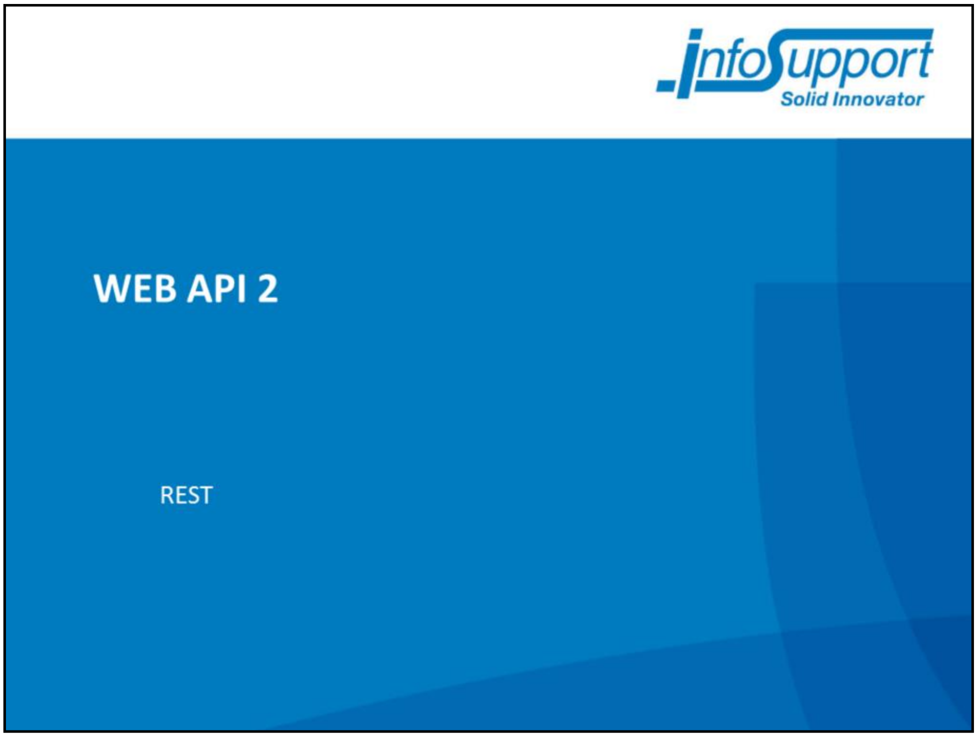
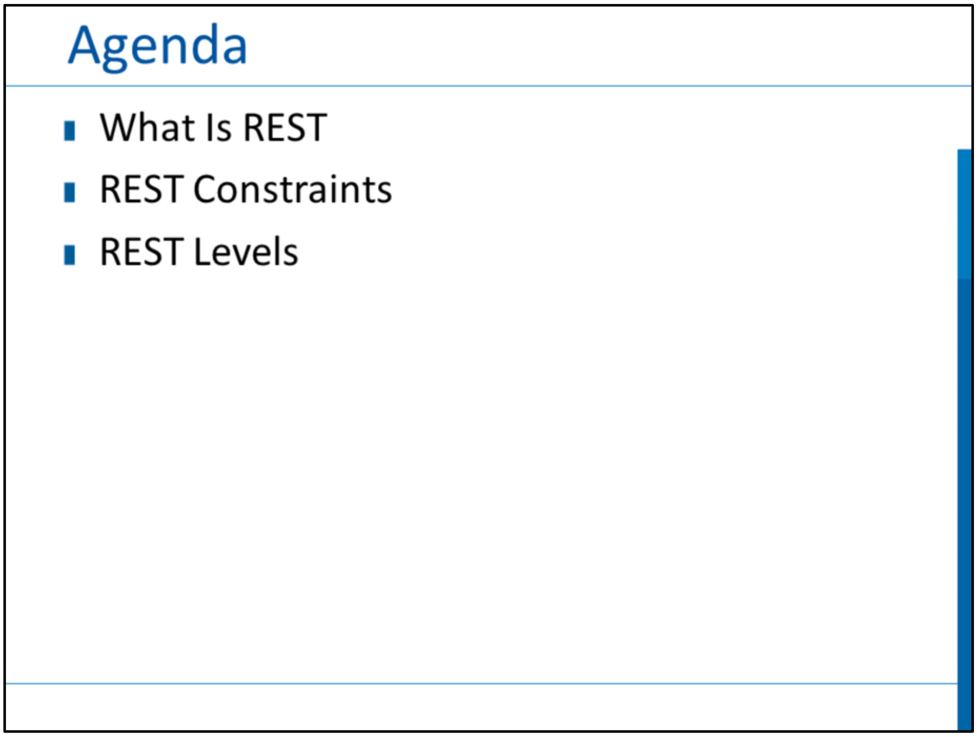


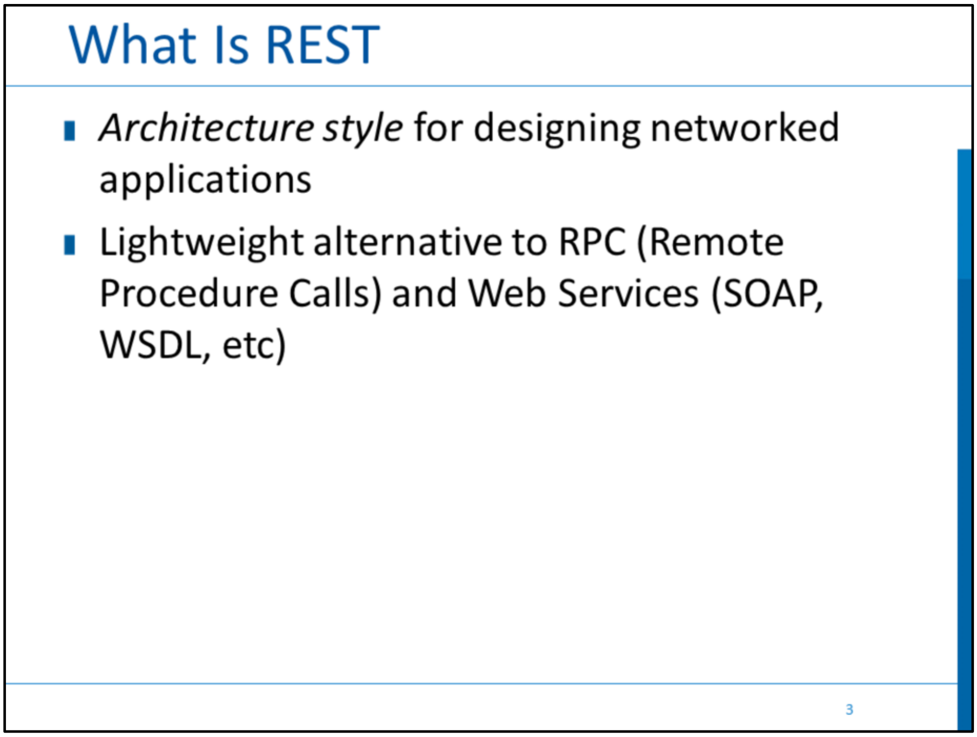
WEB API 2

REST



Agenda

- What Is REST
- REST Constraints
- REST Levels



What Is REST

- *Architecture style* for designing networked applications
- Lightweight alternative to RPC (Remote Procedure Calls) and Web Services (SOAP, WSDL, etc)

http://rest.elkstein.org/

REST stands for **Re**presentational **S**tate **T**ransfer. (It is sometimes spelled "ReST".) It relies on a stateless, client-server, cacheable communications protocol -- and in virtually all cases, the HTTP protocol is used.

REST is *an architecture style* for designing networked applications. The idea is that, rather than using complex mechanisms such as CORBA, RPC or SOAP to connect between machines, simple HTTP is used to make calls between machines.

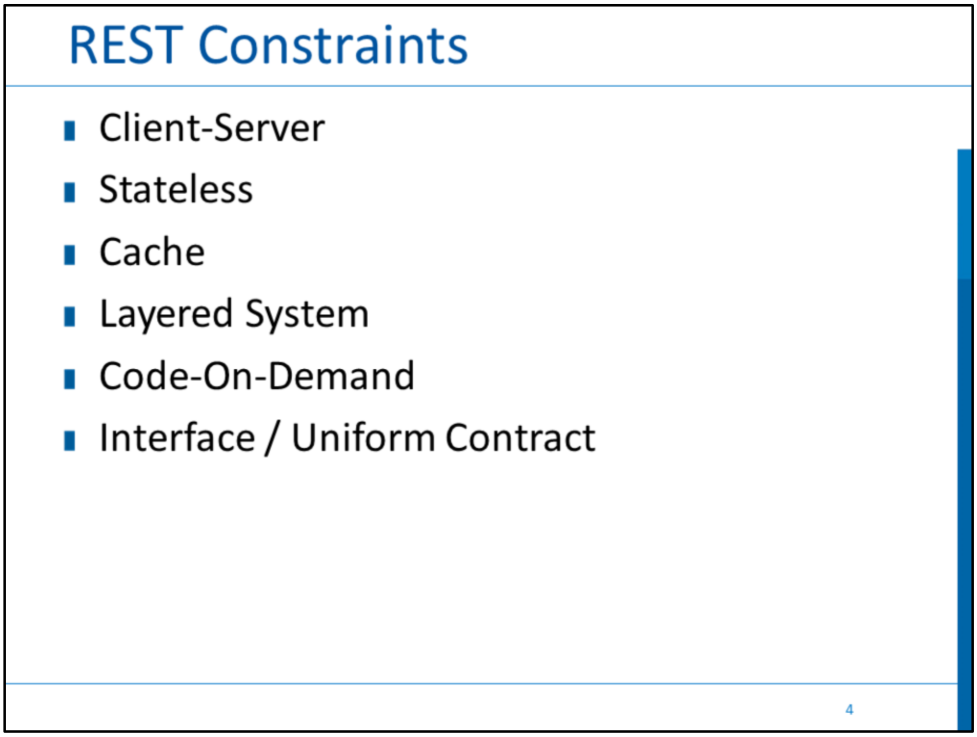
In many ways, the World Wide Web itself, based on HTTP, can be viewed as a REST-based architecture.

RESTful applications use HTTP requests to post data (create and/or update), read data (e.g., make queries), and delete data. Thus, REST uses HTTP for all four CRUD (Create/Read/Update/Delete) operations.

REST is a lightweight alternative to mechanisms like RPC (Remote Procedure Calls) and Web Services (SOAP, WSDL, et al.). Later, we will see how much more simple REST is.

Despite being simple, REST is fully-featured; there's basically nothing you can do in Web Services that can't be done with a RESTful architecture.

REST is not a "standard". There will never be a W3C recommendation for REST, for example. And while there are REST programming frameworks, working with REST is so simple that you can often "roll your own" with standard library features in languages like Perl, Java, or C#.



REST Constraints

- Client-Server
- Stateless
- Cache
- Layered System
- Code-On-Demand
- Interface / Uniform Contract

4

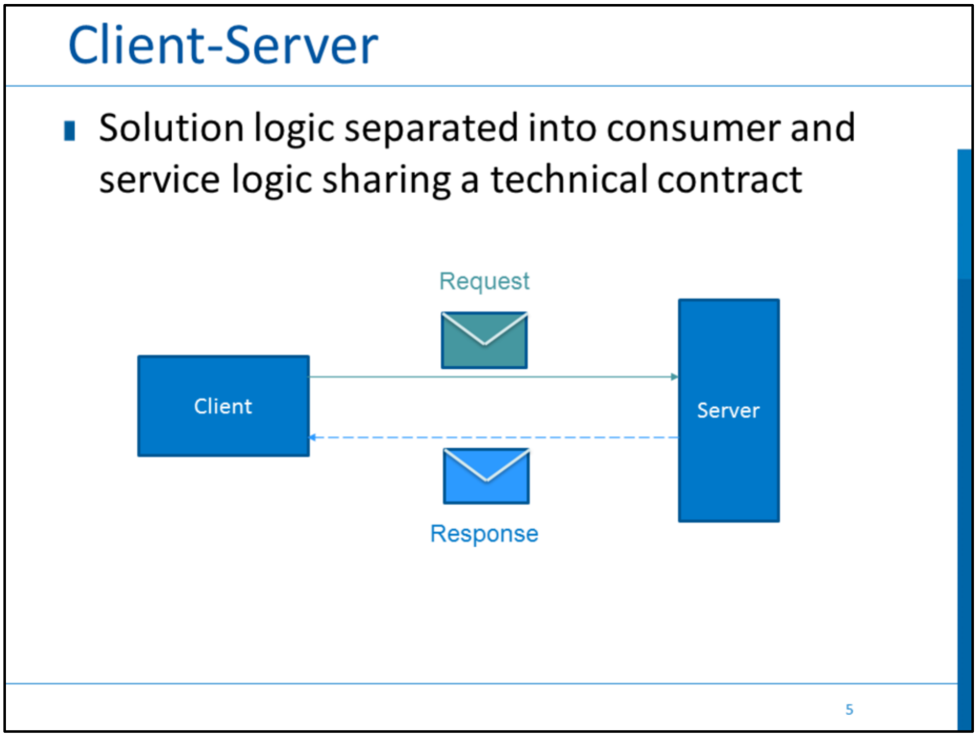
REST Constraints

REST constraints are design rules that are applied to establish the distinct characteristics of the REST architectural style.

The formal REST constraints are:

Client-Server
Stateless
Cache
Interface / Uniform Contract
Layered System
Code-On-Demand
Each constraint is a pre-determined design decision that can have both positive and negative impacts. The intent is for the positives of each constraint to balance out the negatives to produce an overall architecture that resembles the Web.

An architecture that weakens or eliminates a required REST constraint is generally considered to no longer conform to REST. This requires that educated decisions be made to understand the potential trade-offs when deliberately deviating from the application of REST constraints.



Client-Server

- Solution logic separated into consumer and service logic sharing a technical contract

Request

Client

Server

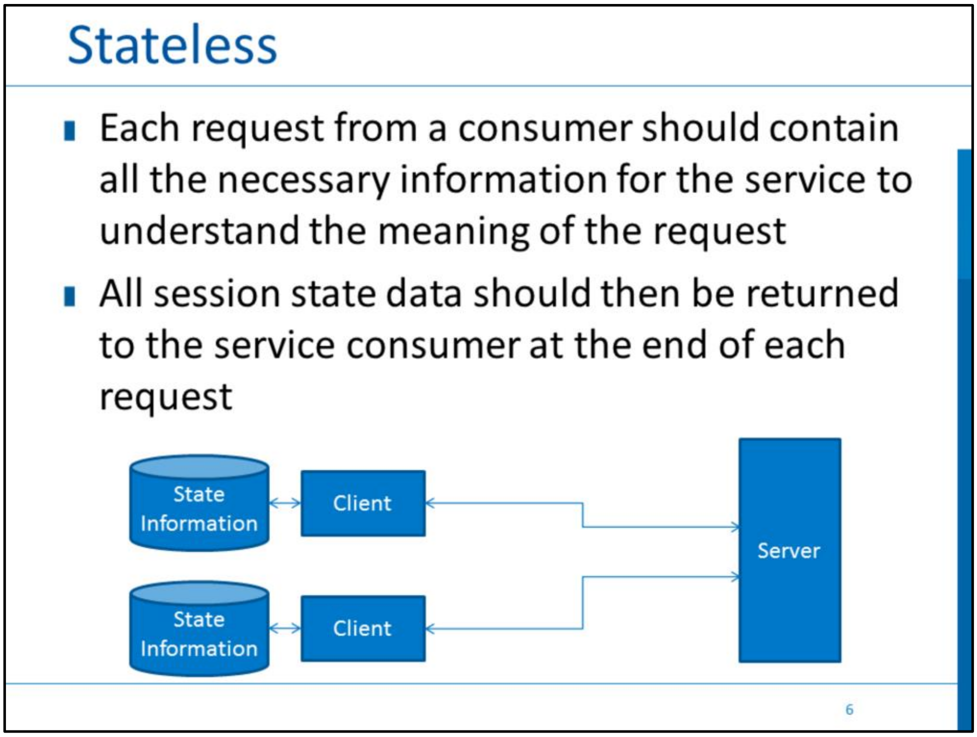
Response

5

http://whatisrest.com/rest_constraints/client_server

Perhaps the most foundational constraint, Client-Server enforces the separation of concerns in the form of a client-server architecture. This helps establish a fundamental distributed architecture, thereby supporting the independent evolution of the client-side logic and server-side logic.

The Client-Server constraint requires that a service offer one or more capabilities and listen for requests on these capabilities. A consumer invokes a capability by sending the corresponding request message, and the service either rejects the request or performs the requested task before sending a response message back to the consumer (Figure 1). Exceptions that prevent the task from proceeding are raised back to the consumer, and the consumer is responsible for taking corrective action.



Stateless

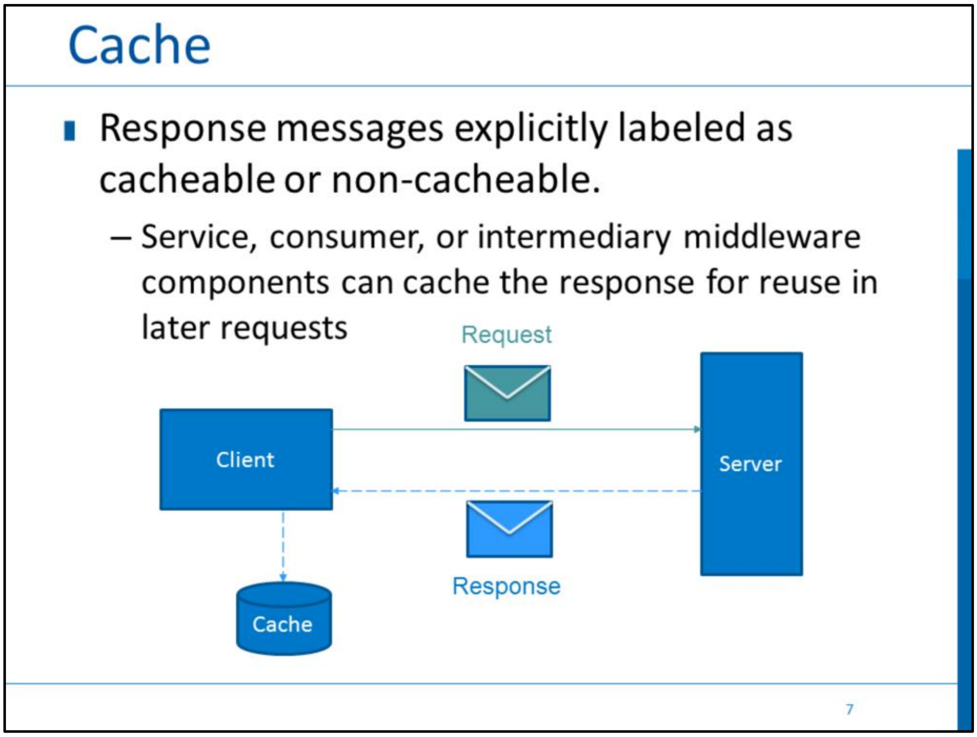
- Each request from a consumer should contain all the necessary information for the service to understand the meaning of the request
- All session state data should then be returned to the service consumer at the end of each request

6

Stateless

The communication between service consumer (client) and service (server) must be stateless between requests. This means that each request from a service consumer should contain all the necessary information for the service to understand the meaning of the request, and all session state data should then be returned to the service consumer at the end of each request.

Statelessness is one of the primary influences over service contract design in REST-style architecture. It imposes significant restrictions on the kinds of communication allowed between services and their consumers in order to achieve its design goals



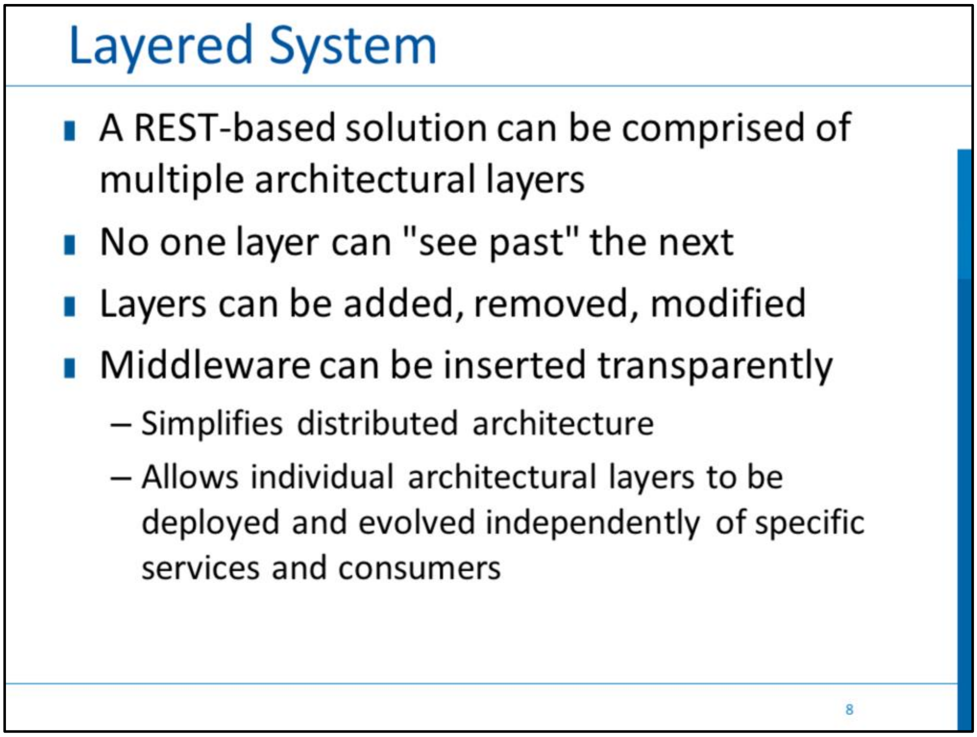
Cache

- Response messages explicitly labeled as cacheable or non-cacheable.
  - Service, consumer, or intermediary middleware components can cache the response for reuse in later requests

Cache

Response messages from the service to its consumers are explicitly labeled as cacheable or non-cacheable. This way, the service, the consumer, or one of the intermediary middleware components can cache the response for reuse in later requests.

The Cache constraint builds upon Client-Server and Stateless with a requirement that responses are implicitly or explicitly labeled as cacheable or non-cacheable. Requests are passed through a cache component, which may reuse previous responses to partially or completely eliminate some interactions over the network (Figure 1. This form of elimination can improve efficiency and scalability, and can further improve user-perceived performance by reducing the average latency during a series of interactions. A common reason for incorporating caching as a native part of a REST architecture is as a counterbalance to some of the negative impacts of applying the Stateless constraint.



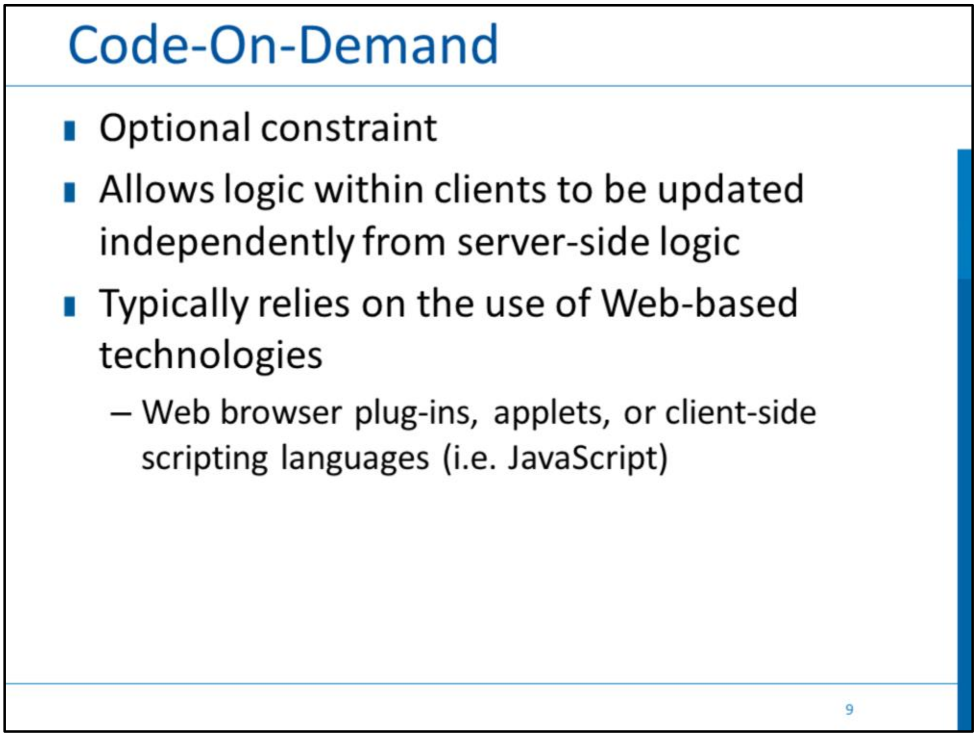
Layered System

- A REST-based solution can be comprised of multiple architectural layers
- No one layer can "see past" the next
- Layers can be added, removed, modified
- Middleware can be inserted transparently
  - Simplifies distributed architecture
  - Allows individual architectural layers to be deployed and evolved independently of specific services and consumers

A REST-based solution can be comprised of multiple architectural layers, and no one layer can "see past" the next. Layers can be added, removed, modified, or reordered in response to how the solution needs to evolve.

The Layered System constraint builds on Client-Server to add middleware components (which can exist as services or service agents) to an architecture. Specifically, Layered System requires that this middleware be inserted transparently so that interaction between a given service and consumer is consistent, regardless of whether the consumer is communicating with a service residing in a middleware layer or a service that represents the ultimate receiver of a message. Similarly, a service does not need to be aware of whether its consumer is further communicating with other services, or whether the consumer itself is also acting as a service for other consumer programs.

This form of information hiding simplifies distributed architecture and allows individual architectural layers to be deployed and evolved independently of specific services and consumers.



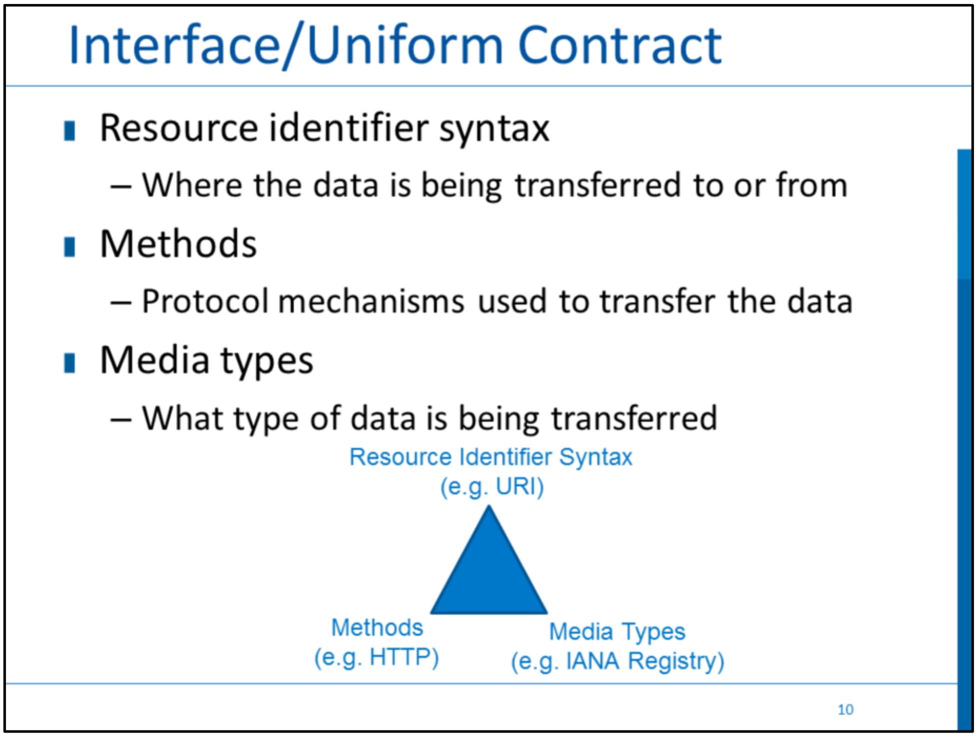
Code-On-Demand

- Optional constraint
- Allows logic within clients to be updated independently from server-side logic
- Typically relies on the use of Web-based technologies
  - Web browser plug-ins, applets, or client-side scripting languages (i.e. JavaScript)

Code-On-Demand

This optional constraint is primarily intended to allow logic within clients (such as Web browsers) to be updated independently from server-side logic. Code-On-Demand typically relies on the use of Web-based technologies, such as Web browser plug-ins, applets, or client-side scripting languages (i.e. JavaScript).

Code-On-Demand can further be applied to services and service consumers. For example, a service can be designed to dynamically defer portions of logic to service consumer programs. For example, this type of functionality can be used in support of Stateless, which dictates whether a session state should be deferred back to the service consumer. Code-On-Demand can also build upon this by further deferring the processing effort. This approach may be justifiable when service logic can be executed by the consumer more efficiently or effectively.



Interface/Uniform Contract

- Resource identifier syntax
  - Where the data is being transferred to or from
- Methods
  - Protocol mechanisms used to transfer the data
- Media types
  - What type of data is being transferred

Interface/Uniform Contract

The Interface constraint states that all services and service consumers within a REST-compliant architecture must share a single, overarching technical interface. As the primary constraint that distinguishes REST from other architecture types, Interface is generally applied using the methods and media types provided by HTTP (Figure 1).

The technical contract established by Interface is typically free of business context because, in order to be reusable by a wide range of services and service consumers, it needs to provide generic, high-level capabilities that are abstract enough to accommodate a broad range of service interaction requirements.

Business-specific or service-specific data and meaning are isolated to the messages that are exchanged via the uniform technical contract.

Uniform Contract Elements

The REST uniform contract is based on three fundamental elements:

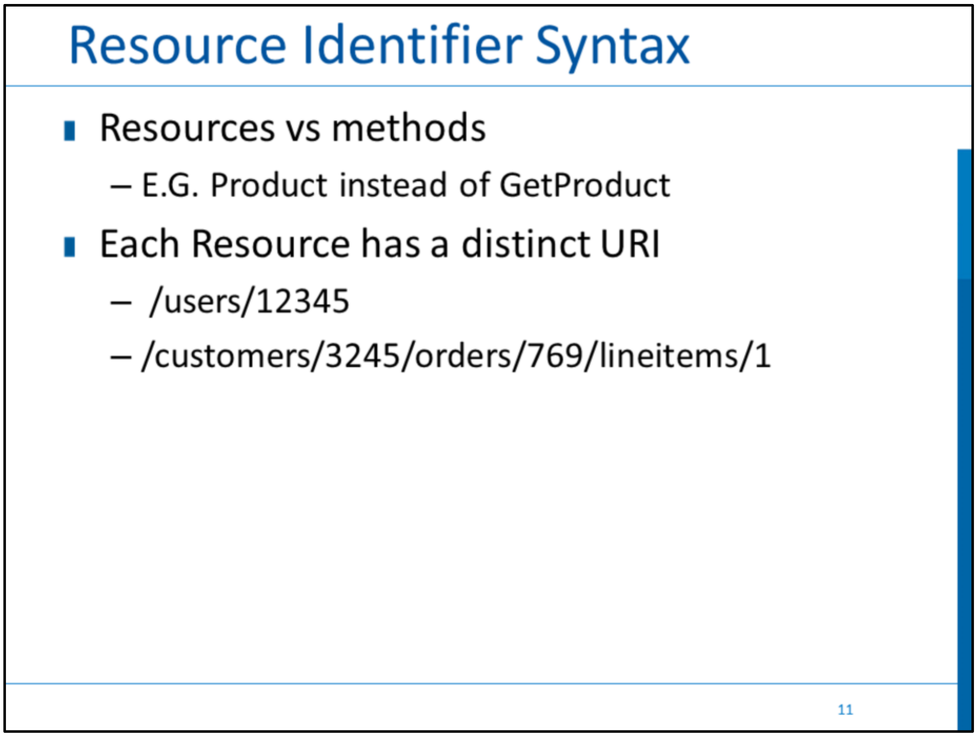
resource identifier syntax - How can we express where is the data being transferred to or from?
methods - What are the protocol mechanisms used to transfer the data?
media types - What type of data is being transferred?
These elements are commonly represented using a triangle symbol, as shown in Figure 1.

Uniform Contract Elements: The REST triangle.
Figure 1 - The REST triangle.

As explained shortly, individual REST services use these elements in different combinations to expose their service capabilities. However, it is important to understand that what makes this type of service contract "uniform" is the fact that a master set of these elements is defined for use by a collection (or inventory) of services. This essentially allows us to standardize the baseline elements of the service contract.

The three elements of the REST triangle are deliberately orthogonal in order to limit the impact of changes to any one element. For example, the resources for a given service inventory are defined separately from the set of methods used for that same inventory, and separately again from the set of supported media types.

Let's take a closer look at each of the uniform contract elements.



Resource Identifier Syntax

- Resources vs methods
  - E.G. Product instead of GetProduct
- Each Resource has a distinct URI
  - /users/12345
  - /customers/3245/orders/769/lineitems/1

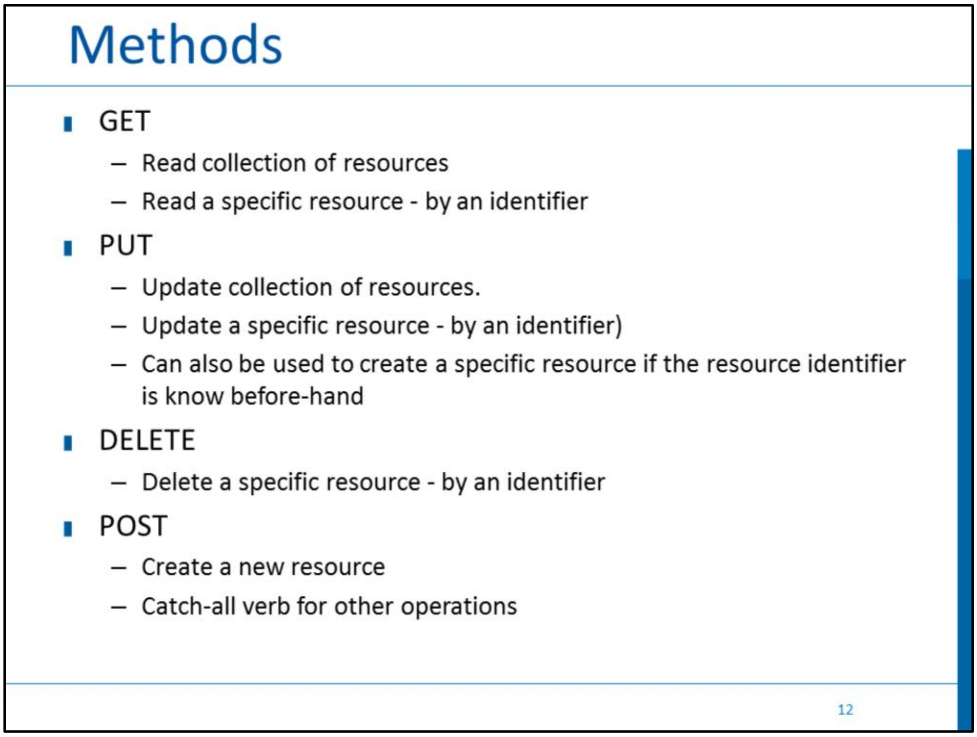
http://rest.elkstein.org/2008/02/rest-architecture-components.html

Resources, which are identified by logical URLs. Both state and functionality are represented using resources.

The logical URLs imply that the resources are universally addressable by other parts of the system.

Resources are the key element of a true RESTful design, as opposed to "methods" or "services" used in RPC and SOAP Web Services, respectively. You do not issue a "getProductName" and then a "getProductPrice" RPC calls in REST; rather, you view the product data as a resource -- and this resource should contain all the required information (or links to it).

A web of resources, meaning that a single resource should not be overwhelmingly large and contain too fine-grained details. Whenever relevant, a resource should contain links to additional information -- just as in web pages.



Methods

- GET
  - Read collection of resources
  - Read a specific resource - by an identifier
- PUT
  - Update collection of resources.
  - Update a specific resource - by an identifier)
  - Can also be used to create a specific resource if the resource identifier is know before-hand
- DELETE
  - Delete a specific resource - by an identifier
- POST
  - Create a new resource
  - Catch-all verb for other operations

12

Use HTTP Verbs to Mean Something

API consumers are capable of sending GET, POST, PUT, and DELETE verbs, and these verbs greatly enhance the clarity of what a given request does. Also, GET requests must not change any underlying resource data. Measurements and tracking may still occur, which updates data, but not resource data identified by the URI.

Generally, the four primary HTTP verb are used as follows:

GET

Read a specific resource (by an identifier) or a collection of resources.

PUT

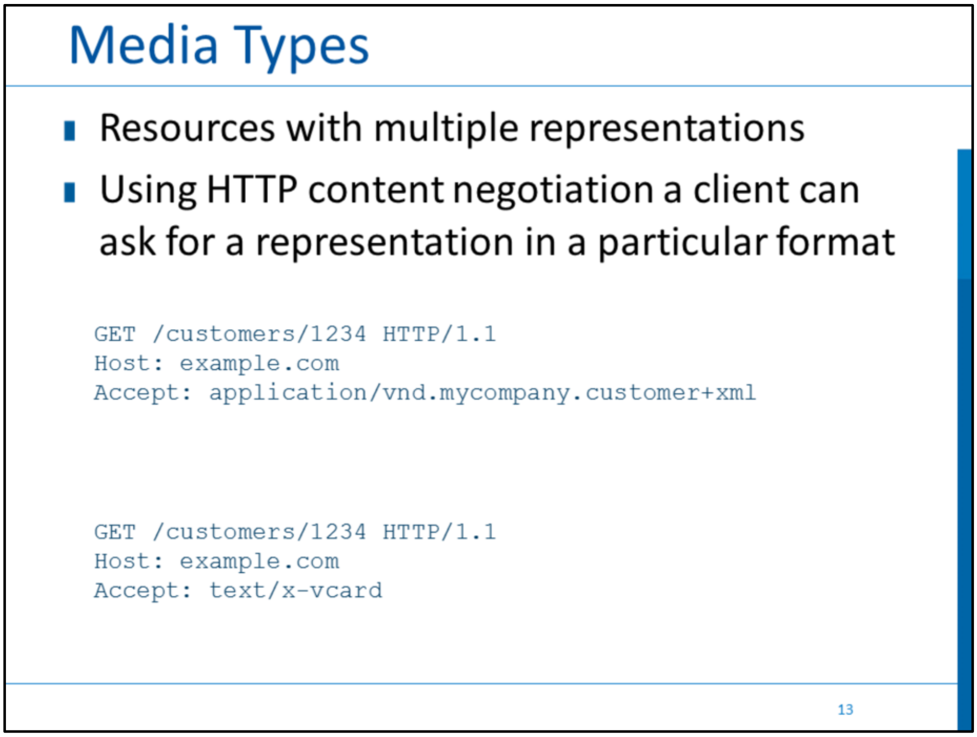
Update a specific resource (by an identifier) or a collection of resources. Can also be used to create a specific resource if the resource identifier is know before-hand.

DELETE

Remove/delete a specific resource by an identifier.

POST

Create a new resource. Also a catch-all verb for operations that don't fit into the other categories.



Media Types

- Resources with multiple representations
- Using HTTP content negotiation a client can ask for a representation in a particular format

```
GET /customers/1234 HTTP/1.1
Host: example.com
Accept: application/vnd.mycompany.customer+xml
```

```
GET /customers/1234 HTTP/1.1
Host: example.com
Accept: text/x-vcard
```

13

http://www.infoq.com/articles/rest-introduction

Resources with multiple representations

We've ignored a slight complication so far: how does a client know how to deal with the data it retrieves, e.g. as a result of a GET or POST request? The approach taken by HTTP is to allow for a separation of concerns between handling the data and invoking operations. In other words, a client that knows how to handle a particular data format can interact with all resources that can provide a representation in this format. Let's illustrate this with an example again. Using HTTP content negotiation, a client can ask for a representation in a particular format:

GET /customers/1234 HTTP/1.1
Host: example.com
Accept: application/vnd.mycompany.customer+xml

The result might be some company-specific XML format that represents customer information. If the client sends a different request, e.g. one like this:

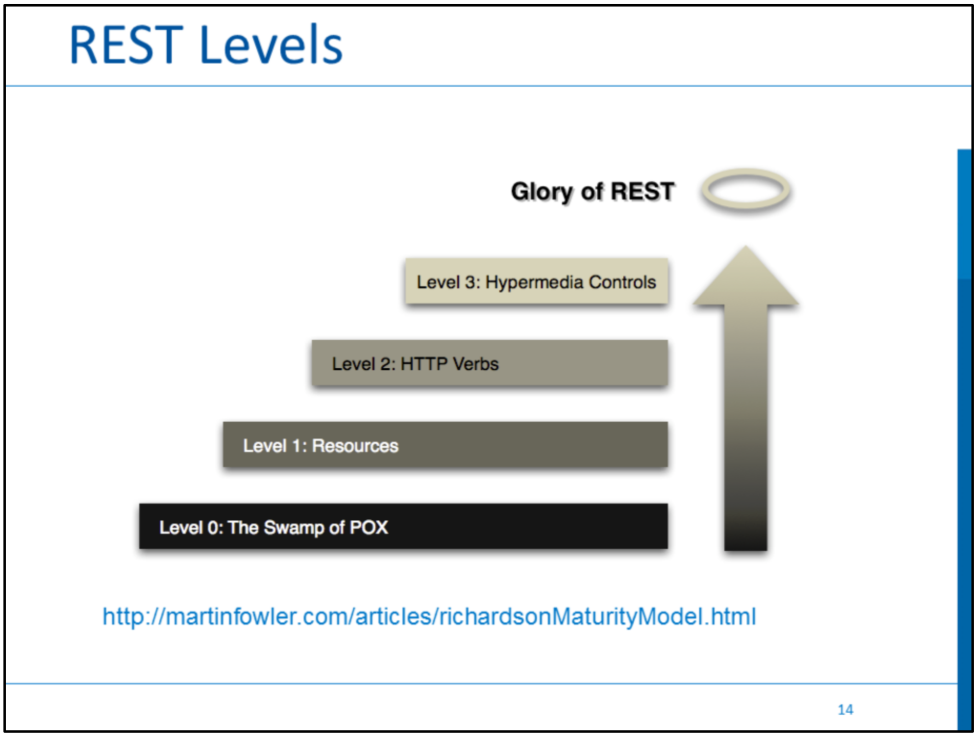
GET /customers/1234 HTTP/1.1
Host: example.com
Accept: text/x-vcard

The result could be the customer address in VCard format. (I have not shown the responses, which would contain metadata about the type of data in the HTTP Content-type header.) This illustrates why ideally, the representations of a resource should be in standard formats — if a client "knows" both the HTTP application protocol and a set of data formats, it can interact with any RESTful HTTP application in the world in a very meaningful way. Unfortunately, we don't have standard formats for everything, but you can probably imagine how one could create a smaller ecosystem within a company or a set of collaborating partners by relying on standard formats. Of course all of this does not only apply to the data sent from the server to the client, but also for the reverse direction — a server that can consume data in specific formats does not care about the particular type of client, provided it follows the application protocol.

There is another significant benefit of having multiple representations of a resource in practice: If you provide both an HTML and an XML representation of your resources, they are consumable not only by your application, but also by every standard Web browser — in other words, information in your application becomes available to everyone who knows how to use the Web.

There is another way to exploit this: You can turn your application's Web UI into its Web API — after all, API design is often driven by the idea that everything that can be done via the UI should also be doable via the API. Conflating the two tasks into one is an amazingly useful way to get a better Web interface for both humans and other applications.

Summary: Provide multiple representations of resources for different needs.



REST Levels

Glory of REST

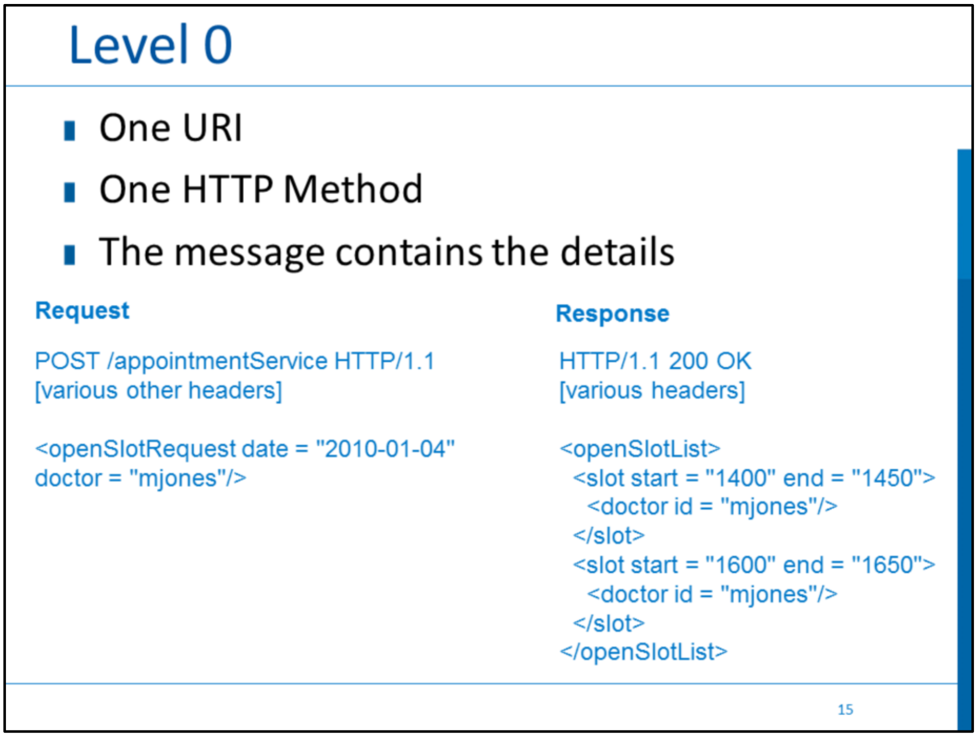
Level 3: Hypermedia Controls

Level 2: HTTP Verbs

Level 1: Resources

Level 0: The Swamp of POX

http://martinfowler.com/articles/richardsonMaturityModel.html



Level 0

- One URI
- One HTTP Method
- The message contains the details

Request

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

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

Response

```
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>
```

15

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

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>
```

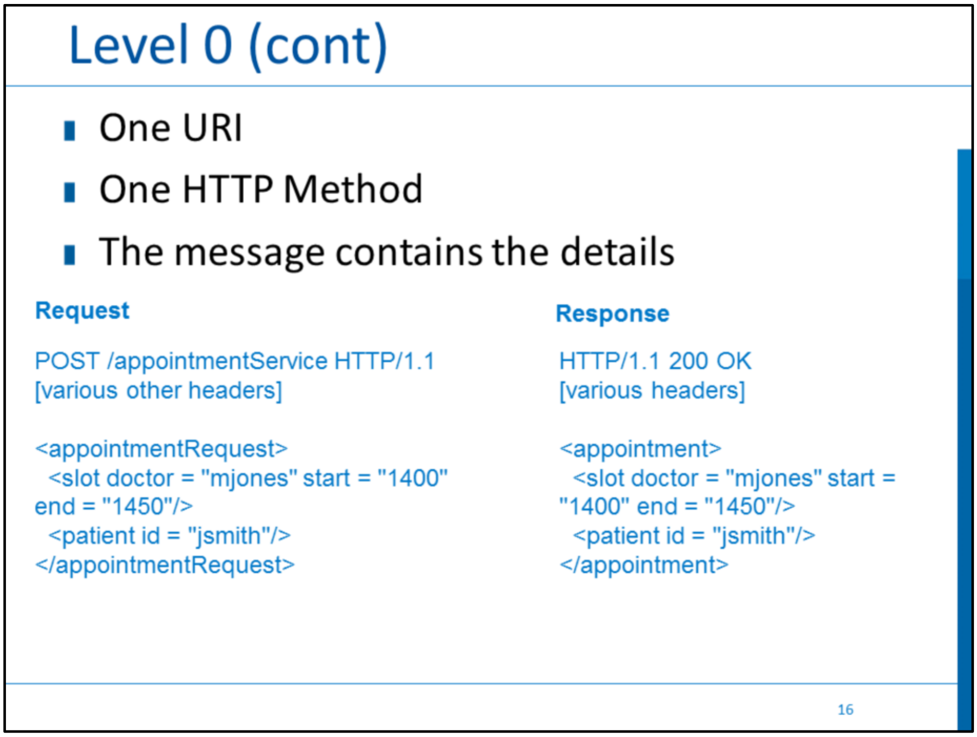
If all is well I get a response saying my appointment is booked.

```
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 0 (cont)

- One URI
- One HTTP Method
- The message contains the details

Request

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

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

Response

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

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

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

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>
```

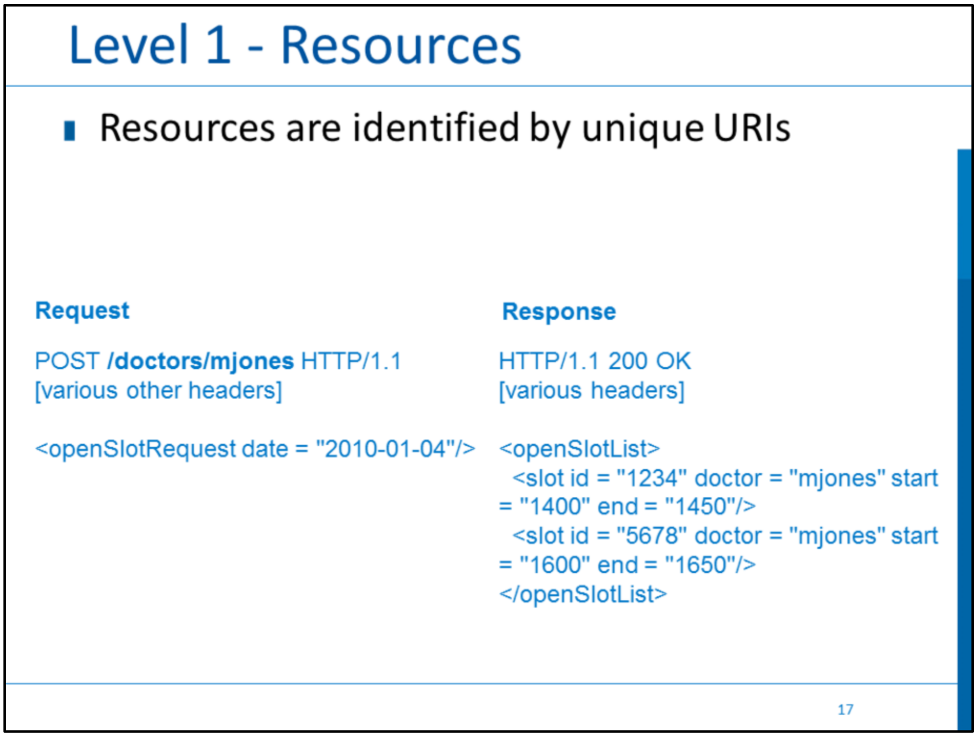
If all is well I get a response saying my appointment is booked.

```
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

- Resources are identified by unique URIs

Request

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

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

Response

```
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>
```

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

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.

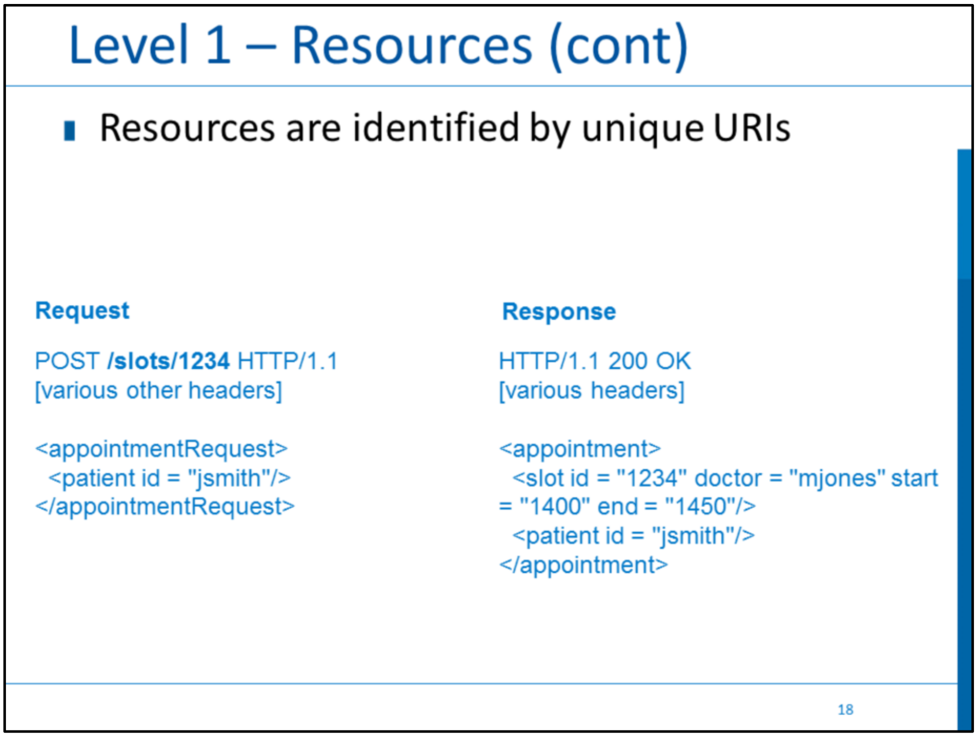
```
POST /slots/1234 HTTP/1.1
[various other headers]

<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 1 – Resources (cont)

- Resources are identified by unique URIs

Request

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

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

Response

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

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

18

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

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.

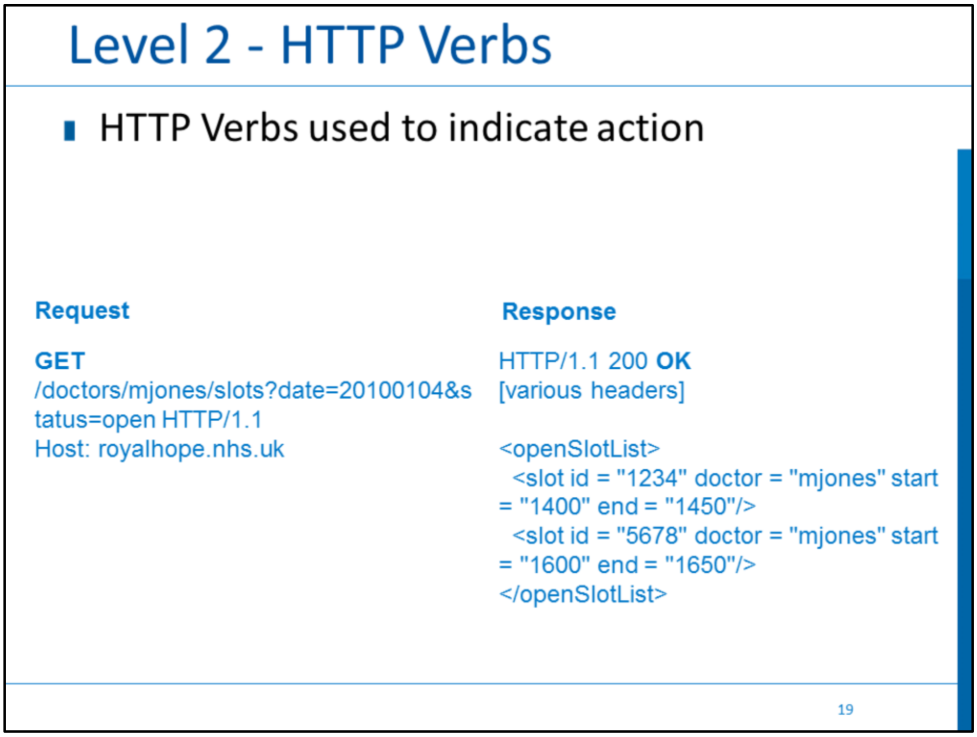
```
POST /slots/1234 HTTP/1.1
[various other headers]

<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

- HTTP Verbs used to indicate action

Request

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

Response

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>
```

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

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.

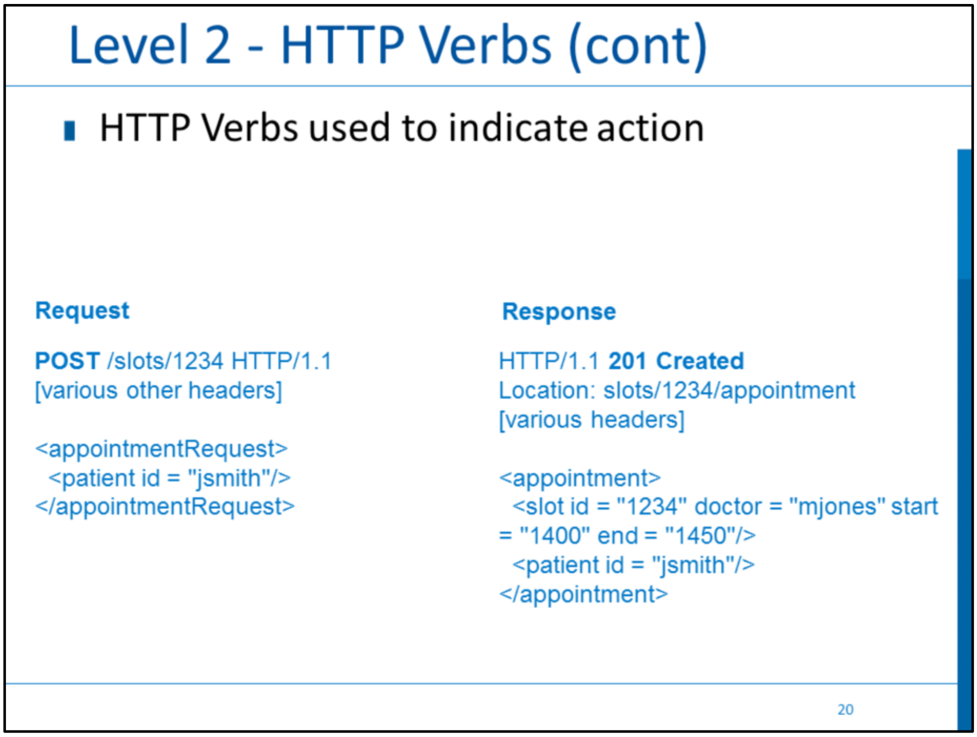
There is another difference if something goes wrong, such as someone else booking the session.

```
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 2 - HTTP Verbs (cont)

- HTTP Verbs used to indicate action

Request

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

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

Response

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>
```

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

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.

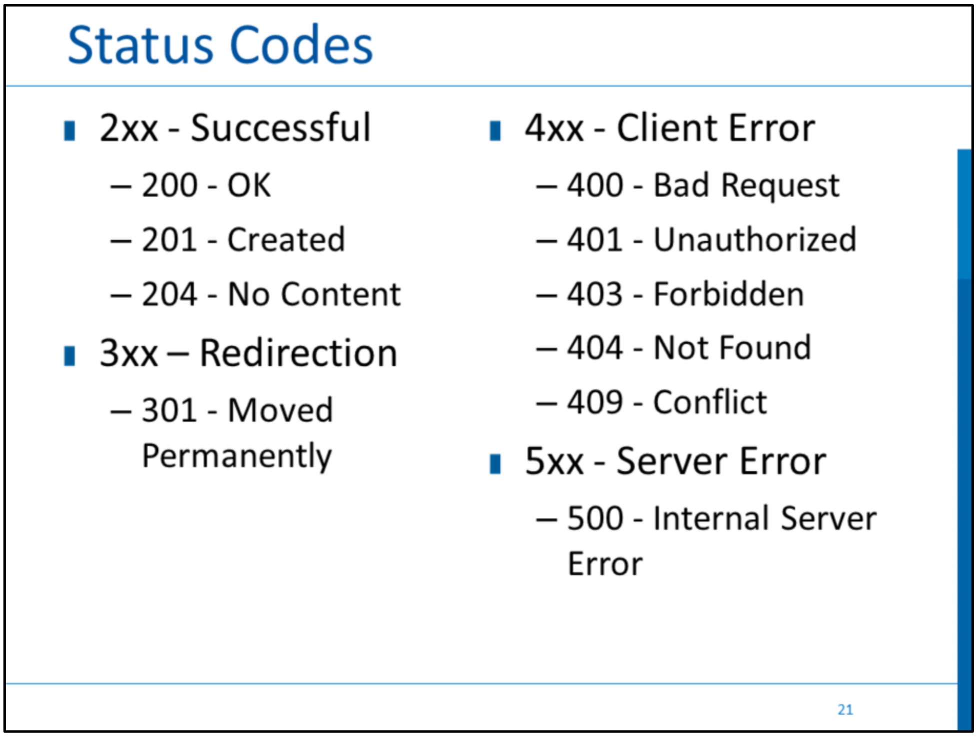
There is another difference if something goes wrong, such as someone else booking the session.

```
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.



Status Codes

- 2xx - Successful
  - 200 - OK
  - 201 - Created
  - 204 - No Content
- 3xx – Redirection
  - 301 - Moved Permanently
- 4xx - Client Error
  - 400 - Bad Request
  - 401 - Unauthorized
  - 403 - Forbidden
  - 404 - Not Found
  - 409 - Conflict
- 5xx - Server Error
  - 500 - Internal Server Error

Each Status-Code is described below, including a description of which method(s) it can follow and any metainformation required in the response.

1xx - Informational

This class of status code indicates a provisional response, consisting only of the Status-Line and optional headers, and is terminated by an empty line. There are no required headers for this class of status code. Since HTTP/1.0 did not define any 1xx status codes, servers MUST NOT send a 1xx response to an HTTP/1.0 client except under experimental conditions.

A client MUST be prepared to accept one or more 1xx status responses prior to a regular response, even if the client does not expect a 100 (Continue) status message. Unexpected 1xx status responses MAY be ignored by a user agent.

Proxies MUST forward 1xx responses, unless the connection between the proxy and its client has been closed, or unless the proxy itself requested the generation of the 1xx response. (For example, if a proxy adds a "Expect: 100-continue" field when it forwards a request, then it need not forward the corresponding 100 (Continue) response(s).)

100 - Continue
101 - Switching Protocols
2xx - Successful

This class of status code indicates that the client's request was successfully received, understood, and accepted.

200 - OK
201 - Created
202 - Accepted
203 - Non-Authoritative Information
204 - No Content
205 - Reset Content
206 - Partial Content
207 - Multi-Status
3xx - Redirection

This class of status code indicates that further action needs to be taken by the user agent in order to fulfill the request. The action required MAY be carried out by the user agent without interaction with the user if and only if the method used in the second request is GET or HEAD. A client SHOULD detect infinite redirection loops, since such loops generate network traffic for each redirection.

Note: previous versions of this specification recommended a maximum of five redirections. Content developers should be aware that there might be clients that implement such a fixed limitation.

300 - Multiple Choices
301 - Moved Permanently
302 - Found
303 - See Other
304 - Not Modified
305 - Use Proxy
306 - (Reserved)
307 - Temporary Redirect
4xx - Client Error

The 4xx class of status code is intended for cases in which the client seems to have erred. Except when responding to a HEAD request, the server SHOULD include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition. These status codes are applicable to any request method. User agents SHOULD display any included entity to the user.

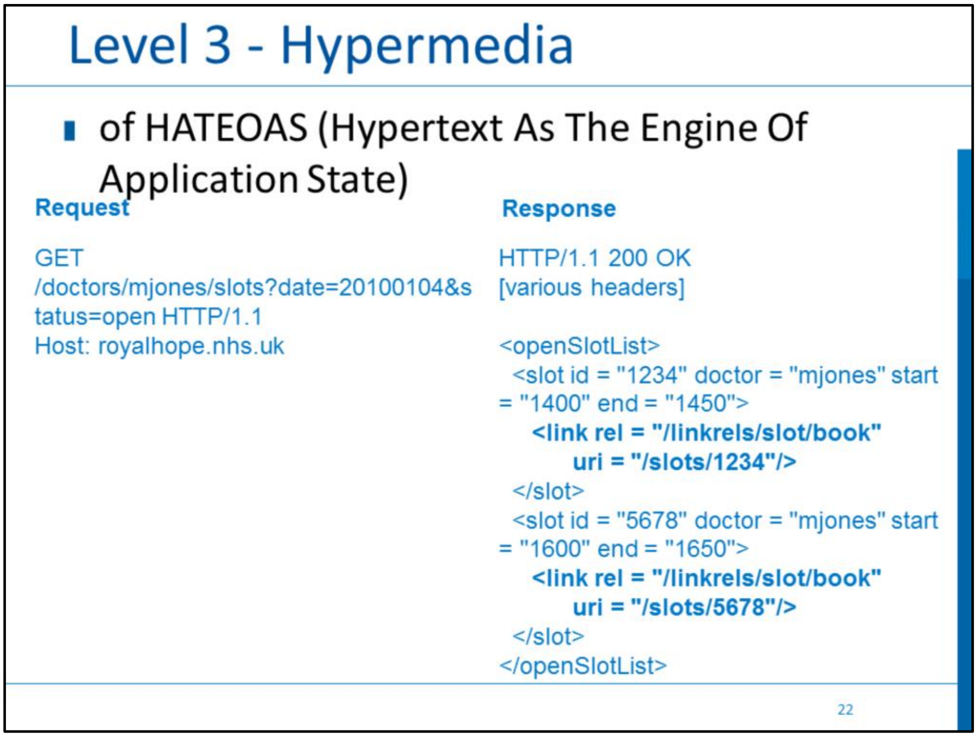
If the client is sending data, a server implementation using TCP SHOULD be careful to ensure that the client acknowledges receipt of the packet(s) containing the response, before the server closes the input connection. If the client continues sending data to the server after the close, the server's TCP stack will send a reset packet to the client, which may erase the client's unacknowledged input buffers before they can be read and interpreted by the HTTP application.

400 - Bad Request
401 - Unauthorized
402 - Payment Required
403 - Forbidden
404 - Not Found
405 - Method Not Allowed
406 - Not Acceptable
407 - Proxy Authentication
408 - Request Timeout
409 - Conflict
410 - Gone
411 - Length Required
412 - Precondition Failed
413 - Request Entity Too Large
414 - Request-URI Too Long
415 - Unsupported Media Type
416 - Requested Range Not Satisfiable
417 - Expectation Failed
422 - Unprocessable Entity
423 - Locked
424 - Failed Dependency
5xx - Server Error

Response status codes beginning with the digit "5" indicate cases in which the server is aware that it has erred or is incapable of performing the request. Except when responding to a HEAD request, the server SHOULD include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition. User agents SHOULD display any included entity to the user. These response codes are applicable to any request method.

500 - Internal Server Error
501 - Not Implemented
502 - Bad Gateway
503 - Service Unavailable
504 - Gateway Timeout
505 - HTTP Version Not Supported
507 - Insufficient Storage

Status CodeAPI meaning
200All is good; response will include applicable resource information, as well
201Resource created; will include the Location header specifying a URI to the newly created resource
202Same as 200, but used for async; in other words, all is good, but we need to poll the service to find out when completed
301The resource was moved; should include URI to new location
400Bad request; caller should reformat the request
401Unauthorized; should respond with an authentication challenge, to let the caller resubmit with appropriate credentials
403Access denied; user successfully authenticated, but is not allowed to access the requested resource
404Resource not found, or, caller not allowed to access the resource and we don’t want to reveal the reason
409Conflict; used as a response to a PUT request when another caller has dirtied the resource
500Server error; something bad happened, and server might include some indication of the underlying problem



Level 3 - Hypermedia

- of HATEOAS (Hypertext As The Engine Of Application State)

Request

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

Response

```
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>
```

22

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

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.

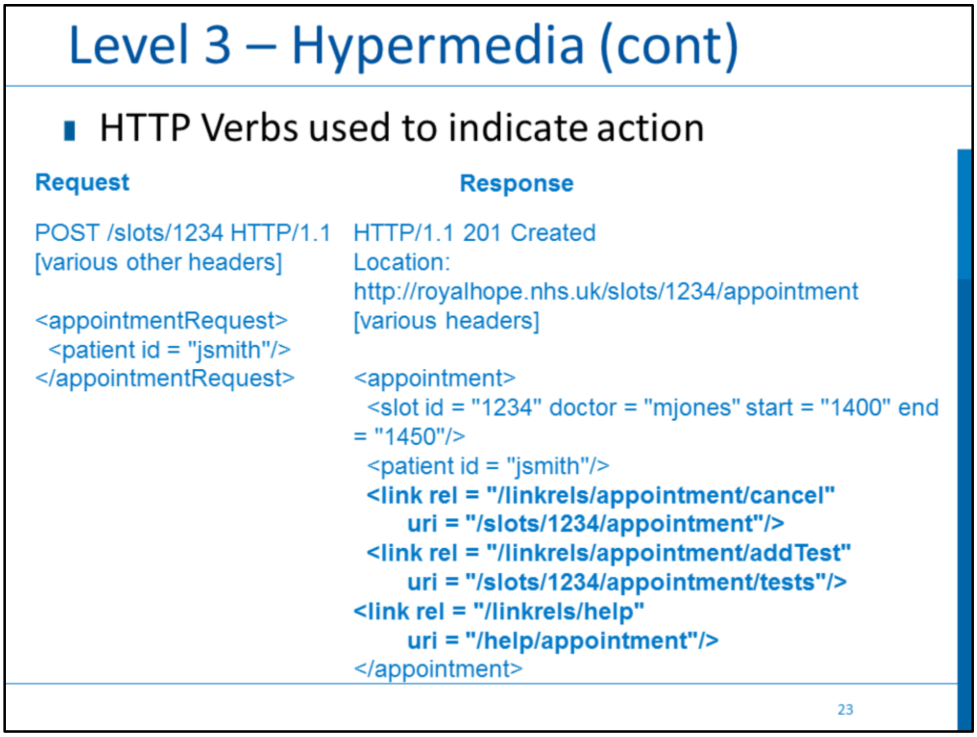
```
HTTP/1.1 201 Created
Location: http://royalhope.nhs.uk/slots/1234/appointment
[various headers]

<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 "latest" 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.



Level 3 – Hypermedia (cont)

- HTTP Verbs used to indicate action

Request

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

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

Response

```
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 = "/linkrels/help"
       uri = "/help/appointment"/>
</appointment>
```

23

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

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.

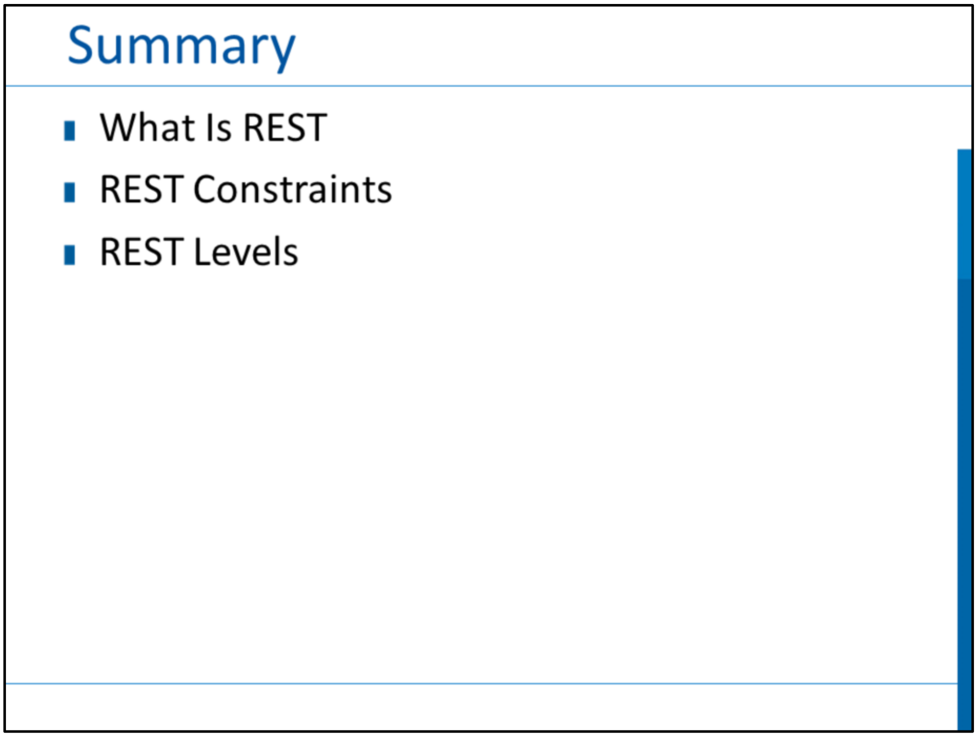
```
HTTP/1.1 201 Created
Location: http://royalhope.nhs.uk/slots/1234/appointment
[various headers]

<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 "latest" 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.



Summary

- What Is REST
- REST Constraints
- REST Levels