Richardson Maturity Model

Steps toward the glory of REST

A model (developed by Leonard Richardson) that breaks down the principal elements of a REST approach into three steps. These introduce resources, http verbs, and hypermedia controls.

Contents

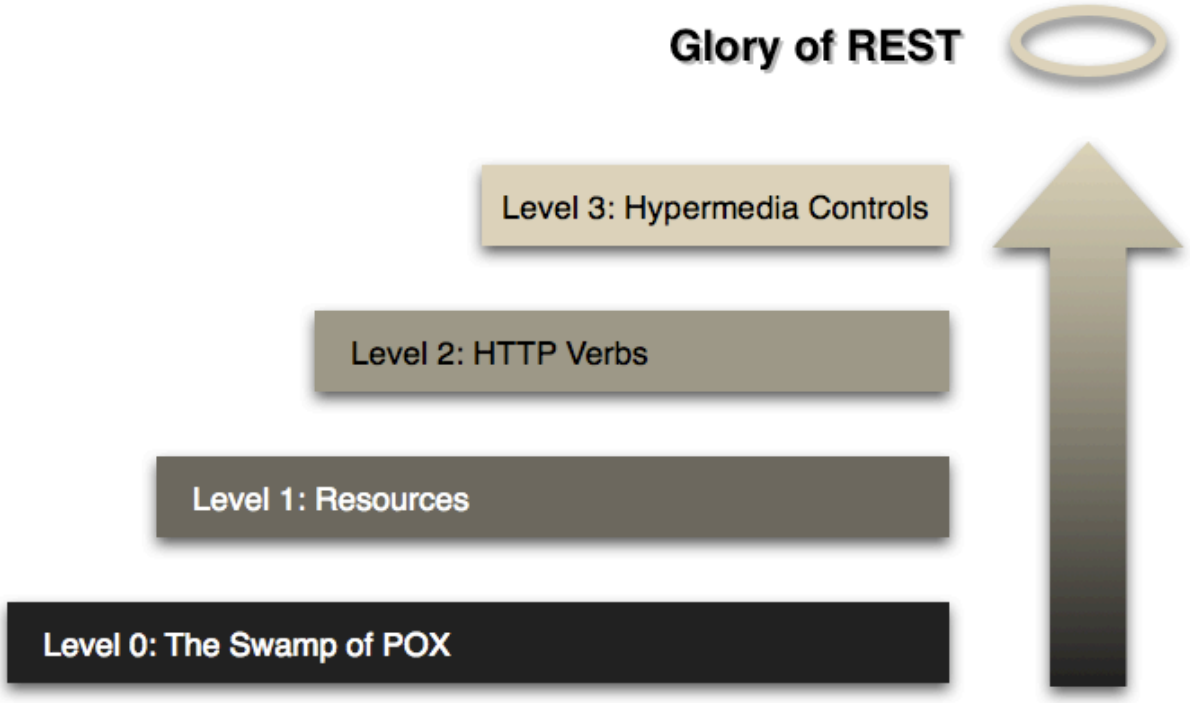

Figure 1: Steps toward REST

To help explain the specific properties of a web-style system, the authors use a model of restful maturity that was developed by Leonard Richardson and explainedat a QCon talk. The model is nice way to think about using these techniques, so I thought I'd take a stab of my own explanation of it. (The protocol examples here are only illustrative, I didn't feel it was worthwhile to code and test them up, so there may be problems in the detail.)

Level 0

The starting point for the model is using HTTP as a transport system for remote interactions, but without using any of the mechanisms of the web. Essentially what you are doing here is using HTTP as a tunneling mechanism for your own remote interaction mechanism, usually based on Remote Procedure Invocation.

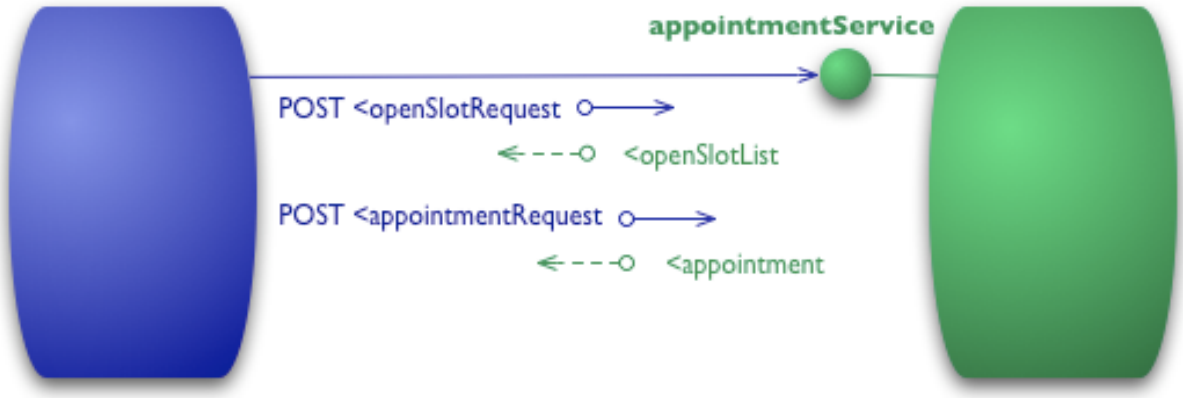

Figure 2: An example interaction at Level 0

Let's assume I want to book an appointment with my doctor. My appointment software first needs to know what open slots my doctor has on a given date, so it makes a request of the hospital appointment system to obtain that information. In a level 0 scenario, the hospital will expose a service endpoint at some URI. I then post to that endpoint a document containing the details of my request.

```
POST /appointmentService HTTP/1.1
[various other headers]

<openSlotRequest date = "2010-01-04" doctor = "mjones"/>
```

The server then will return a document giving me this information

```
HTTP/1.1 200 OK
[various headers]

<openSlotList>
  <slot start = "1400" end = "1450">
    <doctor id = "mjones"/>
  </slot>
  <slot start = "1600" end = "1650">
    <doctor id = "mjones"/>
  </slot>
</openSlotList>
```

I'm using XML here for the example, but the content can actually be anything: JSON, YAML, key-value pairs, or any custom format.

My next step is to book an appointment, which I can again do by posting a document to the endpoint.

```
POST /appointmentService HTTP/1.1
[various other headers]
```

```
<appointmentRequest>
  <slot doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointmentRequest>
```

If all is well I get a response saying my appointment is booked.

```
HTTP/1.1 200 OK
[various headers]

<appointment>
  <slot doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointment>
```

If there is a problem, say someone else got in before me, then I'll get some kind of error message in the reply body.

```
HTTP/1.1 200 OK
[various headers]

<appointmentRequestFailure>
  <slot doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
  <reason>Slot not available</reason>
</appointmentRequestFailure>
```

So far this is a straightforward RPC style system. It's simple as it's just slinging plain old XML (POX) back and forth. If you use SOAP or XML-RPC it's basically the same mechanism, the only difference is that you wrap the XML messages in some kind of envelope.

Level 1 - Resources

The first step towards the Glory of Rest in the RMM is to introduce resources. So now rather than making all our requests to a singular service endpoint, we now start talking to individual resources.

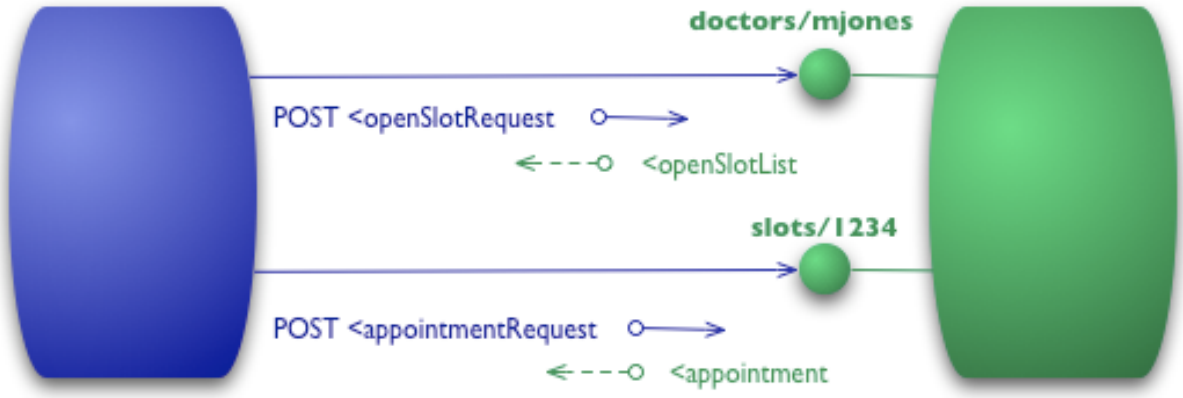

Figure 3: Level 1 adds resources

So with our initial query, we might have a resource for given doctor.

```
POST /doctors/mjones HTTP/1.1
[various other headers]

<openSlotRequest date = "2010-01-04"/>
```

The reply carries the same basic information, but each slot is now a resource that can be addressed individually.

```
HTTP/1.1 200 OK
[various headers]


<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650"/>
</openSlotList>
```

With specific resources booking an appointment means posting to a particular slot.

```
POST /slots/1234 HTTP/1.1
[various other headers]

<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```

If all goes well I get a similar reply to before.

```
HTTP/1.1 200 OK
[various headers]
```

```
<appointment>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointment>
```

The difference now is that if anyone needs to do anything about the appointment, like book some tests, they first get hold of the appointment resource, which might have a URI like `http://royalhope.nhs.uk/slots/1234/appointment`, and post to that resource.

To an object guy like me this is like the notion of object identity. Rather than calling some function in the ether and passing arguments, we call a method on one particular object providing arguments for the other information.

Level 2 - HTTP Verbs

I've used HTTP POST verbs for all my interactions here in level 0 and 1, but some people use GETs instead or in addition. At these levels it doesn't make much difference, they are both being used as tunneling mechanisms allowing you to tunnel your interactions through HTTP. Level 2 moves away from this, using the HTTP verbs as closely as possible to how they are used in HTTP itself.

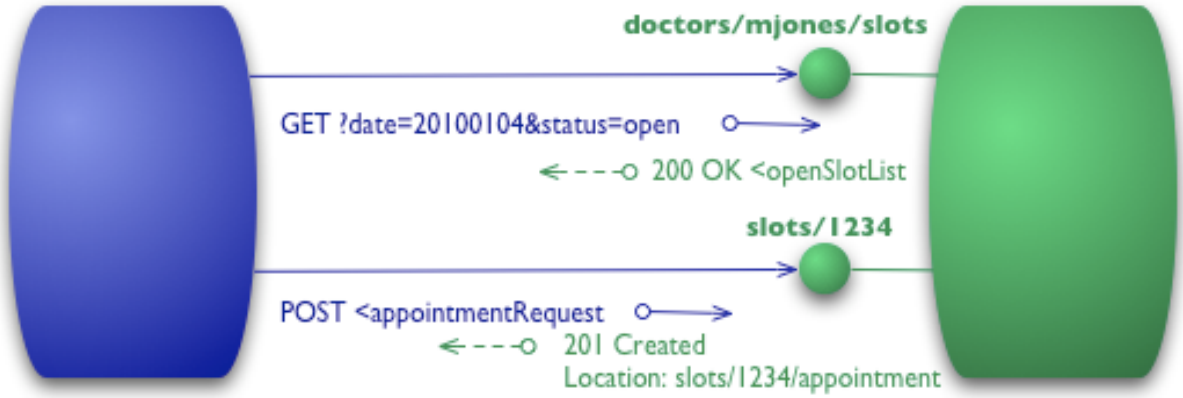

Figure 4: Level 2 addes HTTP verbs

For our the list of slots, this means we want to use GET.

```
GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1
Host: royalhope.nhs.uk
```

The reply is the same as it would have been with the POST

```
HTTP/1.1 200 OK
[various headers]

<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650"/>
</openSlotList>
```

At Level 2, the use of GET for a request like this is crucial. HTTP defines GET as a safe operation, that is it doesn't make any significant changes to the state of anything. This allows us to invoke GETs safely any number of times in any order and get the same results each time. An important consequence of this is that it allows any participant in the routing of requests to use caching, which is a key element in making the web perform as well as it does. HTTP includes various measures to support caching, which can be used by all participants in the communication. By following the rules of HTTP we're able to take advantage of that capability.

To book an appointment we need an HTTP verb that does change state, a POST or a PUT. I'll use the same POST that I did earlier.

```
POST /slots/1234 HTTP/1.1
[various other headers]
```

```
<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```

The trade-offs between using POST and PUT here are more than I want to go into here, maybe I'll do a separate article on them some day. But I do want to point out that some people incorrectly make a correspondence between POST/PUT and create/update. The choice between them is rather different to that.

Even if I use the same post as level 1, there's another significant difference in how the remote service responds. If all goes well, the service replies with a response code of 201 to indicate that there's a new resource in the world.

```
HTTP/1.1 201 Created
Location: slots/1234/appointment
[various headers]

<appointment>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointment>
```

The 201 response includes a location attribute with a URI that the client can use to GET the current state of that resource in the future. The response here also includes a representation of that resource to save the client an extra call right now.

There is another difference if something goes wrong, such as someone else booking the session.

```
HTTP/1.1 409 Conflict
[various headers]

<openSlotList>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650"/>
</openSlotList>
```

The important part of this response is the use of an HTTP response code to indicate something has gone wrong. In this case a 409 seems a good choice to indicate that someone else has already updated the resource in an incompatible way. Rather than using a return code of 200 but including an error response, at level 2 we explicitly use some kind of error response like this. It's up to the protocol designer to decide what codes to use, but there should be a non-2xx response if an error crops up. Level 2 introduces using HTTP verbs and HTTP response codes.

There is an inconsistency creeping in here. REST advocates talk about using all the HTTP verbs. They also justify their approach by saying that REST is attempting to learn from the practical success of the web. But the world-wide web doesn't use PUT or DELETE much in practice. There are sensible reasons for using PUT and DELETE more, but the existence proof of the web isn't one of them.

The key elements that are supported by the existence of the web are the strong separation between safe (eg GET) and non-safe operations, together with using status codes to help communicate the kinds of errors you run into.

Level 3 - Hypermedia Controls

The final level introduces something that you often hear referred to under the ugly acronym of HATEOAS (Hypertext As The Engine Of Application State). It addresses the question of how to get from a list open slots to knowing what to do to book an appointment.

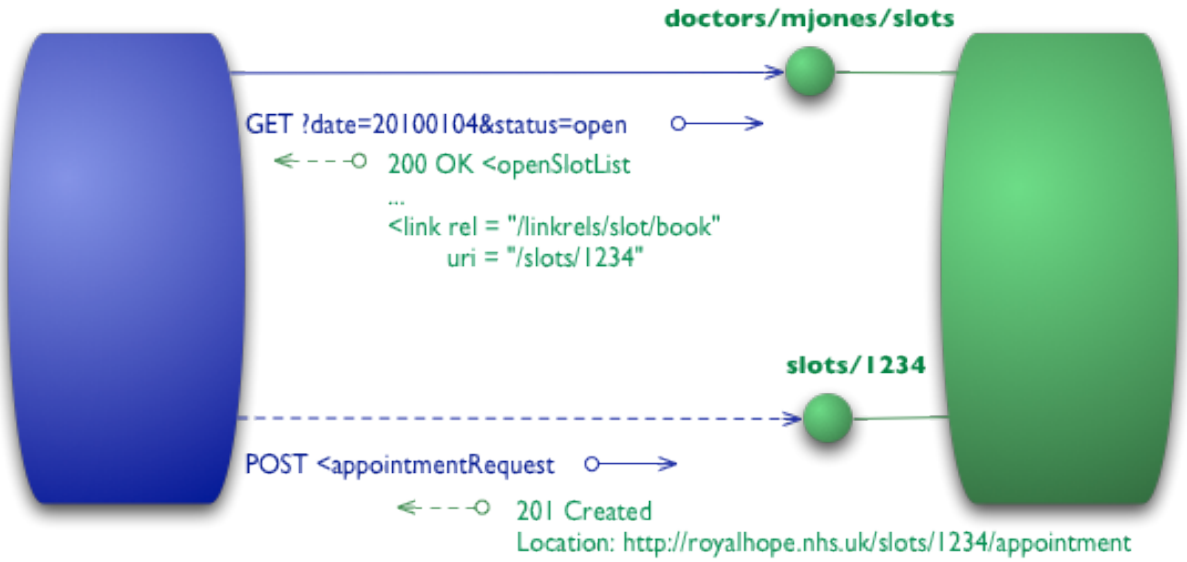

Figure 5: Level 3 adds hypermedia controls

We begin with the same initial GET that we sent in level 2

```
GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1
Host: royalhope.nhs.uk
```

But the response has a new element

```
HTTP/1.1 200 OK
[various headers]

<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450">
     <link rel = "/linkrels/slot/book"
           uri = "/slots/1234"/>
  </slot>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650">
     <link rel = "/linkrels/slot/book"
           uri = "/slots/5678"/>
  </slot>
</openSlotList>
```

Each slot now has a link element which contains a URI to tell us how to book an appointment.

The point of hypermedia controls is that they tell us what we can do next, and the URI of the resource we need to manipulate to do it. Rather than us having to know

where to post our appointment request, the hypermedia controls in the response tell us how to do it.

The POST would again copy that of level 2

```
POST /slots/1234 HTTP/1.1
[various other headers]

<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```

And the reply contains a number of hypermedia controls for different things to do next.

```
HTTP/1.1 201 Created
Location: http://royalhope.nhs.uk/slots/1234/appointment
[various headers]

<appointment>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
  <link rel = "/linkrels/appointment/cancel"
        uri = "/slots/1234/appointment"/>
  <link rel = "/linkrels/appointment/addTest"
        uri = "/slots/1234/appointment/tests"/>
  <link rel = "self"
        uri = "/slots/1234/appointment"/>
  <link rel = "/linkrels/appointment/changeTime"
        uri = "/doctors/mjones/slots?date=20100104&status=open"/>
  <link rel = "/linkrels/appointment/updateContactInfo"
        uri = "/patients/jsmith/contactInfo"/>
  <link rel = "/linkrels/help"
        uri = "/help/appointment"/>
</appointment>
```

One obvious benefit of hypermedia controls is that it allows the server to change its URI scheme without breaking clients. As long as clients look up the "addTest" link URI then the server team can juggle all URIs other than the initial entry points.

A further benefit is that it helps client developers explore the protocol. The links give client developers a hint as to what may be possible next. It doesn't give all the information: both the "self" and "cancel" controls point to the same URI - they need to figure out that one is a GET and the other a DELETE. But at least it gives them a starting point as to what to think about for more information and to look for a similar URI in the protocol documentation.

Similarly it allows the server team to advertise new capabilities by putting new links in the responses. If the client developers are keeping an eye out for unknown links these links can be a trigger for further exploration.

There's no absolute standard as to how to represent hypermedia controls. What I've done here is to use the current recommendations of the REST in Practice team, which is to follow ATOM (RFC 4287) I use a `<link>` element with a `uri` attribute for

the target URI and a `rel` attribute for to describe the kind of relationship. A well known relationship (such as `self` for a reference to the element itself) is bare, any specific to that server is a fully qualified URI. ATOM states that the definition for well-known linkrels is the Registry of Link Relations . As I write these are confined to what's done by ATOM, which is generally seen as a leader in level 3 restfulness.

The Meaning of the Levels

I should stress that the RMM, while a good way to think about what the elements of REST, is not a definition of levels of REST itself. Roy Fielding has made it clear that level 3 RMM is a pre-condition of REST. Like many terms in software, REST gets lots of definitions, but since Roy Fielding coined the term, his definition should carry more weight than most.

What I find useful about this RMM is that it provides a good step by step way to understand the basic ideas behind restful thinking. As such I see it as tool to help us learn about the concepts and not something that should be used in some kind of assessment mechanism. I don't think we have enough examples yet to be really sure that the restful approach is the right way to integrate systems, I do think it's a very attractive approach and the one that I would recommend in most situations.

Talking about this with Ian Robinson, he stressed that something he found attractive about this model when Leonard Richardson first presented it was its relationship to common design techniques.

- Level 1 tackles the question of handling complexity by using divide and conquer, breaking a large service endpoint down into multiple resources.
- Level 2 introduces a standard set of verbs so that we handle similar situations in the same way, removing unnecessary variation.
- Level 3 introduces discoverability, providing a way of making a protocol more self-documenting.

The result is a model that helps us think about the kind of HTTP service we want to provide and frame the expectations of people looking to interact with it.

Reference : https://martinfowler.com/articles/richardsonMaturityModel.html