How to GET a Cup of Coffee

Posted by Jim Webber, Savas Parastatidis, Ian Robinson on Oct 02, 2008. Estimated reading time: 34 minutes 16 Discuss

We are used to building distributed systems on top of large middleware platforms like those implementing CORBA, the Web Services protocols stack, J2EE, etc. In this article, we take a different approach, treating the protocols and document formats that make the Web tick as an application platform, which can be accessed through lightweight middleware. We showcase the role of the Web in application integration scenarios through a simple customer-service interaction scenario. In this article, we use the Web as our primary design philosophy to distil and share some of the thinking in our forthcoming book “GET /connected - Web-based integration” (working title).

Introduction

The integration domain as we know it is changing. The influence of the Web and the trend towards more agile practices are challenging our notions of what constitutes good integration. Instead of being a specialist activity conducted in the void between systems – or even worse, an afterthought – integration is now an everyday part of successful solutions.

Yet, the impact of the Web is still widely misunderstood and underestimated in enterprise computing. Even those who are Web-savvy often struggle to understand that the Web isn't about middleware solutions supporting XML over HTTP, nor is it a crude RPC mechanism. This is a shame because the Web has much more value than simple point-to-point connectivity; it is in fact a robust integration platform.

In this article we'll showcase some interesting uses of the Web, treating it as a pliant and robust platform for doing very cool things with enterprise systems. And there is nothing that typifies enterprise software more than workflows…

Why Workflows?

Workflows are a staple of enterprise computing, and have been implemented in middleware practically forever (at least in computing terms). A workflow structures work into a number of discrete steps and the events that prompt transitions between steps. The overarching business process implemented by a workflow often spans several enterprise information systems, making workflows fertile ground for integration work.

Starbucks: Standard generic coffee deserves standard generic integration

If the Web is to be a viable technology for enterprise (and wider) integration, it has to be able to support workflows – to reliably coordinate the interactions between disparate systems to implement some larger business capability.

To do justice to a real-world workflow, we'd no doubt have to address a wealth of technical and domain-specific details, which would likely obscure the aim of this article, so we've chosen a more accessible domain to illustrate how Web-based integration works: Gregor Hohpe's Starbucks coffee shop workflow. In his popular blog posting, Gregor describes how Starbucks functions as a decoupled revenue-generating pipeline:

> "Starbucks, like most other businesses is primarily interested in maximizing throughput of orders. More orders equals more revenue. As a result they use asynchronous processing. When you place your order the cashier marks a coffee cup with your order and places it into the queue. The queue is quite literally a queue of coffee cups lined up on top of the espresso machine. This queue decouples cashier and barista and allows the cashier to keep taking orders even if the barista is backed up for a moment. It allows them to deploy multiple baristas in a Competing Consumer scenario if the store gets busy."

While Gregor prefers EAI techniques like message-oriented middleware to model Starbucks, we'll model the same scenario using Web resources – addressable entities that support a uniform interface. In fact, we'll show how Web techniques can be used with all the dependability associated with traditional EAI tools, and how the Web is much more than XML messaging over a request/response protocol!

We'll apologise in advance for taking liberties with the way Starbucks works because our goal here isn't to model Starbucks completely accurately, but to illustrate workflows with Web-based services. So with belief duly suspended, let's jump in.

Stating the Obvious

Since we're talking about workflows, it makes sense to understand the states from which our workflows are composed, together with the events that transition the workflows from state to state. In our example, there are two workflows, which we've modelled as state machines. These workflows run concurrently. One models the interaction between the customer and the Starbucks service as shown in Figure 1 the other captures the set of actions performed by a barista as per Figure 2.

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.

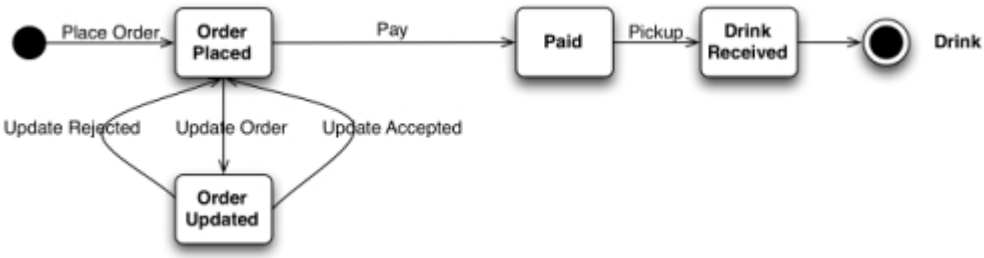


Figure1 The Customer State Machine

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

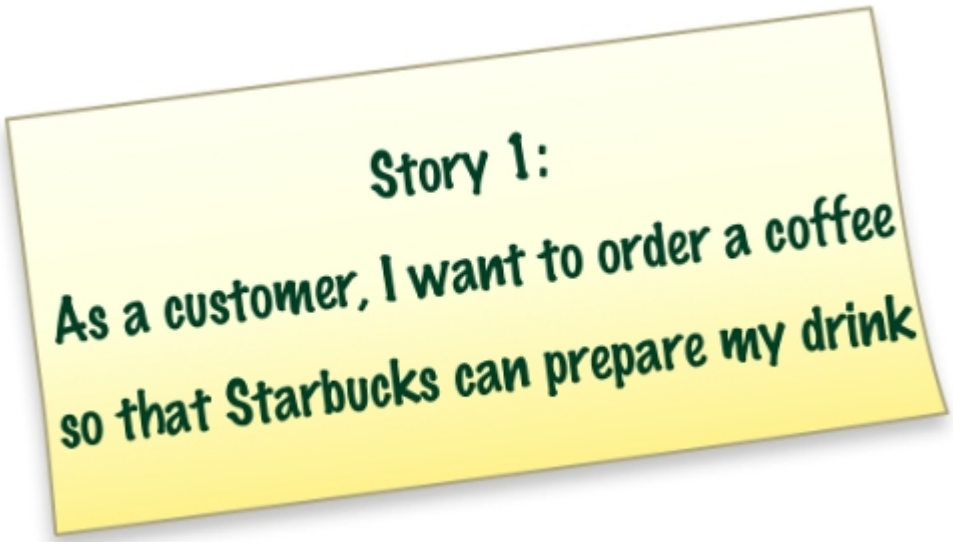
Although all of this might seem a million miles away from Web-based integration, each transition in our two state machines represents an interaction with a Web resource. Each transition is the combination of a HTTP verb on a resource via its URI causing state changes.

> *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 be `http://starbucks.example.org/order`.

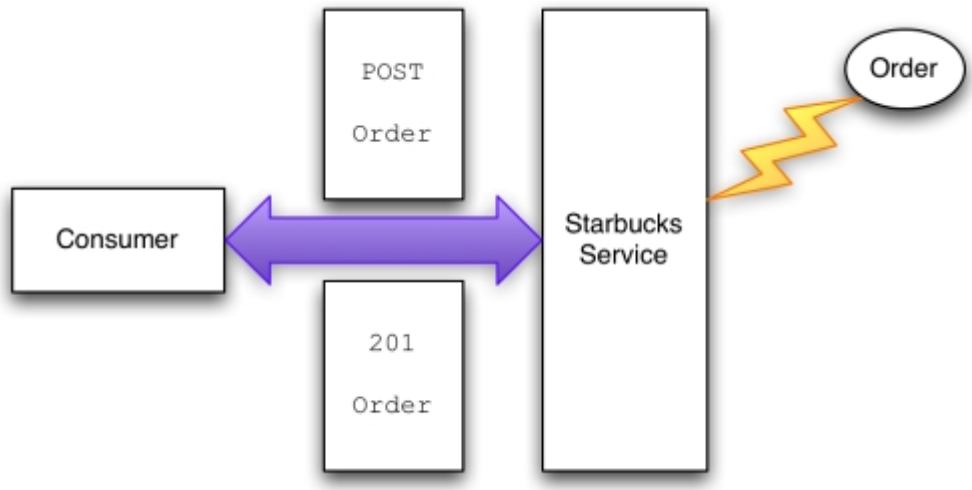


Figure 3 Ordering a coffee

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 using* `GET` *and* `POST`*. 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.*

```
POST /order HTTP 1.1
Host: starbucks.example.org
Content-Type: application/xml
Content-Length: ...

<order xmlns="http://starbucks.example.org/">
  <drink>latte</drink>
</order>
```

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 .

```
201 Created
Location: http://starbucks.example.org/order/1234
Content-Type: application/xml
Content-Length: ...

<order xmlns="http://starbucks.example.org/">
  <drink>latte</drink>
  <cost>3.00</cost>
  <next xmlns="http://example.org/state-machine"
    rel="http://starbucks.example.org/payment"
    uri="https://starbucks.example.com/payment/order/1234"
    type="application/xml"/>
</order>
```

Figure 5 Order created, awaiting payment

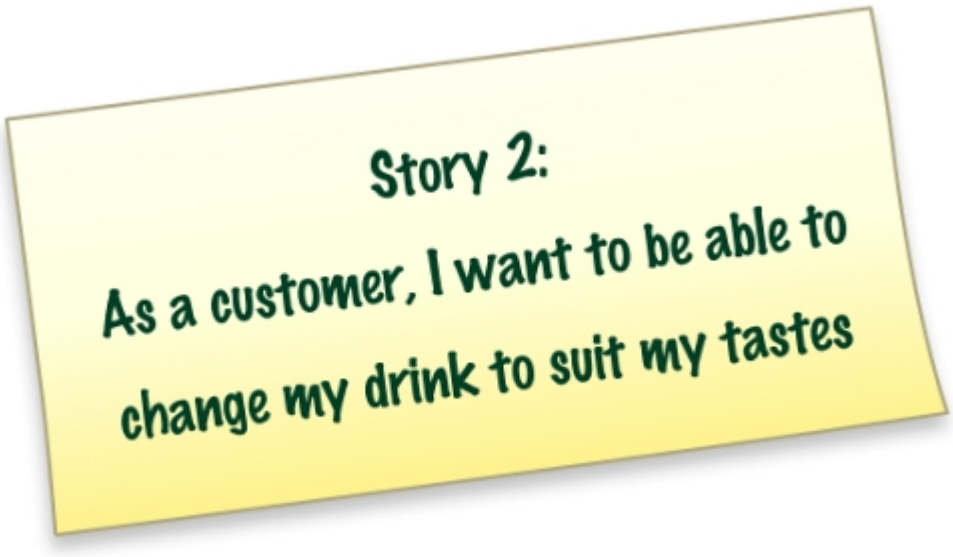
The `201 Created` status indicates that Starbucks successfully accepted the order. The `Location` header gives the URI of the newly created order. The representation in the response body contains confirmation of what was ordered along with the cost. In addition, this representation contains the URI of 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 the `201 Created` status 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 `OPTIONS` verb, 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. `PUT`ting 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.*

```
PUT /order/1234 HTTP 1.1
Host: starbucks.example.com
Content-Type: application/xml
Content-Length: ...

<order xmlns="http://starbucks.example.org/">
  <additions>shot</additions>
</order>
```

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.

```
200 OK
Location: http://starbucks.example.com/order/1234
Content-Type: application/xml
Content-Length: ...

<order xmlns="http://starbucks.example.org/">
  <drink>latte</drink>
  <additions>shot</additions>
  <cost>4.00</cost>
  <next xmlns="http://example.org/state-machine"
    rel="http://starbucks.example.org/payment"
    uri="https://starbucks.example.com/payment/order/1234"
    type="application/xml"/>
</order>
```

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.

> *`OPTIONS` and especially using the `Expect` header 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 Conflict` indicates 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.

```
409 Conflict

<order xmlns="http://starbucks.example.org/">
  <drink>latte</drink>
  <cost>3.00</cost>
    <next xmlns="http://example.org/state-machine"
    rel="http://starbucks.example.org/payment"
    uri="https://starbucks.example.com/payment/order/1234"
    type="application/xml"/>
</order>
```

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] 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 to `PUT` by sending `OPTIONS`. 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 an `If-Unmodified-Since` or `If-Match` header to help the server guard against executing an unnecessary `PUT`. You'll get a `412 Precondition Failed` if the `PUT` subsequently fails. This approach depends either on slowly changing resources (1 second granularity) for `If-Unmodified-Since` or support for ETags for `If-Match`.*
>
> *3. Immediately `PUT` the update and deal with any `409 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 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:

```
<next xmlns="http://example.org/state-machine"
  rel="http://starbucks.example.org/payment"
  uri="https://starbucks.example.com/payment/order/1234"
  type="application/xml"/>
```

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 rel attribute. Consumers that understand the semantics of the http://starbucks.example.org/payment string can use the resource identified by the uri attribute 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].

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]. 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 a `500`, `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 `PUT`ing 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.

```
200 OK
Expires: Thu, 12Jun2008 17:20:33 GMT
Content-Type: application/atom+xml
Content-Length: ...

<?xml version="1.0" ?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>Coffees to make</title>
  <link rel="alternate"
    uri="http://starbucks.example.org/orders"/>
  <updated>2008-06-10T19:18:43Z</updated>
  <author><name>Barista System</name></author>
  <id>urn:starkbucks:barista:coffees-to-make</id>

  <entry>
    <link rel="alternate" type="application/xml"
      uri="http://starbucks.example.org/order/1234"/>
    <id>http://starbucks.example.org/order/1234</id>
      ...
  </entry>
 ...
</feed>
```

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.

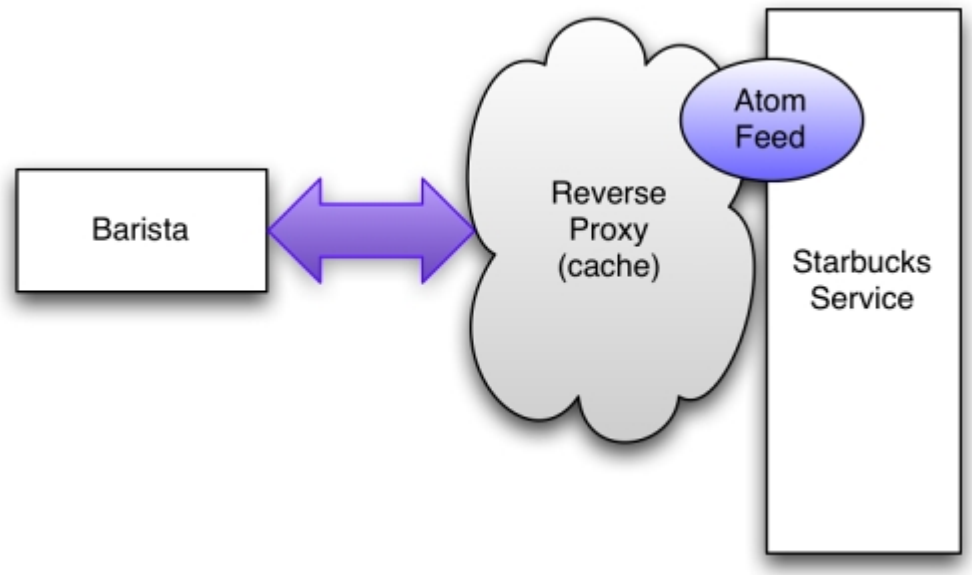


Figure 14 Caching for scalability

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], 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.

```
<entry>
  <published>2008-06-10T19:18:43Z </published>
  <updated>2008-06-10T19:20:32Z</updated>
  <link rel="alternate" type="application/xml"
    uri="http://starbucks.example.org/order/1234"/>
  <id>http://starbucks.example.org/order/1234</id>
  <content type="text+xml">
    <order xmlns="http://starbucks.example.org/">
      <drink>latte</drink>
      <additions>shot</additions>
      <cost>4.00</cost>
    </order>
    <link rel="edit"
      type="application/atom+xml"
      href="http://starbucks.example.org/order/1234/>
  ...
  </content>
</entry>
```

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 the `entry`. 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.

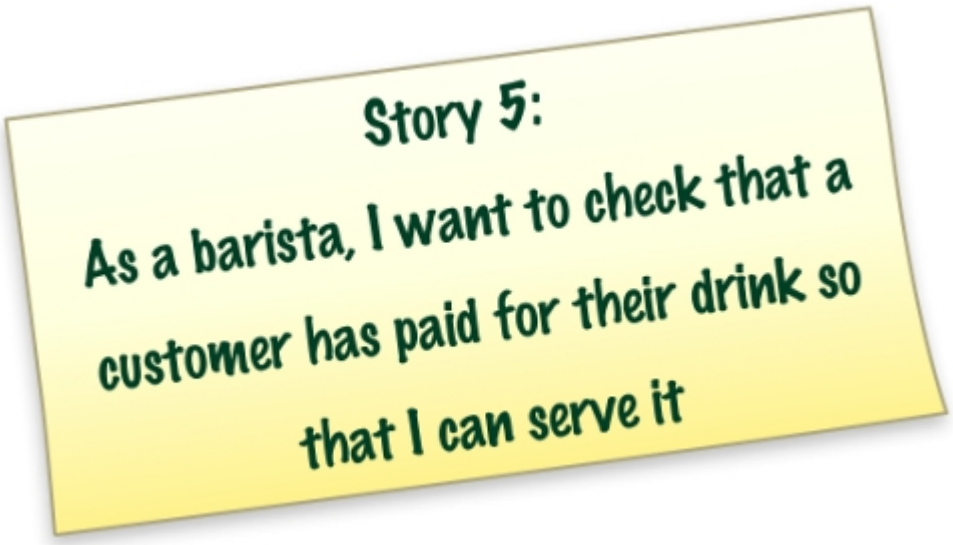
```
PUT /order/1234 HTTP 1.1
Host: starbucks.example.com
Content-Type: application/atom+xml
Content-Length: ...

<entry>
  ...
  <content type="text+xml">
    <order xmlns="http://starbucks.example.org/">
      <drink>latte</drink>
      <additions>shot</additions>
      <cost>4.00</cost>
      <status>preparing</status>
    </order>
    ...
  </content>
</entry>
```

Figure 16 Changing the order status via AtomPub

Once the server has processed the `PUT` request in Figure 16, it will reject anything other than `GET` 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 `GET`ting 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* `/payments` *feed 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 `DELETE`s 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 `DELETE`d from the feed, a fresh `GET` of the feed returns a representation without the `DELETE`d 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

```
...
<next xmlns="http://example.org/state-machine"
    rel="http://wifi.example.org/free-offer"
    uri="http://wifi.example.com/free-offer/order/1234"
    type="application/xml"/>
...
```

  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.

Acknowledgements

The authors would like to thank Andrew Harrison of Cardiff University for the illuminating discussions around “conversation descriptions” on the Web.

About the Authors

Dr. Jim Webber is director of professional services for ThoughtWorks where he works on dependable distributed systems architecture for clients worldwide. Jim was formerly a senior researcher with the UK E-Science programme where he developed strategies for aligning Grid computing with Web Services practices and architectural patterns for dependable Service-Oriented computing and has extensive Web and Web Services architecture and development experience. As an architect with Hewlett-Packard, and later Arjuna Technologies, Jim was the lead developer on the industry's first Web Services Transaction solution. Jim is an active speaker and is invited to speak regularly at conferences across the globe. He is an active author and in addition to "Developing Enterprise Web Services - An Architect's Guide" he is working on a new book on Web-based integration. Jim holds a B.Sc. in Computing Science and Ph.D. in Parallel Computing both from the University of Newcastle upon Tyne. His blog is located at http://jim.webber.name .

Savas Parastatidis is a Software Philosopher, thinking about systems and software. He investigates the use of technology in eResearch and is particularly interested in Cloud Computing, knowledge representation and management, and social networking. He's currently with Microsoft Research's External Research team. Savas enjoys blogging at http://savas.parastatidis.name.

Ian Robinson helps clients create sustainable service-oriented capabilities that align business and IT from inception through to operation. He has written guidance for Microsoft on implementing service-oriented systems with Microsoft technologies, and has published articles on consumer-driven service contracts and their role in the software development lifecycle - most recently in The ThoughtWorks Anthology (Pragmatic Programmers, 2008) and elsewhere on InfoQ. He speaks regularly at conferences on subjects that include RESTful enterprise development and the test-driven foundations of service-oriented delivery.

---

1 An ETag (an abbreviation of Entity Tag) is a unique identifier for the state of a resource. An ETag for a resource is typically an MD5 checksum or SHA1 hash of that resource's data.

2 We'll see how authentication works from Starbucks' point of view later.

3 When safety is at stake, of course, we just prevent things from going too far wrong in the first place! But receiving coffee isn't a safety critical task, even though it might seem that way for our co-workers most mornings!

4 HTTP 1.1 offers some useful request directives, including `max-age`, `max-stale`, and `max-fresh`, which allow the client to indicate the level of staleness it is prepared to accept from a cache.