Building REST Services

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Objectives

- Why REST?
- Exposing REST services using the ASP.NET WebAPI

SOAP == SOA?

- Systems built on SOA often use SOAP
  - defined standard
  - built in extensibility infrastructure
  - higher order protocols agreed
  - agreed metadata formats
  - supports arbitrary network protocols

SOAP != SOA?

- SOAP has issues
- Plumbing can be highly complex
  - e.g. WS-Security
- Service operations at single endpoint
  - scaling out problematic
  - sequence of multiple operations not defined
- "Runs on web" not "part of web"
  - all messages use POST
  - HTTP caching not supported
- Client needs special coding to remember place in series of message exchanges
  - nothing inherent in exchange tells client where they had got to

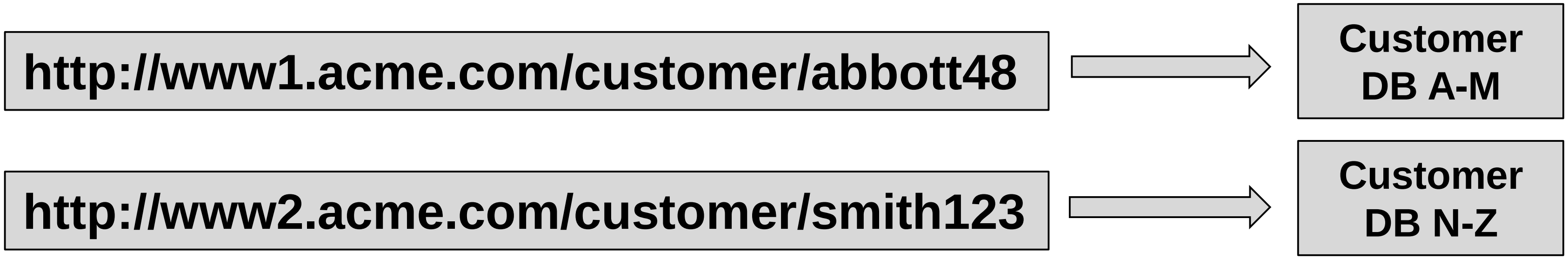
REST = REpresentational State Transfer

- Alternate way to define services
  - all operations identified by resource URI and HTTP verb
    - GET = read – must not change state of system
    - PUT = insert/update - idempotent
    - DELETE = delete - idempotent
    - POST = non-idempotent changes in state
- Many large scale systems built using REST approach
  - Amazon S3
  - Google Search API
  - Azure storage API

GET http://www.acme.com/widgets/bypartno?partno=456

REST: Part of the Web

- Resources identified by URIs
  - http://www.google.com/search?hl=en&q=REST
  - http://news.bbc.co.uk/2/hi/africa/7322468.stm
- GET is commonly cached on the client, proxy server or web server
- Link from one place to another not necessarily on the same machine
  - allows expensive operations to be dealt with by different servers/databases
  - allows simple horizontal partitioning of data



http://www1.acme.com/customer/abbott48

Customer DB A-M

http://www2.acme.com/customer/smith123

Customer DB N-Z

REST: Defining the Application Protocol

- No defined order for SOAP operations
  - InvalidOperationException
- REST response message defines the next valid URIs for message exchange
  - URIs may be data dependent

```
<product>
  <id>123</id>
  <desc>Infinite Improbability Drive</desc>
  <actions>
    <action name="techdetails"
            uri="http://www.heartofgold.com/product/123/techdetails"
            verb="GET"/>
    <action name="purchase"
            uri="http://www.heartofgold.com/basket/add"
            verb="PUT"/>
  </actions>
</product>
```

REST: URI is the State of the System

- URIs change during message exchange
  - next possible operations contained in response message
- Client can stop exchange and continue later
  - URI contains all contextual information
  - may not be possible in all circumstances
    - e.g. loan offer only valid for 48 hours

REST: Flexible Message Types

- REST is not bound to XML
  - URI may contain all data operation requires
  - XML and JSON common for sending complex data
- Response message can be any HTTP content type
  - XML
  - JSON
  - JPEG
  - MPEG
- Applications can define their own media type
  - Often specialized form of other formats

```
application/bookstore+xml
```

REST: Issues

- No metadata standard
  - message “specifications” bespoke
- No standard for “actions”
  - format for next available operations bespoke
- Tool support challenging due to lack of metadata
- Wedded to HTTP
  - not formally but in practical terms
- Building a good REST API harder than first seems
  - very easy to end up with RPC like API rather than relying on URIs

Creating REST services using WebAPI

- WebAPI ships as part of MVC4
  - Originally part of WCF
  - WCF still bootstraps non IIS hosting
- Specialized core components
  - Controller based on ApiController
  - Route registration based on HttpWebRoute

ApiController

- Derive class from ApiController
- Methods map to verbs
  - Get
  - Post
  - Delete
  - Put
- Route maps to correct override

```
public class BooksController : ApiController{
    public IEnumerable<Book> Get(){
        //…
    }
    public Resource<Book> Get(string id){
        // …
    }
}
```

HTTP as the Application Protocol

- HTTP drives the message exchange
- Need to be able to tightly integrate with HTTP
  - HttpRequestMessage
  - HttpResponseMessage

```
public Resource<Book> Get(HttpRequestMessage request, string id)
{
    // use request details
}
```

```
public HttpResponseMessage Post(HttpRequestMessage request)
{
    Book book = request.Content.ReadAsAsync<Book>().Result;

    var response = new HttpResponseMessage(HttpStatusCode.Created);

    response.Headers.Location = new Uri(request.RequestUri.AbsoluteUri +
                                        "/" + book.ISBN);
    return response;
}
```

Raising Errors

- HTTP uses status codes for error conditions
  - Can also send response data
- Raise error conditions by throwing HttpResponseException

```
Book book = AmazonLite.GetBookByISBN(id);

if (book == null)
{
    var msg = new HttpResponseMessage(HttpStatusCode.NotFound);
    throw new HttpResponseException(msg);
}
```

Content Type Negotiation

- Different clients want data in different formats
  - Xml
  - Json
  - XHTML
- Clients states preferences with `Accept` HTTP header

```
Accept: application/xhtml+xml,application/xml
```

- WebAPI looks at Accept header and formats application supports and returns compatible format
  - Json is default
  - XML via DataContractSerializer

Adding Supported Formats

- Can add custom media types to standard serializers

```
GlobalConfiguration.Configuration
                   .Formatters
                   .XmlFormatter
                   .SupportedMediaTypes
                   .Add(new MediaTypeHeaderValue("application/book+xml"));
```

- Can add own custom formatters to control serialization
  - Derive from `MediaTypeFormatter`
  - Add to `Formatters` collection
  - Can take over existing format by `Insert` into `Formatters` collection before existing formatter

Hypermedia Support

- Hypermedia (link) support is a central requirement of REST based systems
  - WebAPI does not do this for you
- Number of options
  - Operation generates links
  - Formatter generates links
  - Use XHTML and Razor

```
public class Resource<T>
{
   public Resource(T item) : this(item, new Dictionary<string, string>()){
   }
   public Resource(T item, Dictionary<string,string> links ) {
      // …
   }
   public T ResourceValue { get; private set; }
   public ILinkCollection Links { get; private set; }
}
```

Consuming REST Services

- Low technology barrier
  - Use HTTP
  - No generated proxies
  - WebRequest
  - HttpClient / System.Json

```
var client = new HttpClient {
    BaseAddress = new Uri("http://localhost.:16523/api/")
};

client.DefaultRequestHeaders
      .Accept
      .Add(new MediaTypeWithQualityHeaderValue("application/json"));

Task<HttpResponseMessage> responseTask = client.GetAsync("books");

Task<string> response = responseTask.Result.Content.ReadAsStringAsync();

dynamic json = JsonValue.Parse(response.Result);
foreach (dynamic item in json)
{
   Console.WriteLine((string)item.ResourceValue.Title);
}
```

Summary

- REST is a powerful model for public APIs
  - Support all clients
- WebAPI makes building REST system straightforward