Chapter 3
RESTful Domain Application Protocols

Ian Robinson

Abstract This chapter discusses the significance of domain application protocols in distributed application design and development. Describing an application as an instance of the execution of a domain application protocol, it shows how we can design RESTful APIs that allow clients to drive the execution of a domain application protocol without binding to the protocol itself. The second half of the chapter provides a step-by-step example of a RESTful procurement application; this application realizes a procurement protocol in a way that requires clients to couple simply to media types and link relations, rather than to the protocol.

Introduction

This chapter reflects the concerns of systems architects and developers charged with satisfying specific business needs – with getting things done. REST's hypermedia constraint (Fielding 2000) is all about getting things done: at the heart of the constraint is a compact of application, application protocol and application state that addresses the need to do useful things with computerized behaviors, to effect the kinds of changes in application state that release value to the providers and consumers of a business capability.

From an analytical perspective, every useful application of computerized behavior can be said to evidence what I call an underlying *domain application protocol* – much as every meaningful utterance evidences an underlying natural-language grammar. The design strategies I present in this chapter represent acts of deliberate discovery through which we come to understand the domain protocols behind specific, domain-sensitive applications of computerized behavior.

I. Robinson (✉)
Neo Technology, Menlo Park, CA, USA
e-mail: iansrobinson@gmail.com

E. Wilde and C. Pautasso (eds.), *REST: From Research to Practice*,
DOI 10.1007/978-1-4419-8303-9_3,

Domain application protocols specialize the interactions between the participants in a distributed application. This specialization is a good thing insofar as it helps support successful domain outcomes. Implemented unwisely, however, specialization inhibits a system's evolution and the serendipitous reuse of its components outside their original context. To overcome this problem, a RESTful API communicates specialization using several of the Web's more generalized mechanisms: namely media types, link relations and HTTP idioms. These artifacts help communicate a domain protocol without our having to import a specific process description into the client part of an application: the resulting domain application protocol is no more written on the surface of the API than a grammar is written on the surface of a sentence.

HTTP is *the* application protocol (Paul Prescod 2002), a domain-agnostic set of rules and conventions for accessing and manipulating resource representations in a uniform manner. Do we really need to introduce the concept of a domain application protocol when we already have the ubiquitous HTTP at our disposal? The answer, I believe, is: yes. Experience suggests that HTTP's domain agnosticism, while enormously beneficial in terms of standardization and interoperability, nonetheless leads to a shortfall in domain semantics. This shortfall must be remedied by every application in its own fashion, most often through prose documentation. HTTP doesn't tell us how to publish web content [the Atom Publication Protocol (Gregorio and de hOra 2007) remedies that], or how to manage cloud resources [The Sun Cloud API (2009) remedies that], or how to procure goods. To achieve a degree of specialization, both AtomPub and Sun's Cloud API apply specific web artifacts – HTTP idioms, media types, and, in the case of AtomPub, link relations – to achieve specific application goals. To retain the generalized benefits of HTTP's uniform interface, both require clients to bind to these web artifacts, rather than to the domain protocols themselves. In doing so, neither restricts a client from applying a system's resources in other contexts and for other purposes. This is the very same approach that I adopt here.

What Is a Domain Application Protocol?

To answer this question, consider the business process shown in Fig. 3.1.

Figure 3.1 illustrates the sequence of interactions that must take place for a customer to purchase some goods from a supplier.[1] The customer asks the supplier for a quote. On receiving a quote, the customer decides to order the goods for which they have been quoted. Once the supplier has confirmed the order, the customer pays for the goods, or cancels the order.

[1]This example simplifies the set of interactions encountered in a real-world application in order to highlight the key points in protocol design.

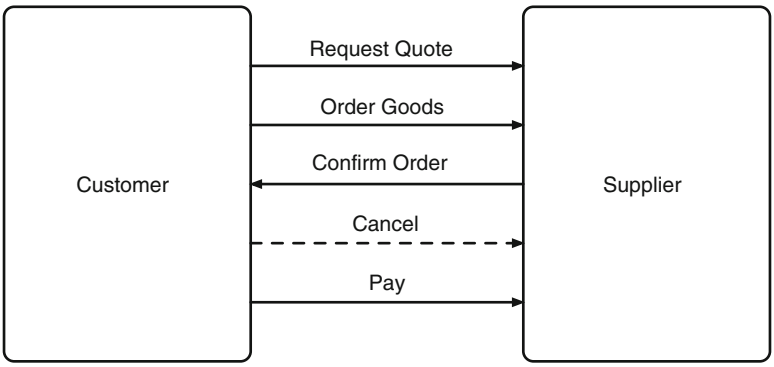

Fig. 3.1 A simple procurement process

Imagine that we have been charged with exposing this procurement process to third parties over the Web. A specific business need – the desire to allow customers to order and pay for goods – motivates a specific engineering task: that of exposing our quoting, order processing and payment functions in a way that allows customers to execute and complete our procurement process in a repeatable, well-understood manner. At the same time, however, we must remain mindful of the fact that other applications may wish to reuse parts of our system for entirely different purposes. Despite having been motivated by the specific business need behind this first project, we do not want to overly specialise our systems' interfaces; rather, we want to implement our APIs in ways that allow them to be composed into other applications and processes.

Fast forward to a time when we've built and deployed a solution to meet our business' procurement needs, and a client has just successfully completed an instance of our procurement process. In order to reach the successful conclusion of the process, the client had to initiate a series of legitimate interactions with whatever systems we'd chosen to expose over the network. The successful completion of the process implies the effective existence of a *domain application protocol*, a set of rules and conventions through which participants in a distributed system coordinate their interactions to achieve a useful, domain-specific application goal.

In the context of a RESTful web application, a domain application protocol is an abstraction of the media types, link relations, and HTTP idioms necessary to achieve a particular application goal. The design of a domain application protocol incorporates the deliberate discovery activities necessary to describe a RESTful API in terms of specific media types and links relations, plus a context-sensitive narrowing of HTTP.

Application

We call the actual occurrence of a set of interactions between participants in a distributed system an *application*. An application, in other words, is computing in

action: computerized behavior directed towards achieving a particular client or end user goal. A distributed application is one in which multiple participants employ computing behavior to realize an application goal. By this definition, an application is not so much a *thing* as a *doing*; it is the very act of putting software to work to realize some benefit. Importantly, an application has duration – it unfolds in time.

Application State

Application state is a snapshot of the state of a distributed application at a particular point in time. Because an application has duration, its state changes over time. Once an application's goal has been achieved, the application can be considered to be in its final state. Prior to achieving this final state, the application passes through one or more intermediate states.

In the context of a conversation between participants in a distributed application, we can also think of application state as being the state of the conversation at a particular point in time. In this respect, application state guarantees the integrity of a sequence of requests. For example, if a client obtains an authenticated token at a certain point in a conversation, it can supply this token with all subsequent requests. Each request then contains sufficient application state information for the server to handle the request without recourse to a server-side session store.

Domain Application Protocol

A domain application protocol is the set of rules and conventions that guides and constrains the interactions between participants in a distributed application.[2] By adhering to a protocol, participants achieve a useful domain or business outcome. Revisiting our definition of application, we can say that an application is an *instance* of the execution of a protocol. In executing the protocol, the participants create an application, which in turn achieves an application goal.

To achieve an application goal in the context of a RESTful web application, a client progressively interacts with a community of resources. These resources can be hosted and governed by a single server, or they can be distributed across the network. Either way, every resource implements the same uniform interface, which in the case of a web application is HTTP.

[2]The term "domain application protocol" and the three-step design methodology described here were first proposed in Webber et al. (2010). We chose the term "domain application protocol" so as to align it both with the book's focus on automating business (domain) processes, and with our use of the terms "application" and "application state." A domain application protocol is more commonly referred to as a coordination protocol: see, for example, Alonso et al. (2004).

Application State in a RESTful Application

Having a server remember the state of each client conversation is expensive, particularly at web scale. To alleviate this burden, a RESTful web application delegates the responsibility for remembering the overall state of an application to the client or clients participating in that application.

As a host of application state, the client in a RESTful web application is responsible for the integrity of a sequence of actions. After each interaction the client is presented with one or more options to interact with additional resources. Servers encode these options in responses using links and forms – otherwise known as *hypermedia controls*.[3] The client decides which control to operate based on its understanding of the current state of the application.

Occasionally, a client may need to add some portion of application state information to its next request in order to provide sufficient application state context for the processing of that request. For example, if the client has received an authorization token in a previous response, it might add this token to all subsequent requests (by sending it in an `Authorization HTTP` request header), thereby conveying to the server the portion of application state information necessary to handle the request.

Design Steps

When automating multi-party business procedures in a RESTful web application, the following three-step process can help guide our design and implementation activities:

1. Model applications as application protocol state machines.
2. Implement them based on resources, resource life cycles and the server-governed rules that associate resources.
3. Document and execute them using media types, link relations, and HTTP idioms.

Step 1 is concerned with the design of an abstract domain application protocol. This step is accomplished without reference to any particular architecture or technology. Steps 2 and 3, on the other hand, focus on the choices particular to the design of a RESTful application, with Step 3 concentrating on the elaboration of a RESTful API.

In practice, the design and implementation of a RESTful web application will not always follow this three-step process. Step 1 in particular is often omitted. For applications whose protocols are relatively trivial, this is perfectly acceptable. Such is the case with simple data services: CRUD (Create, Read, Update and Delete) is a

[3] See Chap. 5, "Hypermedia Types," for a more thorough, and more nuanced, discussion of hypermedia control capabilities.

protocol, albeit a very simple one. Most CRUD-based data services are designed and implemented without reference to the underlying domain application protocol. This doesn't, however, mean that there isn't a protocol – only that we haven't modelled it explicitly. Every application is an instance of a protocol, no matter how simple or implicit.

Whereas Step 1 is optional, Steps 2 and 3 usually proceed iteratively and in parallel. We start by identifying a number of candidate resources, and then detail the HTTP interactions through which a client manipulates representations of these resources. In working through these interactions, we discover additional resources that help adapt the domain to the goals expressed in the protocol. In the worked scenario later in this chapter, for example, we discover some forms-based resources; these resources allow a client to request a quote, submit an order, and cancel an order.

Step 1

As part of the process of articulating a domain application protocol and understanding how it contributes to the successful achievement of an application goal, we create an application state machine representation of the state transitions to be coordinated by the protocol. It is important to point out here that this state machine representation of the protocol is neither an implementation artefact nor public documentation; it simply aids analysis. By explicitly modelling a protocol as a state machine, we gain a better understanding of the "value stream" of application state transitions through which value is released both to the customer and to the organisation(s) owning a process.

Figure. 3.2 shows the several different application state transitions that occur when we execute our procurement protocol. The application terminates when it is in either a *Paid* or a *Cancelled* state. Prior to that, the application passes through the *Quote Requested*, *Goods Ordered* and *Order Confirmed* states.

A procurement application passes through these several different states no matter how it is implemented. Each state refers to the state of the distributed application as a whole (the system), rather than to the state of an individual participant (customer or supplier) or entity (quote, order or payment).

Step 2

On the server side, a RESTful web application is based around resources and resource life cycles.

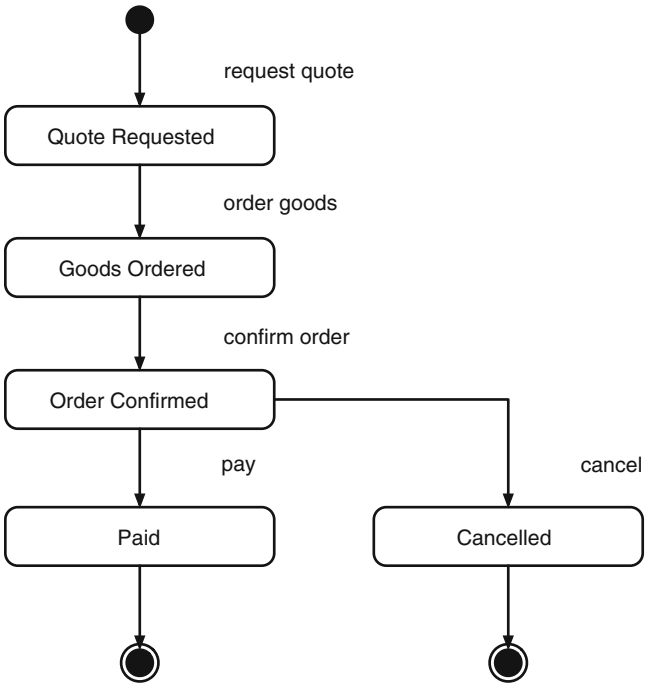

Fig. 3.2 Procurement protocol domain application state machine

Resources

Proponents of web-based systems define the resource abstraction in several different ways. In the most general definition, a resource is simply anything that can be identified by a URI (Berners-Lee et al. 2005). Such a definition lends itself to an inside-out, server-centric view, which sees resources as stateful "things" residing on the server. In contrast, the REST thesis (Fielding 2000) defines a resource as being a membership function, which groups a set of equivalent resource representations and identifiers. Membership of this set can vary over time. In a similar vein, (Booth 2006) sees a resource in terms of a set of state-dependent network functions that accept and return representations. Complementing these several viewpoints, I propose that resources be understood less in terms of what they *are*, and more in terms of what they *do*; resources *adapt* server-based capabilities so that hypermedia clients (i.e., clients that use HTTP's uniform interface to drive an application forwards) can use them.

A hypermedia client applies networked data in pursuit of its application goals. Consequently, a hypermedia system can be regarded as the partial application of networked data to a client or end user goal. Each response to a client request comprises a partial data structure; partial insofar as some of the data items represent links or forms that must be activated to retrieve or produce more data. Clients extend

the structure by applying some of this data back to the network through the uniform interface. Applying the data – operating a link or form – only partially completes the structure; more often than not, it reveals yet more links and forms.

By emphasizing the resource's role in adapting server capabilities for consumption by network-oriented clients, we address one of the downsides of the server-centric perspective, which is the tendency to treat resources in terms of a relatively closed set of domain entities, coarsely manipulated through a small set of verbs. While suitable for simple CRUD-based data services, this entity-oriented attitude to building RESTful systems fails to address the needs of more sophisticated processes. The protocol perspective suggests that resources do not map directly to domain entities; rather, they serve to adapt the domain for its partial application through hypermedia and the uniform interface. Adapting a domain for consumption by a hypermedia client results in our identifying more resources than would normally be identified through a domain-entity-oriented approach. From the client's point of view, domain (i.e. business) behaviors emerge as a side effect of applying a relatively closed set of document-oriented verbs to this open set of resources.

Resource State

A resource has state, and this state, much like application state, can have its own lifecycle. But whereas application state lends integrity to a sequence of interactions with multiple resources, resource state is concerned solely with the state of an individual resource. This resource state is governed and maintained by the server hosting the resource. Attempts to manipulate a resource's state representations must conform to the business rules the server uses to govern the lifecycle of the resource. Such business rules are private to the server and should never be exposed to clients.

For most resources, a resource's state is simply a function of its data. For some resources, however, a resource's state is also partly a function of the state of other resources with which the resource is associated through some server-governed rules. For example, the state of an order is partly a function of the state of any payment with which that order has been associated by the server hosting the order. As with any other business rules governing the state of a resource, these rules remain hidden behind the RESTful interface.

Servers, then, are responsible for maintaining resource state, not application state. Application state remains significant, however. The overall distributed application still moves through several different application states. What's important is that the application state model (and the corresponding protocol) is nowhere reified on the server side. Changes to the state of the overall distributed application emerge as a *side effect* of the client manipulating the states of individual resources through their representations.

Through interacting with a community of resources, a client progressively realizes an application goal in accordance with an implicit domain application protocol. Some client interactions retrieve representations of resource state, others

manipulate that state. Interactions that manipulate representations of resource state manifest an implicit domain application protocol such that resource state transitions occur in a legitimate sequence. It only makes sense to create a payment resource if one has first created an order with which the payment might be associated – and any good system design ought encourage this kind of behavior. How, then, do we encourage such behaviors in a RESTful web application?

Hypermedia

We coordinate a client's interactions with a community of resources by applying REST's hypermedia constraint (Fielding 2000) to the design of our resources and their representations. In this context, the hypermedia constraint is best summarized as, "hypermedia systems change application state."

A hypermedia system comprises a client, one or more server-governed resources, and some systemic behavior. This systemic behavior is initiated when a client makes a request of a resource – in a web application this will be a resource identified by a URI. The resource responds with a representation of its resource state. This representation includes one or more hypermedia controls – links and forms – which advertise legitimate interactions with other resources. The client processes the response and updates its understanding of the current state of the application. If it hasn't yet achieved its application goal, the client chooses the hypermedia control best suited to making forward progress, and operates that control. Operating the control triggers another request, and the cycle begins again.

When generating a response, the server that hosts and governs a resource uses the resource's state plus any application state information supplied by the client in the request to determine which controls to include in the response.

Step 3

A RESTful API is documented using media types, link relations and HTTP idioms.

Media Types

A media type value, such as `application/atom+xml`, is a key into a data format. While not all media types possess the capabilities necessary to implement a hypermedia system, those that do typically define one or more of the following:

- The format to be used for representing content.
- One or more schemas to which content must conform.
- Processing models for schema elements.
- Hypermedia control formats.
- Semantic annotations for hypermedia controls.

Fig. 3.3 A <link> element with semantic annotation

```
<link
  xmlns:rb="http://relations.restbucks.com/"
  rel="rb:order"
  type="application/restbucks+xml"
  href="http://restbucks.com/orders/9876"/>
```

The Atom Syndication Format (Nottingham and Sayre 2005), for example, includes all these elements. In terms of representation format, Atom is based on XML. With regard to schemas, the Atom specification includes two RELAX NG schemas: one for feeds, another for entries. To these it adds a processing model, which determines how content, foreign mark-up and extensions to the Atom vocabulary should be interpreted. In terms of its hypermedia capabilities, it identifies the <atom:link> element as a hypermedia control, and defines five link relation values (*alternate*, *related*, *self*, *enclosure*, and *via*) with which links can be annotated with semantic context. The Atom Syndication Format interpretative scheme is keyed off the application/atom+xml value in Content-Type request and response headers.

Link Relations

On the human web, we intuitively understand what links and forms mean based on the context in which they appear. Machines, on the other hand, cannot reliably infer such implicit semantics. In order to help machine clients decide which hypermedia control to activate in a received resource representation, we must provide some additional, explicit semantics. One of the most popular ways of adding semantic context to hypermedia controls is to annotate links with link relations.

Link relations describe the purpose of a link, the meaning of a target resource, or the relationship between a link's context and the target resource. By stating the purpose of a link, a link relation helps a client use the link according to its purpose. The semantic range of a link relation can vary from describing how the current link's context is related to another resource, to indicating that the target resource has particular attributes or behaviors.

HTML defined the rel attribute for annotating both anchor and link elements with link relations. This attribute convention was adopted by several other formats, including the Atom Syndication Format. Links that have been annotated with a link relation value are called typed links.

Figure. 3.3 shows a typed link taken from the example later in this chapter. The link has been typed with the link relation value *rb:order*. This value acts as a key into a semantic. In this instance, the associated semantic indicates that the linked or destination resource is an order.

Link relations come in one of two flavors: *registered* and *extension* (Nottingham 2010). Registered relations are registered with IANA's Link Relation Type registry (Link Relations 2011). Such well-defined link types take the form of simple string tokens. Examples of registered relation types include *self* and *payment*. Extension

relations, on the other hand, are types that have not been registered with IANA. Such relations are often proprietary to an organisation or application. In order to disambiguate them from any similarly named relations elsewhere, they take the form of a URI. The link relation shown in Fig. 3.3 is an extension relation. It has been formatted as a compact URI (Birbeck and McCarron 2009); expanding the URI returns the absolute link relation value http://relations.restbucks.com/order.

Documenting a Protocol

A RESTful protocol is surfaced using an API composed of media types, link relations and HTTP idioms. Both the Atom Publication Protocol (AtomPub) (Gregorio and de hOra 2007) and Sun's Cloud API (The Sun Cloud API 2009) describe themselves in precisely these terms.

A protocol can draw on pre-existing media types and link relations, as well as invent its own. AtomPub is a good example of this compose-and-invent approach. AtomPub reuses the Atom media type, which is defined in the separate Atom Syndication Format specification; to this, it adds two new media types, `application/atomsvc+xml` and `application/atomcat+xml`, for representing service and category documents. To Atom's five link relations, AtomPub adds two more: *edit* and *edit-media*.

HTTP Idioms

A domain application protocol lends domain meaning to a distributed application's HTTP-based interactions. While all such interactions continue to adhere to the HTTP application protocol, their significance with respect to a client's application goal is determined by the domain application protocol. A domain application protocol constrains HTTP in the context of a particular application; from the client's perspective, this creates a temporally varying subset of HTTP idioms the client can use to manipulate representations of the resources participating in the protocol.

There are two approaches to communicating which HTTP idioms a client should use over the course of an application: upfront, and inline. With the upfront approach, we create a document describing the appropriate idioms. With the inline approach, we use HTTP headers and status codes, plus entity body control data, to communicate at runtime which idioms a client can use to manipulate resource representations.

The Atom Publication Protocol exemplifies the upfront approach. The AtomPub specification explicitly states that resources can be created with `POST`; that successfully creating a resource results in a response with a `201 Created` status code and `Location` header; that `PUT` and `DELETE` can be used to edit resources; and that all edits should be done in a conditional fashion (using an `If-Match` header with a previously supplied entity tag value).

The upfront approach determines which idioms are applicable to an application prior to any client beginning an application instance. In contrast, the inline approach

effectively "programs" the client on the fly. The advantage of the inline approach is that it makes it easier to evolve and extend an application over time.

Here are some of the ways we can use HTTP headers, status codes and entity body control data to describe at runtime which HTTP idioms a client should use to manipulate resource representations:

- `Cache-Control` directives instruct intermediaries to cache content in accordance with HTTP's caching rules.
- Forms (HTML, XForms, etc.) program clients with control data (such as a URI, HTTP verb, and required `Content-Type` value), which the client can then use to encode and submit the form.
- `ETag` headers indicate to the client that subsequent requests for the same resource should use a conditional idiom: either a conditional `GET`, which uses an `If-None-Match` header with an entity tag value to instruct the server to return a full-blown response only if the resource addressed in the request *has* changed since the entity tag value was issued; or a conditional `PUT, POST` or `DELETE`, which uses an `If-None` header with an entity tag value to instruct the server to apply the request if *and only if* the resource addressed in the request *has not* changed since the entity tag value was issued.
- Some of the HTTP status codes determine the next HTTP idiom to be used; `303 See Other`, for example, instructs the client to issue a `GET` request for the resource specified in the accompanying `Location` header.
- `405 Method Not Allowed` tells the client that the verb in the request cannot be used; issuing an `OPTIONS` request for the same resource will return a `200 OK` response with an `Allow` header specifying which verbs can be used.

A RESTful Procurement Application

The remainder of this chapter comprises a narrative exposition of a set of HTTP interactions through which a client executes an instance of our procurement protocol. The example is set in Restbucks, a fictional coffee shop in a world of coffee-loving HTTP robots.[4] Starting from modest roots, Restbucks now has a number of retail outlets. Recently, it has decided to sell coffee beans direct to consumers.

In the course of this narrative, we'll see how the state of the procurement application changes as a result of the client accessing and manipulating representations of resource state. The narrative represents not so much a documented design as it does an act of deliberate discovery, as per the three-step methodology outlined earlier. We've already drawn the abstract protocol (Fig. 3.2) for Step 1: in the narrative that follows we identify a candidate set of resources, media types, link relations and

[4]Restbucks served as the basis for the examples in Webber et al. (2010).

```
Request:
GET /shop HTTP/1.1
Host: restbucks.com

Response:
HTTP/1.1 200 OK
Date: Mon, 26 Jul 2010 10:00:00 GMT
Cache-Control: public, max-age=86400
Content-Type: application/restbucks+xml
Content-Length: ...

<shop xmlns="http://schemas.restbucks.com/shop"
  xmlns:rb="http://relations.restbucks.com/">
  <link rel="rb:rfq"
    href="http://restbucks.com/request-for-quote"
    type="application/restbucks+xml"/>
</shop>
```

Fig. 3.4 Client starts the application

HTTP idioms (Steps 2 and 3, performed in parallel). Throughout, we make design decisions regarding resource boundaries, the connections between resources, HTTP headers, representation formats, and the placement of links and forms – all of which help drive out an API which is both specialized to the protocol *and* amenable to serendipitous reuse.

In the accompanying diagrams, arrow-headed arcs represent requests, while nodes represent responses. A response is shown either as a document containing typed links or a form, or as a status code requiring further action from the client. The round-cornered dashed boxes represent application states. These application states are not built into any of the server-side resources; rather, they have been superimposed onto the diagrams from the perspective of a third-party observer of the entire distributed application.

Start

Every application needs at least one entry point, located at a well-known URI, through which a client can initiate a sequence of interactions – and our procurement application is no different. To start the application, clients navigate to http://restbucks.com/shop, as shown in Fig. 3.4.

The response shown in Fig. 3.4 includes two headers of note: `Cache-Control` and `Content-Type`.

The `Cache-Control` header influences the behavior of any caching intermediaries – local caches, proxies, and reverse proxies – along the request–response path. Caching allows us to store copies of a representation closer to clients, thereby helping to conserve bandwidth, reduce latency, and minimize load on the origin server. In this instance, the header includes two directives: `public`, which

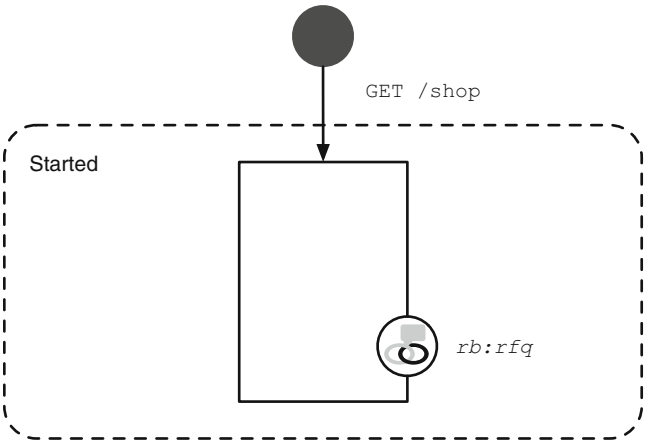

Fig. 3.5 Application begins in a *Started* state

makes the response cacheable by all intermediaries, both private and shared; and `max-age=86400`, which indicates that the response will remain fresh for up to one day after it was issued by the origin server. Together, these two directives ensure that the majority of requests for the procurement "homepage" are satisfied by the caching infrastructure, rather than by the origin server.

The response's `Content-Type` header has a media type value of `application/restbucks+xml`. This is a proprietary, but nonetheless reasonably generalized, format for representing quotes and orders; it is documented in more detail at the end of this chapter.

Below the response header block is the entity body, comprising an XML-formatted representation of the shop's homepage. This entry-point resource representation advertises the procurement application's capabilities. It currently contains a single `<link>` element. The link is typed *rb:rfq*, indicating that the resource at the other end of the link allows the client to request a quote.

An entry-point resource such as this is the ideal place to advertise new capabilities. If, for example, we were to evolve our application to include search functionality, we might advertise this new capability by adding a typed link (leading to a search form) to the shop's entry-point resource representation.

With this first client request, the overall distributed application enters the *Started* state, as shown in Fig. 3.5.

Request Quote

Having started the application, the client now processes the shop representation. The representation contains only one typed link, so to make forward progress, the client issues a request for the request-for-quote form, as shown in Fig. 3.6.

```
Request:
GET /request-for-quote HTTP/1.1
Host: restbucks.com

Response:
HTTP/1.1 200 OK
Date: Mon, 26 Jul 2010 10:00:05 GMT
Cache-Control: public, max-age=86400
Content-Type: application/restbucks+xml
Content-Length: ...

<model xmlns="http://www.w3.org/2002/xforms"
  schema="http://schemas.restbucks.com/shop.xsd">
  <instance/>
  <submission
    resource="http://restbucks.com/quotes"
    method="post"
    mediatype="application/restbucks+xml"/>
</model>
```

Fig. 3.6 Client gets a request-for-quote form

The response shown in Fig. 3.6 contains an XForms form (Boyer 2009). XForms is an XML vocabulary and data processing model for building web forms inside a host application. It is based on a model-view-controller architecture. The form shown in Fig. 3.6 uses an XForms `<model>` element to communicate control data to the client. The `<submission>` element's `resource, method` and `mediatype` attributes specify the URI, HTTP method and `Content-Type` header value to be used when submitting the form. The `<model>` element's schema attribute references an XML Schema instance to which the submitted content must conform. A client programmed with the correct media type library and appropriate HTTP and XForms processing capabilities can use this inline control data to compose and submit its next request.

Note that the representation format used here doesn't explicitly encode the fact that this form allows the client to submit a request for a quote – there's no `<request-for-quote>` element, for example. This is because throughout our procurement application we use a strategy of providing typed links to forms. The link relation associated with a typed link establishes the meaning of the linked resource. When dereferencing the link, the client retains this contextual knowledge (in this instance, the client understands that the linked resource will allow it to submit a request for a quote), and processes the received form accordingly. In doing so, the client navigates a steady state space. Following the link doesn't change the state of the overall distributed application; it does, however, enrich that state. By following a link to a form, the client discovers new opportunities – and appropriate idioms – for progressing the application further.

The client "fills out" the form – that is, it creates a request whose body conforms to the schema at http://schemas.restbucks.com/shop.xsd – and `POST`s it to the URI supplied in the control data, as shown in Fig. 3.7.

```
Request:
POST /quotes HTTP/1.1
Host: restbucks.com
Content-Type: application/restbucks+xml
Content-Length: ...

<shop xmlns="http://schemas.restbucks.com/shop">
  <items>
    <item>
      <description>Costa Rica Tarrazu</description>
      <amount>250g</amount>
    </item>
    <item>
      <description>Elephant Beans</description>
      <amount>250g</amount>
    </item>
  </items>
</shop>

Response:
HTTP/1.1 201 Created
Date: Mon, 26 Jul 2010 10:01:00 GMT
Location: http://restbucks.com/quotes/1234
Content-Type: application/restbucks+xml
Content-Length: ...

<shop xmlns="http://schemas.restbucks.com/shop"
  xmlns:rb="http://relations.restbucks.com/">
  <items>
    <item>
      <description>Costa Rica Tarrazu</description>
      <amount>250g</amount>
      <price currency="GBP">4.40</price>
    </item>
    <item>
      <description>Elephant Beans</description>
      <amount>250g</amount>
      <price currency="GBP">5.30</price>
    </item>
  </items>
  <link rel="self"
    href="http://restbucks.com/quotes/1234"
    type="application/restbucks+xml"/>
  <link rel="rb:order-form"
    href="http://restbucks.com/order-forms/1234"
    type="application/restbucks+xml"/>
</shop>
```

Fig. 3.7 Client submits a request for a quote

The resource at http://restbucks.com/quotes creates a quote based on the details supplied in the request. (Behind the RESTful interface, this resource contacts a quote engine to generate the quote.) The server returns a response with a `201 Created` status code, a `Location` header indicating the URI of the newly created quote, and an entity body containing a representation of the quote itself. This representation contains two typed links: a *self* link, which is the preferred URI for the quote, and an *rb:order-form* link. The *rb:order-form* link relation indicates that the resource at the other end of the link allows the client to submit an order based on the quote.

As an aside, it's worth noting that there's nothing special about the `POST` request that results from filling out the form shown in Fig. 3.6. In accordance with the XForms processing model, the `<model>` and `<submission>` scaffolding elements have been stripped away by the client. What ends up on the wire is simply the data representing a request for a quote. In other words, we could have added a typed link leading directly to the quotes resource to the application's homepage, and documented in our protocol specification that clients can POST a request for a quote to this linked resource. By using a typed link to a form, however, we avoid specifying specific HTTP idioms upfront. Instead, we put the control data in the form. The downside of using a typed link to a form is that it requires an additional request–response interaction – but given that the blank form is highly cacheable, the overhead of this additional request will be mitigated in many instances by the caching infrastructure.

With this `POST` request and response, the client sees that the overall distributed application's state has changed from *Started* to *Quote Requested*, as shown in Fig. 3.8.

Place Order

Assuming the quote is satisfactory, we can now observe what happens when the client wants to place an order. First, the client follows the quote's *rb:order-form* typed link, as shown in Fig. 3.9. The response contains another XForms form, similar to the one in Fig. 3.6. But whereas the form in Fig. 3.6 was empty, this one has been pre-populated by the server.

The response's `Content-Location` header indicates the source for this form data. The header value refers back to the quote issued earlier in the application. In other words, the entity encoded in the form is also accessible from another location: http://restbucks.com/quotes/1234. The result is that we have two resources, both of which share the same underlying domain data. The first adapts the domain so that a client can receive representations of a quote. The second – the pre-filled form – adapts the domain so that a hypermedia client can advance the procurement protocol by submitting an order based on a previously received quote.

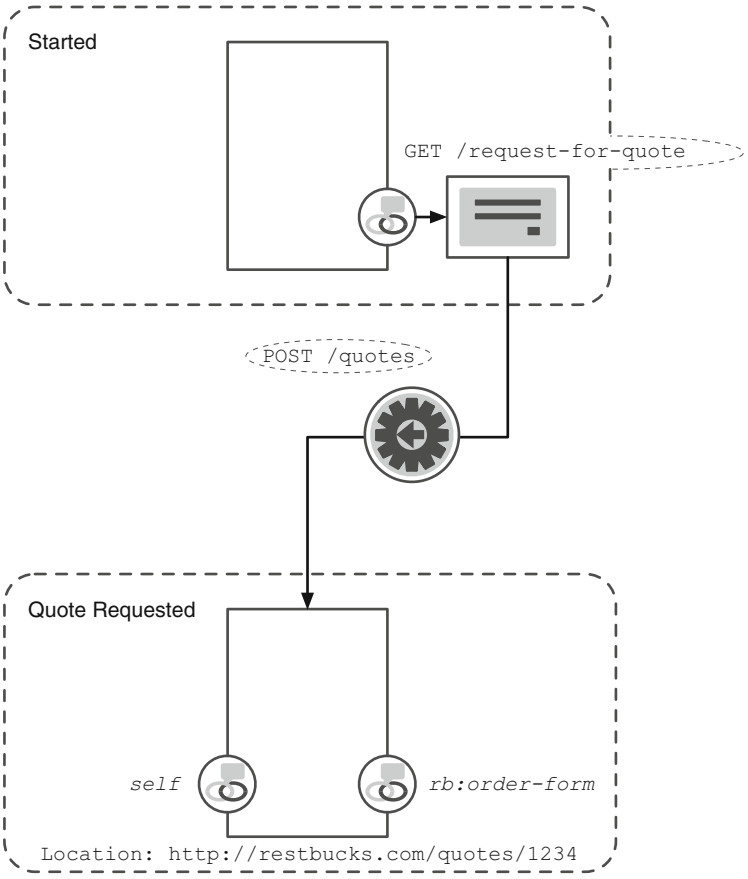

Fig. 3.8 Application state changes from *Started to Quote Requested*

Based on the quote data, the server responsible for this resource has generated a form that can then be `POST`ed to an order processor. The form contains all the information necessary to create an order, thereby eliminating any need for the order processor to look up the original quote. But this strategy, useful as it is in making the message self-sufficient, also raises an issue of message integrity, for the form's target need not be hosted on the same server – that is, the order processor may very well belong in an entirely different subsystem. Because we're passing around quote data, rather than a reference to a quote, a malicious client might be tempted to adjust the quote values prior to submitting the form, thereby earning itself a substantial discount. Given this possibility, how can we prevent clients from tampering with the message?

The solution we've adopted depends on the quoting and order processing subsystems having established a shared key. Prior to sending the response, the quotes resource generates a hash of the form data (the `<shop>` element and its

```
Request:
GET /order-forms/1234
Host: restbucks.com

Response:
HTTP/1.1 200 OK
Cache-Control: public
Date: Mon, 26 Jul 2010 10:01:05 GMT
Expires: Mon, 02 Aug 2010 10:01:00 GMT
Content-Type: application/restbucks+xml
Content-Length: ...
Content-Location: http://restbucks.com/quotes/1234

<model xmlns="http://www.w3.org/2002/xforms">
  <instance>
    <shop xmlns="http://schemas.restbucks.com/shop">
      <items>
        <item>
          <description>
            Costa Rica Tarrazu
          </description>
          <amount>250g</amount>
          <price currency="GBP">4.40</price></item>
        <item>
          <description>Elephant Beans</description>
          <amount>250g</amount>
          <price currency="GBP">5.30</price></item>
      </items>
      <link rel="self"
        href="http://restbucks.com/quotes/1234"
        type="application/restbucks+xml"/>
    </shop>
  </instance>
  <submission
    resource="http://restbucks.com/orders?c=99fe97e1
      ➥&s=k2awEpciJkd2X8rt3NmgDg8AyUo%3D"
    method="post"
    mediatype="application/restbucks+xml"/>
</model>
```

Fig. 3.9 Client gets the order form

children) and signs the hash using this shared secret. It then appends the generated value, together with its client ID, to the form URI, to make http://restbucks.com/orders?c=99fe97e1&s=k2awEpciJkd2X8rt3NmgDg8AyUo%3D. On receiving the `POST`ed form, the ordering subsystem is able to parse out the client ID and signed hash, recalculate its own version of the signed hash, and compare the recalculated value with the received value. [5]

[5]This is an example of a one-time URI. See Allamaraju (2010) for more details of generating one-time URIs.

Note that the design decisions we've made here trade message integrity for increased coupling. The quotes resource and the order processor are coupled through their sharing a secret to sign the hash, and through their sharing a URI template, `/orders?c={clientId}&s={signedHashValue}`, to generate the form URI. Moreover, if the shared secret leaks out, the tamper proofing mechanism will have been compromised.

There is one final thing to note about the response shown in Fig. 3.9. Restbucks has a business rule that says that a quote is valid for up to seven days after it has been issued. As we can see from the quote response in Fig. 3.7, the quote that was recently requested by the client was generated on Monday, 26 July 2010 at 10:01:00 GMT. The `Expires` header attached to the order form response indicates that the form representation can be cached, and will remain fresh, for exactly seven days from when the underlying quote was first issued.

To place its order, the client submits the form, as shown in Fig. 3.10. The order processor responds with `202 Accepted`, indicating that it has successfully received the request but has not yet finished processing it. Both the `Location` header and the typed link in the response body point to a resource that the client can later interrogate to discover the eventual result of processing the request.

The `202 Accepted` status code separates the action of accepting the request from the work necessary to fulfil it. In doing so, it coordinates the successful transfer of the request in the context of an asynchronous server-side task. To create an order in its initial state, a number of potentially slow operations must take place behind the RESTful interface. The order processor must contact a third-party payment provider and set up a transaction (to be completed later by the client); it must also contact the warehouse to determine stock availability. Both of these operations are relatively slow. Rather than have the client hang onto a connection waiting for a response describing the outcome of all this work, we've chosen simply to acknowledge successful delivery of the request while queuing the work itself for subsequent processing.

With this interaction, the client's view of the state of the overall distributed application changes from *Quote Requested* to *Goods Ordered*, as shown in Fig. 3.11.

Confirm Order

The client can now begin to poll the resource identified in the `Location` header of the response shown in Fig. 3.10. In polling, the client becomes responsible for the successful "delivery" of the outcome of its order request. (In contrast, pub/sub solutions depend on either the publisher or a piece of middleware to deliver notifications to subscribers successfully.) Figure. 3.12 shows the client's first attempt at polling the order at http://restbucks.com/orders/9876.

The server responds with `404 Not Found`, indicating that the order has not yet been created (the tasks necessary to create the order in its initial state have not completed). The client waits a couple of seconds, and then tries again, as shown in Fig. 3.13.

```
Request:
POST /orders?c=99fe97e1&s=k2awEpciJkd2X8rt3NmgDg8Ay
➥Uo%3D HTTP/1.1
Host: restbucks.com
Content-Type: application/restbucks+xml
Content-Length: ...

<shop xmlns="http://schemas.restbucks.com/shop">
  <items>
    <item>
      <description>Costa Rica Tarrazu</description>
      <amount>250g</amount>
      <price currency="GBP">4.40</price>
    </item>
    <item>
      <description>Elephant Beans</description>
      <amount>250g</amount>
      <price currency="GBP">5.30</price>
    </item>
  </items>
  <link rel="self"
    href="http://restbucks.com/quotes/1234"
    type="application/restbucks+xml"/>
</shop>

Response:
HTTP/1.1 202 Accepted
Cache-Control: no-store
Date: Mon, 26 Jul 2010 10:02:00 GMT
Location: http://restbucks.com/orders/9876
Content-Type: application/restbucks+xml
Content-Length: ...

<shop xmlns="http://schemas.restbucks.com/shop"
  xmlns:rb="http://relations.restbucks.com/">
  <status>Initializing</status>
  <link rel="rb:order"
    href="http://restbucks.com/orders/9876"
    type="application/restbucks+xml"/>
</shop>
```

Fig. 3.10 Client submits the order form

This time, all the server-side tasks necessary to create an order in its initial state have been completed, so the server responds with a representation of the newly created order. As the value of the order's `<status>` element indicates, the order is *Awaiting Payment*. This is resource state – and a particularly interesting kind of resource state at that, for the state of this order is not only a function of the data proper to the resource, it is also (partly) a function of the state of the payment with which the order was associated when it was created. While the payment is waiting to be completed by the client, this order is in the state of *Awaiting Payment*. The

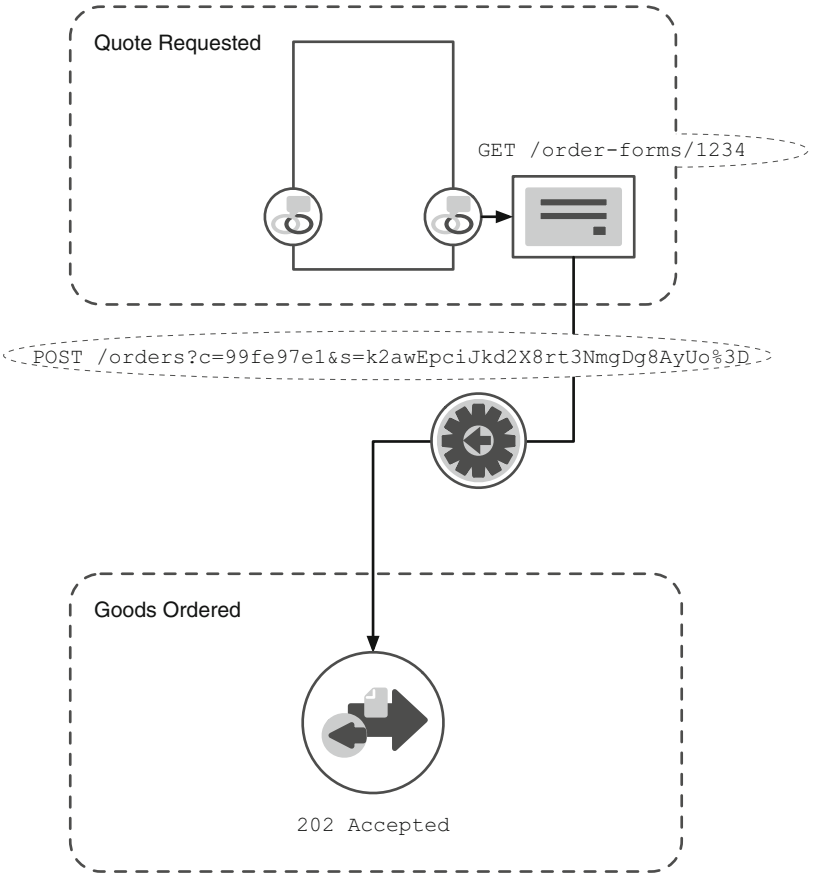

Fig. 3.11 Application state changes from *Quote Requested to Goods Ordered*

Fig. 3.12 The client polls the order resource

```
Request:
GET /orders/9876 HTTP/1.1
Host: restbucks.com

Response:
HTTP/1.1 404 Not Found
```

server responsible for the order resource can "watch" the payment resource in order to compute the state of the order.

The order's resource state, then, can change over time; moreover, it can change as a function of other resources changing state. This kind of situation requires us to make some tradeoffs between consistency and efficient use of network resources. The client here desires a view of the order consistent with the view held on the server; we, however, as designers of a networked application, want to use caching to conserve bandwidth, reduce latency, and save processing cycles.

```
Request:
GET /orders/9876 HTTP/1.1
Host: restbucks.com

Response:
HTTP/1.1 200 OK
Cache-Control: public, max-age=0
Date: Mon, 26 Jul 2010 10:05:00 GMT
ETag: "4d3e88c9"
Content-Type: application/restbucks+xml
Content-Length: ...

<shop xmlns="http://schemas.restbucks.com/shop"
  xmlns:rb="http://relations.restbucks.com/">
  <status>Awaiting Payment</status>
  <items>
    <item>
      <description>Costa Rica Tarrazu</description>
      <amount>250g</amount>
      <price currency="GBP">4.40</price>
    </item>
    <item>
      <description>Elephant Beans</description>
      <amount>250g</amount>
      <price currency="GBP">5.30</price>
    </item>
  </items>
  <link rel="self"
    href="http://restbucks.com/orders/9876"
    type="application/restbucks+xml"/>
  <link rel="rb:quote"
    href="http://restbucks.com/quotes/1234"
    type="application/restbucks+xml"/>
  <link rel="payment"
    href="https://example.org/payments/1010"
    type="application/xhtml+xml"/>
  <link rel="rb:cancellation"
    href="http://restbucks.com/orders/9876/cancel"
    type="application/restbucks+xml"/>
</shop>
```

Fig. 3.13 The client polls the order a second time

Fortunately, there is a way to provide both consistency and – to an extent – cacheability, using, as we have done here, `ETag` and `Cache-Control` headers.

The `ETag` header attached to the response in Fig. 3.13 contains an opaque string token – an entity tag value. An entity tag represents in digest form the state of a resource at the time the entity tag was generated. When the resource changes, the entity tag value changes. Clients and caches can use a previously supplied entity tag value to make efficient queries of the server governing the resource to which the entity tag belongs, as we'll see shortly.

Before we look at how a client or cache can use an entity tag value to maintain consistency in a reasonably network-efficient manner, let's examine the order's `Cache-Control` header. We've made the order resource representation cacheable using a cache-but-revalidate strategy, implemented using two `Cache-Control` directives. The first of these directives, `public`, makes the response cacheable by all caches; the second, `max-age=0`, indicates that the cached response must immediately be treated as stale.

This cache-but-revalidate strategy provides consistency, but at the expense of a small increase in network traffic. Anyone holding a copy of the order must revalidate with the origin server *with every request* using a conditional `GET`. Conditional `GET` requests look like normal `GET` requests, except they also include an `If-None-Match` header, which takes a previously received entity tag as a value. If the resource hasn't changed since the supplied entity tag was generated, the server responds `304 Not Modified`, thereby allowing the requestor to use its cached copy of the order. If the resource *has* changed since the supplied entity tag value was generated, the origin server replies with a full-blown response. This response travels all the way to the client, replacing any cached copies along the response path as it does so.

Returning to the entity body, we see that it contains four typed links: two with registered link relations (*self* and *payment*), and two with extension link relations (*rb:cancellation* and *rb:quote*):

- The *self* link indicates the preferred URI for the order.
- The *rb:quote* link points back to the quote used to created the order.
- The *rb:cancellation* link points to a resource that allows the client to cancel the order.
- The *payment* link refers to a resource that can be used to pay for the order.

With the transmission of the order response, the state of the overall distributed application has changed from *Goods Ordered* to *Order Confirmed*, as shown in Fig. 3.14.

Pay

Choosing now to pay for the order, the client `GET`s the *payment* typed link, as shown in Fig. 3.15. This request is made over a secure channel to a third-party payment provider.

The payment provider's response comprises an XHTML form representation of a payment waiting to be filled out with the client's payment details. The client fills out the form and `POST`s it back to itself. The outcome of this `POST` request depends on the current state of the payment. `POST`ing the client's payment details back to the payment resource for the first time changes the state of the payment from *Awaiting Payment* to *Paid*, and causes the payment to return a `200 OK` response, as shown in Fig. 3.16. Once is in the *Paid* state, however, the payment will no longer

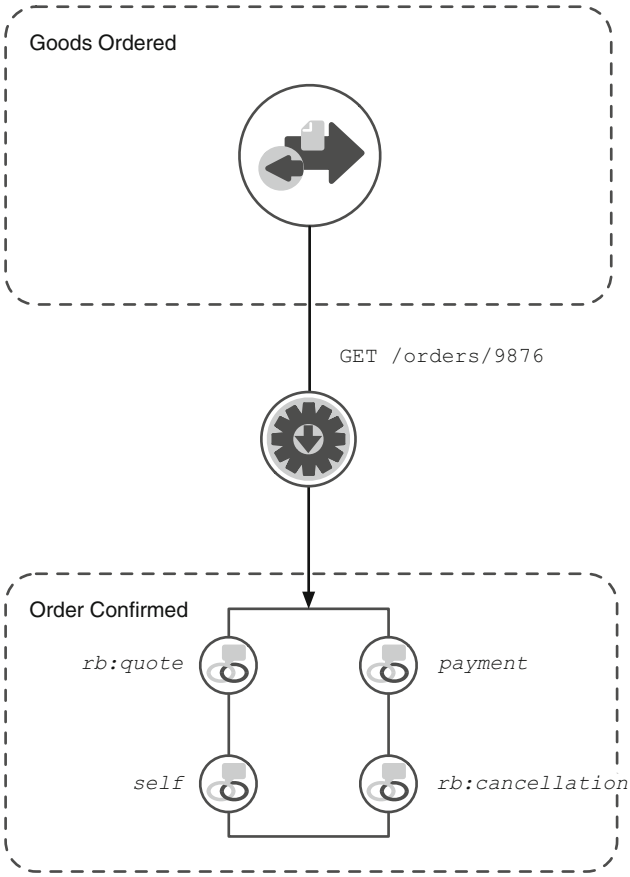

Fig. 3.14 Application state changes from *Goods Ordered to Order Confirmed*

accept `POST` requests; subsequent `POST` requests will cause the resource to return a `405 Method Not Allowed` response instead. In effect, the payment resource implements idempotent `POST`; that is, multiple `POST`s to the payment cause the transaction to be completed only once.

The response shown in Fig. 3.16 comprises another form. When the order processor set up the payment, it supplied the payment provider with a callback URI and confirmation ID. The payment provider uses these details to create a pre-filled payment confirmation form, which the client now submits, as shown in Fig. 3.17.

The resource to which the form data is `POST`ed validates the received confirmation ID and sets the state of the underlying order domain entity to *Paid*. It then redirects the client to the order resource with a `303 See Other` response. As shown in Fig. 3.18, the client makes a `GET` request of the URI supplied in the redirect's `Location` header.

```
Request:
GET https://example.org/payments/1010 HTTP/1.1

Response:
HTTP/1.1 200 OK
Cache-Control: no-store
Date: Mon, 26 Jul 2010 10:05:05 GMT
Content-Type: application/xhtml+xml
Content-Length: ...

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Payment</title>
  </head>
  <body>
    <form method="post"
      action="https://example.org/payments/1010"
      enctype="application/x-www-form-urlencoded">
      <input type="text" name="card-type"/>
      <input type="text" name="name"/>
      <input type="text" name="card-number"/>
      <input type="text" name="security-code"/>
      <input type="submit" value="Submit"/>
    </form>
  </body>
</html>
```

Fig. 3.15 Client gets the payment form

When following the redirect to the order, the client adds the entity tag value it received the last time it requested the order to an `If-None-Match` request header, thereby making the request conditional. This conditional request requires the server to return a full-blown response only if the entity tag associated with the requested entity differs from the entity tag value supplied in the request. Because the order *has* changed since the client last requested it (its resource state has changed from *Awaiting Payment* to *Paid*, and therefore its entity tag value is different), the server returns a full response. This response includes a new entity tag value.

With this last series of interactions, the payment's state has changed to *Paid*, as has the order's. And with these two resource state changes, the client's view of the overall distributed application's state has changed from *Order Confirmed* to *Paid*, as shown in Fig. 3.19. The procurement application has reached a terminal state.

Cancel

Instead of paying for an order, a client may choose to cancel it. (In a real-world application there would likely be several points where the client could choose to cancel the order.) Following a link typed with *rb:cancellation* leads the client to a form, which the client then uses to `PUT` a reason for cancelling the order

```
Request:
POST https://example.org/payments/1010 HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: ...

card-type=Visa+Debit&name=MR+JOHN+SMITH&
➥card-number=4876512418675010&security-code=212

Response:
HTTP/1.1 200 OK
Cache-Control: no-store
Date: Mon, 26 Jul 2010 10:06:00 GMT
Content-Type: application/xhtml+xml
Content-Length: ...

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Payment Confirmation</title>
  </head>
  <body>
    <form method="post"
      action="http://restbucks.com/payments/9876"
      enctype="application/x-www-form-urlencoded">
      <input type="hidden"
        name="confirmation-id">6a0806ca</input>
      <input type="continue" value="Continue"/>
    </form>
  </body>
</html>
```

Fig. 3.16 Client submits payment details

```
Request:
POST /payments/9876 HTTP/1.1
Host: restbucks.com
Content-Type: application/x-www-form-urlencoded
Content-Length: ...

confirmation-id=6a0806ca

Response:
HTTP/1.1 303 See Other
Date: Mon, 26 Jul 2010 10:06:02 GMT
Location: http://restbucks.com/orders/9876
```

Fig. 3.17 Client is redirected to the order

to a cancellation resource. This cancellation resource adapts the underlying order domain entity on behalf of clients wishing to cancel orders. Much as `POST`ting a payment confirmation modifies the underlying order and sets its state to *Paid* (as shown in Fig. 3.17), creating a new cancellation cancels the underlying order.

```
Request:
GET /orders/9876 HTTP/1.1
Host: restbucks.com
If-None-Match: "4d3e88c9"

Response:
HTTP/1.1 200 OK
Cache-Control: public, max-age=0
Date: Mon, 26 Jul 2010 10:06:05 GMT
ETag: "5612cfaa"
Content-Type: application/restbucks+xml
Content-Length: ...

<shop xmlns="http://schemas.restbucks.com/shop"
  xmlns:rb="http://relations.restbucks.com/">
  <status>Paid</status>
  <items>
    <item>
      <description>Costa Rica Tarrazu</description>
      <amount>250g</amount>
      <price currency="GBP">4.40</price>
    </item>
    <item>
      <description>Elephant Beans</description>
      <amount>250g</amount>
      <price currency="GBP">5.30</price>
    </item>
  </items>
  <link rel="self"
    href="http://restbucks.com/orders/9876"
    type="application/restbucks+xml"/>
  <link rel="rb:quote"
    href="http://restbucks.com/quotes/1234"
    type="application/restbucks+xml"/>
</shop>
```

Fig. 3.18 Order is now in a *Paid* state

Documenting the Procurement API

Having described a likely sequence of interactions through which a client can drive the procurement protocol forwards, together with the representation formats, processing models and link relation values necessary to realize these interactions, we're in a position to begin documenting the public face of our system. In large part, this documentation comprises descriptions of the media types and link relations we use throughout the application. It does *not* include any reference to the underlying protocol state machine. By coupling to our media types and link relations, clients allow themselves to be guided towards successfully completing the procurement protocol; at the same time, they are free to compose our resources and their interactions with those resources into entirely different applications.

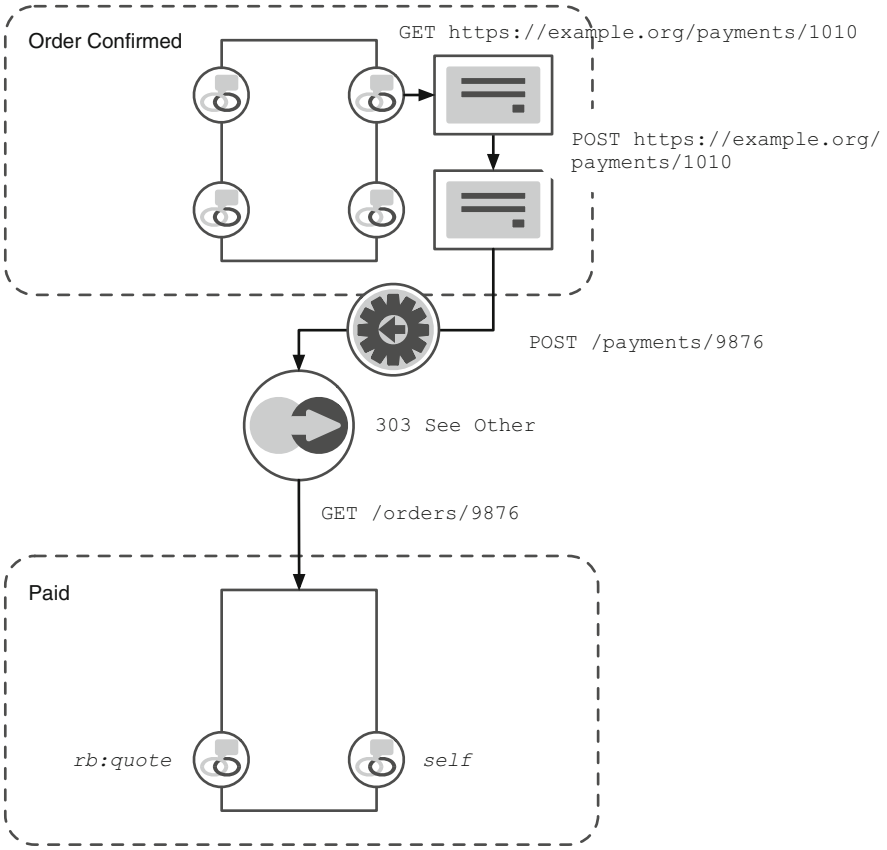

Fig. 3.19 Application state changes from *Order Confirmed to Paid*

The documentation we provide client developers indicates that our procurement application uses the `application/restbucks+xml` media type, together with a couple of registered link relations: *self* and *payment*. We also note that we use a third party payment provider whose protocol uses `application/xhtml+xml`.

The Restbucks Media Type

The documentation for the `application/restbucks+xml` media type says that:

- Responses will contain either a <shop> entity corresponding to the schema described at http://schemas.restbucks.com/shop.xsd, or an XForms `<model>`.

- We use <link> elements to represent links, and XForms <model> elements to represent forms and runtime control data.
- A <shop> may contain zero or more <link> elements, at most one <items> element containing zero or more child <item> elements, and at most one <status> element.
- We use five extension link relation values:

– http://relations.restbucks.com/quote – Indicates that the linked resource is a quote.
– http://relations.restbucks.com/order – Indicates that the linked resource is an order.
– http://relations.restbucks.com/cancellation – Indicates a resource where an order can be cancelled.
– http://relations.restbucks.com/rfq – Indicates a resource where a quote can be requested.
– http://relations.restbucks.com/order-form – Indicates a resource where orders can be submitted.

- User agents can automatically activate links typed with *rb:cancellation*, *rb:rfq* or *rb:order-form*. That is, these link relations indicate external resources that a client can prefetch to enrich its view of a steady state without changing the application's state.
- Clients wishing to use forms to further the application must understand and implement the XForms 1.1 Core Module.

With this documentation, client developers can develop media type libraries that parse and produce representations belonging to each media type, and which implement any processing models particular to those types; they can then compose these libraries into their client-side part of the application.

References

Subbu Allamaraju. *RESTful Web Services Cookbook*. O'Reilly, 2010.
Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag, Berlin, Heidelberg, New York, 2004.
Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. 2005. http://www.ietf.org/rfc/rfc3986.
Mark Birbeck and Shane McCarron (eds). *CURIE Syntax 1.0*. 2009. http://www.w3.org/TR/curie/.
David Booth. *URIs and the Myth of Resource Identity*. 2006. http://dbooth.org/2006/identity/.
John M. Boyer (ed). *XForms 1.1*. 2009. http://www.w3.org/TR/xforms11/.
Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
Joe Gregorio and Bill de hOra (eds). *The Atom Publishing Protocol*. 2007. http://tools.ietf.org/html/rfc5023.
Link Relations. 2011. http://www.iana.org/assignments/link-relations
M. Nottingham. *Web Linking*. 2010. http://www.rfc-editor.org/rfc/rfc5988.txt.
M. Nottingham and R. Sayre (eds). *The Atom Syndication Format*. 2005. http://tools.ietf.org/html/rfc4287.

Paul Prescod. *Roots of the REST/SOAP Debate*, 2002 Extreme Markup Languages Conference, Montréal, Canada, Aug 2002.

The Sun Cloud API. 2009. http://kenai.com/projects/suncloudapis/pages/Home.

Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly, 2010.