REST Easy: API Design, Evolution, and Connection

lean • enterprise • middleware

Version 1.1 (April 18, 2014)

Table of Contents

RESTful design increases API performance, reduces development effort, and minimizes operational support burden. By conforming web applications, web services, and web APIs with proven RESTful constraints, teams can create systems as scalable, pervasive, and prolific as The Web itself. Because REST requires a new development mindset, many API implementations are not truly based on REST constraints and principles, and do not exhibit expected scalability, evolvability, and interoperability. By following a few best practices and selecting RESTful tooling, teams can easily design, evolve, and connect RESTful APIs.

Easy REST Definition

RESTful web services and RESTful APIs are the de facto standard in consumer/web scale enterprises (e.g. Twitter, Google). How can we define REST, and understand how to adeptly build RESTful systems?

Roy Fielding's research dissertation defines popular network based software architecture styles, and contains a chapter proposing REST constraints and principles. According to Roy Fielding, *REST itself is just a name chosen for a specific style that I defined as characteristic of the best designed parts of the Web. - November 2007*

REST Constraints

By conforming web applications, web services, and web APIs with proven REST constraints, teams can create systems as scalable, pervasive, and prolific as The Web itself. REST constrains system to

- Client server interactions
- Stateless communication
- Cacheable data
- Uniform interfaces
- Layered systems
- Code-on-demand

Each constraint adds beneficial properties to the web system. By incorporating the constraints, teams can build simple, visible, usable, accessible, evolvable, flexible, maintainable, reliable, scalable, and performant systems. Table 1 maps how following specific REST constraints will result in gaining valuable system properties.

Table 1: REST Constraints and System Properties

By Following This Constraint	Gain This System Property
Client-server interactions	Simple, evolvable, scalable
Stateless communications	Simple, visible, maintainable, evolvable, and reliable
Cacheable data	Visible, scalable, and performant
Uniform Interfaces	Simple, usable, visible, accessible, evolvable, and reliable
Layered system	Flexible, scalable, reliable, and performant
Code on demand	Evolvable

When building APIs, web applications, or web services, developers often unevenly apply REST constraints. For example, a web application may exhibit code on demand (i.e. client-side JavaScript), layered system design, and client-server interactions, but not exhibit uniform interfaces, statelessness, and cacheable data.

If the browser back-button doesn't work, your web application (or web API) is not fully RESTful.

APIs may exhibit uniform interfaces, client-server interactions, and layered system but not exhibit stateless communication and cacheable data.

If your API client must be bound to a sticky session, or if servers must run session clustering, or if your server doesn't check the last-modified header before responding, your API is not fully RESTful.

Client Server Interactions

Client server interactions are the foundation underneath Internet-scale systems, independent component evolution, and pervasive reach. Client server interactions decouple REST clients from REST servers, and enforce a separation of concerns. By restricting interactions to common Internet protocols, message formats, and scripting languages (i.e. HTTP, JSON, XML, JavaScript), diverse client platforms can connect to server-side resources.

Stateless Communication

When communications are stateless, every request happens in isolation, and the client is responsible for maintaining state. The client request contains all information required to process the request, and a RESTful server does not use local state information from previous requests. A stateless communication litmus test is to turn off session cookies, and determine if the API, web service, or web application still works as designed.

RESTful APIs, services, and applications **can still exhibit state**. However, state information must be embedded in representation hyperlinks (e.g. URLs, message hrefs). By explicitly transferring state to the client, the response message provides an explicitly valid operational path through the system to the client consumer.

With stateless communication, shared temporary context is not stored on servers. Because temporary state context must be sent across repetitive requests, stateless communication does increase network traffic. But with network connectivity increasing, the benefits often outweigh the cost.

Cacheable Data

RESTful system mark data as cacheable, and respect cache markers. Cacheable data decreases network traffic and reduces back-end system load. Intermediary nodes can return cached data on behalf of back-end servers, increasing scalability, availability, reliability and performance. In a RESTful system, intermediaries recognize information freshness, idempotent operations (i.e. GET), and versioning schemes.

Uniform Interfaces

While the uniform interface constraint is the most defining and visible REST constraint, the uniform interface constraint is commonly a source of incomplete adherence to RESTful principles and techniques. Uniform interfaces:

- Identify resources
- Manipulate resources through representations
- Present self-descriptive messages
- Rely on Hypermedia As The Engine Of Application State (HATEOAS)

Uniform interfaces should identify message behavior and semantics without looking in the message body. The URL or URI should state the resource being manipulated. Uniform interfaces do not depend on saved message states. Uniform interfaces rely on standard messages (i.e. GET, HEAD, POST, DELETE) and communicate media type processing information.

When layering REST constraints on top of HTTP and URLs, RESTful systems are optimized for large-grain hypermedia data transfers, yet sub-optimized for other interactions.

Layered Systems

By creating a layered system, teams can independently evolve client and server components. An API Facade hides internal implementation complexity, and presents a simple interface to external consumers. Simple RESTful API interfaces hide multiple back-end databases and aggregated services (see Figure 1).

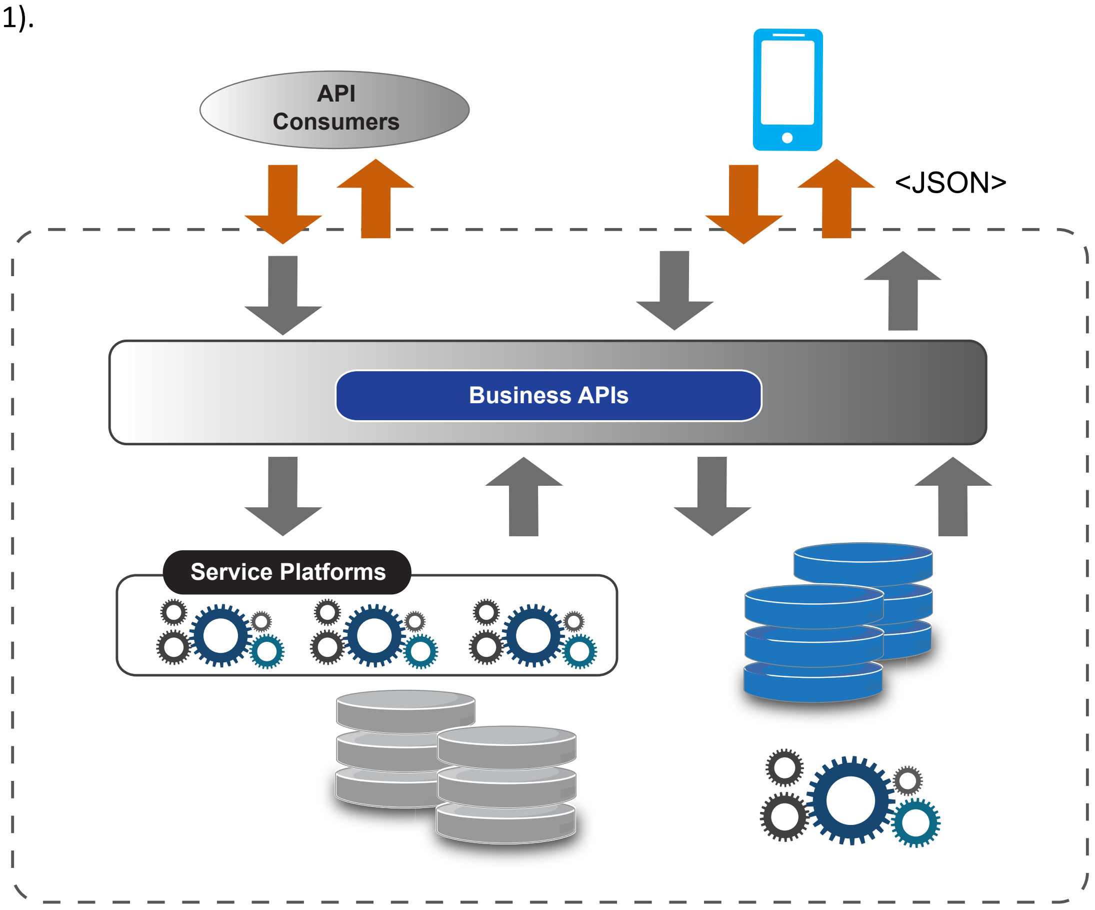

Figure 1: API Façade Pattern

By employing the API Façade pattern, teams can easily connect legacy systems using complex mediation techniques, but also provide a simple, business friendly API to consumers. Under the hood, the solution may chain services, orchestrate services, and transform messages.

The API façade pattern lets teams independently scale each architecture layer, and apply infrastructure policies (e.g. authorization, authentication, response time, rate limits) per system service or API. The API façade bounds the system's complexity, and by applying RESTful principles and constraints to the API façade, the team further reduces integration complexity.

Code on Demand

Code on demand allows clients to extend functionality by downloading and executing code locally. By enabling code on demand, systems reduce the required number of pre-implemented features, and improve system extensibility. Downloading and executing client side Javascript or plug-ins within a browser or REST client conforms to REST constraints.

Structures at REST

RESTful APIs must conform to REST constraints, and only contain REST structures built from REST data elements (Table 2). REST simplifies distributed network architecture by defining resources, representation, and control data.

Table 2: REST Data Elements

Data Element	Description
Resource	Conceptual target of a hypertext reference e.g. customer/order
Resource identifier	A uniform resource locator (URL) or uniform resource name (URN) identifying a specific resource **e.g.** http://my.rest.com/customer/3435 http://my.rest.com/order/503
Resource metadata	Information describing the resource **e.g.** tag, author, source link, alternate location, alias names
Representation	The resource content JSON message, HTML document, JPEG image
Representation metadata	Information describing how to process the representation **e.g.** media type, last-modified time
Control data	DInformation describing how to optimize response processing **e.g.** if-modified-since, cache-control

Resources

A resource is a logical information entity, and any web entity (i.e. API, service, application, location, action, video, or image) can be referenced as a resource. A URI identifies every resource, and RESTful clients interact with resources using a uniform interface.

Representations

In REST-speak, a resource is an identifier, and representations specify the actual item. Resources are manipulated through representations. A representation may be a JSON message describing a customer, a video file, or an XML configuration file. A representation contains the resource state, and the representation may contain resource identifiers pointing towards the next valid state. For example, an expense report may contain links to expense detail items, employee record, and approval status resource.

By decoupling resource from representation, **systems may expose multiple representations per resource identifier.** For example, a representation may be returned in JSON, XML, or CSV based on the REST client's processing capabilities. Client processing capabilities are communicated via the Accept header.

Getting REST

Fourteen years after Roy Fielding's paper, not many people understand how to build RESTful systems. Impedance mismatches between REST constraints and structures with programming languages, frameworks, and architectural paradigms often lead teams to built immature RESTful applications. To 'get REST', remember these three principles

- Hypermedia design
- Application behaviour modelled as state machines
- CRUD only operations

REST is incompatible with the commonly understood method oriented programming style. When defining method-oriented interfaces, activities are the abstraction and services encapsulate many operational methods. In contrast, with resource-oriented interfaces, information is the abstraction, and a resource model is the service interface. When building resource-oriented systems, a few fixed operations are used to operate on resource interfaces. Many people understand how to manipulate data resources by following the Create, Read, Updated, and Delete (CRUD) metaphor, and CRUD design techniques can be utilized to build RESTful systems. Additionally, CRUD reduces systems to four verbs, which are readily mapped to the HTTP protocol and web servers used to build large scale distributed systems.

When modelling application behaviour as state machines, specify resource identifiers for states and representations contain state transition logic. Hypermedia design specifies links as resource relationships and transitions between resources states. By defining semantics as media types, hypermedia design inherently extensible to new message formats and code on demand.

With hypermedia design, hypermedia as the engine of state, and resource orientation, you can easily design RESTful Web API, promote easy connections, and easily evolve RESTful systems.

Easy Connections

Complexity and machine readability often hinder pervasive consumer connections. While SOAP's flexibility (i.e. RPC, document wrapped, document styles) and XML's modularity (via namespaces) create well-architected service architecture, efficiently building a client connection requires specialized tooling (e.g. Java2WSDL, WSDL2Java, SOAPUI). By reducing integration components to resources, representations, and optional metadata, human developers can readily explore the interface with simple tooling. Simple, human readable JSON and HTTP tools (i.e. curl, wget, apache HTTPD) allow teams to easily create connections.

Creating easy connections does require thinking about information systems in terms of nouns, not verbs. Identifiers should change infrequently, and the interface should be general purpose. Rather than expose internal models, layer Enterprise Integration Patterns (EIP) behind the RESTful API.

Design an Easy to Use API

API design does not equal service design. In addition to designing the interface message (REST representation), carefully craft URL identifiers (REST resources), define usage policy (i.e. rate limits), and service level agreement (i.e. response time, availability). Take a resource identifier first focus and allow flexible representations. Allow clients to negotiate the representation format, and RESTful clients will permissively process the representation. An example of permissive processing is a Web browser that will render malformed HTML rather than displaying a blank page.

Design resource identifiers for faster interaction by conforming design to uniform interface and HATEOAS conventions.

Conform with Uniform Interface

To build uniform interfaces, restrict interactions to simple, easy to understand CRUD methods, and think in terms of nouns instead of verbs.

When crafting REST interactions over HTTP, adhere to the following chart:

Table 3: Uniform Interface Methods

HTTP Method	Purpose
GET	Retrieve information (possibly cached)
PUT	Create or update with a known ID
POST	Create or append a sub-resource
DELETE	Remove (perhaps logically)

Follow a few simple uniform interface guidelines

1. Don't change data with GET
2. Don't tunnel methods over POST. Most webapps fail this test. Breaks uniform interface constraint
3. Don't tunnel methods over URIs. Breaks uniform interface constraint
   http://NOTrest.com/customers?method=delCustomer&name=Smith
   http://NOTrest.com/orders?method=deleteOrder&id=34534
4. Change typical web application verbs into nouns
   http://NOTrest.com/user/3453?method=checkCredit
   http://RESTful.com/user/3453/creditDoc?date=2014-04-11

Conform with HATEOAS

To design Hypertext As The Engine Of Application State (HATEOAS) systems, build state into the message representation, and remove server-side state.

Eliminate (or minimize) server-side state. Remove session cookies with session replication or sticky server affinity sessions with stateless design. Current state is stored in client-side message. Most web applications break this principle.

Present the next state and related resources using embedded hyperlinks. Follow these simple design guidelines:

1. Give everything an identifier. Assign an identifier to any resource important enough to directly manipulate or access.
2. Link things to each other (API response representations contain additional hyperlinks)
3. Use only standard methods

Easily Evolve APIs

To easily evolve APIs, think through versioning, eventing, and hypermedia maturity.

Versioning Considerations

API evolution must consider forward compatibility, backwards compatibility, and client processing flexibility.

When representation modifications don't trigger a message processing exception in existing clients and preserves expected client-processing output, the API modification is backwardly compatible. Permissive client processing 'fails gracefully' and can accept a wide range of message format modifications. When modifications are backward compatible, you may choose to automatically grant existing API subscriptions to the new API definition.

When a representation modification will trigger a client message processing exception or modify expected outputs, the modification is not backwardly compatible. A best practice is to define a new resource identifier, and force clients to re-subscribe to the new API containing the resource identifier.

Modeling Events and Queues

Pushing events out to clients requires managing subscription lists (creating server state) and maintaining persistent client-server connections. Scaling push-based systems to web scale is complicated and expensive. In REST, events are published as timelines, and clients pull the resources.

Hypermedia Maturity

The Richardson maturity model recognizes that systems may only partially adhere to REST principles and constraints. The maturity model specifies four distinct levels.

Level Zero: The SWAMP of POX

A single URL and a single HTTP method are used invoke custom methods. Most commonly, POST is used to exchange XML, and the specific invocation method is tunneled using SOAP, XML RPC, or Plain-Old-XML (POX) custom schemes.

Level One: Define Explicit Resources

Many public resource URIs are exposed, and methods are 'tunneled' by appending the method definition onto the end of the URL. Resource definitions commonly align with basic data entities, yet actions are not expressed as standalone documents.

Level Two: Thinking Only in Nouns

Level two treats data entities similar to level one, and additionally treats actions as standalone URIs. Clients can now directly post representations to the action resource.

Level Three: Hypermedia Controls

Level three augments level two by embedding stateful links in the resource representation (i.e. response message). Clients do not require a prior knowledge of the application flow. Applications can follow root entity definitions and navigate the systems workflow.

REST Tooling

In section 6.3 of Roy's dissertation, he explains how REST applies to HTTP. But implementing the explanation required painstaking assembly. Java JAX-RS and API Management infrastructure reduce the learning curve, increase API adoption, and decrease development effort by simplifying API creation, publication, and consumption.

The Java API for RESTful Web Services: JAX-RS

JSR 311, JAX-RS is Java's RESTful programming model. In JAX-RS, a single class corresponds to a resource. Java annotations are used to specify URI mappings, mime type information, and representation meta-data conforming with REST constraints (see Table 4).

Table 4: Mapping REST concepts to JAX-RS

REST concept	JAX-RS Annotation or class	Examples
Addressability	@Path and URI Path Template	@Path("/user/{username}")
Uniform Interface	@GET, @PUT, @POST, @DELETE, @HEAD	@GET @Produces("application/json") public String getUser (String username) { return erService(username)); } getUserService(username)); }
Self-descriptive messages	@Produces, @Consumes	@roduces({"application/xml", ap- plication/json"})
HATEOAS	UriBuilder	UriBuilder.fromUri("http:// localhost/"). path("{a}"). queryParam("name", "{value}"). build("segment", "value");

WSO2 Application Server relies on Apache CXF to process JAX-RS annotations and expose a RESTful API. Your existing Apache CXF code can be readily migrated to WSO2 Application Server.

API Management

RESTful APIs may be either naked or managed. A naked API is not wrapped in security, subscription, usage tracking, and service level management. A managed API increases reliability, availability, security, and operational visibility. By placing an API gateway in front of your naked RESTful API or service, you can easily gain advanced capabilities (see Figure 2).

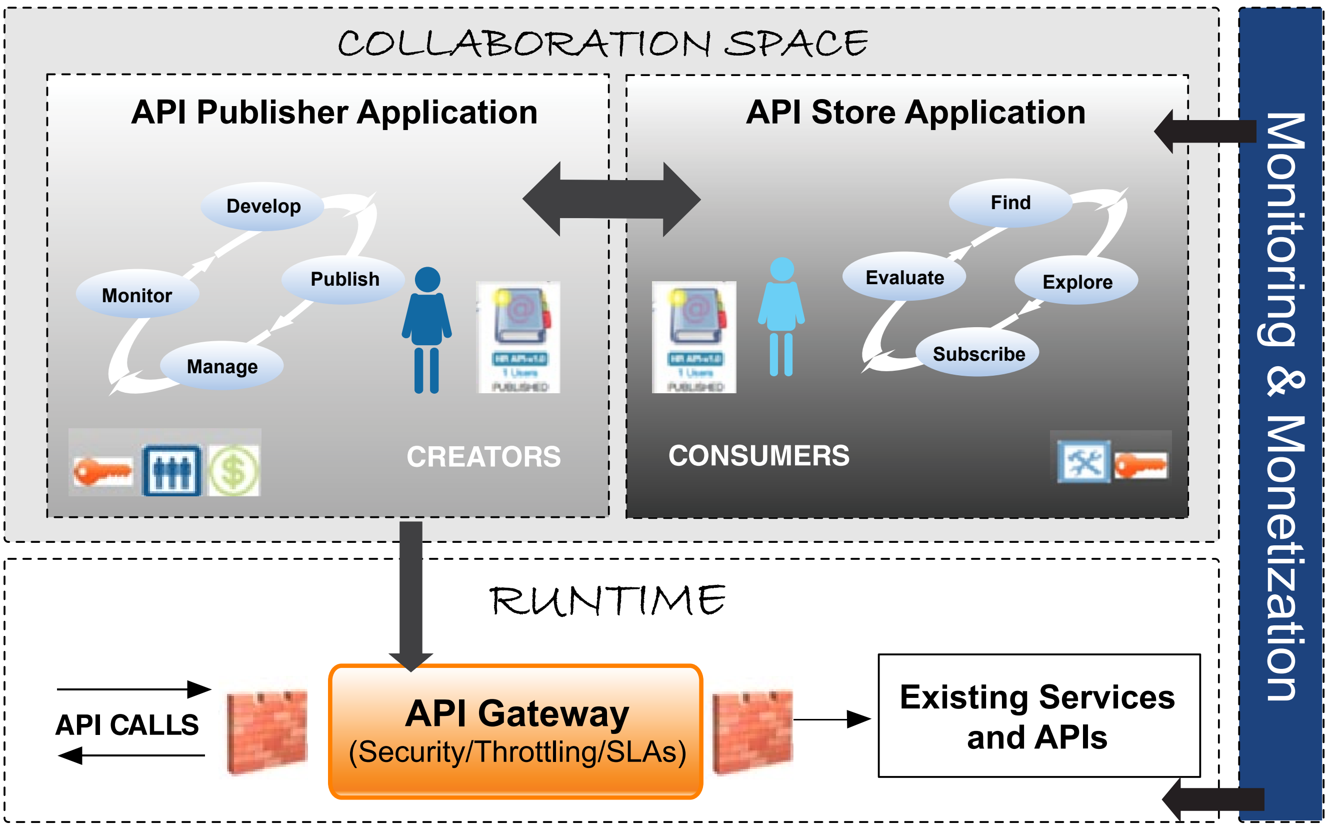

Figure 2: API Management Capabilities and Topology

The API gateway systematizes the API façade pattern, and enforces authorization, quality of service compliance, and usage monitoring without requiring any back-end API modifications. Figure 3 demonstrates API facade actions commonly provided by industry leading API gateway products.

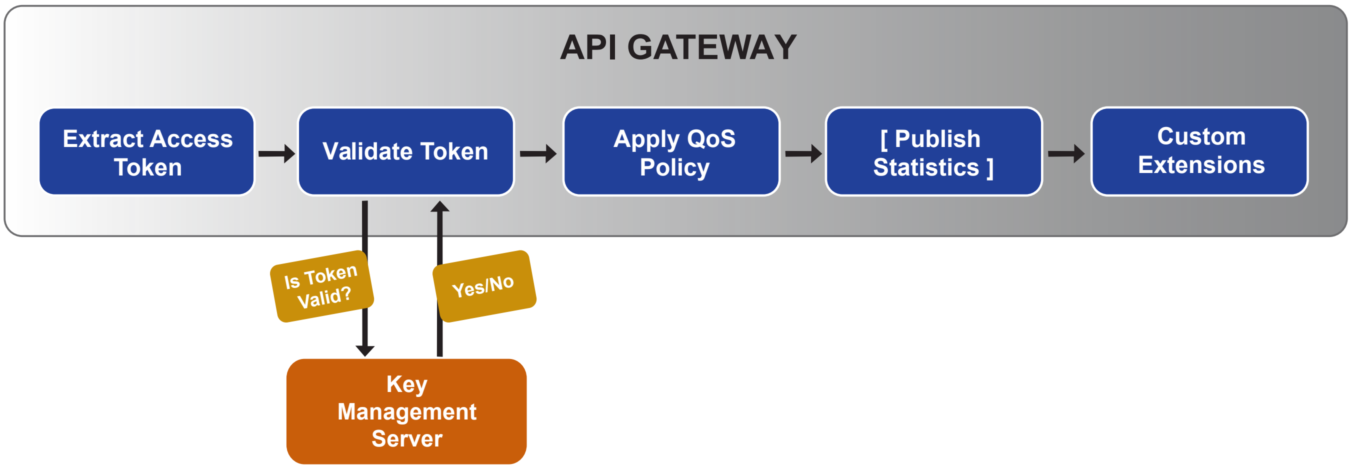

Figure 3: API Façade Operations

WSO2 API Manager can easily integrate with your RESTful system and rapidly add advanced capabilities. For more information on API management, read the technical evaluation guide.

Aligning Work with REST

RESTful systems must consider security, separation of concerns, and legacy web services. Support for open standards (OAuth, OpenID Connect, UMA, SAML 2.0, XACML)

Build an API Security Ecosystem

Security is not an afterthought. It has to be an integral part of any development project. The same applies to APIs as well. API security has evolved significantly in the past five years. The growth of standards to date has been exponential. OAuth is the most widely adopted standard, and is possibly now the de-facto standard for API security. To learn more, read the Build an API Security Ecosystem white paper.

Promote Legacy Web Service Re-use with API Facades

RESTful APIs are a strategic component within your Service Oriented Architecture initiative. Many development teams publish services, yet struggle to create a service architecture that is widely shared, re-used, and adopted across internal development teams. RESTful APIs extend the reach of legacy web services. To learn more, read the Promoting Service Re-use white paper.

Converging RESTful API Strategies and Tactics with Service Oriented Architecture

While everyone acknowledges RESTful APIs and Service Oriented Architecture (SOA) are best practice approaches to solution and platform development, the learning curve and adoption curve can be steep. To gain significant business benefits, teams must understand their IT business goals, define an appropriate SOA & API mindset, describe how to implement shared services and popular APIs, and tune governance practices. To learn how REST and SOA coexist, read the Converging API Strategy with SOA white paper.

About WSO2

WSO2 is the only company that provides a completely integrated enterprise application platform for enabling a business to build and connect APIs, applications, Web services, iPaaS, PaaS, software as a service and legacy connections without having to write code; using big data and mobile; and fostering reuse through a social enterprise store. Only with WSO2 can enterprises use a family of governed secure solutions built on the same code base to extend their ecosystems across the cloud and on mobile devices to employees, customers and partners in anyway they like. Hundreds of leading enterprise customers across every sector—health, financial, retail, logistics, manufacturing, travel, technology, telecom and more—in every region of the world rely on WSO2's award-winning, 100% open source platform for their mission-critical applications. To learn more, visit http://wso2.com or check out the WSO2 community on the WSO2 Blog, Twitter, LinkedIn, Facebook, and FriendFeed..

Check out more **WSO2 Whitepapers** and **WSO2 Case Studies**.

For more information about WSO2 products and services,
please visit http://wso2.com or email bizdev@wso2.com