What is REST?

From SOA to REST: Designing and Implementing RESTful Services

Tutorial at ICWE 2009

[http://icwe2009.webengineering.org/] (San Sebastián, Spain)

Erik Wilde (UC Berkeley School of Information)

June 22, 2009

This work is licensed under a CC Attribution 3.0 Unported License

Contents

- Abstract
- 1 Abstraction Layers
  - What is REST?
  - What is Architecture?
  - Architecture Examples
  - Architecture vs. Design
  - Architectural Styles
  - REST is not an Architecture
  - SOA is not an Architecture
- 2 REST: The Definition
  - The REST Architectural Style
  - Resource Identification
  - Uniform Interface
  - Self-Describing Messages
  - Hypermedia Driving Application State
  - Stateless Interactions
- 3 Web Architecture
  - What is the Web?
  - 3.1 Uniform Resource Identifier (URI)
    - Identifying Resources on the Web
    - URI Schemes
    - Query Information
    - Processing URIs
  - 3.2 Hypertext Transfer Protocol (HTTP)
    - How RESTful Applications Talk

Erik Wilde: What is REST?
**Abstract**

Representational State Transfer (REST) is defined as an architectural style, which means that it is not a concrete systems architecture, but instead a set of constraints that are applied when designing a systems architecture. We briefly discuss these constraints, but then focus on explaining how the Web is one such systems architecture that implements REST. In particular, the mechanisms of the Uniform Resource Identifiers (URIs), the Hypertext Transfer Protocol (HTTP), media types, and markup languages such as the Hypertext Markup Language (HTML) and the Extensible Markup Language (XML). We also introduce Atom and the Atom Publishing Protocol (AtomPub) as two established ways on how RESTful services are already provided and used on today's Web.
What is REST?

- Defining Representational State Transfer: 3 popular definitions
  1. An architectural style for building loosely coupled systems
     - defined by a set of very general constraints (principles)
     - the Web (URI/HTTP/HTML/XML) is an instance of this style
  2. The Web used correctly (i.e., not using the Web as transport)
     - HTTP is built according to RESTful principles
     - services are built on top of Web standards without misusing them
     - most importantly, HTTP is an application protocol (not a transport protocol)
  3. Anything that uses HTTP and XML (XML without SOAP)
     - XML-RPC was the first approach for this
     - violates REST because there is no uniform interface

What is Architecture?

- What is the "A" in SOA?
- Architecture is constraint-based design
  - constraints are derived from requirements ("contextualized requirements")
  - design without constraints probably is art
- Constraints can be derived from a wide variety of sources
  - technical infrastructure (current landscape and expected developments)
  - business considerations (current landscape and expected developments)
  - time horizon (short-term vs. long-term requirements)
  - existing architecture
  - scalability
  - performance (based on performance requirements and definitions)
  - cost (development, deployment, maintenance)
Architecture Examples

Architecture vs. Design
Architectural Styles

- Architectural Style vs. Architecture
  - Architectural Style: General principles informing the creation of an architecture
  - Architecture: Designing a solution to a problem according to given constraints
  - Architectural styles inform and guide the creation of architectures

- Architecture: [Louvre](http://en.wikipedia.org/wiki/Louvre)
- Architectural Style: [Baroque](http://en.wikipedia.org/wiki/Baroque_architecture)
- Architecture: [Villa Savoye](http://en.wikipedia.org/wiki/Villa_Savoye)
- Architectural Style: [International Style](http://en.wikipedia.org/wiki/International_Style_(architecture))

REST is not an Architecture

- REST is an architectural style
  - distilled from the Web a posteriori
  - some of the Web's standards and practices are not perfectly RESTful
- The Web is an information system following REST
- It is possible to design other RESTful information systems
  - different uniform interfaces (not using HTTP's methods)
  - different representations (not using HTML or XML)
  - different identification (not using URIs)
SOA is not an Architecture

- SOA is more a style than an architecture
- SOA's biggest problem: What is a service?
  - is a service something that is described by RPC-like custom functions?
  - is a service exposed through a uniform interface?
  - the Reference Model [http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf] defines a “service” as “a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description.”
- SOA can be done RESTfully or not
  - whether a RESTful approach makes sense depends on the constraints
  - if the constraints allow REST, there should be a good reason for ignoring REST

REST: The Definition

The REST Architectural Style

- A set of constraints that inform an architecture

  1. Resource Identification [Resource Identification (1)]
  2. Uniform Interface [Uniform Interface (1)]
  3. Self-Describing Messages [Self-Describing Messages (1)]
  4. Hypermedia Driving Application State [Hypermedia Driving Application State (1)]
  5. Stateless Interactions [Stateless Interactions (1)]

- Claims: scalability, mashup-ability, usability, accessibility
Resource Identification

- Name everything that you want to talk about
- "Thing" in this case should refer to anything
  - products in an online shop
  - categories that are used for grouping products
  - customers that are expected to buy products
  - shopping carts where customers collect products
- Application state also is represented as a resource
  - next links on multi-page submission processes
  - paged results with URIs identifying following pages

Uniform Interface

- The same small set of operations applies to everything
- A small set of verbs applied to a large set of nouns
  - verbs are universal and not invented on a per-application base
  - if many applications need new verbs, the uniform interface can be extended
  - natural language works in the same way (new verbs rarely enter language)
- Identify operations that are candidates for optimization
  - GET and HEAD are safe operations
  - PUT and DELETE are idempotent operations
  - POST is the catch-all and can have side-effects
- Build functionality based on useful properties of these operations
Self-Describing Messages

- Resources are abstract entities (they cannot be used *per se*)
  - Resource Identification guarantees that they are clearly identified
  - they are accessed through a Uniform Interface
- Resources are accessed using resource representations
  - resource representations are sufficient to represent a resource
  - it is communicated which kind of representation is used
  - representation formats can be negotiated between peers
- Resource representations can be based on different constraints
  - XML and JSON can represent the same model for different users
  - whatever the representation is, it must support links

Hypermedia Driving Application State

- Resource representations contain links to identified resources
- Resources and state can be used by navigating links
  - links make interconnected resources navigable
  - without navigation, identifying new resources is service-specific
- RESTful applications navigate instead of calling
  - representations contain information about possible traversals
  - the application navigates to the next resource depending on link semantics
  - navigation can be delegated since all links use identifiers
Stateless Interactions

- This constraint does not say "Stateless Applications"!
  - for many RESTful applications, state is an essential part
  - the idea of REST is to avoid long-lasting transactions in applications
- Statelessness means to move state to clients or resources
  - the most important consequence: avoid state in server-side applications
- Resource state is managed on the server
  - it is the same for every client working with the service
  - when a client changes resource state other clients see this change as well
- Client state is managed on the client
  - it is specific for a client and thus has to be maintained by each client
  - it may affect access to server resources, but not the resources themselves
- Security issues usually are important with client state
  - clients can (try to) cheat by lying about their state
  - keeping client state on the server is expensive (but may be worth the price)

Web Architecture

What is the Web?

- Web = URI + HTTP + ( HTML | XML )
- RESTful Web uses HTTP methods as the uniform interface
  - non-RESTful Web uses GET/POST and tunneled RPC calls
  - a "different RESTful Web" uses Web Distributed Authoring and Versioning (WebDAV)
- Imagine your application being used in "10 browsers"
  - resources to interact with should be identified (Resource Identification (1)) and linked (Hypermedia Driving Application State (1))
  - a user’s preferred font size could be modeled as client state
  - what about an access count associated with an API key?
- Imagine your application being used in "10 browser tabs"
  - no difference as long as client state is representation-based
  - cookies are shared across browser windows (different "client scope")
Identifying Resources on the Web

- Essential for implementing a Resource Identification
  - URIs are human-readable universal identifiers for "stuff"
    - many identification schemes are not human-readable (binary or hex strings)
    - many RPC-based systems do not have universally identified objects
- Making every thing a universally unique identified thing is important
  - it removes the necessity to scope non-universal identifiers
  - it allows to talk about all things in exactly the same way

URI Schemes

URI = scheme ":" hier-part [ "?" query ] [ "#" fragment ]

- URIs in their general case are very simple
  - the scheme identifies how resources are identified
  - the identification may be hierarchical or non-hierarchical
- Many URI schemes are hierarchical
  - it is then possible to use relative URIs such as in a href="../"
  - the slash character is not just a character, in URIs it has semantics

[...] the URI syntax is a federated and extensible naming system wherein each scheme's specification may further restrict the syntax and semantics of identifiers using that scheme.

Query Information

- Query components specify additional information
  - it is non-hierarchical information further identifying the resource
  - in most cases, it can be regarded as "input" to the resource
- Query components often influence caching
  - successful GET/HEAD requests may be cached
  - only cache query string URIs when explicitly requested (Expires/Cache-Control)

The query component contains non-hierarchical data that, along with data in the path component [...], serves to identify a resource within the scope of the URI's scheme and naming authority [...].


Processing URIs

- Processing URIs is not as trivial as it may seem
  - escaping and normalization rules are non-trivial
  - many implementations are broken
  - complain about broken implementations
  - even more complicated when processing an Internationalized Resource Identifier (IRI)
- URIs are not just strings
  - URIs are strings with a considerable set of rules attached to them
  - implementing all these rules is non-trivial
  - implementing all these rules is crucial
  - application development environments provide functions for URI handling
Hypertext Transfer Protocol (HTTP)

How RESTful Applications Talk

- Essential for implementing a Uniform Interface
- Misusing HTTP turns application into non-RESTful applications
- Extending HTTP turns applications into more specialized RESTful applications

HTTP Methods

- **Safe methods** can be ignored or repeated without side-effects
  - arithmetically safe: $41 \times 1 \times 1 \times 1 \times 1 \ldots$
  - in practice, "without side-effects" means "without relevant side-effects"
- **Idempotent methods** can be repeated without side-effects
  - arithmetically idempotent: $41 \times 0 \times 0 \times 0 \times 0 \ldots$
  - in practice, "without side-effects" means "without relevant side-effects"
- Unsafe and non-idempotent methods should be treated with care
- HTTP has two main safe methods: GET, HEAD
- HTTP has two main idempotent methods: PUT, DELETE
- HTTP has one main overload method: POST
Cookies

- Cookies are client site state bound to a domain
  - they are convenient because they work without having to use a representation
  - they are inconvenient because they are not embedded in representations
- Cookies are managed by the client
  - they are shared across browser tabs
  - they are not shared across browsers used by the same user
  - essentially, the client model of cookies is a bit outdated
- Two major things to look out for when using cookies:
  1. session IDs are application state (i.e., non-resource state)
  2. cookies break the back button (requests contain a "URI/cookie" combo)
- The ideal RESTful cookie is never sent to the server
  - cookies as persistent data storage on the client
  - interactions with the server are only using URIs and representations

Representations

Structured Documents

What is identified by a URI?

- Essential for implementing Self-Describing Messages
  - also should provide support for Hypermedia Driving Application State
- Resource Identification only talks about an abstract resource
  - resources are never exchanged or otherwise processed directly
  - all interactions use resource representations
- Representations depend on various factors
  - the nature of the resource
  - the capabilities of the server
  - the capabilities or the communications medium
  - the capabilities of the client
  - requirements and constraints from the application scenario
  - negotiations to figure out the "best" representation
- Representations are identified by media type (sometimes called MIME type)
Extensible Markup Language (XML)

- The language that started it all
  - created as a streamlined version of SGML
  - took over as the first universal language for structured data
- XML is a metalanguage (a language for representing languages)
  - many domain-specific languages are defined as XML vocabularies
  - some metalanguages use XML syntax (RDF [Resource Description Framework (RDF)] is a popular example)
- XML is only syntax and has almost zero semantics
  - very minimal built-in semantics (language identification, IDs, relative URIs)
  - semantics are entirely left to the XML vocabularies
- XML is built around a tree model
  - each XML document is a tree and thus limited in structure
  - RESTful XML introduces hypermedia to turn XML data into a graph

JavaScript Object Notation (JSON)

- The XMLHttpRequest API has been built for requesting XML via HTTP
  - this is useful because XML is the most popular data format
  - all requested data has to be processed by using XML access methods in JavaScript
- JavaScript does not have XML as its internal data model
  - the XML received via XMLHttpRequest has to be parsed into a DOM tree
  - DOM access in JavaScript is inconvenient for complex operations
  - alternatively, the XML can be mapped to JavaScript objects (also requires parsing)
- JavaScript Object Notation (JSON) encodes data as JavaScript objects
  - more efficient for the consumer if the consumer is written in JavaScript
  - this turns the generally usable XML service into a JavaScript-oriented service
  - for large-scale applications, it might make sense to provide XML and JSON
  - this can be negotiated with HTTP content negotiation
JSON Example

```xml
<?xml version="1.0"?>
<menu id="file" value="File">
  <popup>
    <menuitem value="New" onclick="CreateNewDoc()"/>
    <menuitem value="Open" onclick="OpenDoc()"/>
    <menuitem value="Close" onclick="CloseDoc()"/>
  </popup>
</menu>

{ "menu" : {
  "id" : "file",
  "value" : "File",
  "popup" : {
    "menuitem" : [
      { "value" : "New", "onclick" : "CreateNewDoc()" },
      { "value" : "Open", "onclick" : "OpenDoc()" },
      { "value" : "Close", "onclick" : "CloseDoc()" }
    ]
  }
}
```

Resource Description Framework (RDF)

- Developed around the same time as XML was developed
  - based on the idea of machine-readable/understandable semantics
    - builds the Semantic Web as a parallel universe on top of the Web
- RDF uses URIs for naming things
- RDF's data model is based on (URI, property, value) triples
  - triples are combined and inference is used to produce a graph
- RDF is a metalanguage built on the triple-based data model
  - RDF has a number of syntaxes (one of them is XML [Extensible Markup Language (XML)](1)-based)
  - RDF introduces a number of schema languages (often referred to as ontology languages)
Atom

- A language for representing syndication feeds
- Much more modest in its goal than XML (Extensible Markup Language (XML) (1)) or RDF (Resource Description Framework (RDF) (1))
  - models feeds as a sets of entries with associated metadata
  - uses an XML vocabulary for representing the data model
  - uses links for expressing relationships in the data model
- Will be discussed in detail as a good foundation for REST (REST in Practice)

Linked Documents

Making Resources Navigable

- Essential for using Hypermedia Driving Application State (Hypermedia Driving Application State (1))
- RPC-oriented systems need to expose the available functions
  - functions are essential for interacting with a service
  - introspection or interface descriptions make functions discoverable
- RESTful systems use a Uniform Interface (Uniform Interface (1))
  - no need to learn about functions
  - but how to find resources?
  1. find them by following links from other resources
  2. learn about them by using URI Templates (URI Templates (1))
  3. understand them by recognizing representations
**URI Templates**

- REST does not care about URI details
- Apart from the scheme, URIs should be semantically opaque
  - media types should not be guessed by URI (breaks content negotiation)
  - semantics should not be inferred from inspecting URIs
  - URIs should not be guessed based on previously encountered URIs
- "URI hacking" on the Web works and can be useful
  - Firefox Go Up [http://dret.typepad.com/dretblog/2008/07/go-up.html] allows easy navigation up one level
  - good URIs and bad UIs sometimes turn the address bar into a useful UI
- Technically speaking, URI templates are not required by REST
  - practically speaking, URI templates are a useful best practice
  - all URI navigable resources should also be navigable using representations

---

**State**

**State Management on the Web**

- Essential for supporting Stateless Interactions [Stateless Interactions (1)]
- Cookies [Cookies (1)] are a frequently used mechanism for managing state
  - in many cases used for maintaining session state (login/logout)
  - more convenient than having to embed the state in every representation
  - some Web frameworks switch automatically between cookies and URI rewriting
- Cookies have two interesting client-side side-effects
  - they are stored persistently independent from any representation
  - they are "shared state" within the context of one browser
- Session ID cookies require expensive server-side tracking
  - not associated with any resource and thus potentially global
  - load-balancing must be cookie-sensitive or cookies must be global
- Resource-based state allows RESTful service extensions
State in HTML or HTTP

State in the Server Application
State as a Resource

Typical “session scenarios” can be mapped to resources [http://www.peej.co.uk/articles/no-sessions.html]

- Client: Show me your products
- Server: Here’s a list of all the products
- Client: I’d like to buy 1 of http://ex.org/product/X, I am "John"/"Password"
- Client: I’d like to buy 1 of http://ex.org/product/Y, I am "John"/"Password"
- Client: I don’t want http://ex.org/product/X, remove it, I am "John"/"Password"
- Client: Okay I’m done, username/password is "John"/"Password"
- Server: Here is the total cost of the items in http://ex.org/users/john/basket

- This is more than just renaming “session” to “resource”
  - all relevant data is stored persistently on the server
  - the shopping cart’s URI can be used by other services for working with its contents
  - instead of hiding the cart in the session, it is exposed as a resource
**Reusing Resources**

- REST is simple to learn and use
- Unlearning RPC in most cases is the hardest part
  - OO is all about identifying classes and methods
  - distributed systems very often are built around RPC models
  - many classical IT architectures are RPC-centric by design
- REST and RPC do not mix
  - resource orientation ↔ function orientation
  - cooperation ↔ integration
  - openly distributed ↔ hiding distribution
  - coarse-grained ↔ fine-grained
  - complexity in resources formats ↔ complexity in function set