220

Distribution and Remoting

Dr Robert Chatley - rbc@imperial.ac.uk

@rchatley #doc220

In this lecture we look at some techniques for remote and distributed execution of different parts of systems, particularly looking at web services.

Applications Composed from Components

#doc220

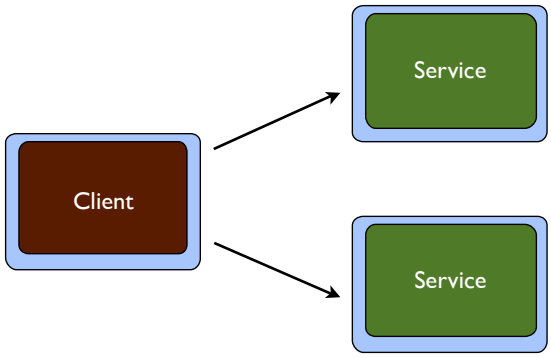
A theme of this course has been our desire to find ways to re-use code that we write in different systems. It has long been a goal of software engineers to produce software components that could be bought or downloaded and assembled into new systems. The wide-scale connectivity of the internet makes it easier for us to find and re-use code, and also to re-use components that operate over the network - we call these services.

Splitting an application into components

#doc220

As well as combining components and services in order to build up our application, forces may act upon us that make us decide to split up our application into parts. This may be for performance reasons - by spreading our computation over multiple machines we may be able to handle a higher load or complexity of computation. It may also be an organisational issue. If the team working on a project becomes too big, it can be difficult to manage communication. Breaking the team up into several smaller teams each working on a separate component of the system may work better.

Remote Services

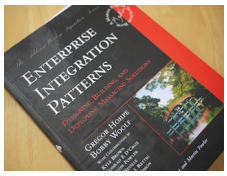


#doc220

When we split our application up, it is common to have some components that provide services, and other components that are clients of these services. This may extend to a complex graph, where clients of one service provide their own services to other clients higher up the chain. We saw this when we looked at layered and n-tier architectures. Clients and servers may run on the same machine, or on different ones. When we communicate with other machines, we have additional complexities to consider caused by networking.

Making Network Connections

```
String hostName = "www.eaipatterns.com";
int port = 80;

IPHostEntry hostInfo = Dns.GetHostByName(hostName);
IPAddress address = hostInfo.AddressList[0];

IPEndPoint endpoint = new IPEndPoint(address, port);

Socket socket = new Socket(address.AddressFamily,
  SocketType.Stream, ProtocolType.Tcp);
socket.Connect(endpoint);

byte[] amount = BitConverter.GetBytes(1000);
byte[] name   = Encoding.ASCII.GetBytes("Joe");

int bytesSent = socket.Send(amount);
bytesSent    += socket.Send(name);

socket.Close();
```



Example from http://www.eaipatterns.com/ChapterI.html

At a low level, we can create a connection to a remote machine using a socket, and send bytes of data over that socket to the remote machine for it to interpret. This can be problematic though, as it depends on us knowing (or assuming) a lot of detail about the implementation of the receiving process. In sending raw bytes, we assume that the machine at the other end of the connection is going to interpret the way that we intended. What if the sending machine is 32-bit, but the receiver is 64-bit. It will probably try to read the amount and the name together, and interpret them as a single number. That won't work well.

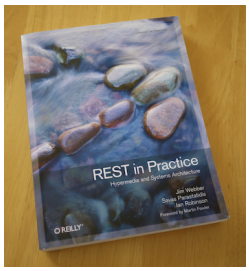
Similar problems exist when serializing objects and sending them across the network. We expect that the machine at the other end knows how to interpret a string of bytes and turn it back into a functioning object.

The Web as an Application Platform

#doc220

Rather than rely on this low level networking, it has become common to use the mechanisms provided by the web as a means of communicating between computers and applications. The web has evolved into a very robust and well-used distributed system through which a huge amount of data is managed and consumed. We can use the same sorts of techniques to build web services, to be read by machines rather than humans.

HTTP and REST



Hypertext Transfer Protocol
Methods - "verbs"
Status Codes

#doc220

The basic transport protocol that we use in building web services is HTTP - the Hypertext Transfer Protocol. This allows us to transfer documents between computers over the network. It has two other useful features: methods and status codes. There are a number of different HTTP methods that give meaning to the type of request that is being made, and allow the receiver the make decisions about what is required. The most common methods are GET and POST, but there others, like PUT, DELETE, HEAD. See http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

HTTP status codes allow the sender to know whether or not their request was successful, and if not (to some degree) why not. Status codes are normally numbers. 200 is OK. 403 is Forbidden. 500 is Internal Server Error. 503 is Temporarily Unavailable. For a full list see: http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

Resources and URIs

http://tennis-club.com/members/sam — a particular member

http://tennis-club.com/courts/5 — a particular court

http://tennis-club.com/bookings/today — all today's bookings

#doc220

REST stands for Representational State Transfer. It was characterised by Roy Fielding in his PhD thesis. REST services are built around the idea of resources and representations. Resources are things in the world, either physical or conceptual. Resources are identified by URIs - Uniform Resource Indicators (similar to URLs). We transfer a representation of a resource between computers over the network in order to communicate between different services in our system.

Wire Formats

XML

```
<schedule>
  <booking>
    <member>tim</member>
    <date>13-04-2012</date>
    <start-time>14:00</start-time>
    <end-time>16:00</end-time>
    <court>5</court>
  </booking>
  <booking>
    <member>andy</member>
    <date>13-04-2012</date>
    <start-time>15:00</start-time>
    <end-time>16:30</end-time>
    <court>1</court>
  </booking>
</schedule>
```

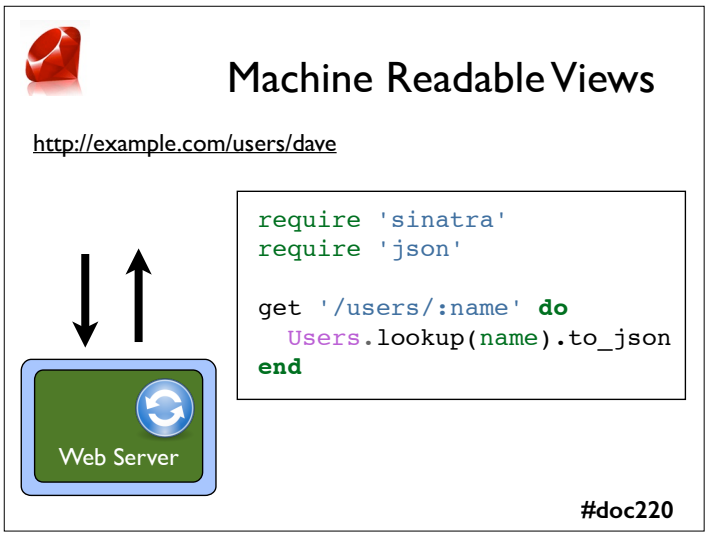
JSON

```
{
  "schedule": {
    "booking": [
      {
        "member": "tim",
        "date": "13-04-2012",
        "start-time": "14:00",
        "end-time": "16:00",
        "court": "5"
      },
      {
        "member": "andy",
        "date": "13-04-2012",
        "start-time": "15:00",
        "end-time": "16:30",
        "court": "1"
      }
    ]
  }
}
```

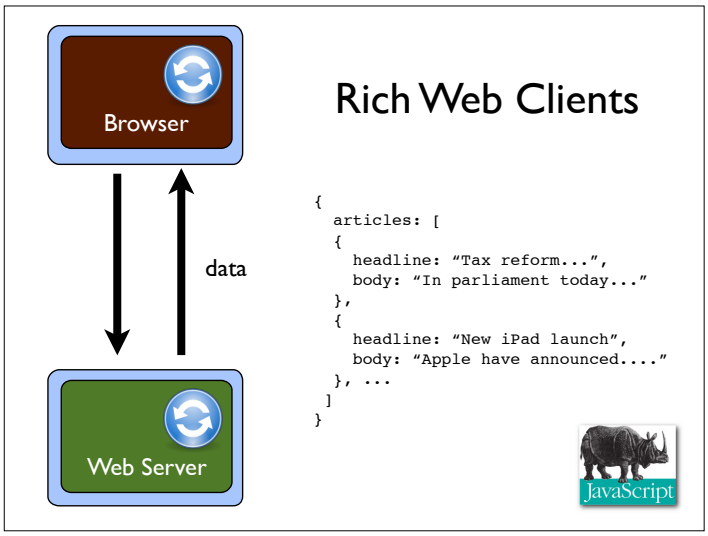
#doc220

We can have different representations of the same resource. These can be expressed in different data formats. Here we see an XML and a JSON representation of the same data. Sometimes it is possible for the client to request a particular representation - other times only one particular representation is available. That depends on the service implementation.

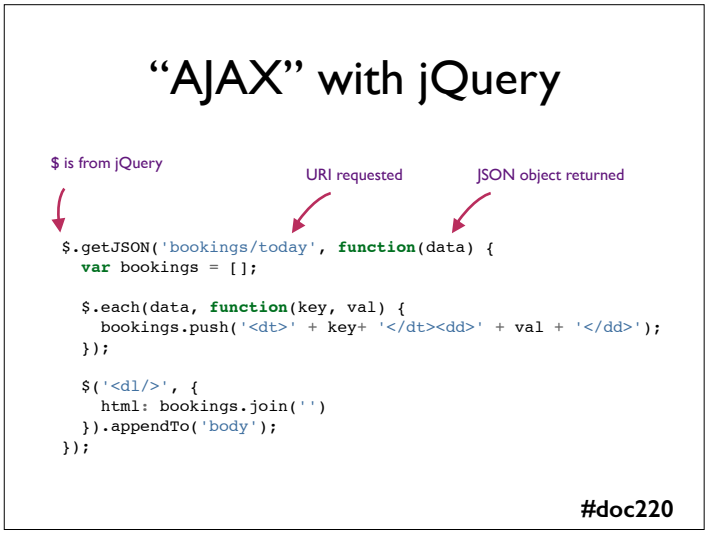
As well as these human-readable representations, some services use binary formats like Protocol Buffers, Thrift or Avro to pack data more tightly in a binary form - especially when a lot of data needs to be exchanged for processing. These formats are all platform independent, so as long as you have a suitable parser at both ends you can send data between systems implemented in different languages.



Implementing a service is quite similar to implementing a web application - it is just that the the views that we render will be in a format for consumption by machines rather than humans. In the above example we use the 'json' Ruby gem to convert our User objects to json format before returning them from the controller. Apart from that, the routing of URLs, extraction of parameters etc is the same as for a standard web app.



Now that we can build a service that produces JSON, we can consume this from a client application - for example a rich JavaScript application running in a browser. As we saw in the web applications lecture, this could be fetched asynchronously in the background by the application to update the display that the user sees.

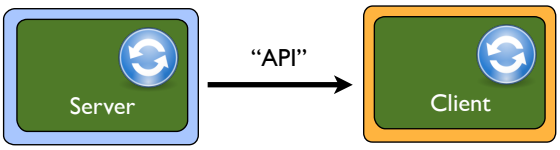


Here is an example using the jQuery JavaScript library to asynchronously load a JSON document from a URL and render its contents in the middle of the current web page.

The getJSON() function takes a service URI to query, and a callback function which will be called once the data has been retrieved. The jQuery library takes care of reading the string of characters is reads over the network, and parsing it as JSON to form JavaScript objects. We can therefore manipulate the `data` parameter directly as a JavaScript object. The mechanics of parsing are hidden from us.

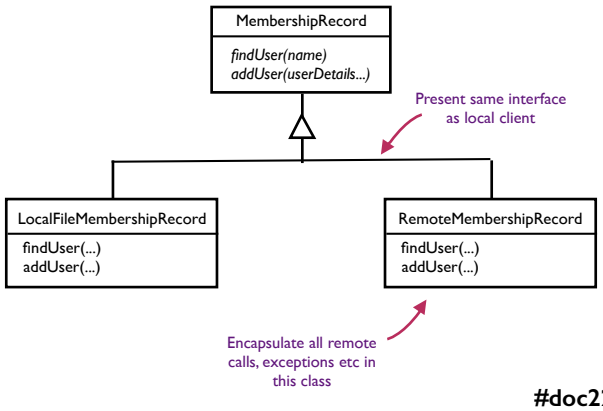
Other Clients

```
try {
   String content = Request.Get("http://tennis-club.com/bookings/today")
                           .execute().returnContent().asString();

   // process the content as XML or JSON, parse it to
   // extract data or create objects...

   return bookings;

} catch (Exception e) {
   // malformed documents or network errors may cause exceptions
}
```



Alternatively, it may be another "server side" application that is calling our web service. It is not just browsers that can connect to a URI and read data back, we can create an HTTP client in pretty much any programming language - it is easy to find convenient implementations in libraries for most common languages. Here we show an example written in Java using Apache commons HttpClient. We create a GET request for a URI, execute it and grab the content returned as a string which we can then parse as appropriate.

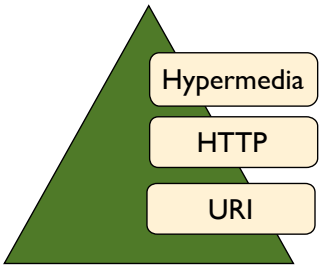
Remote Service Clients



#doc220

Most of the time we want to hide away the mechanics of how a service is accessed. We don't want to deal with exceptions caused by possible networking or parsing errors in the middle of our business logic code. In most cases we would benefit from having a service interface that the client can use that looks the same, regardless of whether the implementation being used is local or remote. This also gives us more flexibility to switch between implementations.

Richardson Maturity Model

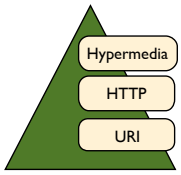


#doc220

The Richardson Maturity Model is used to describe and categorise webservices based on the degree to which they take advantage of each of URIs for identifying resources, HTTP and the various methods that it supports, and hypermedia in terms of linked data.

RMM - Level 0

Single URI
Request payload describes request
Normally HTTP POST
e.g. SOAP/WS-*



#doc220

At the lowest level of the Richardson Maturity Model we have Level 0 services. These services use HTTP (and in that sense, the web) as a means of transporting data, but do not take advantage of URIs to identity resources, different HTTP methods to describe actions, or hypermedia. They normally use a single URI to identify the service "endpoint", to which requests are posted.

SOAP

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <gs:doGoogleSearch xmlns:gs="urn:GoogleSearch">
      <q>dinosaurs</q>
      <start>0</start>
      <maxResults>10</maxResults>
      <filter>true</filter>
      <restrict/>
      <safeSearch>false</safeSearch>
      <lr/>
      <ie>latin1</ie>
      <oe>latin1</oe>
    </gs:doGoogleSearch>
  </soap:Body>
</soap:Envelope>
```

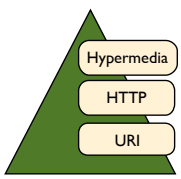
POST document to http://api.google.com/search/beta2

#doc220

In a Level 0 service, each request consists of a document or a set of parameters that describes the request. All requests generally go to the same URI by the same method. The service parses the request document to determine what the client wants. An example of this is using a SOAP (Simple Object Access Protocol) envelope to wrap an XML document describing a request. The response typically takes the same format. Technologies such as WSDL attempt to describe the protocol expected by a service, and describe the format of request/response documents.

RMM - Level 1

many URIs
only one HTTP method (GET)
side effects on GET



#doc220

Level 1 services make use of more URIs to represent different types of resources in the system, but typically do not take advantage of all of the available HTTP methods. They also do not respect the correct semantics for HTTP methods like GET, as they often use GET requests to cause side-effects on the state of the system.

Verbs tend to appear in URLs

verb: create

```
GET http://tennis-club.com/bookings/create?member=tim&court=5&time=..
GET http://tennis-club.com/bookings/cancel?member=tim&court=5&time=..
```

verb: cancel

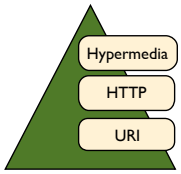
These URLs are not resource identifiers

#doc220

The above example shows the sort of URLs often presented by Level 1 services. They do not identify resources, they correspond to actions. Instead of nouns, the URLs are verbs. These URLs may all respond to GET requests, with additional parameters being passed after the question mark. To properly respect the HTTP protocol, GET requests should be idempotent and should not affect the state of the system.

RMM - Level 2

many URIs
four HTTP verbs for CRUD
HTTP status codes



#doc220

Level 2 services use URIs to represent different types of resources, and also respond to different HTTP methods (typically GET, POST, PUT and DELETE) in order to update the state of these resources. They also send appropriate HTTP status codes with their responses, which allow the client to track the effects of the calls they have made.

CRUD with HTTP verbs

```
<booking>
  <member>tim</member>
  <date>13-04-2012</date>
  <start-time>14:00</start-time>
  <end-time>16:00</end-time>
  <court>5</court>
</booking>
```

Add this booking by POSTing this representation to http://tennis-club.com/bookings

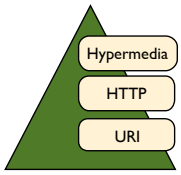
Retrieve all bookings by GETting http://tennis-club.com/bookings

#doc220

In the above example the URI http://tennis-club.com/bookings identifies a collection of all of the bookings currently known to the system. We can Create, Read, Update and Delete items in this collection. In order to add a booking, we POST a representation of our new booking (here shown as an XML document) to the collection. If we subsequently GET the URI for the bookings collection it should contain our new booking. If we wanted to update an existing booking, we could GET it from its identifying URI (e.g. http://tennis-club.com/bookings/15437) update the document, and PUT it back to the same URI. To delete a booking we issue a DELETE request to the identifying URI.

RMM - Level 3

many URIs
resource representations contain links
clients follow links
REST



#doc220

Level 3 is the highest level in the Richardson model, and characterises fully RESTful services. It builds on Level 2, with the same ideas of identifying resources using URIs, and acting upon these resources using different HTTP methods. The key point about Level 3 services is that the representations that they use contain hyperlinks to other resources that the client can follow.

With Level 2 services, clients often follow URI templates to construct the URI for a particular resource. For example, if we want to look up the user Sam, we may construct the URI http://tennis-club.com/members/sam . This reveals more about the implementation than we would like. In Level 3 services we might do a search, and then get back a document containing links to the user records, which we can follow, without having to assume the structure of those links.

Resources represented as Hypermedia

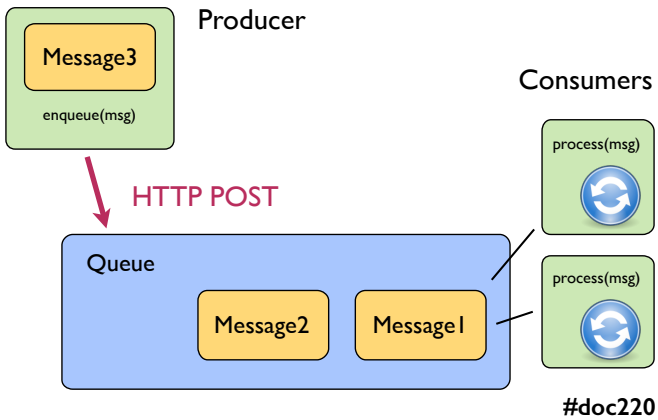
```
<schedule>
  <booking>
    <member>http://tennis-club.com/members/tim</member>
    <date>13-04-2012</date>
    <start-time>14:00</start-time>
    <end-time>16:00</end-time>
    <court>http://tennis-club.com/courts/5</court>
  </booking>
  <booking>
    <member>http://tennis-club.com/members/andy</member>
    <date>13-04-2012</date>
    <start-time>15:00</start-time>
    <end-time>16:30</end-time>
    <court>http://tennis-club.com/courts/1</court>
  </booking>
</schedule>
```

URIs rather than simple names/ids

#doc220

Here we seem an example of all of today's bookings for tennis course, returned as an XML document containing links to related resources. If we want to find information about a member or a court, we can follow the link to retrieve that resource. If one day the URIs for members change, this doesn't affect the client, as it just follows the updated links. This is taking full advantage of the mechanisms that have proved successful in building the web to implement our services.

Request-Response vs Queues



#doc220

In this lecture we have mostly looked at request-response services, where a client makes a request to another system for a resource and waits for the response. In previous lectures we looked at using queues and messages to decouple systems. We can combine the techniques to build a queue as a RESTful web service, and have producers POST messages to the queue, and consumers GET and then DELETE messages from the queue. This is how Amazon's SQS service is implemented.