RestFul Services:

**RestWebservice Advance**

**OPTIONS ::**

This method allows the client to

determine the options and/or requirements associated with a resource,

or **the capabilities of a server**, without implying a resource action

or initiating a resource retrieval.

Responses to this method are not cacheable.

If the Request-URI is an asterisk ("\*"), the OPTIONS request is

intended to apply to the server in general rather than to a specific

resource. Since a server's communication options typically depend on

the resource, the "\*" request is only useful as a "ping" or "no-op"

type of method; it does nothing beyond allowing the client to test

the capabilities of the server.

Example :

**I want to figure out which methods are supported on a resource.**

use the OPTIONS method for this:

OPTIONS /my/resource HTTP/1.1

Host: example.org

HTTP/1.1 200 OK

Allow: HEAD,GET,DELETE,OPTIONS

Even though it's not defined, this method CAN return a message body. It will return an ALLOW header, that returns all the methods the current resource is capable of handling.

- See more at: <http://restcookbook.com/HTTP%20Methods/options/#sthash.UTyaRz29.dpuf>

**When should we use PUT and when should we use POST?**

The HTTP methods POST and PUT aren't the HTTP equivalent of the CRUD's create and update. They both serve a different purpose. It's quite possible, valid and even preferred in some occasions, to use POST to create resources, or use PUT to update resources.

Use PUT when you can update a resource completely through a specific resource. For instance, if you know that an article resides at http://example.org/article/1234, you can PUT a new resource representation of this article directly through a PUT on this URL.

If you do not know the actual resource location, for instance, when you add a new article, but do not have any idea where to store it, you can POST it to an URL, and let the server decide the actual URL.

PUT /article/1234 HTTP/1.1

<article>

<title>red stapler</title>

<price currency="eur">12.50</price>

</article>

POST /articles HTTP/1.1

<article>

<title>blue stapler</title>

<price currency="eur">7.50</price>

</article>

HTTP/1.1 201 Created

Location: /articles/63636

As soon as you know the new resource location, you can use PUT again to do updates to the blue stapler article. But as said before: you CAN add new resources through PUT as well. The next example is perfectly valid if your API provides this functionality:

PUT /articles/green-stapler HTTP/1.1

<article>

<title>green stapler</title>

<price currency="eur">9.95</price>

</article>

HTTP/1.1 201 Created

Location: /articles/green-stapler

Here, the client decided on the actual resource URL.

PUT and POST are both unsafe methods. However, PUT is idempotent, while POST is not

- See more at: <http://restcookbook.com/HTTP%20Methods/put-vs-post/#sthash.PXvpWpqd.dpuf>

**When should we use the PATCH HTTP method?**

The HTTP methods PATCH can be used to update partial resources. For instance, when you only need to update one field of the resource, PUTting a complete resource representation might be cumbersome and utilizes more bandwidth

PATCH /user/jthijssen HTTP/1.1

<user>

<firstname>Joshua</firstname>

</user>

Also, the PUT method is idempotent. PUTting the same data multiple times to the same resource, should not result in different resources, while POSTing to the same resource can result in the creation of multiple resources.

### See also

* [RFC 5789 - HTTP PATCH](http://tools.ietf.org/html/rfc5789)

### Caveats

* PATCH is neither safe nor idempotent.
* An API implementing PATCH must patch atomically. It MUST not be possible that resources are half-patched when requested by a GET.
* **What are idempotent and/or safe methods?**
* Safe methods are HTTP methods that do not modify resources. For instance, using GET or HEAD on a resource URL, should NEVER change the resource. However, this is not completely true. It means: it won't change the resource representation. It is still possible, that safe methods do change things on a server or resource, but this should not reflect in a different representation.
* This means the following is incorrect, if this would actually delete the blogpost:
* GET /blog/1234/delete HTTP/1.1
* Safe methods are methods that can be cached, prefetched without any repercussions to the resource.
* Idempotent methods
* An idempotent HTTP method is a HTTP method that can be called many times without different outcomes. It would not matter if the method is called only once, or ten times over. The result should be the same. Again, this only applies to the result, not the resource itself. This still can be manipulated (like an update-timestamp, provided this information is not shared in the (current) resource representation.
* Consider the following examples:
* a = 4;
* a++;
* The first example is idempotent: no matter how many times we execute this statement, a will always be 4. The second example is not idempotent. Executing this 10 times will result in a different outcome as when running 5 times. Since both examples are changing the value of a, both are non-safe methods.
* Idempotency is important in building a fault-tolerant API. Suppose a client wants to update a resource through POST. Since POST is not a idempotent method, calling it multiple times can result in wrong updates. What would happen if you sent out the POST request to the server, but you get a timeout. Is the resource actually updated? Does the timeout happened during sending the request to the server, or the response to the client? Can we safely retry again, or do we need to figure out first what has happened with the resource? By using idempotent methods, we do not have to answer this question, but we can safely resend the request until we actually get a response back from the server.
* Be careful when dealing with safe methods as well: if a seemingly safe method like GET will change a resource, it might be possible that any middleware client proxy systems between you and the server, will cache this response. Another client who wants to change this resource through the same URL (like: http://example.org/api/article/1234/delete), will not call the server, but return the information directly from the cache. Non-safe (and non-idempotent) methods will never be cached by any middleware proxies
* Overview of (some) HTTP methods

|  |  |  |
| --- | --- | --- |
| HTTP Method | Idempotent | safe |
| OPTIONS | YES | YES |
| GET | YES | YES |
| HEAD | YES | YES |
| POST | NO | NO |
| PUT | YES | NO |
| DELETE | YES | NO |
| PATCH | NO | NO |

**Are REST and HTTP the same thing?**

No, they are not. HTTP stands for **H**yper**T**ext **T**ransfer **P**rotocol and is a way to transfer files. This protocol is used to link pages of hypertext in what we call the world-wide-web. However, there are other transfer protocols available, like FTP and gopher, yet they are less popular.

**RE**presentational **S**tate **T**ransfer, or REST, is a set of constraints that ensure a scalable, fault-tolerant and easily extendible system. The world-wide-web is an example of such system (and the biggest example, one might say). REST by itself is not a new invention, but it's the documentation on such systems like the world-wide-web.

One thing that confuses people, is that REST and HTTP seem to be hand-in-hand. After all, the world-wide-web itself runs on HTTP, and it makes sense, a RESTful API does the same. However, there is nothing in the REST constraints that makes the usage of HTTP as a transfer protocol mandatory. It's perfectly possible to use other transfer protocols like SNMP, SMTP and others to use, and your API could still very well be a RESTful API

In practice, most - if not all - RESTful APIs currently use HTTP as a transport layer, since the infrastructure, servers and client libraries for HTTP are widely available already

Note that there is also a big difference between a RESTful API and a HTTP API. A RESTful API adheres ALL the REST constraints set out in its "format" documentation (in the dissertation of Roy Fielding). A HTTP API is ANY API that makes use of HTTP as their transfer protocol. This means that even SOAP can be considered a HTTP API, as long as it will use HTTP for transport, but most HTTP APIs will make more and better use of the infrastructure and possibilities of HTTP. Most HTTP APIs can be very close to becoming a truly RESTful API. This can be defined by their Richardsons maturity level.

- See more at: http://restcookbook.com/Miscellaneous/rest-and-http/#sthash.xV7XmGLJ.dpuf

**When should we return 4xx or 5xx status codes to the client?**

4xx codes are used to tell the client that a fault has taken place on THEIR side. They should not retransmit the same request again, but fix the error first.

5xx codes tell the client something happened on the server and their request by itself was perfectly valid. The client can continue and try again with the request without modification.

**Caching your REST API**

The goal of caching is never having to generate the same response twice. The benefit of doing this is that we gain speed and reduce server load. The best way to cache your API is to put a gateway cache (or reverse proxy) in front of it. Some frameworks provide their own reverse proxies, but a very powerful, open-source one is [Varnish](https://www.varnish-cache.org/).

When a safe method is used on a resource URL, the reverse proxy should cache the response that is returned from your API. It will then use this cached response to answers all subsequent requests for the same resource before they hit your API. When an unsafe method is used on a resource URL, the cache ignores it and passes it to the API. The API is responsible for making sure that the cached resource is invalidated.

The result will look like this:

GET /article/1234 HTTP/1.1

- The resource is not cached yet

- Send request to the API

- Store response in cache and return

GET /article/1234 HTTP/1.1

- The resource is cached

- Return response from cache

PUT /article/1234 HTTP/1.1

- Unsafe method, send to API

PURGE /article/1234 HTTP/1.1

- API sends PURGE method to the cache

- The resources is removed from the cache

GET /article/1234 HTTP/1.1

- The resource is not cached yet

- Send request to the API

- Store response in cache and return

**What is HATEOAS and why is it important for my REST API?**

HATEOAS stands for **Hypertext As The Engine Of Application State**. It means that hypertext should be used to find your way through the API. An example:

GET /account/12345 HTTP/1.1

HTTP/1.1 200 OK

**<?xml version="1.0"?>**

<account>

<account\_number>12345</account\_number>

<balance currency="usd">100.00</balance>

<link rel="deposit" href="/account/12345/deposit" />

<link rel="withdraw" href="/account/12345/withdraw" />

<link rel="transfer" href="/account/12345/transfer" />

<link rel="close" href="/account/12345/close" />

</account>

Apart from the fact that we have 100 dollars (US) in our account, we can see 4 options: deposit more money, withdraw money, transfer money to another account, or close our account. The "link"-tags allows us to find out the URLs that are needed for the specified actions. Now, let's suppose we didn't have 100 usd in the bank, but we actually are in the red:

GET /account/12345 HTTP/1.1

HTTP/1.1 200 OK

**<?xml version="1.0"?>**

<account>

<account\_number>12345</account\_number>

<balance currency="usd">-25.00</balance>

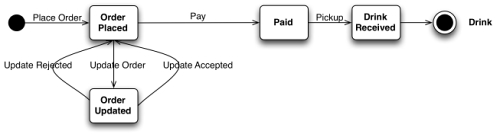
<link rel="deposit" href="/account/12345/deposit" />

</account>

Now we are 25 dollars in the red. Do you see that right now we have lost many of our options, and only depositing money is valid? As long as we are in the red, we cannot close our account, nor transfer or withdraw any money from the account. The hypertext is actually telling us what is allowed and what not: HATEOAS

=====================-----------------------------

In the customer workflow, customers advance towards the goal of drinking some coffee by interacting with the Starbucks service. As part of the workflow, we assume that the customer places an order, pays, and then waits for their drink. Between placing and paying for the order, the customer can usually amend it – by, for example, asking for semi-skimmed milk to be used.



**Figure1 The Customer State Machine**

The barista has his or her own state machine, though it's not visible to the customer; it's private to the service's implementation. As shown in Figure 2, the barista loops around looking for the next order to be made, preparing the drink, and taking the payment. An instance of the loop can begin when an order is added to the barista's queue. The outputs of the workflow are available to the customer when the barista finishes the order and releases the drink.



**Figure 2 The Barista's State Machine**

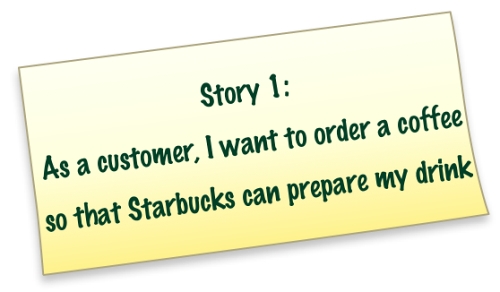
Although all of this might seem a million miles away from Web-based integration, each transition in our two state machines represents an interaction with a Web resource. Each transition is the combination of a HTTP verb on a resource via its URI causing state changes.

GET and HEAD are special cases since they don't cause state transitions. Instead they allow us to inspect the current state of a resource.

But we're getting ahead of ourselves. Thinking about state machines and the Web isn't easy to swallow in one big lump. So let's revisit the entire scenario from the beginning, look at it in a Web context, and proceed one step at a time.

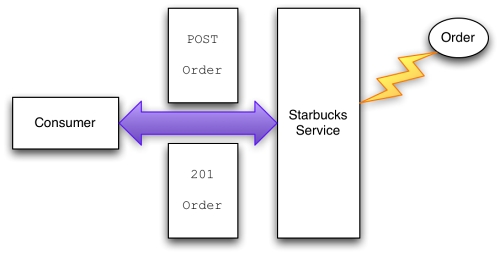
### The Customer's Viewpoint

We'll begin at the beginning, with a simple story card that kick-starts the whole process:



This story contains a number of useful actors and entities. Firstly, there's the customer actor, who is the obvious consumer of the (implicit) Starbucks service. Secondly, there are two interesting entities (coffee and order), and an interesting interaction (ordering), which starts our workflow.

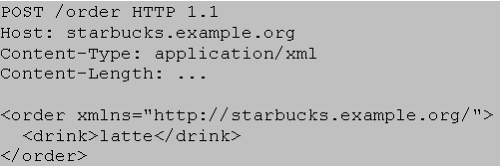
To submit an order to Starbucks, we simply POST a representation of an order to the well-known Starbucks ordering URI, which for our purposes will behttp://starbucks.example.org/order.



**Figure 3 Ordering a coffee**

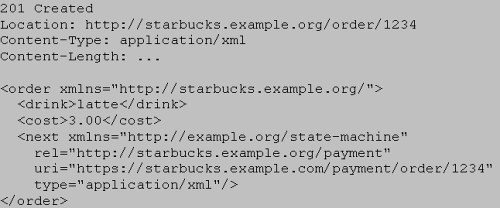
Figure 3 shows the interaction to place an order with Starbucks. Starbucks uses an XML dialect to represent entities from its domain; interestingly, this dialect also allows information to be embedded so that customers can progress through the ordering process – as we'll see shortly. On the wire the act of posting looks something like Figure 4.

In the human Web, consumers and services use HTML as a representation format. HTML has its own particular semantics, which are understood and adopted by all browsers:<a/>, for example,**means**“an anchor that links to another document or to a bookmark within the same document.” The consumer application – the Web browser – simply renders the HTML, and the state machine (that's you!) follows links usingGETandPOST. In Web-based integration the same occurs, except the services and their consumers not only have to agree on the interaction protocols, but also on the format and semantics of the representations.



**Figure 4 POSTing a drinks order**

The Starbucks service creates an order resource, and then responds to the consumer with the location of this new resource in the Location HTTP header. For convenience, the service also places the representation of the newly created order resource in the response. The response looks something like .



**Figure 5 Order created, awaiting payment**

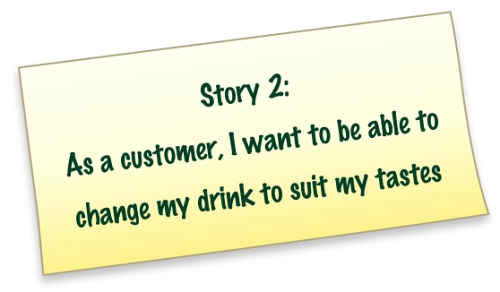
The 201 Created status indicates that Starbucks successfully accepted the order. TheLocation header gives the URI of the newly created order. The representation in the response body contains confirmation of what was ordered along with the cost. In addition, this representation contains the URI of a resource with which Starbucks expects us to interact to make forward progress with the customer workflow; we'll use this URI later.

Note that the URI is contained in a <next/> tag, not an HTML <a/> tag. <next/> is here meaningful in the context of the customer workflow, the semantics of which have been agreed a priori.

We've already seen that the201 Createdstatus code indicates the successful creation of a resource. We'll need a handful of other useful codes both for this example and for Web-based integration in general:200 OK- This is what we like to see: everything's fine; let's keep going.201 Created- We've just created a resource and everything's fine.202 Accepted- The service has accepted our request, and invites us to poll a URI in the Location header for the response. Great for asynchronous processing.303 See Other- We need to interact with a different resource. We're probably still OK.400 Bad Request- We need to reformat the request and resubmit it.404 Not Found- The service is far too lazy (or secure) to give us a real reason why our request failed, but whatever the reason, we need to deal with it.409 Conflict- We tried to update the state of a resource, but the service isn't happy about it. We'll need to get the current state of the resource (either by checking the response entity body, or doing a GET) and figure out where to go from there.412 Precondition Failed- The request wasn't processed because an Etag, If-Match or similar guard header failed evaluation. We need to figure out how to make forward progress.417 Expectation Failed- You did the right thing by checking, but please don't try to send that request for real.500 Internal Server Error- The ultimate lazy response. The server's gone wrong and it's not telling why. Cross your fingers…

### Updating an Order

One of the nice things about Starbucks is you can customise your drink in a myriad of different ways. In fact, some of the more advanced customers would be better off ordering by chemical formula, given the number of upgrades they demand! But let's not be that ambitious – at least not to start with. Instead, we'll look at another story card:



Looking back on Figure 4, it's clear we made a significant error: for anyone that really likes coffee, a single shot of espresso is going to be swamped by gallons of hot milk. We're going to have to change that. Fortunately, the Web (or more precisely HTTP) provides support for such changes, and so does our service.

Firstly, we'll make sure we're still allowed to change our order. Sometimes the barista will be so fast our coffee's been made before we've had a chance to change it – and then we're stuck with a cup of hot coffee-flavoured milk. But sometimes the barista's a little slower, which gives us the opportunity to change the order before the barista processes it. To find out if we can change the order, we ask the resource what operations it's prepared to process using the HTTP OPTIONSverb, as shown on the wire in Figure 6.

|  |  |
| --- | --- |
| **Request** | **Response** |
| OPTIONS /order/1234 HTTP 1.1 Host: starbucks.example.org | 200 OK Allow: GET, PUT |

**Figure6 Asking for OPTIONS**

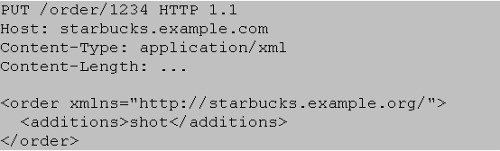
From Figure 6 we see that the resource is readable (it supports GET) and it's updatable (it supports PUT). As we're good citizens of the Web, we can, optionally, do a trial PUT of our new representation, testing the water using the Expect header before we do a real PUT – like in Figure 7.

|  |  |
| --- | --- |
| **Request** | **Response** |
| PUT /order/1234 HTTP 1.1 Host: starbucks.example.com Expect: 100-Continue | 100 Continue |

**Figure 7 Look before you leap!**

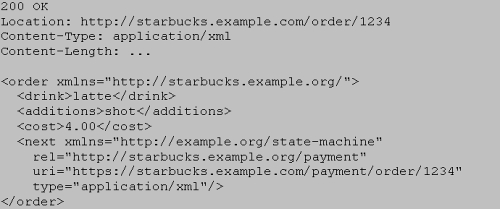
If it had no longer been possible to change our order, the response to our “look before you leap” request in Figure 7 would have been 417 Expectation Failed. But here the response is 100 Continue, which allows us to try to PUT an update to the resource with an additional shot of espresso, as shown in Figure 8. PUTting an updated resource representation effectively changes the existing one. In this instance PUT lodges a new description with an <additions/> element containing that vital extra shot.

Although partial updates are the subject of deep philosophical debates within the REST community, we take a pragmatic approach here and assume that our request for an additional shot is processed in the context of the existing resource state. As such there is little point in moving the whole resource representation across the network for each operation and so we transmit deltas only.



**Figure 8 Updating a resource's state**

If we're successfully able to PUT an update to the new resource state, we get a 200 response from the server, as shown in Figure 9.



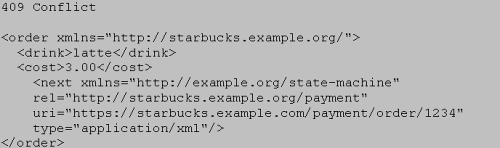
**Figure 9 Successfully updating the state of a resource**

Checking OPTIONS and using the Expect header can't totally shield us from a situation where a change at the service causes subsequent requests to fail. As such we don't mandate their use, and as good Web citizens we're going to handle 405 and 409 responses anyway.

OPTIONSand especially using theExpectheader should be considered optional steps.

Even with our judicious use of Expect and OPTIONS, sometimes our PUT will fail; after all, we're in a race with the barista – and sometimes those guys just fly!

If we lose the race to get our extra shot, we'll learn about it when we try to PUT the updates to the resource. The response in Figure 10 is typical of what we can expect. 409 Conflictindicates the resource is in an inconsistent state to receive the update. The response body shows the difference between the representation we tried to PUT and the resource state on the server side. In coffee terms it's too late to add the shot – the barista's already pouring the hot milk.



**Figure 10 Losing a race**

We've discussed using Expect and OPTIONS to guard against race conditions as much as possible. Besides these, we can also attach If-Unmodified-Since or If-Match headers to our PUT to convey our intentions to the receiving service. If-Unmodified-Since uses the timestamp and If-Match the ETag[1](http://www.infoq.com/articles/webber-rest-workflow" \l "sdfootnote1sym) of the original order. If the order hasn't changed since we created it – that is, the barista hasn't started preparing our coffee yet – then the change will be processed. If the order has changed, we'll get a 412 Precondition Failed response. If we lose the race, we're stuck with milky coffee, but at least we've not transitioned the resource to an inconsistent state.

There are a number of patterns for consistent state updates using the Web. HTTP PUT is idempotent, which takes much of the intricate work out of updating state, but there are still choices that need to be made. Here's our recipe for getting updates right:

1. Ask the service if it's still possible toPUTby sendingOPTIONS. This step is optional. It gives clients a clue about which verbs the server supports for the resource at the time of asking, but there are no guarantees the service will support those same verbs indefinitely.

2. Use anIf-Unmodified-SinceorIf-Matchheader to help the server guard against executing an unnecessaryPUT. You'll get a412 Precondition Failedif thePUTsubsequently fails. This approach depends either on slowly changing resources (1 second granularity) forIf-Unmodified-Sinceor support for ETags forIf-Match.

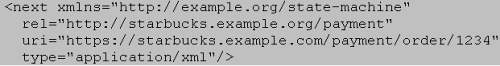
3. ImmediatelyPUTthe update and deal with any409 Conflict responses. Even if we use (1) and (2), we may have to deal with these responses, since our guards and checks are optimistic in nature.

The W3C has[*a non-normative note*](http://www.w3.org/1999/04/Editing/)on detecting and dealing with inconsistent updates that argues for using ETag. ETags are our preferred approach.

After all that hard work updating our coffee order, it seems only fair that we get our extra shot. So for now let's go with our happy path, and assume we managed to get our additional shot of espresso. Of course, Starbucks won't hand our coffee over unless we pay (and it turns out they've already hinted as much!), so we need another story:



Remember the <next/> element in the response to our original order? This is where Starbucks embedded information about another resource in the order representation. We saw the tag earlier, but chose to ignore it while correcting our order. But now it's time to look more closely at it:



There are a few aspects to the next element worth pointing out. First is that it's in a different namespace because state transitions are not limited to Starbucks. In this case we've decided that such transition URIs should be held in a communal namespace to facilitate re-use (or even eventual standardisation).

Then, there's the embedded semantic information (a private microformat, if you like) in the relattribute. Consumers that understand the semantics of thehttp://starbucks.example.org/payment string can use the resource identified by the uriattribute to transition to the next state (payment) in the workflow.

The uri in the <next/> element points to a payment resource. From the type attribute, we already know the expected resource representation is XML. We can work out what to do with the payment resource by asking the server which verbs that resource supports using OPTIONS.

Microformats are a way to embed structured, semantically-rich data inside existing documents. Microformats are most common in the human readable Web, where they are used to add structured representations of information like calendar events to Web pages. However, they can just as readily be turned to integration purposes. Microformat terminology is agreed by the microformats community, but we are at liberty to create our own private microformats for domain-specific semantic markup.

Innocuous as they seem, simple links like the one of Figure 10 are the crux of what the REST community rather verbosely calls “Hypermedia as the engine of application state.” More simply, URIs represent the transitions within a state machine. Clients operate application state machines, like the ones we saw at the beginning of this article, by following links.

Don't be surprised if that takes a little while to sink in. One of the most surprising things about this model is the way state machines and workflows gradually describe themselves as you navigate through them, rather than being described upfront through WS-BPEL or WS-CDL. But once your brain has stopped somersaulting, you'll see that following links to resources allows us to make forward progress in our application's various states. At each state transition the current resource representation includes links to the next set of possible resources and the states they represent. And because those next resources are just Web resources, we already know what to do with them.

Our next step in the customer workflow is to pay for our coffee. We know the total cost from the<cost/> element in the order, but before we send payment to Starbucks we'll ask the payment resource how we're meant to interact with it, as shown in Figure 11.

How much upfront knowledge of a service does a consumer need? We've already suggested that services and consumers will need to agree the semantics of the representations they exchange prior to interacting. Think of these representation formats as a set of possible states and transitions. As a consumer interacts with a service, the service chooses states and transitions from the available set and builds the next representation. The process – the “how” of getting to a goal – is discovered on the fly; what gets wired together as part of that process is, however, agreed upfront.

Consumers typically agree the semantics of representations and transitions with a service during design and development. But there's no guarantee that as service evolves, it won't confront the client with state representations and transitions the client had never anticipated but knows how to process – that's the nature of the loosely coupled Web. Reaching agreement on resource formats and representations under these circumstances is, however, outside the scope of this article.

Our next step is to pay for our coffee. We know the total cost of our order from the <cost>element embedded in the order representation, and so our next step is to send a payment to Starbucks so the barista will hand over the drink. Firstly we'll ask the payment resource how we're meant to interact with it, as shown in Figure 11.

|  |  |
| --- | --- |
| **Request** | **Response** |
| OPTIONS/payment/order/1234 HTTP 1.1 Host: starbucks.example.com | Allow: GET, PUT |

**Figure 11 Figuring out how to pay**

The response indicates we can either read (via GET) the payment or update it (via PUT). Knowing the cost, we'll go ahead and PUT our payment to the resource identified by the payment link. Of course, payments are privileged information, so we'll protect access to the resource by requiring authentication[2](http://www.infoq.com/articles/webber-rest-workflow" \l "sdfootnote2sym).

|  |
| --- |
| **Request** |
| PUT /payment/order/1234 HTTP 1.1 Host: starbucks.example.com Content-Type: application/xml Content-Length: ... Authorization: Digest username="Jane Doe" realm="starbucks.example.org“  nonce="..." uri="payment/order/1234" qop=auth nc=00000001 cnonce="..." reponse="..." opaque="..."  <payment xmlns="http://starbucks.example.org/">    <cardNo>123456789</cardNo>    <expires>07/07</expires>    <name>John Citizen</name>    <amount>4.00</amount> </payment> |
| **Response** |
| 201 Created Location: https://starbucks.example.com/payment/order/1234 Content-Type: application/xml Content-Length: ...  <payment xmlns="http://starbucks.example.org/">    <cardNo>123456789</cardNo>    <expires>07/07</expires>    <name>John Citizen</name>    <amount>4.00</amount> </payment> |

**Figure 12 Paying the bill**

For successful payments, the exchange shown in Figure 12 is all we need. Once the authenticated PUT has returned a 201 Created response, we can be happy the payment has succeeded, and can move on to pick up our drink.

But things can go wrong, and when money is at stake we'd rather things either didn't go wrong or are recoverable when they do[3](http://www.infoq.com/articles/webber-rest-workflow" \l "sdfootnote3sym). A number of things can obviously go wrong with our payment:

* We can't connect to the server because it is down or unreachable;
* The connection to the server is severed at some point during the interaction;
* The server returns an error status in the 4xx or 5xx range.

Fortunately, the Web helps us in each of these scenarios. For the first two cases (assuming the connectivity issue is transient), we simply PUT the payment again until we receive a successful response. We can expect a 200 response if a prior PUT had in fact succeeded (effectively an acknowledgement of a no-op from the server) or a 201 if the new PUT eventually succeeds in lodging the payment. The same holds true in the third case where the server responds with a500, 503 or 504 response code.

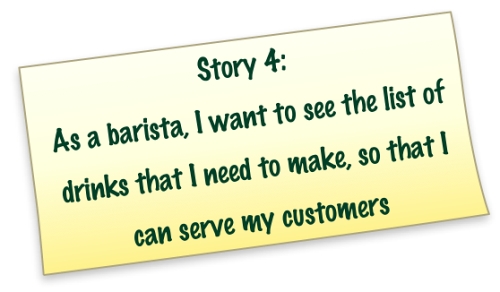
Status codes in the 4xx range are trickier, but they still indicate how to make forward progress. For example, a 400 response indicates that we PUT something the server doesn't understand, and should rectify our payload before PUTing it again. Conversely, a 403 response indicates that the server understood our request but is refusing to fulfil it and doesn't want us to re-try. In such cases we'll have to look for other state transitions (links) in the response payload to make alternative forward progress.

We've used status codes several times in this example to guide the client towards its next interaction with the service. Status codes are semantically rich acknowledgments. By implementing services that produce meaningful status codes and clients that know how to handle them, we can layer a coordination protocol on top of HTTP's simple request-response mechanism, adding a high degree of robustness and reliability to distributed systems.

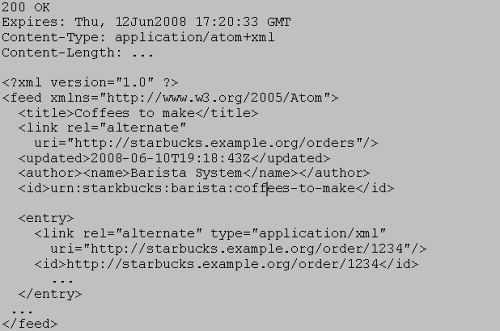
Once we've paid for our drink we've reached the end of our workflow, and the end of the story as far as the consumer goes. But it's not the end of the whole story. Let's now go inside the service boundary, and look at Starbucks' internal implementation.

### The Barista's Viewpoint

As customers we tend to put ourselves at the centre of the coffee universe, but we're not the only consumers of a coffee service. We know already from our “race” with the barista that the service serves at least one other set of interested parties, not the least of which is the barista. In keeping with our incremental delivery style, it's time for another story card.



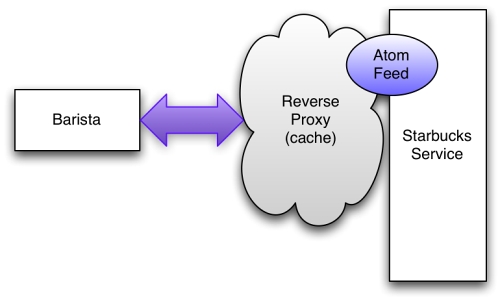
Lists of drinks are easily modelled using Web formats and protocols. Atom feeds are a perfectly good format for lists of practically anything, including outstanding coffee orders, so we'll adopt them here. The barista can access the Atom feed with a simple GET on the feed's URI, which for outstanding orders is http://starbucks.example.org/orders in Figure 13.



**Figure 13 Atom feed for drinks to be made**

Starbucks is a busy place and the Atom feed at /orders is updated frequently, so the barista will need to poll it to stay up to date. Polling is normally thought of as offering low scalability; the Web, however, supports an extremely scalable polling mechanism – as we'll see shortly. And with the sheer volume of coffees being manufactured by Starbucks every minute, scaling to meet load is an important issue.

We have two conflicting requirements here. We want baristas to keep up-to-date by polling the order feed often, but we don't want to increase the load on the service or unnecessarily increase network traffic. To avoid crushing our service under load, we'll use a reverse proxy just outside our service to cache and serve frequently accessed resource representations, as shown in Figure 14.



**Figure 14 Caching for scalability**

For most resources – especially those that are accessed widely, like our Atom feed for drinks – it makes sense to cache them outside of their host services. This reduces server load and improves scalability. Adding Web caches (reverse proxies) to our architecture, together with caching metadata, allows clients to retrieve resources without placing load on the origin server.

A positive side effect of caching is that it masks intermittent failures of the server and helps crash recovery scenarios by improving the availability of resource state. That is, the barista can keep working even if the Starbucks service fails intermittently since the order information will have been cached by a proxy. And if the barista forgets an order (crashes) then recovery is made easier because the orders are highly available.

Of course, caching can keep old orders around longer than needed, which is hardly ideal for a high-throughput retailer like Starbucks. To make sure that cached orders are cleared, the Starbucks service uses the Expires header to declare how long a response can be cached. Any caches between the consumer and service (should) honour that directive and refuse to serve stale orders[4](http://www.infoq.com/articles/webber-rest-workflow" \l "sdfootnote4sym), instead forwarding the request onto the Starbucks service, which has up-to-date order information.

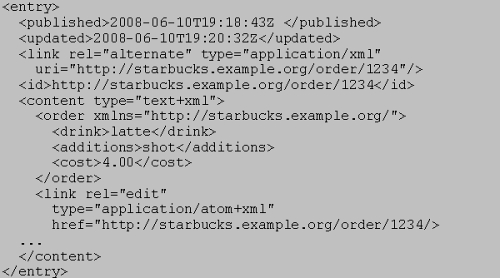
The response in Figure 13 sets the Expires header on our Atom feed so that drinks turn stale 10 seconds into the future. Because of this caching behaviour, the server can expect at most 6 requests per minute, with the remainder handled by the cache infrastructure. Even for a relatively poorly performing service, 6 requests per minute is a manageable workload. In the happiest case (from Starbucks' point of view) the barista's polling requests are answered from a local cache, resulting in no increased network activity or server load.

In our example, we use only one cache to help scale-out our master coffee list. Real Web-based scenarios, however, may benefit from several layers of caching. Taking advantage of existing Web caches is critical for scalability in high volume situations.

The Web trades latency for massive scalability. If you have a problem domain that is highly sensitive to latency (e.g. foreign exchange trading), then Web-based solutions are not a great idea. If, however, you can accept latency in the order of seconds, or even minutes or hours, then the Web is likely a suitable platform.

Now that we've addressed scalability, let's return to more functional concerns. When the barista begins to prepare our coffee, the state of the order should change so that no further updates are allowed. From the point of view of a customer, this corresponds to the moment we're no longer allowed to PUT updates of our order (as in Figure 6, Figure 7, Figure 8, Figure 9, and Figure 10).

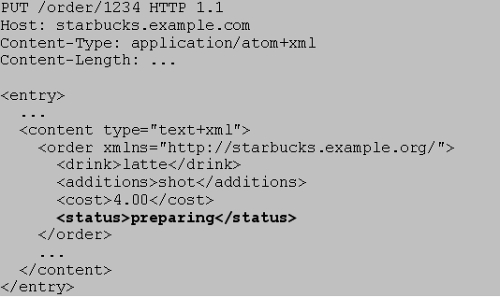
Fortunately there is a well-defined protocol that we can use for this job: the Atom Publishing Protocol (also known as APP or AtomPub). AtomPub is a Web-centric (URI-based) protocol for managing entries in Atom feeds. Let's take a closer look at the entry representing our coffee in the /orders Atom feed.



**Figure 15 Atom entry for our coffee order**

The XML in Figure 15 is interesting for a number of reasons. First, there's the Atom XML, which distinguishes our order from all the other orders in the feed. Then there's the order itself, containing all the information our barista needs to make our coffee – including our all-important extra shot! Inside the order entry, there's a link element that declares the edit URI for theentry. The edit URI links to an order resource that is editable via HTTP. (The address of the editable resource in this case happens to be the same address as the order resource itself, but it need not be.)

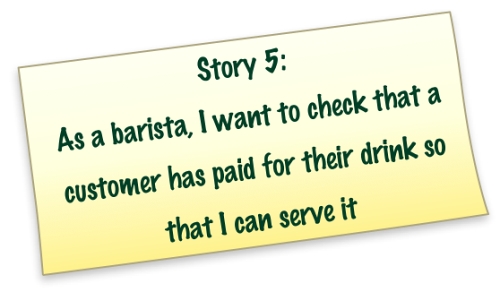
When a barista wants to change the state of the resource so that our order can no longer be changed, they interact with it via the edit URI. Specifically they PUT a revised version of the resource state to the edit URI, as shown in Figure 16.



**Figure 16 Changing the order status via AtomPub**

Once the server has processed the PUT request in Figure 16, it will reject anything other thanGET requests to the /orders/1234 resource.

Now that the order is stable the barista can safely get on with making the coffee. Of course, the barista will need to know we've paid for the order before they release the coffee to us, so before handing the coffee over, the barista checks to make sure we've paid. In a real Starbucks, things are a little different: there are conventions, such as paying as you order, and other customers hanging around to make sure you don't run off with their drinks. But in our computerised version it's not much additional work to add this check, and so onto our penultimate story card:



The barista can easily check the payment status by GETting the payment resource using the payment URI in the order.

In this instance the customer and barista know about the payment resource from the link embedded in the order representation. But sometimes it's useful to access resources via URI templates.

URI templates are a description format for well-known URIs. The templates allow consumers to vary parts of a URI to access different resources.

A URI template scheme underpins Amazon's S3 storage service. Stored artefacts are manipulated using the HTTP verbs on URIs created from this template:http://s3.amazonaws.com/{bucket\_name}/{key\_name}.

It's easy to infer a similar scheme for payments in our model so that baristas (or other authorised Starbucks systems) can readily access each payment without having to navigate all orders:http://starbucks.example.org/payment/order/{order\_id}

URI templates form a contract with consumers, so service providers must take care to maintain them even as the service evolves. Because of this implicit coupling some Web integrators shy away from URI templates. Our advice is to use them only where inferable URIs are useful and unlikely to change.

An alternative approach in our example would be to expose a/paymentsfeed containing (non-inferable) links to each payment resource. The feed would only be available to authorised systems.

Ultimately it is up to the service designer to determine whether URI templates are a safe and useful shortcut through hypermedia. Our advice: use them sparingly!

Of course, not everyone is allowed to look at payments. We'd rather not let the more creative (and less upstanding) members of the coffee community check each-others' credit card details, so like any sensible Web system, we protect our sensitive resources by requiring authentication.

If an unauthenticated user or system tries to retrieve the details of a particular payment, the server will challenge them to provide credentials, as shown in Figure 17.

|  |  |
| --- | --- |
| **Request** | **Response** |
| GET /payment/order/1234 HTTP 1.1 Host: starbucks.example.org | 401 Unauthorized WWW-Authenticate: Digest realm="starbucks.example.org", qop="auth", nonce="ab656...", opaque="b6a9..." |

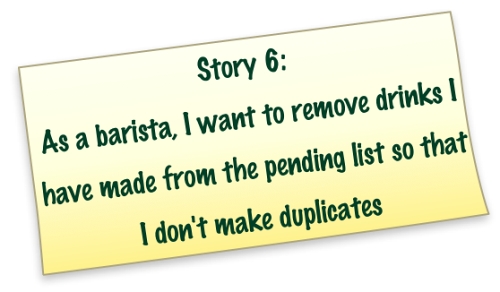
**Figure 17 Unauthorised access to a payment resource is challenged**

The 401 status (with helpful authentication metadata) tells us we should try the request again, but this time provide appropriate credentials. Retrying with the right credentials (Figure 18), we retrieve the payment and compare it with the resource representing the total value of the order at http://starbucks.example.org/total/order/1234.

|  |  |
| --- | --- |
| **Request** | **Response** |
| GET /payment/order/1234 HTTP 1.1 Host: starbucks.example.org Authorization: Digest username="barista joe" realm="starbucks.example.org“ nonce="..." uri="payment/order/1234" qop=auth nc=00000001 cnonce="..." reponse="..." opaque="..." | 200 OK Content-Type: application/xml Content-Length: ... <payment xmlns="http://starbucks.example.org/">    <cardNo>123456789</cardNo>    <expires>07/07</expires>    <name>John Citizen</name>    <amount>4.00</amount> </payment> |

**Figure 18 Authorised access to a payment resource**

Once the barista has prepared and dispatched the coffee and collected payment, they'll want to remove the completed order from the list of outstanding drinks. As always we'll capture this as a story:



Because each entry in our orders feed identifies an editable resource with its own URI, we can apply the HTTP verbs to each order resource individually. The barista simply DELETEs the resource referenced by the relevant entry to remove it from the list, as in Figure 19.

|  |  |
| --- | --- |
| **Request** | **Response** |
| DELETE /order/1234 HTTP 1.1 Host: starbucks.example.org | 200 OK |

**Figure 19 Removing a completed order**

With the item DELETEd from the feed, a fresh GET of the feed returns a representation without the DELETEd resource. Assuming we have well behaved caches and have set the cache expiry metadata sensibly, trying to GET the order entry directly results in a 404 Not Found response.

You might have noticed that the Atom Publishing Protocol meets most of our needs for the Starbucks domain. If we'd exposed the /orders feed directly to customers, customers could have used AtomPub to publish drinks orders to the feed, and even change their orders over time.

## Evolution: A fact of Life on the Web

Since our coffee shop is based around self-describing state machines, it's quite straightforward to evolve the workflows to meet changing business needs. For example Starbucks might choose to offer a free Internet promotion shortly after starting to serve coffee:

* July – Our new Starbucks shop goes live offering the standard workflow with the state transitions and representations that we've explored throughout this article. Consumers are interacting with the service with these formats and representations in mind.
* August – Starbucks introduces a new representation for a free wireless promotion. Our coffee workflow will be updated to contain links providing state transitions to the offer. Thanks to the magic of URIs, the links may be to a 3rd party partner just as easily as they could be to an internal Starbucks resource



Because the representations still include the original transitions, existing consumers can still reach their goal, though they may not be able to take advantage of the promotion because they have not been explicitly programmed for it.

* September – Consumer applications and services are upgraded so that they can understand and use the free Internet promotion, and are instructed to follow such promotional transitions whenever they occur.

The key to successful evolution is for consumers of the service to anticipate change by default. Instead of binding directly to resources (e.g. via URI templates), at each step the service provides URIs to named resources with which the consumer can interact. Some of these named resources will not be understood and will be ignored; others will provide known state transitions that the consumer wants to make. Either way this scheme allows for graceful evolution of a service while maintaining compatibility with consumers.

## The Technology you're about to enjoy is extremely hot

Handing over the coffee brings us to the end of the workflow. We've ordered, changed (or been unable to change) our order, paid and finally received our coffee. On the other side of the counter Starbucks has been equally busy taking payment and managing orders.

We were able to model all necessary interactions here using the Web. The Web allowed us to model some simple unhappy paths (e.g. not being able to change an in process order or one that's already been made) without us having to invent new exceptions or faults: HTTP provided everything we needed right out of the box. And even with the unhappy paths, clients were able to progress towards their goal.

The features HTTP provides might seem innocuous at first. But there is already worldwide agreement and deployment of this protocol, and every conceivable software agent and hardware device understands it to a degree. When we consider the balkanised adoption of other distributed computing technologies (such as WS-\*) we realise the remarkable success that HTTP has enjoyed, and the potential it releases for system-to-system integration.

The Web even helped non-functional aspects of the solution. Where we had transient failures, a shared understanding of the idempotent behaviour of verbs like GET, PUT and DELETE allowed safe retries; baked-in caching masked failures and aided crash recovery (through enhanced availability); and HTTPs and HTTP Authentication helped with our rudimentary security needs.

Although our problem domain was somewhat artificial, the techniques we've highlighted are just as applicable in traditional distributed computing scenarios. We won't pretend that the Web is simple (unless you are a genius), nor do we pretend that that it's a panacea (unless you are an unrelenting optimist or have caught REST religion), but the fact is that the Web is a robust framework for integrating systems at local, enterprise, and Internet scale.

**What is the Richardson Maturity Model?**

The Richardson Maturity Model is a way to grade your API according to the constraints of REST. The better your API adheres to these constraints, the higher its score is. The Richardson Maturity Model knows 4 levels (0-3), where level 3 designates a truly RESTful API.

### Level 0: Swamp of POX

Level 0 uses its implementing protocol (normally HTTP, but it doesn't have to be) like a transport protocol. That is, it tunnels requests and responses through its protocol without using the protocol to indicate application state. It will use only one entry point (URI) and one kind of method (in HTTP, this normally is the POST method). Examples of these are SOAP and XML-RPC.

### Level 1: Resources

When your API can distinguish between different resources, it might be level 1. This level uses multiple URIs, where every URI is the entry point to a specific resource. Instead of going through http://example.org/articles, you actually distinguish between http://example.org/article/1 and http://example.org/article/2. Still, this level uses only one single method like POST.

### Level 2: HTTP verbs

To be honest, I don't like this level. This is because this level suggests that in order to be truly RESTful, your API **MUST** use HTTP verbs. It doesn't. REST is completely protocol agnostic, so if you want to use a different protocol, your API can still be RESTful.

This level indicates that your API should use the protocol properties in order to deal with scalability and failures. Don't use a single POST method for all, but make use of GET when you are requesting resources, and use the DELETE method when you want to delete a resources. Also, use the response codes of your application protocol. Don't use [200](http://httpstatus.es/200) (OK) code when something went wrong for instance. By doing this for the HTTP application protocol, or any other application protocol you like to use, you have reached level 2.

### Level 3: Hypermedia controls

Level 3, the highest level, uses HATEOAS to deal with discovering the possibilities of your API towards the clients. More information about HATEOAS can be found below.

**How can I create my own custom content-types that are representations of users, categories, articles etc?**

Do not use a standard text/xml content-type. A client is not capable of handling this kind of information. Instead, use a custom format in the following form:

Content-type: application/vnd+company.category+xml

Content-type: application/vnd+company.category+html

Content-type: application/vnd+company.category+json

This allows your clients to process the information (in this case: your categories) as the specified content-type. All three of these content-types are categories, but they are represented in different formats (xml, html and json). Even though this is still information that clients need to know in advance, clients know what to expect, since they can ask for the specific information (categories, in this case), without outside knowledge like the URL etc.

All application/vnd are vendor-specific content-types and are not standardized. These are for non-standard content-types only.

**How can I let users create resources that might take a considerable amount of time. I cannot have my users wait on the API to finish.**

Instead of creating the actual resources, create a temporary one. Instead of returning a [201](http://httpstatus.es/201) (Created) HTTP response, you can then issue at [202](http://httpstatus.es/202) (Accepted) response code. This informs the client that the request has been accepted and understood by the server, but the resource is not (yet) created. Send the temporary resource inside the Location header.

### Request:

POST /blogs HTTP/1.1

<xml>

blogdata

</xml>

### Response:

HTTP/1.1 202 Accepted

Location: /queue/12345

This location can store information about the status of the actual resource: an ETA on when it will be created, what is currently being done or processed.

When the actual resource has been created, the temporary resources can return a [303](http://httpstatus.es/303) (see other) response. The location header returns the URI to the definitive resource. A client can either DELETE the temporary resource, or the server can expire this resource and return a [410](http://httpstatus.es/410) GONE later on.

- See more at: http://restcookbook.com/Resources/asynchroneous-operations/#sthash.fRGltjoL.dpuf

++++++++++++++++\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_+==============

POST VS PUT

POST means "create new" as in "Here is the input for creating a user, create it for me".

PUT means "insert, replace if already exists" as in "Here is the data for user 5".

You POST to example.com/users since you don't know the URL of the user yet, you want the server to create it.

You PUT to example.com/users/id since you want to replace/create a *specific* user.

POSTing twice with the same data means create two identical users. PUTing twice with the same data creates the user the first and updates him to the same state the second time (no changes). Since you end up with the same state after a PUT no matter how many times you perform it, it is said to be "equally potent" every time - idempotent. This is useful for automatically retrying requests. No more 'are you sure you want to resend' when you push the back button on the browser.

A general advice is to use POST when you need the server to be in control of URL generation of your resources. Use PUT otherwise. Prefer PUT over POST.

**ClientBuilder.newBuilder() vs ClientBuilder.newBuilder().newClient()**

Starting with the newBuilder is more flexible, you can add SSL   
configuration, etc

****