r'/api/things', my.Handler),  # replace with...
    (r'/things', my.Handler)
])
```

The router is provided by the `prestans.rest` package. It's the keeper of all Prestans API requests, it (with the help of other members of the `prestans.rest` package) parases the HTTP request, setups the appropriate handler and hands over control.

The router also handles error states raised by the API and informs the requesting client to what went wrong. These can include issues with parsing the request or exceptions raised by the handler (this is covered later in the chapter) while dealing with the request.

The constructor takes the following parameters:

- `routes` a list of tuples that maps a URL with a request handler
- `serializers` takes a list of serializer instances, if you omit this parameter Prestans will assign JSON as serializer to the API.
- `default_serializer` takes a serializer instance which it uses if the client does not provide an `Accept` header.
- `deserializers`, a set of deserializers that the clients use via the `Content-Type` header, this default to `None` and will result in Prestans using JSON as the default deserializer.
- `default_deserializer`, default serializer to be used if a client doesn't provide a `Content-Type` header, defaults to `None` which results in Prestans using the JSON deserializer.
- `charset`, to be used to parse strings, defaulted to `utf-8`
- `application_name` the name of your API, `prestans`
- `logger`, an instance of a Python logger, defaulted to `None` which results in Prestans creating a default logger instance.
- `debug`, runs Prestans under debug mode (results in increased logging, error reporting), it's defaulted to `False`

```
import prestans.rest
import myapp.rest.handlers

api = prestans.rest.RequestRouter(routes=[
        (r'/([0-9]+)', myapp.rest.handlers.DefaultHandler)
    ],
    serializers=[prestans.serializer.JSON()],
    default_serializer=prestans.serializer.JSON(),
    deserializers=[prestans.deserializer.JSON()],
    default_deserializer=prestans.deserializer.JSON(),
```

```
    charset="utf-8",
    application_name="music-db",
    logger=None,
    debug=True)
```

The router is a standalone WSGI application you pass onto server environment.

If your application prefers a diaclet other than JSON as it's default, ensure you configure this as part of the router. It's recommended that the default diaclet for serialization and deserialization is the same.

The default logger uses is a configured Python Logger, it logs in detail the lifecycle of a request along with the requested URL. Refer to documentation on how to configure your system logger to control verbosity.

Once your router is setup, Prestans is ready to route requests to nominated handlers.

If were deploying under mod_wsgi, your the above would be the contents of your WSGI file. mod_wsgi requires the endpoint to be called `application`. The WSGI configuration variable might look something like.

Note: *since 2.1*

```
WSGIScriptAlias     /api   /srv/musicdb/api.wsgi
```

Under AppEngine, if this above was declared under a script named `entry.py`, you would reference it in `app.yaml` as follows:

```
- url: /api/.*
  script: entry.api
  login: required
```

3.3 Handling Requests

REST requests primarily use the following HTTP verbs to handle requests:

- `HEAD` to check if the entity the client has is still current
- `GET` to retrieve entities
- `POST` to create a new entity
- `PUT` to update an entity
- `PATCH` to update part of an entity
- `DELETE` to delete an entity

Prestans maps each one of these verbs to a python function of the same name in your REST handler class. Each REST request handler in your application derives from `prestans.rest.RequestHandler`. Unless your handler overrides the functions `get`, `post`, `put`, `patch`, `delete` the base implementation tells the Prestans router that the requested end point does not support the particular HTTP verb.

Your handler must accept an equal number of parameters as defined the router regular expression.

Our Regex premier highlights the use of two handlers per entity, one deals with collections the other entities. APIs generally let clients get a collection of entities, add to a collection and get a particular entity, update an entity or delete an entity. The later require an identifier for the entity, where as the collection does not.

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

This is no way says that your API can't provide an endpoint to delete all entities of a type or update a collection of entities, in which instances your collection handler would implement the appropriate HTTP verb handlers.

A typical collection handlers would typically look like (implementing `GET` and `POST` and does not require an identifier):

```
import prestans.rest
import prestans.parser

import myapp.rest.models

class MyCollectionRESTRequestHandler(prestans.rest.RequestHandler):

    __parser_config__ = prestans.parser.Config(
        GET=prestans.parser.VerbConfig(
            response_template=prestans.types.Array(element_template=myapp.rest.models.Track())
        ),
        POST=prestans.parser.VerbConfig(
            body_template=myapp.rest.models.Track(),
            response_template=myapp.rest.models.Track()
        )
    )

    def get(self):
        ... return a collection of entities

    def post(sef):
        ... add a new type
```

A typical entity handler would look like (implementing `GET`, `PUT` and `DELETE` expecting an identifier):

```
import prestans.rest
import prestans.parser

import myapp.rest.models

class MyEntityRESTRequestHandler(prestans.rest.RequestHandler):

    __parser_config__ = prestans.parser.Config(
        GET=prestans.parser.VerbConfig(
            response_template=myapp.rest.models.Track()
        ),
        PUT=prestans.parser.VerbConfig(
            body_template=myapp.rest.models.Track(),
        )
    )

    def get(self, track_id):
        ... return an individual entity

    def put(self, track_id):
        ... update an entity

    def delete(self, track_id):
        ... delete an entity
```

Notice that since deleting an entity only requires an identifier, and does not have to parse the body of a request. The update request can also choose to use attribute filters to pass in partial objects.

Note: At this point if you'd rather learn about how to parse requests and responses, then head to the chapter on Validating Requests & Responses. This chapter continues to talk about how handlers work assuming you are going to read the chapter on validation shortly after.

Each handler allows accessing the environment as follows:

- `self.request` is the parsed request based on the rules defined by your handler, this is an instance of `prestans.rest.Request`
- `self.response` is response Prestans will eventually write out to the client, this is an instance of `prestans.rest.Response`
- `self.logger` is an instance of the logger the API expects you to write any information to, this must be an instance of a Python logger
- `self.debug` is a boolean value passed on by the router to indicate if we are running in debug mode

Each request handler instance is run in the following order (all of these methods can be overridden by your handler):

- `register_seriailzers` is called on the handler, this allows the handler to a list of additional serializers it would like to use
- `register_deserializers` is called on the handler, this allows the handler to a list of additional deserializers it would like to use
- `handler_will_run` is called, perform any handler specific warm up acts here
- The function that corresponds to the requests HTTP verb is called
- `handler_did_run` is called, perform any handler specific tear downs here

`prestans.rest.RequestHandler` can be subclassed as your project's Base Handler class, this generally contains common code e.g getting access to a database sessions, etc. A SQLAlchemy specific example would look like:

```
import prestans.rest
import prestans.parser

import myapp.rest.models

class Base(prestans.rest.RequestHandler):

    def handler_will_run(self):
        self.db_session = myapp.db.Session()
        self.__provider_config__.authentication = myapp.rest.auth.AuthContextProvider(self.request.er

    def handler_did_run(self):
       myapp.db.Session.remove()

    @property
    def user_profile(self):
        return self.__provider_config__.authentication.get_current_user()

    @property
    def auth_context(self):
        return self.__provider_config__.authentication
```

Note: You'd typically place this in `myapp.rest.handlers.__init__.py` and place all your handlers grouped by entity type in that package.

3.4 Constructing Response

The end result of all handler call is to send a response back to the client. This can be as simple as a status code, or as elaborate as a group of entities. Prestans is unforgiving (unless requested otherwise) while accepting requests and writing responses.

In accordance with the REST standard, each handler must declare what sort of entities (if any) the handler will return if it successfully processes a request. We will cover error scenarios later in this chapter.

Declaration of response types are defined as part of your parser configuration per HTTP verb. Handlers typically return a collection of entities, defined as:

```
import prestans.rest
import prestans.parser
import prestans.types

import myapp.rest.models

class MyEntityRESTRequestHandler(prestans.rest.RequestHandler):

    __parser_config__ = prestans.parser.Config(
        GET=prestans.parser.VerbConfig(
            response_template=prestans.types.Array(element_template=myapp.rest.models.Album())
        )
    )

    def get(self):

        albums = prestans.types.Array(element_template=myapp.rest.models.Album())
        albums.append(myacc.rest.models.Album(name="Journeyman", artist="Eric Clapton"))
        albums.append(myacc.rest.models.Album(name="Dark Side of the Moon", artist="Pink Floyd"))
        return albums
```

or an individual entity, defined as:

```
import prestans.rest
import prestans.parser

import myapp.rest.models

class MyEntityRESTRequestHandler(prestans.rest.RequestHandler):

    __parser_config__ = prestans.parser.Config(
        GET=prestans.parser.VerbConfig(
            response_template=myapp.rest.models.Album()
        )
    )

    def get(self, album_id):

        return myapp.rest.models.Album(name="Journeyman", artist="Eric Clapton")
```

More often than not, the content your handler sends back, would have been read queried from a persistent data store. Sending persistent data verbatim nearly never fits the user case. A useful API sends back appropriate amount of information to the client to make the request useful without bloating the response. This becomes a cases by case consideration of what a request handler sends back. Sending out persistent objects verbatim could sometimes pose to be a security threat.

Prestans requires you to transform each persistent object into a REST model. To ease this tedious task Prestans

provides a feature called Data Adapters. Data Adapters perform the simple task of converting persistent instances or collections of persistent instances to paired Prestans REST models, while ensuring that the persistent data matches the validation rules defined by your API.

Data Adapters are specific to backends, and it's possible to write your own your backend specific data adapter. All of this is discussed in the chapter dedicated to Data Adapters.

3.4.1 Minifying Content

Prestans tries to make your API as efficient as possible. Minimizing content size is one of these tricks. Complimentary to Attribute Filters (which allows clients to dynamically turn attributes on or off in the response) is response key minification.

This is particularly useful for large amounts of repetitive data, e.g several hundred order lines.

Setting the `Prestans-Minification` header to `On` is all that's required to use this feature. This is a per request setting and is set to `Off` by default.

Prestans also sends `Prestans-Minification-Map` header back containing a one to one map of the original attribute names it's minified counterpart.

Note: Our Google Closure extensions provides a JSON REST Client, which can automatically unpack minified requests to fully formed Prestans client side models.

3.4.2 Serving Binary Content

It's perfectly legitimate for your REST handler to return binary content. Prestans provides a built in model to assign to your handler's `VerbConfig`.

Your handler must return an instance of `prestans.types.BinaryContent`, and Prestans will do what's right to deliver binary content.

Instances of `BinaryContent` accept the following parameters:

- `mime_type`, the mime type of the file that you are sending back
- `file_name`, the filename that Prestans is to send back in the HTTP header, this is what the browser thinks the name of the file is. File names can be generated by applications or they might have been stored as meta information when the file was uploaded by the user
- `as_attachment`, tells Prestans if the file is to be delivered as an attachment (forces the user to save the file) or deliver it inline (generally opens in the browser).
- `contents`, binary contents that you've read up from disk or generated.

```
import prestans.types

class Download(st.cs.rest.handlers.Base):

    __parser_config__ = prestans.parser.Config(
        GET=prestans.parser.VerbConfig(
            response_template=prestans.types.BinaryResponse()
        )
    )

    def get(self, document_id):
```

```
        file_contents = open(file_path).read()

        self.response.status = prestans.http.STATUS.OK
        self.response.body = prestans.types.BinaryResponse(
            mime_type='application/pdf',
            file_name='invoice.pdf',
            as_attachment=True,
            contents=file_contents)
```

If you set `as_attachment` to False the file will be delivered inline into the browser. It's up to the browser to handle the content properly.

3.5 Raising Exceptions

As alluded to in our Thoughts on API design chapter, Prestans provides two distinct set of Exceptions. The first raised if you've configured your API incorrectly and the later used to send back meaningful error messages to the requesting client.

This section deals with Exceptions that Prestans expects you to use to raise meaningful REST error messages. These are generally caused by the client sending you an inappropriate e.g the logged in user is not allowed to access or update an entity, or something simply not being found on the persistent data store.

Note: **PEP 008** recommends that Exceptions that are errors should end with the Error suffix.

You do not have to raise exceptions for request and response data validation. If the data does not match the rules defined by your models, parameter sets or attribute filters, it's prestans' responsibility to graceful fail and respond back to the client.

3.5.1 Unsupported Vocabulary or Content Type

Prestans uses the `Accept` and `Content-Type` HTTP headers to negotiate the format the client is sending their request as or the format they expect the response in. Prestans ships with a standard set of vocabularies (serialization formats) and allow you to add your own. Prestans automatically adjusts the serializer or de-serializer to use based on the mime types. If Prestans is unable to service the request, the following exceptions are raised:

- `UnsupportedVocabularyError` raised if Prestans can't find a suitable serializer for the requested mime type in the `Accept` header
- `UnsupportedContentTypeError` raised if Prestans can't find a suitable de-serializer for the requested mime type in the `Content-Type` header

If these exceptions are raised as part of parsing the request Prestans generates an appropriate message using the default serializer (internally set to JSON by default).

3.5.2 Configuration Exceptions

Along with Python's *TypeError* Prestans raises the following exceptions if you've configured portions of your application incorrectly.

- `InvalidType` raised if the Python typed passed in for validate is incorrect.
- `ParseFailedError` raised if the data type fails to evaluate the value as an appropriate Python data type

- `InvalidMetaValueError` raised if you've passed a unacceptable configuration option for an attribute
- `MissingParameterError` raised if a required parameter for an attribute is missing
- `UnregisteredAdapterError` raised if the Data Adapters registry can't locate a REST to persistent map

3.5.3 Data Validation Exceptions

Prestans raises the following exceptions (See **PEP 008** for naming conventions of Exceptions) if data passed through Prestans types fails to validate. If the validation fails as part of the request Prestans captures the stack trace and responses to the client with an appropriate error code. If a request fails to parse your handler code will not be executed. If the exception is raised a result of your code (e.g using Data Adapters or writing responses) Prestans will halt the execution and write the error message to the logger.

- `RequiredAttributeError` raised if an attribute is required and an appropriate value wasn't provided, note that you can relax these by using attribute filters
- `LessThanMinimumError` raised if the value provided for an attribute is less than the floor
- `MoreThanMaximumError` raised if the value provided for an attribute is greater than the ceiling
- `InvalidChoiceError` raised if the value provided is not defined in the set of choices for the attribute
- `UnacceptableLengthError` raised if the value provided is longer than the acceptable length
- `InvalidFormatError` raised if the value provided does not pass the regular expression format

3.5.4 Parser Exceptions

Prestans raises the following exceptions when parsing data. If the exception is raise while parsing a request Prestans will fail gracefully by responding to the client with an error message and a track trace. If the exception is raised while your handler returns a response Prestans will stop the execution and expect you fix the issue.

Note: Since all data is strictly validated your application should to maintain consistent data at all times.

- `UnimplementedVerbError` raised if a client requests an HTTP verb that an end point does not handle
- `NoEndpointError` raised if a client requests an endpoint that does not exists
- `AuthenticationError` raised if the client does not have an authenticated session and the handler requires them to do so
- `AuthorizationError` raised if the logged is user fails to pass the authorisation rules
- `SerializationFailedError` raised if serialization of the data fails, would only happen if the handler passed a data type that cannot be handled by the nominated serializer.
- `DeSerializationFailedError` raised if deserialization failed, this would only happen if the client passed data that cannot be parsed by the nominated deserializer
- `AttributeFilterDiffers` raised if the attribute filter differs from the nominated response template
- `InconsistentPersistentDataError`

3.5.5 Handler Exceptions

Prestans make the following exceptions available for your handler to use. They are associated to error codes defined in the HTTP specification and when raised Prestans gracefully returns to the client with an appropriate error message and where applicable a stack trace. Each exception allows you to pass a custom error message which is returned to the client. Prestans will also set the appropriate HTTP error code in the response header.

- `ServiceUnavailable` to be raised if the service called cannot satisfy the request at the present time, this could be caused due to exhausted quotas on cloud services or a database services that may have failed to respond
- `BadRequest` raised when the client placed an inappropriate request, a typical example is that the data Prestans parsed is in the right format but your service is unable to process the combination.
- `Conflict` raised when the data provided to the end point is valid but conflicts with the business logic
- `NotFound` raised when a requested entity is not found, for example the entity ID provided as part of the request is valid in format but does not exists on the system
- `Unauthorized` raised if the particular user is not allowed to access an entity or related children
- `MovedPermanently` raised if the particular endpoint has moved, this should cause the client to request the newly appointed URL
- `PaymentRequired` raised if your service requires the user to subscribe to the service and their subscription has expired
- `Forbidden` raised if the the particular entity the client requested should not be accessed by the currently logged in user

CHAPTER 4

Validating Requests & Responses

A robust and secure API is responsible for validating all incoming and outgoing data to ensure that it matches business rules. Validation is at the heart of Prestans design and a cornerstone feature.

APIs have three major areas where data validation is paramount:

- Query String parameters supplied to the API as key value pairs in the URL `http://localhost/api/album?offset=10&limit=20`
- Serialized data in the body of the request, which is unpacked and made available for use to the handler
- Data that's serialized down to the client as the response to each request. This is typically but not necessarily read back from persistent data stores.

Prestans allows each request handler to elegantly define the rules it wants to adhere to by declaring what we refer to as a `VerbConfig` (the Verb refers to the HTTP verb).

4.1 Attribute Filters

Attribute Filters are Prestans way of making temporary exceptions to validation rules otherwise defined by Prestans `Models`. Quality of code written using Prestans thrives on strong validation. Certainly uses cases in every application demands relaxing rules temporarily. Attribute filters are used both incoming and outgoing data.

At it's heart Attribute Filters are a configuration template for exceptions Prestans is to make while parsing data. Attribute Filters contain a boolean flag for each attribute defined in instances that will be parsed by prestans. They can be created by subclassing `prestans.parser.AttributeFilter` and it containing an identical structure to the corresponding model with boolean flags to denote visibility during a parse operation.

However we recommend using our convenience method that dynamically creates a filter based on a model (unless of course you have non trivial use case):

```
attribute_fitler = prestans.parser.AttributeFilter.from_model(model_instance=mysicdb.rest.models.Albu
```

the above will generate an attribute filter with all attributes turned off for parsing. We can then selectively turn attributes on by setting the keys to `True`:

```
attribute_filter.id = True
attribute_filter.name = True
```

The attribute filter will have a corresponding definition for every attribute visible in the `Model` with the default boolean value assigned to it. This goes for children objects.

If a child attribute is a collection then the child Attribute Filter is based on an instance that would appear in the collection, this allows fine grained control of toggling parse rules. Assigning a `boolean` value to a collection applies the state to all attributes of it's instances.

4.2 Setting up validation rules

Validation rules are set up per HTTP verb your handler intends to service. By default there are no validation rules defined for any HTTP verb, this does not mean that your handler can't respond to a particular verb, it simply means that Prestans takes no responsibility of validating incoming or outgoing data. By design if you wish to send data back to the client Prestans insist on validating what a handler sends down, however it's perfectly valid for a handler to return no content (which is what Prestans expects if you aren't specific).

Each handler has a meta attribute called __parser_config__ this must be an instance of `prestans.parser.Config` which accepts six named parameters one for each supported HTTP verb (`HEAD`, `GET`, `POST`, `PUT`, `DELETE`, `PATCH`) each one of which must be an instance of `prestans.parser.VerbConfig`. A `VerbConfig` accepts the following named parameters (not all of them are supported across all HTTP verbs):

- `response_template` an instance of a `prestans.types.DataCollection` subclass i.e a Model or an Array of Prestans `DataType`. This is what Prestans will use to validate the response your handler sends back to the client.
- `response_attribute_filter_default_value` Prestans automatically creates an attribute filter based on the `response_template`. By default the visibility of all attributes is off.
- `parameter_sets` an array of `prestans.parser.ParameterSet` instances (see *Parameters Sets*)
- `body_template` an instance of a `prestans.types.DataCollection` subclass i.e a Model or an Array of Prestans `DataType`, this is what Prestans will use to validate the request sent to your handler. If validation of the incoming data fails, Prestans will not execute the associated verb in your handler.
- `request_attribute_filter` is an attribute filter used to relax or tighten rules for the incoming data. This is particularly useful if you want to use portions of a model. Particularly useful for `UPDATE` requests.

Warning: By default Prestans exposes none of the attributes in the response, setting the `response_attribute_filter_default_value` parameter to `True` will show all attributes in the response. Your handler code is responsible for toggling visibility in either instance.

```
import prestans.rest
import prestans.parser

import musicdb.rest.models


class CollectionRequestHandler(prestans.rest.RequestHandler):

    __parser_config__ = prestans.parser.Config(
        GET =  prestans.parser.VerbConfig(
            body_template=prestans.types.Array(element_template=musicdb.rest.models.Album())
        ),
        POST =  prestans.parser.VerbConfig(
            body_template=musicdb.rest.models.Album()
        )
    )

    def get(self):
        ... do stuff here
```

```
    def post(self):
        ... do stuff here
```

Prestans is aggressive when it comes to validating requests and responses. However in the cases where you wish to relax the rules we recommend that you use *Attribute Filters*. You can define an `AttributeFilter` in context and assign it to the appropriate `VerbConfig`.

```
update_filter = prestans.parser.AttributeFitler(model_instance=musicdb.prest.models.Album(), default_
update_filter.name = True

class EntityRequestHandler(prestans.rest.RequestHandler):

    __parser_config__ = prestans.parser.Config(
        GET = prestans.parser.VerbConfig(
            response_template=musicdb.rest.models.Album(),
            response_attribute_filter_default_value=False,
            parameters_sets=[]
        ),
        PUT =  prestans.parser.VerbConfig(
            body_template=musicdb.rest.models.Album(),
            request_attribute_fitler=update_filter
        )
    )

    def get(self, album_id):
        ... do stuff here

    def put(self, album_id):
        ... do stuff here

    def delete(self, album_id):
        ... do stuff here
```

Lastly a reminder parameters that were part of your URL scheme will be passed in as positional arguments to your handler verb (see Routing & Handling Requests). Prestans runs your handler code if the the request succeeds to parse and will only respond back to the client if the response you intend to return passes the validation test.

4.3 Working with parsed data

The following sections detail how you access the parsed data and how you provide Prestans with a valid response to send back to the client. Remember that your handler's objective is to send back information the client can reliably use.

4.3.1 Parameters Sets

`ParmeterSets` refer to sets of data sent as key value pairs in the query string. Typically if you handler is expecting data as part of the query string you would expect it to be follow similar patterns as `Models`. Prestans extends the use of it's `types` (see Types & Models) to validate data passed in a query string.

Each `ParameterSet` is made of a group of keys that you're expecting along with rules to be used to parse the value. `ParameterSets` are defined by subclassing `prestans.parser.ParameterSet`.

```
class SearchByKeywordParameterSet(prestans.parser.ParameterSet):

    keyword = prestans.types.String(min_length=5)
    offset = prestans.types.Integer(defauflt=0)
```

```
    limit = prestans.types.Integer(default=10)

class SearchByCategoryParameterSet(prestans.parser.ParameterSet):

    category_id = prestans.types.Integer(min_length=5)
    offset = prestans.types.Integer(defauflt=0)
    limit = prestans.types.Integer(default=10)
```

these would then be assigned to your handler's `VerbConfig` as follows:

```
__parser_config__ = prestans.parser.Config(
    GET = prestans.parser.VerbConfig(
        response_template=musicdb.rest.models.Album(),
        response_attribute_filter_default_value=False,
        parameters_sets=[SearchByKeywordParameterSet(), SearchByCategoryParameterSet()]
    ),
    PUT =  prestans.parser.VerbConfig(
        body_template=musicdb.rest.models.Album(),
        request_attribute_fitler=update_filter
    )
)
```

Note: Parameter Set can only use basic data types i.e `Strings`, `Integer`, `Float`, `Date`, `Time`, `DateTime`.

Using serialized data as values for query string keys is not a good idea.

All web servers have limitations on how large query strings can be, if you experience issues with sending information via the query string you should check your web server configuration before attempting to debug your code.

For each request:

- If the data provided as part of a query string matches, Prestans will make an instance of that `ParameterSet` available at `self.request.parameter_set`.
- If a query string would result in matching more than one `ParameterSet` Prestans will stop parsing at the first match and make it available to your handler
- Failure in matching a `ParameterSet` still results in your handler code being called. Prestans would simply set `self.request.parameter_set` to `None`.

You can access the attributes defined in your `ParameterSet` as you would any ordinary Python object.

If your handler assigned multiple `ParameterSets` to a handler `VerbConfig` you can always check for the `type` of `self.request.paramter_set` for conditional code execution.

4.3.2 Request Body

By assigning a `DataCollection` object to the `body_template` configuration of a `VerbConfig` asks Prestans to strictly parse the data received as part of every request. If the data sent as part of the body successfully parses your handler code is executed and the parsed object is available as `self.request.parsed_body`.

Should you wish to relax the rules of a model for particular use cases you should consider using *Attribute Filters* as opposed to relaxing the validation rules. The attribute filter used while parsing the request is available as `self.request.attribute_filter`.

Note: `GET` requests cannot have a request body and Prestans will not attempt to parse the body for `GET` requests.

Your handler code can access the `parsed_body` as a regular Python object. If your handler accepts collections of elements then Prestans makes available a Prestans Array in the `parsed_body` which is a Python iterable.

4.3.3 Response Body

Once your handler has completed what it needed to do, it can optionally return a response body. If you aren't returning a body then your handler is simply required to set `self.response.status` to a valid HTTP status, Prestans has wrapper constants available in `prestans.http.STATUS`.

Prestans will respect the `response_template` configuration set by your handler's `VerbConfig`. You must return an object that matches the rules. There are generally two scenarios:

- Your handler will return an entity that is an instance of a `prestans.rest.Model` subclass. This is typically the case for Entity handlers.
- Your handler returns a collection (i.e a Prestans Array) which contains instances of a Prestans `DataType` usually a Model. This is typically the case for Collection handlers.

REST end-points must always return the same `type` of response. This is discussed in detail in :doc:api_design.

Note: Prestans does not allow sending down instances of python types because they do not conform to a serialization format making it difficult for the client to determine the reliability of the response.

```
def get(self, album_id):

    ... assuming you have an object that you can return

    self.response.status = prestans.http.STATUS.OK
    self.response.body = new musicdb.rest.models.Album(name="Journeyman", artist="Eric Clapton")
```

If you read your response from a persistent store you would be required to convert that object (typically by copying the values) to a similarly formatted Prestans REST model. Prestans features Data Adapters which automate this process.

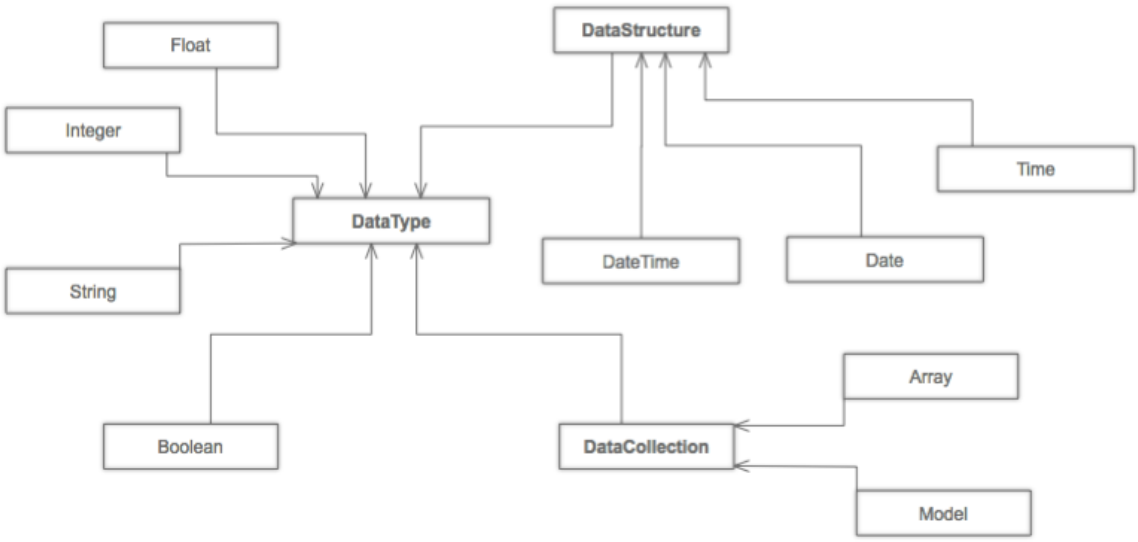
Prestans allows clients to dynamically configure the response payload by sending a serialized version of an Attribute Filter as the HTTP header `Prestans-Response-Attribute-List`. This allows the client to adjust the response from your API without you having to do extra work. Of course the server has the final say, if your handler outright sets a rule there's nothing a client can do to override it.

The response object that your handler has access to has a reference to an *Attribute Filters* which is made up of the rules defined by your `response_template` with the client's request preferences applied, accessible at `self.response.attribute_filter`. If your handler changes the state of the attribute filter before the verb method returns, Prestans used the modified state of the attribute filter, giving you the final say.

CHAPTER 5

Types & 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 & Responses.

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
- `Date`, wraps Python `date`
- `Time`, wraps Python `time`
- `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`.

5.1 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
```

5.1.1 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.

5.1.2 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`.

5.1.3 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))
```

5.1.4 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())
```

5.2 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.

5.2.1 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.DATETIME.CONSTANT.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.

5.2.2 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/gi
```

5.3 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.http.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 Data Adapters 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 section.

5.4 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.

5.4.1 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.

5.4.2 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.

5.4.3 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.

5.4.4 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.

5.4.5 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`

5.4.6 Date

Date Time is a complex structure that parses strings to Python `datetime` and vice versa. Default string format is `%Y-%m-%d` 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.Date.CONSTANT.TODAY` if you want to use the date / time of execusion.
- `format` default format `%Y-%m-%d`

5.4.7 Time

Date Time is a complex structure that parses strings to Python `datetime` and vice versa. Default string format is `%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.Time.CONSTANT.NOW` if you want to use the date / time of execusion.
- `format` default format `%H:%M:%S`

5.4.8 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.DateTime.CONSTANT.NOW` if you want to use the date / time of execusion.
- `format` default format `%Y-%m-%d %H:%M:%S`

5.5 Collections

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

5.5.1 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,

5.5.2 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

Providers

Prestans is a micro-framework designed to streamline building REST APIs. Providers are Prestans' bridge to other infrastructure level services that are specific to each application but are required for REST APIs to function.

6.1 Authentication

Provided by the `prestans.provider.auth` package and is designed to plug into an existing authentication system. Prestans does not provide an authentication layer because it's understood that every applicaiton and environment has it's own requirements, and this outside the realm of Prestans.

Security typically has two parts to the problem, Authentication to check if a 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 (i.e 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.

Once you've defined and assigned an authentication provider for your handler you can use the following decorators (see :pep::*318*) provided by `prestans.providers.auth`:

- `prestans.provider.auth.login_required` checks with the the registered authentication provider if a user is logged in
- ‘‘prestans.provider.auth.role_required‘‘checks with the registered authentication provider if the user has the appropriate role

Note: Providers are inherited, consider assigning the provider to a base handler class of your application.

6.1.1 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.provider.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`. It defines the following method stubs:

- `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.

```
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")
```

6.1.2 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.provider.auth.Bae):

    ## @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)
```

6.1.3 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.

6.1.4 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.rest.RequestHandler):

    __provider_config__ = prestans.provider.Config(
            authentication=musicdb.rest.auth.AuthContextProvider()
        )
```

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.provider.auth.login_required`.
- If your handler method wants to test final grained roles use the `@prestans.provider.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.

Note: If your `AuthenticationContextProvider` needs access to the request or server envrionment you can choose to set it in the setup method called `handler_will_run` which is executed before handler specific code but after the environment and request have been parsed.

```
class MyRESTHandler(prestans.rest.RequestHandler):

    def handler_will_run(self):
        self.__provider_config__ = prestans.provider.Config(
                authentication=musicdb.rest.auth.AuthContextProvider(self.request.environ)
            )

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

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

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

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

CHAPTER 7

Data Adapters

The Validating Requests & Responses chapter demonstrates the use of Prestans `Models` to validate requests, build rules complaint responses and the use of `AttributeFilters` to make temporary exceptions to the validation rules.

`DataAdapters` automate morphing persistent objects to Prestans models, it provides the following features:

- A static registry `prestans.ext.data.adapters.registry`, that maps persistent models to REST models
- An instance convertor, used to convert an instance. Convertors are specific to backends and uses the registry to determine relationships between persistent and REST models.
- A collection iterator, that iterates through collections of persistent results and turns them into REST models. It follows all the same rules as the instance convertor.

Out of the box Prestans supports:

- SQLAlchemy which in turn should allow you to support most popular RDBMS backends.
- AppEngine's Python NDB which is built on top of DataStore.

Note: REST services provide views of persistent data, DataAdapters allow you to map multiple REST models to the same persistent model.

For the purposes of this example lets assume our rest models live in the namespace `musicdb.rest.models` and the persistent models live in `musicdb.models`, and is written for AppEngine.

Writing custom `DataAdapters` is quite straight forward. The Prestans project welcomes third party contributions.

7.1 Pairing REST models to persistent models

Before you can ask Prestans to convert persistent objects to REST model instances, you must use the `registry` you to pair persistent definitions to REST definitions. Prestans uses the REST model as the template for the data that will be transformed. While adapting data Prestans will:

- Inspect the REST model for a list of attributes
- Inspect the persistent model for the data
- Ensure that the data provided by the persistent model matches the rules definied by the REST model
- If the persistent model does not define an attribute, Prestans reverts to using the default value or `None`

- If the value provided by the persistent model fails to validate, Prestans raises an `prestans.exception.DataValidationException` which will graceful respond to the requesting client.

If a persistent definition maps to more than one REST model defintion, DataAdapters will try and make the sensible choice unless you explicitly provide the REST model you wish to adapt the data to.

Registering the persistent model is done by calling the `register_adapter` method on `prestans.ext.data.adapters.registry`, and an appropriate `ModelAdapter` instance.

Consider the following REST models defined in `musicdb.rest.models`:

```
import prestans.types

class Album(prestans.types.Model):

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

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)
```

along with it's corresponding NDB persistent model defined in `musicdb.models`:

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

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

class Album(ndb.Model):

    name = ndb.StringProperty()

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

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()
```

Note: By convention we recommend the use of the namespace `yourproject.rest.adapters` to hold all your adapter registrations.

```
import musicdb.models
import musicdb.rest.models


# 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=musicdb.rest.models.Band,
        persistent_model_class=musicdb.models.Band
    )
)
```

7.2 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 appropriate HTTP status code
- Use the `adapt_persistent_instance` or `adapt_persistent_collection` from the appropriate package to transform your persitent objects to REST objects.
- Prestans will query the registry for any children objects that appear in the object it's attempt to adapt.
- Assign the returned collection to `self.response.body` to send a response to the client

Each `DataAdapter` provides two convenience methods:

- `adapt_persistent_collection` which iterates over a collection of persistent objects to a collection of REST models
- `adapt_persistent_instance` which iterates over an instance of a persistent object to a REST model

```
from google.appengine.ext import ndb

import musicdb.models
import musicdb.rest.handlers
import musicdb.rest.models
import musicdb.rest.adapters

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


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

    request_parser = CollectionRequestParser()

    def get(self):
```

```
        bands = musicdb.models.Band().query()

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

If you are using `AttributeFilters`, you should pass the filter along to the adapter method enabling it to skip accessing that property all together. This can significantly reduce read stress on backends that support lazy loading properties:

```
# Collection of objects
class BandCollection(musicdb.rest.handlers.Base):

    def get(self):

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

        self.response.http_status = prestans.http.STATUS.OK
        self.response.body = prestans.ext.data.adapters.ndb.adapt_persistent_collection(
            collection=bands,
            target_rest_instance=musicdb.rest.models.Band,
            attribute_filter = self.response.attribute_filter
        )

# Adapting a single instance
class BandEntity(musicdb.rest.handlers.Base):

    def get(self, band_id):

        .. use the appropriate query to get the appropriate instance

        self.response.http_status = prestans.http.STATUS.OK
        self.response.body = prestans.ext.data.adapters.ndb.adapt_persistent_instance(
            collection=band,
            target_rest_instance=musicdb.rest.models.Band,
            attribute_filter = self.response.attribute_filter
        )
```

Note: Each handler has access to the approprite attribute filter at `self.response.attribute_filter` (see Validating Requests & Responses)

7.3 Writing your own DataAdapter

`DataAdapter` can be easily extended to support custom backends. Writing an adapter for a custom backend involves providing:

- an `adapt_persistent_collection` method that iterates over a collection of persistent objects and transforms them into a collection of REST objects
- an `adapt_persistent_instance` method that converts a single instance of a persistent object to a REST object

- an implementation of a `ModelAdapter` class that implements `adapt_persistent_to_rest` method which converts a persistent object to a REST object. An instance of this is returned by the `registry` and is used by the convenience methods. This method detail how each property is transformed.

It's very likely that you will be able to reuse the code for the convenience methods from one of the `DataAdapters` shipped with Prestans. They are responsible for wrapping backend specific operations like accessing collection lengths.

A scaffold of the a custom `DataAdapter` looks as follows:

```
def adapt_persistent_instance(persistent_object, target_rest_class=None, attribute_filter=None):
    ... your custom implementation

def adapt_persistent_collection(persistent_collection, target_rest_class=None, attribute_filter=None)
    ... your custom implementation


class ModelAdapter(adapters.ModelAdapter):

    def adapt_persistent_to_rest(self, persistent_object, attribute_filter=None):
        ... your custom implementation here
```

CHAPTER 8

Google Closure Library Extensions

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.

8.1 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.

Note: Code referenced in this section is available on Github.

8.1.1 Unit testing

Adjust the `DEPSWRITER` variable in the calcdeps.sh script and run it in the prestans-client directory.

```
cd prestans-client
./calcdeps.sh
```

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

```
/Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome --allow-file-access-from-files
```

8.2 Extending JavaScript namespaces

Types & 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 losing 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 `User.js` in directory `pdemo/data/extension`, this will provide the namespace `pdemo.data.extension.User` and depend on `pdemo.data.model.User`.

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

# Closure will ensure that the namespace pdemo.data.extension.User
# 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.extension.User');
```

8.3 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 & Responses.

Note: prestans.types.Integer only support integers in signed 32 bit range as anything outside this range does not work correctly with JavaScript bitwise operators.

8.3.1 Array

`prestans.types.Array` extends `goog.iter.Iterator`, allowing you to use the methods from goog.iter including:

- `goog.iter.filter`
- `goog.iter.forEach`
- `goog.iter.limit`

An array takes the following object as its constructor.

```
{
    elementTemplate: Subclass of prestans.types.Model or instance of prestans.types.Integer, prestans
    opt_elements: Array of elements to append to the array,
    opt_json: Array of json elements to append to the array,
    opt_minified: Whether or not the json has been minified,
    opt_maxLength: An integer value representing the maximum length of the array,
    opt_minLength: An integer value representing the minimum length of the array
}
```

Prestans provides wrappers for the following Google closure goog.array methods:

- `isEmpty` checks to see if an Array is empty returns a `Boolean`
- `binarySearch(target, opt_compareFn)` performs a binary search for an object returns an index `Number`
- `binaryInsert(value, opt_compareFn)` performs a binary insert and returns a `Boolean`
- `binaryRemove(value, opt_compareFn)` performs a binary remove and returns a `Boolean`
- `insertAt(obj, opt_i)` inserts and object at the given index
- `indexOf(obj, opt_fromIndex)` returns the index as a `Number` for a particular object
- `removeAt(i)` removes an object at a particular index and returns a `Boolean`
- `removeIf(f, opt_obj)` removes an object if the supplied function returns `true` and returns a `Boolean`
- `remove(obj)` removes the provided object from the collection and returns a `Boolean`
- `sort(opt_compareFn)` performs a sort based on a comparison function
- `clear` clears the contents of an array
- `find(f, opt_obj)` performs a fund using a user provided function, returns the `Element` or `null`
- `slice(start, opt_end)` returns an `prestans.types.Array` which is a portion of the original `Array`
- `contains(obj)` returns a `Boolean` to see if the collection contains a particular `object`

Prestans then provides the following additional methods:

- `getMinLength` returns a `Number`
- `getMaxLength` returns a `Number`
- `append (element)` returns a `Boolean`
- `insertAfter(newValue, existingValue)` returns a `Boolean`
- `length` returns a `Number`
- `containsIf(condition, opt_context)` returns an `Element|null`
- `objectAtIndex(index)` returns an `Element`
- `asArray` returns an `Array`
- `clone` returns an `prestans.types.Array`
- `getJSONObject` returns an `Object`
- `getJSONString` returns a `String`

8.3.2 Attribute Change Events

Models generated by Prestans raise the `prestans.types.Model.EventType.ATTRIBUTE_CHANGED` event whenever a mutator is fired. You can listen for this event on any instance of a Prestans Model subclass.

`event.getIdentifier()` provides you a camel cased representation of the attribute that changed:

```
var album_ = new pdemo.data.model.Album();

// Use the Event Handler provided by a Google Closure Component
this.getHandler().listen(album_, prestans.types.Model.EventType.ATTRIBUTE_CHANGED, function(event){

    if(event.getIdentifier() == "name") {
```

```
        // name changed, I might update the user interface
    }

});
```

8.3.3 Generating Model Code

Based on the server model definition Prestans can generate JavaScript versions of your models. You can use this in tandem with the Xhr client to ensure the data you receive from is intact and matches the business rules (hence ensuring nothing went wrong on the way), and that the data you send to the server will pass the validation test.

The generated model code centralizes references to JSON keys allowing you to access the data via accessors and mutators (which also validate the data) allowing Closure Compiler to optimise your code.

Prestans provides a utility called pride (Prestans Integrated Development Environment) which resides in */usr/local/bin* and is responsible for generating the Model stubs. You provide it a reference to the Python file that contains your Model definitions along with target name spaces and paths and for each server Model definition it produces a JavaScript Model or Attribute Filter.

To generate models using pride use the following command:

```
pride gen --template closure.model --model pdemo/rest/models.py --namespace pdemo.data.model --output
```

Prestans assumes that your filters files live in the same level as models i.e `pdemo.data.model` corresponds to `pdemo.data.filter`, you can optionally provide a `--filter-namespace` to override the default filter namespace

To generate filters using pride use the following command:

```
pride gen --template closure.filter --model pdemo/rest/models.py --namespace pdemo.data.filter --outp
```

8.4 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

8.4.1 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({
        baseUrl: "/api",
        opt_numRetries: 0,
        opt_minified: true
    })
};
```

Note: Minification support is built into our Xhr client. All you have to do is set the optional parameter to `true` and the client negotiates with the server.

8.4.2 Composing a Request

To place an Xhr request you compose a request by instantiating a `prestans.rest.Request` object, it accepts the following parameters as a JSON configuration:

- `identifier` unique string identifier for this request type, these are used to cancel requests
- `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`

Example of a GET request which optionally passes two parameters and expects the server to return a set of `Album` entities:

```
var config_ = {
    identifier: "AlbumSearchFetch",
    httpMethod: prestans.net.HttpMethod.GET,
    responseFilter: opt_filter,
    responseModel: pdemo.data.model.Album,
    isArray: true,
    urlFormat: "/album",
    parameters: [
        {
            key: "search_text",
            value: searchText
        },
        {
            key: "limit",
            value: limit
        }
    ]
};

var request = prestans.rest.json.Request(config_);
```

Example of a GET request which fetches a particular entity:

```
// Optional filter to turn off all attributes bar Album Id and Name
var opt_filter = new pdemo.data.filter.Album(false);
opt_filter.enableName();
opt_filter.enableAlbumId();

var config_ = {
    identifier: "AlbumFetch",
    httpMethod: prestans.net.HttpMethod.GET,
    responseFilter: opt_filter,
    responseModel: pdemo.data.model.Album,
    isArray: false,
    urlFormat: "/album/%i",
    urlArgs: [albumId]
};
```

```
var request = prestans.rest.json.Request(config_);
```

8.4.3 Dispatching a Request

Once you have a request object use the `dispatchRequest` 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`, optionally asks the Request Manager to cancel all pending requests.

```
# Assume you have a request object
pdemo.GLOBALS.API_CLIENT.dispatchRequest(
    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.

8.4.4 Working with Responses

The Xhr calls back the nominated functions on success and failure. Prestans passes an instance of `prestans.rest.json.Response` which has access to th following:

- `requestIdentifier` The string identifier for the request type,
- `statusCode` HTTP status code,
- `responseModel` Class used to unpack response body,
- `arrayElementTemplate` prestans.types.Model subclass
- `responseModelElementTemplates`
- `responseBody` JSON Object (Optional), this won't be available for failures

In case of a successful request you will be able to retrieve the parsed Prestans JavaScript model using the `getUnpackedBody()` method.

```
/**
 * @private
 */
pdemo.ui.album.Renderer.prototype.successCallback_ = function(response) {
```

```
    // response.getUnpackedBody() will return a parsed Prestans model
    // based on the rules you defined in the request
    console.log(response.getUnpackedBody());

};
```

In case of failures you get access to the response which has the status code and the original body (if any) of the response. For failures you are expected to do what your application deems appropriate.

8.4.5 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
```

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.

CHAPTER 9

Thoughts on API design

Prestans was a result of our careful study into the REST standards, popular frameworks and appraoches. Following are useful lessons we've learnt along the way. We've also compiled an extensive list of extremely useful Reference Material we found during our research.

9.1 REST resources are *not* persistent models

The word entity traditionally referred to persistent objects. That concept applied on the Web results in a HTTP implementation of approaches like XML-RPC. Many frameworks implement REST as a gateway to directly access persistent object on the server.

Entity in the REST world refers to entities returned by REST service; not what's stored in the persistence layer. In most instances it refers to an application specific view of the persistent data; that the requesting client will find most useful.

REST endpoints form the business logic layer of your Ajax/Mobile application. REST services should do the heavy lifting and return the most useful view of the data for the use-case; this may include related data i.e Order may include Order line.

Server round trips (or latency) is one of biggest performance overheads for REST services.

9.2 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 separate handlers.

9.3 Response Size

Running local development HTTP servers is common practice. Due to negligible latency; local server are notorious for masking server round trip issues, specially ones related to response size. Only when you've deployed your application to a remote server can you judge how long noticing how long round trips take.

Database, Web Servers, Prestans your handlers, servers are generally pretty quick (if you have written most things well). Network latency caused by the sheer size of the response can make your REST services appear to be slow.

It's important not to loose sight of the response size written out by your REST services.

CHAPTER 10

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.

10.1 WSGI & REST

- 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 (YouTube).
- What every developer should know about REST - Michael Mahemoff's Web Directions Code 2013 presentation on REST (YouTube).
- Thoughts on RESTfulAPI Design - by Geert Jansen

10.2 HTTP

- Unacceptable Browser HTTP Accept Headers - Kris Jordan's comprehensive article on how different browser interpret the Accept header.
- The Accept Header - Chris Shifflett describes the Accept Header.
- A Beginner's Guide to HTTP Cache Headers - Kyle Young's excellent description of HTTP cache headers.
- Know your HTTP headers well - Andrei Neculau's guide to the HTTP protocol

10.3 Advanced Python

- Python WSGI reference implementation - available in Python 2.5
- Effbot - (A Semi-Official) Python FAQ Zone
- 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.
- Python Regular Expressions - Google Developer article on regular expressions.
- Regular Expressions by Example - specific regular expression examples.
- 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.
- An Intro to logging - learn about how to use and extend the Python logging feature.
- Python Style Guide - PEP 0008 standards on naming stuff in Python
- Getting Stated with WSGI - Armin Ronarcher's introduction to WSGI.

Implementation specific posts:

- Faux function type signatures in Python - using a Python decorator to ensure that your functions get values in the right type from WSGI calls. Originally posted as a response on Stackoverflow.

Frameworks:

- Another Do-It-Yourself Framework

10.4 Serialization format

- YAML ain't a markup language - Jess Noller talks about YAML.

10.5 Server 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.
- Werkzeug - The Python WSGI Utility Library (including a development HTTP server)
- MongoDB - MongoDB (from "humongous") is a scalable, high-performance, open source NoSQL database. Written in C++.

10.6 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 11

Feedback

If you find any inconsistencies in our documentation please feel free to lodge an issue via our Issue Tracker on Github. Or Tweet at us @anomalymade.

CHAPTER 12

Discussions

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.

CHAPTER 13

License

Sphinx Source available on GitHub.

P

R