Thoughts on RESTful API Design

Lessons learnt from designing the Red Hat Enterprise Virtualization API

Author: Geert Jansen <gjansen@redhat.com>

Date: November 15th, 2012

The source of this document can be found on github.

Introduction

This essay is an attempt to put down my thoughts on how to design a real-world yet beautiful RESTful API. It draws from the experience I have gained being involved in the design of theRESTful API for Red Hat's Enterprise Virtualization product, twice. During the design phase of the API we had to solve many of the real-world problems described above, but we weren't willing to add non-RESTful or "RPC-like" interfaces to our API too easily.

In my definition, a real-world RESTful API is an API that provides answers to questions that you won't find in introductory texts, but that inevitably surface in the real world, such as whether or not resources should be described formally, how to create useful and automatic command-line interfaces, how to do polling, asynchronous and other non-standard types of requests, and how to deal with operations that have no good RESTful mapping.

A beautiful RESTful API on the other hand is one that does not deviate from the principles of RESTful architecture style too easily. One important design element for example that is not always addressed is the possibility for complete auto-discovery by the API user, so that the API can be used by a human with a web browser, without any reference to external documentation. I will discuss this issue in detail in *Forms*.

This essay does not attempt to explain the basics about REST, as there are many other texts about that. For a good description of REST, I would recommend reading at least the Atom Publishing Protocol and this blog post by Roy Fielding.

While the views exposed in this essay are my own, they are heavily based on the public discussion on the rhevm-api mailing list. My thanks goes to all people who have contributed and are still contributing to it, including Mark McLoughlin, Michael Pasternak, Ori Liel, Sander Hoentjen, Ewoud Kohl van Wijngaarden, Tomas V.V. Cox and everybody else I forgot.

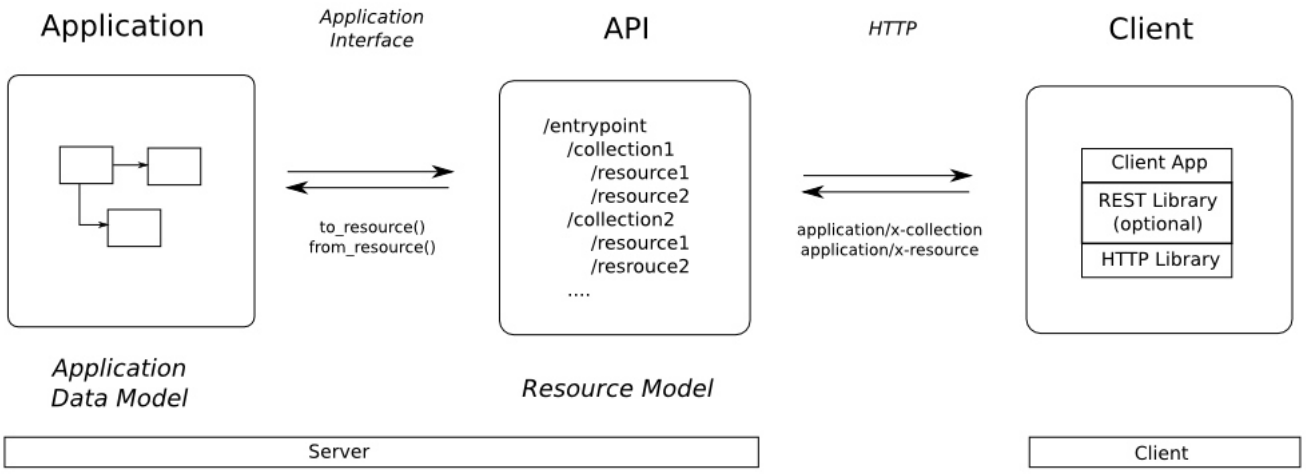
The Job of the API Designer

Before we dive fully into RESTful API design, it makes sense to look in a little more detail what the job of a RESTful API designer is.

APIs don't exist in isolation. Instead, APIs expose functionality of an application or service that exists independently of the API. In my view, the job of the RESTful API designer is two-fold.

1. Understanding enough of the important details of the application for which an API is to be created, so that an informed decision can be made as to what functionality needs to be exposed, how it needs to be exposed, and what functionality can instead be left out.
2. Modeling this functionality in an API that addresses all use cases that come up in the real world, following the RESTful principles as closely as possible.

There are three distinct components involved in RESTful API design: the application, the API code, and the client. The image above illustrates how these three components interact.



The Application

The application for which an API is to be provided exists independently of that API. Maybe the application has a GUI, and you have a requirement to add a programmatic interface to it. Or maybe the application was designed under the assumption that it would be accessed only via the API you are designing.

Like any application, the application for which the API is to be created contains state.

That state is dynamic, and will change due to various operations that are executed on it. This state, and the operations on it, need to be modeled and exposed, and will form the API you are designing.

The easiest way to think about the application state, is to assume that it is is described by an*application data model*, which can be described by an *Entity-Relationship diagram*. The ER-Diagram lists the details of the *entities* (I will call them *objects*) in the application state, and the *relationships* between them.

In some cases, it is very easy to create the ER diagram. For example, in case of a web application that stores all its state in a database, it can be trivially created from the schema of the database. In other cases where there is not such a precise definition, the job of the API designer becomes a little more difficult. In such a case it would actually make sense to create an ER diagram for the application in question. That is a useful exercise in its own right, as it will make you better understand the application you're designing the API for. But more importantly, it will also help you in designing a good RESTful API. We will talk more about that in a minute. Going forward, I will assume that an ER diagram is available.

In addition to understanding the application data model, and the operations on it, you of course need an entry point into the application that allows you to access and change the application state. This "way in" is fully application dependent and could take many forms. We will call this "way in" the *application interface*. Formally, this application interface could be considered an API as well. The difference, though, is that this interface is usually not intended for external consumption or even fully documented. In order not to introduce any confusion, we will not refer to this interface as an API: that term will be reserved for the RESTful API that is being designed.

The API Code

The job of the API code is to access the application state, as well as the operations on that state, via the application interface, and expose it as a RESTful API. In between the application interface and the RESTful API, there is a transformation step that adapts the application data model and makes it comply with the RESTful architecture style.

The result of this transformation would be RESTful *resources*, operations on those resources, and relationships between the resources. All of these are described by what we call the RESTful *resource model*.

Resources are the foundation behind any RESTful API, and we will go into a lot of detail on them in *Resources*. For now, just think of resources as being very similar to the entities from the ER diagram (which is why I encouraged you to create an ER diagram for your application in case it didn't exist).

Relationships between resources are expressed as hyperlinks in the representation of the resource. This is one of the fundamental principles of RESTful API design. Resources also respond to a very limited set of operations (usually just 4), which is a second principle of

the RESTful architectural style.

When transforming objects from the application data model to RESTful resources, you may find it useful to define two utility functions:

`to_resource`()

This function is assumed to take an object from the application data model, and convert it into a resource.

`from_resource`()

This function is assumed to take a resource, and translate it into an object in the application data model.

We don't discuss these methods further, other than mentioning that these can be rather simple functions if the application data model is similar to the resource model you're exposing, and can be quite complicated if they are very different.

The Client

The client consumes the RESTful API via the standard HTTP protocol. In theory, the service could be provided on top of other protocols as well. Since HTTP is so ubiquitous, however, it is not certain how valuable such a mapping to another protocol would be in the real world. Therefore, this essay limits itself to describing our RESTful protocol in terms of HTTP.

Clients would typically use an HTTP library to access the RESTful API. HTTP has become a moderately complex protocol, and very good libraries exist for many target platforms / languages. It therefore makes a lot of sense to use one of those libraries.

In some cases, it may make sense to use a generic REST library on top of an HTTP library. However, since there are so many different conventions in RESTful APIs, this library may actually be specific to the API that you are consuming.

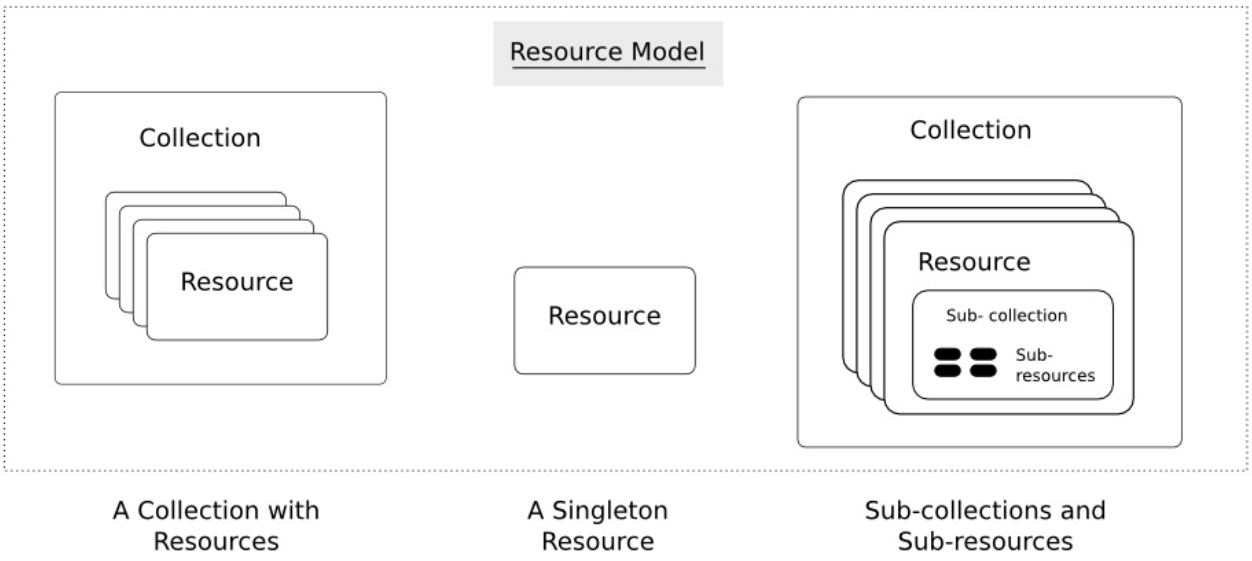
Resources

The fundamental concept in any RESTful API is the *resource*. A resource is an object with a type, associated data, relationships to other resources, and a set of methods that operate on it. It is similar to an object instance in an object-oriented programming language, with the important difference that only a few standard methods are defined for the resource (corresponding to the standard HTTP GET, POST, PUT and DELETE methods), while an object instance typically has many methods.

Resources can be grouped into *collections*. Each collection is homogeneous so that it contains only one type of resource, and unordered. Resources can also exist outside any collection. In this case, we refer to these resources as *singleton resources*. Collections are themselves resources as well.

Collections can exist globally, at the top level of an API, but can also be contained inside a single resource. In the latter case, we refer to these collections as *sub-collections*. Sub-collections are usually used to express some kind of "contained in" relationship. We go into more detail on this in *Relationships*.

The diagram below illustrates the key concepts in a RESTful API.



We call information that describes available resources types, their behavior, and their relationships the *resource model* of an API. The resource model can be viewed as the RESTful mapping of the application data model.

Resource Data

Resources have data associated with them. The richness of data that can be associated

with a resource is part of the *resource model* for an API. It defines for example the available data types and their behavior.

Based on my experience, I have developed a strong conviction that the JSON data model has just the right "richness" so that it is an ideal data model for RESTful resources. I would recommend that everybody use it.

In JSON, just three types of data exist:

- scalar (number, string, boolean, null).
- array
- object

Scalar types have just a single value. Arrays contain an ordered list of values of arbitrary type. Objects consist of a unordered set of key:value pairs (also called attributes, not to be confused with XML attributes), where the key is a string and the value can have an arbitrary type. For more detailed information on JSON, see the JSON web site.

Why the strong preference for JSON? In my view, JSON has the right balance between expressiveness, and broad availability. The three types of data (scalars, arrays and objects) are powerful enough to describe in a natural way virtually all the data that you might want to expose as resource, and at the same time these types are minimal enough so that almost any modern language has built-in support for them.

XML would be the other obvious contender. Actually, in the final incarnation of the RHEV-M API, XML is used to describe resources, via an XMLSchema definition. With hindsight, I believe that the XML data model is a bad choice for a RESTful API. On one side, it is too rich, and on the other side, it lacks features. XML, as an SGML off-shoot, is in my view great for representing structured documents, but not for representing structured data.

Features of XML that are too rich include:

1. Attributes vs elements. An XML element can have both attributes as well as sub-elements. A data item associated with a resource could be encoded in either one, and it would not be clear beforehand which one a client or a server should use.
2. Relevance of order. The order between child-elements is relevant in XML. It is not natural in my view for object attributes to have ordering.

The limitations of the XML data model are:

1. Lack of types. Elements in XML documents do not have types, and in order to use types, one would have to use e.g. XMLSchema. XMLSchema unfortunately is a strong contender for the most convoluted specification ever written.
2. Lack of lists. XML cannot natively express lists. This can lead to issues whereby it is not clear whether a certain element is supposed to be a list or an object, and where that element ends up being both.

Application Data

We define the data that can be associated with a resource in terms of the JSON data model, using the following mapping rules:

1. Resources are modeled as a JSON object. The type of the resource is stored under the special key:value pair "_type".
2. Data associated with a resource is modeled as key:value pairs on the JSON object. To prevent naming conflicts with internal key:value pairs, keys must not start with "_".
3. The values of key:value pairs use any of the native JSON data types of string, number, boolean, null, or arrays thereof. Values can also be objects, in which case they are modeling nested resources.
4. Collections are modeled as an array of objects.

We will also refer to key:value pairs as attributes of the JSON object, and we will be sloppy and use that same term for data items associated with resources, too. This use of attributes is not to be confused with XML attributes.

REST Metadata

In addition to exposing application data, resources also include other information that is specific to the RESTful API. Such information includes URLs and relationships.

The following table lists generic attributes that are defined and have a specific meaning on all resources. They should not be used for mapping application model attributes.

Attribute	Type	Meaning
id	String	Identifies the unique ID of a resource.
href	String	Identifies the URL of the current resource.
link	Object	Identifies a relationship for a resource. This attribute is itself an object and has "rel" "href" attributes.

Other Data

Apart from application data, and REST metadata, sometimes other data is required as well. This is usually "RPC like" data where a setting is needed for an operation, and where that setting will not end up being part of the resource itself.

One example that I can give here is where a resource creation needs a reference to another resource that is used during the creation, but where that other resource will not become part of the resource itself.

It is the responsibility of the API code to merge the application data together with the REST metadata and the other data into a single resource, resolving possible naming conflicts that may arise.

Representations

We have defined resources, and defined the data associated with them in terms of the JSON data model. However, these resources are still abstract entities. Before they can be communicated to a client over an HTTP connection, they need to be serialized to a textual representation. This representation can then be included as an entity in an HTTP message body.

The following representations are common for resources. The table also lists the appropriate content-type to use:

Type	Content-Type
JSON	application/x-resource+json application/x-collection+json
YAML	application/x-resource+yaml application/x-collection+yaml
XML	application/x-resource+xml application/x-collection+xml
HTML	text/html

Note: all these content types use the "x-" experimental prefix that is allowed by RFC2046.

JSON Format

Formatting a resource to JSON is trivial because the data model of a resource is defined in terms of the JSON model. Below we give an example of a JSON serialization of a virtual machine:

```
{
  "_type": "vm",
  "name": "A virtual machine",
  "memory": 1024,
  "cpu": {
    "cores": 4,
    "speed": 3600
  },
  "boot": {
    "devices": ["cdrom", "harddisk"]
  }
}
```

YAML Format

Formatting to YAML is only slightly different than representing a resource in JSON. The resource type that is stored under the "_type" key/value pair is serialized as a YAML "!type" annotation instead. The same virtual machine as above, now in YAML format:

```
!vm
name: A virtual machine
memory: 1024
cpu:
  cores: 4
  speed: 3600
boot:
  devices:
  - cdrom
  - harddisk
```

XML Format

XML is the most complex representation format due to both its complexity as well as its limitations. I recommend the following rules:

- Resources are mapped to XML elements with a tag name equal to the resource type.
- Attributes of resources are mapped to XML child elements with the tag name equal to the attribute name.
- Scalar values are stored as text nodes. A special "type" attribute on the containing element should be used to refer to an XML Schema Part 2 type definition.
- Lists should be stored as a single container element with child elements for each list item. The tag of the container element should be the English plural of the attribute name. The item tag should be the English singular of the attribute name. Lists should have the "xd:list" type annotation.

The same VM again, now in XML:

```
<vm xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <name type="xs:string">My VM</name>
  <memory type="xs:int">1024</memory>
  <cpu>
    <cores type="xs:int">4</cores>
    <speed type="xs:int">3600</speed>
  </cpu>
  <boot>
    <devices type="xs:list">
```

```
      <device type="xs:string">cdrom</device>
      <device type="xs:string">harddisk</device>
    </devices>
  </boot>
</vm>
```

HTML Format

The exact format of a HTML response can be API dependent. HTML is for human consumption, and the only requirement is therefore that it be easy to understand. A simple implementation may choose the following representation:

- For a collection, a <table> with a column per attribute where each object is listed in a separate row.
- For a resource, a <table> with two columns, one with all the attribute names, one with the corresponding attribute value.

Content-Types

As can be seen above, I am advocating the use of a generic content types "application/x-resource+format" and "application/x-collection+format". In my view this represents the right middle ground between two extremes that are commonly found in RESTful APIs:

Some RESTful APIs only use the "bare" XML, JSON or YAML content types. An example of such as API is the Red Hat Enterprise Virtualization API. In this case, the content type expresses nothing but the fact that an entity is in XML, JSON or YAML format. In my view, this is not sufficient. Resources and collections have some specific semantics around for example the use of "href" attributes, "link" attributes, and types. Therefore, these are a specialization of XML, JSON and YAML and should be defined as such.

Other RESTful APIs define a content-type for every resource type that exists in the resource model. Examples of such APIs include for example VMware's vSphere Director API. In my view, this is not proper either. Specifying detailed content types invites both the API implementer, as well as a client implementer to think about these types as having specific interfaces. In my view though, all resources should share the same basic interface, which is defined by the RESTful design principles and expressed by the "application/x-resource" content type.

One reason that is sometimes given in favor of defining detailed content types is that this way, the content type can be associated with a specific definition in some type definition language (such as XMLSchema). This, supposedly, facilitates client auto-discovery because a client can know available attributes for a certain type. I go into a lot of detail on this topic in *Forms* but the summary is that I do not agree with this argument.

Selecting a Representation Format

Clients can express their preference for a certain representation format using the HTTP "Accept" header. The HTTP RFC defines an elaborate set of rules in which multiple formats can be requested, each with its own priority. In the following example, the client tells the API that it accepts only YAML input:

```
GET /api/collection
Accept: application/x-collection+yaml
```

URLs

Entry Point

A RESTful API needs to have one and exactly one entry point. The URL of the entry point needs to be communicated to API clients so that they can find the API.

Technically speaking, the entry point can be seen as a singleton resource that exists outside any collection. It is common for the entry point to contain some or all of the following information:

- Information on API version, supported features, etc.
- A list of top-level collections.
- A list of singleton resources.
- Any other information that the API designer deemed useful, for example a small summary of operating status, statistics, etc.

URL Structure

Each collection and resource in the API has its own URL. URLs should never be constructed by an API client. Instead, the client should only follow links that are generated by the API itself.

The recommended convention for URLs is to use alternate collection / resource path segments, relative to the API entry point. This is best described by example. The table below uses the ":name" URL variable style from Rail's "Routes" implementation.

URL	Description
/api	The API entry point
/api/:coll	A top-level collection named "coll"
/api/:coll/:id	The resource "id" inside collection "coll"
/api/:coll/:id/:subcoll	Sub-collection "subcoll" under resource "id"
/api/:coll/:id/:subcoll/:subid	The resource "subid" inside "subcoll"

Even though sub-collections may be arbitrarily nested, in my experience, you want to keep the depth limited to 2, if possible. Longer URLs are more difficult to work with when using simple command-line tools like curl.

Relative vs Absolute

It is strongly recommended that the URLs generated by the API should be absolute URLs.

The reason is mostly for ease of use by the client, so that it never has to find out the

correct base URI for a resource, which is needed to interpret a relative URL. The URL RFC specifies the algorithm for determining a base URL, which is rather complex. One of the options for finding the base URL is to use the URL of the resource that was requested. Since a resource may appear under multiple URLs (for example, as part of a collection, or stand-alone), it would be a significant overhead to a client to remember where he retrieved a representation from. By using absolute URLs, this problem doesn't present itself.

URL Templates

A draft standard for URL templates exists. URL templates can be useful in link attributes where the target URL takes query arguments. It is recommended though to make only conservative use of these. So far the only good use case I have come across is when searching in a collection. In that case, it seems fair that the search criteria can be specified as GET style query arguments appended to the collection URL.

I would recommend to only use the "literal expansion" part of the URL templates spec, because the "expression expansion" part adds too much complexity to a client in my view, for very little benefit.

Variants

Sometimes you need to work on a variant of a resource. In our RHEV-M API for example, some attributes on a virtual machine can be updated while a virtual machine is running. This amounts to a hot plug/unplug of the resource, which is a different operation altogether from changing the saved representation. A nice way to implement this is using ";variant" identifier in a URL. For example:

URL	Description
/api/:coll/:id;saved	Identifies the saved variant of a resource.
/api/:coll/:id;current	Identifies the current variant of a resource.

The use of the semicolon to provide options that are specific to a path segment is explicitly alowed in RFC3986 The advantage over using a "?variant" query argument is that this format is specific to a path segment only, and would allow you to e.g. work on the saved variant of a sub-resource in a sub-collection of the current variant of another resource.

Methods

Standard Methods

We already discussed that resources are the fundamental concept in a RESTful API, and that each resource has its own unique URL. Methods can be executed on resources via their URL.

The table below lists the standard methods that have a well-defined meaning for all resources and collections.

Method	Scope	Semantics
GET	collection	Retrieve all resources in a collection
GET	resource	Retrieve a single resource
HEAD	collection	Retrieve all resources in a collection (header only)
HEAD	resource	Retrieve a single resource (header only)
POST	collection	Create a new resource in a collection
PUT	resource	Update a resource
PATCH	resource	Update a resource
DELETE	resource	Delete a resource
OPTIONS	any	Return available HTTP methods and other options

Normally, not all resources and collections implement all methods. There are two ways to find out which methods are accepted by a resource or collection.

1. Use the OPTIONS method on the URL, and look at the "Allow" header that is returned. This header contains a comma-separated list of methods are are supported for the resource or collection.
2. Just issue the method you want to issue, but be prepared for a "405 Method Not Allowed" response that indicates the method is not accepted for this resource or collection.

Actions

Sometimes, it is required to expose an operation in the API that inherently is non RESTful. One example of such an operation is where you want to introduce a state change for a resource, but there are multiple ways in which the same final state can be achieved, and those ways actually differ in a significant but non-observable side-effect. Some may say such transitions are bad API design, but not having to model all state can greatly simplify an API. A great example of this is the difference between a "power off" and a "shutdown" of a virtual machine. Both will lead to a vm resource in the "DOWN" state. However, these operations are quite different.

As a solution to such non-RESTful operations, an "actions" sub-collection can be used on

a resource. Actions are basically RPC-like messages to a resource to perform a certain operation. The "actions" sub-collection can be seen as a command queue to which new action can be POSTed, that are then executed by the API. Each action resource that is POSTed, should have a "type" attribute that indicates the type of action to be performed, and can have arbitrary other attributes that parameterize the operation.

It should be noted that actions should only be used as an exception, when there's a good reason that an operation cannot be mapped to one of the standard RESTful methods. If an API has too many actions, then that's an indication that either it was designed with an RPC viewpoint rather than using RESTful principles, or that the API in question is naturally a better fit for an RPC type model.

PATCH vs PUT

The HTTP RFC specifies that PUT must take a full new resource representation as the request entity. This means that if for example only certain attributes are provided, those should be removed (i.e. set to null).

An additional method called PATCH has been proposed recently. The semantics of this call are like PUT inthat it updates a resource, but unlike PUT, it applies a delta rather than replacing the entire resource. At the time of writing, PATCH was still a proposed standard waiting final approval.

For simple resource representations, the difference is often not important, and many APIs simply implement PUT as a synonym for PATCH. This usually doesn't give any problems because it is not very common that you need to set an attribute to null, and if you need to, you can always explicitly include it.

However for more complex representations, especially including lists, it becomes very important to be able to express accurately the changes you want to make. Therefore, it is my recommendation now to both provide PATCH and PUT, and make PATCH do an relative update and have PUT replace the entire resource.

It is important to realize that the request entity to PATCH is of a different content-type than the entity that it is modifying. Instead of being a full resource, it is a resource that describes modifications to be made to a resource. For a JSON data model, which is what this essay is advocating, I believe that there are two sensible ways to define the patch format.

1. An informal approach where you accept a dict with a partial representation of the object. Only attributes that are present are updated. Attributes that are not present are left alone. This approach is simple, but it has the drawback that if the resource has a complex internal structure e.g. containing a big list of dicts, then that entire list of dicts need to be given in the entity. Effectively PATCH becomes similar to PUT again.
2. A more formal approach would be to accept a list of modifications. Each

modification can be a dict specifying the JSON path of the node to modify, the modification ('add', 'remove', 'change') and the new value.

Link Headers

An Internet draft exists defining the "Link:" header type. This header takes the "link" attributes of the response entity, and formats them as an HTTP header. It is argued that the Link header is useful because it allows a client to quickly get links from a response without having to parse the response entity, or even without retrieving the response entity at all using the HEAD HTTP method.

In my view, the usefulness of this feature is dubious. First of all, it can increase response sizes quite significantly. Second, it can only be used when a resource is being returned; it does not make sense to be used with collections. Because I have not yet seen any good use of this header in the context of a RESTful API, I recommend not to implement Link headers.

Asynchronous Requests

Sometimes an action takes too long to be completed in the context of a single HTTP request. In that case, a "202 Accepted" status can be returned to the client. Such a response should only be returned for POST, PUT, PATCH or DELETE.

The response entity of a 202 Accepted response should be a regular resource with only the information filled in that was available at the time the request was accepted. The resource should contain a "link" attribute that points to a status monitor that can be polled to get updated status information.

When polling the status monitor, it should return a "response" object with information on the current status of the asynchronous request. If the request is still in progress, such a response could look like this (in YAML):

```
!response
status: 202 Accepted
progress: 50%
```

If the call has finished, the response should include the same headers and response body had the request been fulfilled synchronously:

```
!response
status: 201 Created
headers:
  - name: content-type
    value: applicaton/x-resource+yaml
response: !!str
```

```
Response goes here
```

After the response has been retrieved once with a status that is not equal to "202 Accepted", the API code may garbage collect it and therefore clients should not assume it will continue to be available.

A client may request the server to modify its asynchronous behavior with the following "Expect" headers:

- "Expect: 200-ok/201-created/204-no-content" disables all asynchronous functionality. The server may return a "417 Expectation Failed" if it is not willing to wait for an operation to complete.
- "Expect: 202-accepted" explicitly request an asynchronous response. The server may return a "417 Expectation Failed" if it is not willing to perform the request asynchronously.

If no expectation is provided, client must be prepared to accept a 202 Accepted status for any request other than GET.

Ranges / Pagination

When collections contain many resources, it is quite a common requirement for a client to retrieve only a subset of the available resources. This can be implemented using the Range header with a "resource" range unit:

```
GET /api/collection
Range: resources=100-199
```

The above example would return resources 100 through 199 (inclusive).

Note that it is the responsibility of the API implementer to ensure a proper and preferably meaningful ordering can be guaranteed for the resources.

Servers should provide an "Accept-Ranges: resource" header to indicate to a client that they support resource-based range queries. This header should be provided in an OPTIONS response:

```
OPTIONS /api/collection HTTP/1.1

HTTP/1.1 200 OK
Accept-Ranges: resources
```

Notifications

Another common requirement is where a client wants to be notified immediately when

some kind of event happens.

Ideally, such a notification would be implemented using a call-out from the server to the client. However, there is no good portable standard to do this over HTTP, and it also breaks with network address translation and HTTP proxies. A second approach called busy-loop polling is horribly inefficient.

In my view, the best approach is what is is called “long polling”. In long polling, the client will retrieve a URL but the server will not generate a response yet. The client will wait for a configurable amount of time, until it will close the connection and reconnect. If the server becomes aware of an event that requires notification of clients, it can provide that event immediately to clients that are currently waiting.

Long polling should be disabled by default, and can be enabled by a client using an Expect header. For example, a client could long poll for new resources in a collection using a combination of long-polling and a resource-based range query:

```
GET /api/collection
Range: 100-
Expect: nonempty-response
```

In this case, resource “100” would be the last resource that was read, and the call is requesting the API to return at least one resource with an ID > 100.

Server implementers need to decide whether they want to implement long polling using one thread per waiting client, or one thread that uses multiplexed IO to wait for all clients. This is a trade-off to be made between ease of implementation and scalability (that said, threads are pretty cheap on modern operating systems).

Relationships

As we have seen in *Resources*, the resource is the fundamental unit in RESTful API design. Resources model objects from the application data model.

Resources do not exist in isolation, but have relationships to other other resources. Sometimes these relationships exist between the mapped objects in the application data model as well, sometimes they are specific to the RESTful resources.

One of the principles of the RESTful architecture style is that these relationships are expressed by hyperlinks to the representation of a resource.

In our resource model, we interpret any object with an "href" attribute as a hyperlink. The value of the href attribute contains an absolute URL that can be retrieved with GET. Using GET on a such a URL is guaranteed to be side-effect free.

Two types of hyperlinks are normally used:

1. Using "link" objects. This is a special object of type "link" with "href" and a "rel" attributes. The "rel" attribute value identifies the semantics of the relationship. The link object is borrowed from HTML.
2. Using any object with a "href" attribute. The object type defines the semantics of the relationship. I'll call these "object links," not to be confused with the "link objects" above.

Below is an example of a virtual machine representation in YAML that illustrates the two different ways in which a relationship can be expressed:

```
!vm
id: 123
href: /api/vms/123
link:
  rel: collection/nics
  href: /api/vms/123/nics
cluster:
  href: /api/clusters/456
```

The "link" object is used with rel="collection/nics" to refer to a sub-collection to the VM holding the virtual network interfaces. The cluster link instead points to the cluster containing this VM by means of a "cluster" object.

Note that the VM itself has a href attribute as well. This is called a "self" link, and it is a convention for a resource to provide such a link. That way, the object can later be easily re-retrieved or updated without keeping external state.

There are no absolute rules when to use link objects, and when not. That said, we have been using the rules below in the rhevm-api project with good success. They seem to be

a good trade-off between compactness (favoring object links) and the ability to understand links without understanding the specific resource model (favoring link objects).

- Link objects are used to express structural relationships in the API. So for example, the top-level collections, singleton resources and sub-collections (including actions) are all referenced using link objects.
- Object links are used to express semantic relationships from the application data model. In the example above, the vm to cluster link comes directly from the application data model and is therefore modeled as a link.

Note however that sub-collections can express both a semantic relationship, as well as a structural relationship. See for example the "collection/nics" sub-collection to a VM in the example above. In such a case, our convention has been to use link objects.

Standard Structural Relationships

It makes sense to define a few standard structural relationship names that can be relevant to many different APIs. In this way, client authors know what to expect and can write to some extent portable clients that work with multiple APIs. The table below contains a few of these relationship names.

Relationship	Semantics
collection/{name}	Link to a related collection {name}.
resource/{name}	Link to a related resource {name}.
form/{name}	Link to a related form {name}.

Here, {name} is replaced with a term that has meaning for the specific API. For example, a collection of virtual machines could be linked from the entry point of a virtualization API using rel="collection/vms".

Modeling Semantic Relationships

Semantic relationships can be modeled either by an object link, or by a sub-collection. I believe that choosing the right way to represent a sub-collection is important to get a consistent API experience for the client. I would advocate using the following rules:

1. In case of a 1:N relationship, where the target object is **existentially dependent** on the source object, I'd recommend to use a sub-collection. By "existentially dependent" I mean that a target object cannot exist without its source. In database parlance, this would be a FOREIGN KEY relationship with an ON DELETE CASCADE. An example of such a relationship are the NICs that are associated with a VM.
2. In case of a 1:N relationship, where there is data that is associated with the link,

I’d recommend to use a sub-collection. Note that we are talking about data here that is neither part of the source object, nor the target object. The resources in the sub-collection can hold the extra data. In this case, the data in the sub-resource would therefore be a merge of the data from the mapped object from the application data model, and link data.

3. In case of any other 1:N relationship, I’d recommend to use an object link.
4. In case of a N:M relationship, I’d recommend to use a sub-collection. The sub-collection can be defined initially to support the lookup direction that you most often need to follow. If you need to query both sides of the relationship often, then two sub-collections can be defined, one on each linked object.

Forms

In this section, a solution is proposed to a usability issue that is present in a lot of RESTful APIs today. This issue is that, without referring to external documentation, an API user does not know what data to provide to operations that take input. For example, the POST call is used to create a new resource in a collection, and an API user can find out with the OPTIONS call that a certain collection supports POST. However, he does not know what data is required to create the new resource. So far, most RESTful APIs require the user to refer to the API documentation to get that information. This violates an important RESTful principle that APIs should be self-descriptive.

As far as I can see, this usability issue impacts two important use cases:

1. A person is browsing the API, trying to use and learn it as he goes. The person can either be using a web browser, or a command-line tool like "curl".
2. The development of a command-line or graphical interface to a RESTful API. Without a proper description of required input types, knowledge about these has to be encoded explicitly in the client. This has the disadvantage that new features are not automatically exposed, and that there are strict requirements around not making backwards incompatible changes.

When we look at the web, this issue doesn't exist. Millions of users interact with all kinds of IT systems over the web at any given moment, and none of them has to resort to API documentation to understand how to do that. Interactions with IT systems on the web are all self-explanatory using hypertext, and input is guided by forms.

In fact, forms are what I see as the solution here. But before I go into that, let's look into two classes of solutions that have been proposed to address this issue, and why I think they actually do not solve it. The first is using a type definition language like XMLSchema to describe the input types, the second is using some kind of service description language like WADL.

Type Definition

As alluded to in *Resources*, some RESTful APIs use a type definition language to provide information about how to construct request entities. In my view this is a bad approach, and has the following issues:

1. It creates a tight coupling between servers and clients.
2. It still does not allow a plain web browser as an HTTP client.
3. It does not help in the automatic construction of CLIs.

Often, XMLSchema is used as the type definition language. This brings in a few more

issues:

1. XMLSchema is a horribly complicated specification on one side, but on the other side it can't properly solve the very common requirement of an unordered set of key:value pairs (Note that <xs:all> does not solve this. Its limitations make it almost useless.).
2. Type definitions in XMLSchema are context-free. This means that you cannot just define one type, you need to define one type for PUT, one for POST, etc. And since one XML element can only have one type, this means you're PUTing a different element that you'd POST or GET. This breaks that "homogeneous collection" principle of REST.

Service Definition

WADL is an approach to define a service description language for RESTful APIs. It tries to solve the problem by meticulously defining entry points and parameters to those. The problem I have with WADL, is that it feels very non-RESTful. I can't see a lot of difference between a method on a WADL resource, and an RPC entry point.

It would be possible to construct a more RESTful service description language. It would focus on mapping the RESTful concepts of resource, collection, relationships, and links. It would probably need a type definition language as well to define the constraints on types for each method (again, PUT may have different constraints than POST). There have been discussions in the RHEV-M project on this.

In the end though, this approach also feels wrong. What I think is needed is something that works like the web, where everything is self descriptive, and guided only by the actual URL flow the user is going through, not by reference to some external description.

Using Forms to Guide Input

The right solution to guide client input in my view is to use forms. Note that I'm using the word "forms" here in a generic sense, as an entity that guides user input, and not necessarily equal to an HTML form.

In my view, forms need to specify three pieces of information:

1. How to contact the target and format the input.
2. A list of all available input fields.
3. A list of constraints which which the input fields must comply.

HTML forms provide for #1 and #2, but not for #3 above. #3 is typically achieved using client-side Javascript. Because Javascript is not normally available to clients that use the API, we need to define another way to do validation. The approach I propose here is to

define a form definition language that captures the information #1 through #3. A form expressed in this language can be sent over to the client as-is, in case the client understands the form language. It can also be translated into an HTML form with Javascript in case the client is a web browser.

Form Definition Language

Our forms are defined as a special RESTful resource with type "form". This means they use the JSON data model, can be represented in JSON, YAML and XML according to the rules in*Resources*.

Forms have their own content type. This is listed in the table below:

Type	Content-Type
Form	application/x-form+json application/x-form+yaml application/x-form+xml

Rather than providing a formal definition, I will introduce the form definition language by using an example. Below is an example of a form that guides the the creation of a virtual machine. In YAML format:

```
!form
method: POST
action: {url}
type: vm
fields:
- name: name
  type: string
  regex: [a-zA-Z0-9]{5,32}
- name: description
  type: string
  maxlen: 128
- name: memory
  type: number
  min: 512
  max: 8192
- name: restart
  type: boolean
- name: priority
  type: number
  min: 0
  max: 100
constraints:
- sense: mandatory
```

```
  field: name
- sense: optional
  field: description
- sense: optional
  field: cpu.cores
- sense: optional
  field: cpu.sockets
- sense: optional
  exclusive: true
  constraints:
  - sense: mandatory
    field: highlyavailable
  - sense: optional
    field: priority
```

As can be seen the form consists of 3 parts: form metadata, field definitions, and constraints.

Form Metadata

The form metadata is very simple. The following attributes are defined:

Attribute	Description
method	The HTTP method to use. Can be GET, POST, PUT or DELETE.
url	The URL to submit this form to.
type	The type of the resource to submit.

Fields

The list of available fields are specified using the "fields" attribute. This should be a list of field definitions. Each field definition has the following attributes:

Attribute	Description
name	The field name. Should be in dotted-name notation.
type	One of "string", "number" or "boolean"
min	Field value must be greater than or equal to this (numbers)
max	Field value must be less than or equal to this (numbers)
minlen	Minimum field length (strings)
maxlen	Maximum field length (strings)
regex	Field value needs to match this regular expression (strings)
multiple	Boolean that indicates if multiple values are accepted (array).

Constraints

First we need to answer the question of what kind of constraints we want to express in

our form definition language. I will start by mentioning that in my view, it is impossible to express each and every constraint on the client side. Some constraints for example require access to other data (e.g. when creating relationships), are computationally intensive, or even unknown to the API designer because they are undocumented for the application the API is written for. So in my view we need to find a good subset that is useful, without making it too complicated and without worrying about the fact that some constraints possibly cannot not be expressed.

This leads me to define the following two kinds of constraints, which in my view are both useful, as well as sufficient for our purposes:

1. Constraints on individual data values.
2. Presence constraints on fields, i.e. whether a field is allowed or not allowed, and if allowed, whether it is mandatory.

The constraints on data values are useful because CLIs and GUIs could use this information to help a user input data. For example, depending on the type, a GUI could render a certain field a checkbox, a text box, or a dropdown list. For brevity, constraints on individual data values are defined as part of the field definition, and were discussed in the previous section.

Presence constraints are also useful, as they allow an API user to generate a concise synopsis on how to call a certain operation to be used e.g. in CLIs. Presence constraints specify which (combination of) input fields can and cannot exist. Each presence constraint has the following attributes:

Attribute	Description
sense	One of “mandatory” or “optional”
field	This constraint refers to a field.
constraints	This constraint is a group with nested contrainst.
exclusive	This is an exclusive group (groups only).

Either “field” or “constraints” has to be specified, but not both. A constraint that has the field attribute set is called a simple constraint. A constraint that has the constraints attribute set, is called a group.

Checking Constraints

Value constraints should be checked first, and should be checked only on non-null values.

After value constraints, the presence constraints should to be checked. This is a bit more complicated because the constraints are not only used for making sure that all mandatory fields exist, but also that no non-optional fields are present. The following algorithm should be used:

1. Start with an empty list called “referenced” that will collect all referenced fields.

2. Walk over all constraints in order. If a constraint is a group, you need to recurse into it, depth first, passing it the "referenced" list.
3. For every simple constraint that validates, add the field name to "referenced".
4. In exclusive groups, matching stops at the first matching sub-constraint, in which case the group matches. In non-exclusive groups, matching stops at the first non-matching sub-constraint, in which case the group does not match.
5. When matching a group, you need to backtrack to the previous value of "referenced" in case the group does not match.
6. A constraint only fails if it is mandatory and it is a top-level constraint. If a constraint fails, processing may stop.
7. When you've walked through all constraints, it is an error if there are fields that have a non-null value but are not in the referenced list.

Building the Request Entity

After all constraints have been satisfied, a client should build a request entity that it will pass in the body of the POST, PUT or DELETE method. In case the form was requested in JSON, YAML or XML format, it is assumed that the client is not a web browser, and the following applies:

First, a new resource is created of the type specified in the form metadata. Dotted field names should be interpreted as follows. Each dot creates a new object, and stores it under the name immediately left of the dot in its parent object. This means that the parent must be an object as well, which means it cannot correspond to a field definition with "multiple" set to "true" (which would make it a list). This resource is then represented in a format that the server supports, using the rules described in *Resources*.

If a client requested a "text/html" representation of the form, it is assumed that the client is a web browser, and we assume the form will be processed as a regular HTML form. In this case:

- An HTML <form> should be generated, with an appropriate <input> element for each field.
- The HTML form's "method" attribute should be set to POST, unconditionally. In case the RESTful form's method is not POST, the server should include a hidden input element with the name "_method" to indicate to the server the original method. (HTML does not support PUT or DELETE in a form).
- The form's "enctype" should be set to "multipart/form-data" or "application/x-www-form-urlencoded", as appropriate for the input elements.
- A hidden field called "_type" is generated that contains the value of the "type" attribute in the form metadata.
- The server may generate Javascript and include that in the HTML to check the value and presence constraints.

Linking to Forms

Forms are associated with resources and collections by link objects. The name of the link object defines the meaning of the form. The following standard forms names are defined:

Name	Scope	Description
form/search	collection	Form to search for resources
form/create	collection	Form to create a new resource
form/update	resource	Form to update a resource
form/delete	resource	Form to delete a resource

Miscellaneous Topics

Offline Help

A requirement that sometimes comes up is that to have an offline description of the API that can be used to generate offline help for a command-line interface (CLI) for that API. When connected to an API, all this information is of course available because of the self-descriptive nature of a RESTful API. But the request is to have that information offline as well, so that CLI users can get help without have to connect first.

In my view it is a bad idea to offer this offline help. It introduces a tight coupling between a client and a server, which will break the RESTful model. A server cannot be independently upgraded anymore from a client, and clients become tied to a specific server version. Also I doubt the usefulness of this feature as having to connect first to get help does not seem like a big issue to me.

That said, it is relatively straightforward to define a way that such an offline description can be facilitated.

One server-side change is required for this. For each collection, the API should implement a placeholder resource with a special ID (let's say "_", but it doesn't matter as long as it cannot be a valid ID), and some fake data. When a special HTTP Expect header is set, only this placeholder resource is returned when querying a collection.

Having the placeholder resources in place for every collection, the following procedure can be used by the client to retrieve all relevant metadata:

1. Retrieve the entry point of the API and store it in memory as a resource.
2. For every hyperlink in the entry point, fetch the target, and store it under the "target" property under the link object, recursively. When collections are retrieved, the special Expect header must be set.
3. For every hyperlink in the entry point, execute an OPTIONS call on the target, and store the resulting headers under the "options" property of the link object, recursively.
4. Serialize the resulting object to a file.

The result of this is basically a recursive dump of the "GET"-able part of an API with one placeholder resource in every collection. It will contain all metadata including forms, options, available resource types and collections, in a format that is very similar to the real on-line data. It can be used as an off-line store to generate any help that you would be able to generate online as well.

An API can store a version number on its entry point, so that clients can see when their offline cached copy expired and can therefore generate a new one.