REST Eye for the SOA Guy

Steve Vinoski

Member of Technical Staff
Verivue
Westford, MA USA

QCon November 2007

Paul Downey

Just to be Clear

- REST: Representational State Transfer
- SOA: Service-Oriented Architecture

How many of you are “SOA Guys?”

SOA Basics

- Recognize similar abstractions across applications, separate them out into reusable services
- Service contracts are public, service implementations are private
- During development, continually alternate between top-down (application) and bottom-up (service) views
- Minimize coupling, maximize cohesion
- Gain buy-in across the organization to achieve common practices across the enterprise

Typical Features of Service-Oriented Systems

- *Registries*, where services advertise and applications lookup and find services
- *Repositories*, where services store metadata useful for application design and deployment
- *Definition languages*, for service contracts
- *Service platforms*, providing design- and run-time support for service creation, deployment, and execution

SOA Governance

- Enterprises contain internal organizational boundaries
- SOA systems tend to be distributed and more likely to cross such boundaries
- Want to maximize reuse and avoid duplicated effort across the enterprise
- Need rules regarding service ownership, deployment, usage, monitoring, management, security, maintenance, etc.

SOA Does Not Mean...

SOA Does Not Mean...

- Scrap Old Applications

SOA Does Not Mean...

- Scrap Old Applications
- State Of (the) Art

SOA Does Not Mean...

- Scrap Old Applications
- State Of (the) Art
- Special Object Annotations

SOA Does Not Mean...

- Scrap Old Applications
- State Of (the) Art
- Special Object Annotations
- Same (vendor) On All

SOA Does Not Mean...

- Scrap Old Applications
- State Of (the) Art
- Special Object Annotations
- Same (vendor) On All
- Scalable Optimal Architecture

Fundamentally, SOA is…

Fundamentally, SOA is...

- ...comprised of loose recommendations that essentially reiterate elementary software development practices

Fundamentally, SOA is...

- ...comprised of loose recommendations that essentially reiterate elementary software development practices
- ...lacking when it comes to actual *architecture*
  - no elements, relationships, properties, or constraints
  - technical SOA systems have these, but they’re not consistent

Fundamentally, SOA is...

- ...comprised of loose recommendations that essentially reiterate elementary software development practices
- ...lacking when it comes to actual *architecture*
  - no elements, relationships, properties, or constraints
  - technical SOA systems have these, but they’re not consistent
- ...really about organizational IT culture —”the business of IT” — and also partly about control

SOA Guy Says…

“Steve, I’m not sure I like where you’re going with this!”

REST Basics

- The term “Representational State Transfer” was coined by Roy T. Fielding in his Ph.D. thesis, published in 2000: “*Architectural Styles and the Design of Network-based Software Architectures*”
- REST is an *architectural style* that targets large-scale distributed hypermedia systems
- It imposes certain *constraints* to achieve desirable *properties* for such systems

Desired System Properties

- Performance, scalability, portability
- Simplicity: simple systems are easier to build, maintain, more likely to operate correctly
- Visibility: monitoring, mediation
- Modifiability: ease of changing, evolving, extending, configuring, and reusing the system
- Reliability: handling failure and partial failure, and allowing for load balancing, failover, redundancy

SOA Guy Says...

"So what? All distributed or network programming approaches also want to achieve those properties, including SOA."

Constraints Induce Desired Properties

- REST intentionally places constraints on the system to induce these properties
- In general, software architecture is about imposing constraints and choosing from the resulting trade-offs in order to achieve desired properties
- Contrast with SOA: it imposes zero constraints

REST Constraints

- Client-Server
- Statelessness
- Caching
- Layered System
- Uniform Interface
- Code-on-demand (optional, so we'll skip this)

SOA Guy Says...

“Well, what about that client-server constraint? Isn’t that the same old client-server idea that’s been around for forever? Next you’re gonna tell me the REST guys invented it, right?”

Client-Server Constraint

- *What*: clients and servers are distributed; clients send requests to servers, and servers reply
- *Why*: enables separation of concerns, sharing, and reuse, especially across organizational boundaries, which coincidentally is a basic goal of SOA
- *Why*: allows applications to be distributed, replicated, fault-tolerant, etc.
- *Contrast with SOA*: OASIS SOA Reference Model includes distribution in its definition of SOA

Statelessness Constraint

- *What*: resources hold resource state, clients hold application state
- *Why*: makes for simpler, more reliable, partitionable, more scalable servers that can more easily manage their resources
- *Trade-off*: clients get slightly more complicated for having to hold application state, but this approach works best for distributed systems
- *Contrast with SOA*: undefined

Caching Constraint

- *What*: servers control cacheability of their responses
- *What*: when clients use cached responses, they avoid unnecessary network and server activity
- *Why*: obviously, this constraint can **significantly** help system scalability and performance
- *Contrast with SOA*: huge hole here — many SOA-based systems don't perform caching, nor do they allow statements of cacheability
    - In SOA you cache at your own risk, or invent your own caching protocols

Layered System Constraint

- *What*: system layers interact only with adjacent layers
- *Why*: allows for hiding/encapsulation, proxies, gateways, policy management at boundaries
- *Why*: simplifies system by confining interactions, so you get better observability, management, evolution
- *Contrast with SOA*: undefined

Uniform Interface Constraint

- *What*: all servers present the same general interface to clients
  - In HTTP, this interface comprises the protocol’s verbs: GET, PUT, POST, DELETE
- *Why*: important for implementation hiding, visibility of interactions, intermediaries, scalability
- This constraint induces several more constraints, described later

SOA Guy Says...

“A uniform what? That’s unworkable! My services are all different, how can they all have the same interface?”

Revisiting SOA Discovery

- Earlier I said SOAs typically support registries and repositories for service discovery and metadata
- Finding & using a service requires knowing its interface ahead of time, otherwise you can’t use what you find
- In such systems, code is constantly dealing with interface issues
- Consider just how many pages CORBA, COM, WS-* devote to interface definitions alone

Interfaces and Scalability

- Specialized interfaces inhibit scalability
  - they require custom client coding
  - they also limit service discoverability
  - every interface is essentially a custom protocol that might keep us from achieving our desired properties
  - versioning is a big problem
- Specialized interfaces inhibit *serendipitous reuse*

SOA Guy Says...

"But services all have different semantics! You can't just call any random service through its uniform interface and expect the right thing to happen!"

Service Semantics

- In REST, interface methods deal only with resource state representations, with reasonably strong but sometimes bendable semantics
- For example, consider HTTP:
  - GET gets resource state (idempotent, no side effects)
  - PUT sets resource state (idempotent)
  - DELETE deletes a resource (idempotent)
  - POST creates/extends a resource (not idempotent)

SOA Guy Says...

“With this approach, all type safety goes out the window. How can I generate services from my programming language classes? How do I ensure service type safety?”

Remember, It’s *Distributed!*

- SOA systems typically try to give distributed systems the illusion of just extending a programming language
- You can’t pretend a distributed system is a local one
- Distributed systems generally don’t have distributed compile-time type safety, and they only fake run-time type safety
- REST focuses on fully heterogeneous distributed systems, because that’s what “large-scale” implies

SOA Guy Says...

"But, but...where's my IDL? Where's my WSDL? What describes a resource? How do I even know how to invoke these resources?"

The IDL Question

- Traditional IDLs exist for code generation of programming language interfaces/classes and method parameter data types
  - (there’s that programming language focus again)
- No automatic systems exist that download any ol’ IDL and generate fully-operational client applications
- Nobody reads *only* WSDL or IDL to write their clients
- In reality, actual human programmers read documentation in order to code against resources or services

Uniform Interface Sub-constraints

- Resource identification via Uniform Resource Identifiers (URIs)
- Resource manipulation through the exchange of representations
- Self-describing messages and possibly multiple representation formats
- Hypermedia as the engine of application state

Media Types

- REST uses media (MIME) types for data definitions
- Many such types are standardized/registered through the IANA (http://www.iana.org/assignments/media-types/)
- This approach allows resources to produce representations in multiple formats
- It allows clients to indicate the formats they’d prefer
- IDL-based systems typically tie data definition directly to the interface language, i.e. you have no choice

URIs Are Cheap, Use ‘Em

- Applications can have many states and involve many resources
- If you can name a resource, give it a URI
- In each resource representation, include URIs to related resources to guide clients through the application state
- Use standardized MIME types for representations, e.g., (X)HTML, JSON, Atom
- Keep verbs out of your URIs

Summary

- There's nothing inherently wrong with SOA, but it's all about IT culture and organizations, not architecture
- REST is an architectural style for distributed hypermedia systems featuring specific constraints to induce desired system properties
- REST-style applications are built around the exchange of resource state representations in standard data formats through a fixed uniform interface

Get This Book!

- This book is excellent. It will open your eyes to the possibilities of REST and help you choose the best ways of designing REST-based systems.

Web Services for the Real World
RESTful
Web Services
O'REILLY
Leonard Richardson & Sam Ruby

For More Information

- Attend the rest of the talks in this track
- Fielding's thesis
  - http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
- Read the blogs of Mark Baker, Bill de hÓra, Joe Gregorio, Paul Downey, Benjamin Carlyle, Stu Charlton, Mark Nottingham, and the host and speakers of this track
- Sign up to the *rest-discuss* Yahoo mailing list
- My "Toward Integration" columns in IEEE Internet Computing sometimes discuss REST (all columns are available from http://steve.vinoski.net/)

Thanks