



Service Application Engine™

A Collaborative Computing Platform for Cloud and Web

HTTP and REST Application Programming Interface

Release 3.3

Copyright 2007-2012 StreamScape Technologies. The trade name StreamScape and the product names Service Application Engine™ and Service Event Fabric™ are trademarks of StreamScape Technologies. This document may not be reproduced in any medium without the express permission of StreamScape Technologies LLC.

For problems and issues with this documentation please contact us at support@streamscape.com

Table of Contents

| | |
|--|----|
| Table of Contents | 3 |
| Document Conventions | 4 |
| REST API Overview..... | 5 |
| Representational State Transfer (REST) | 5 |
| REST Architecture | 5 |
| REST Interface Principles | 7 |
| RESTful Web Services and API | 8 |
| Using the REST API..... | 9 |
| Accessing Service Engine Resources | 9 |
| Application Engine Clients | 13 |
| Security Considerations | 13 |
| RESTful API Guide | 19 |
| Invoke Service | 22 |
| Raise Event..... | 25 |
| Raise Request..... | 27 |
| Receive Event..... | 30 |
| Receive Event with NoWait | 32 |
| Querying Service Configuration | 33 |
| Get Service List..... | 33 |
| Get Service Reference..... | 33 |
| Get Service Actionable Events | 34 |
| Get Service Event Handlers..... | 34 |
| Get Service Event Handler | 35 |
| Get Service Configuration Object | 35 |
| Get Service Request Object | 36 |
| Get Service Response Object | 37 |
| Get Service Request Event..... | 38 |
| Get Service Response Event..... | 39 |
| Browsing the Entity Repository | 40 |

Document Conventions

This document uses the following font conventions:

Italic text is intended to express conceptual terms that are part of StreamScape taxonomy and terminology. The intent is to provide users with indicators of terms that are used to define architectural concepts and functions.

Source code samples and output are presented in boxed `Courier New` font for example:

```
tnode -init -log -dir C:\StreamScape\nodes\demo -ddx C:\StreamScape\deploy\demo
```

General script and command syntax examples are provided using embedded `Courier New` text, for example:

The Java Archive is located in the `<install_root>/platform/lib` directory.

Semantic Language Commands (SLANG), Event Definition Language (EDL) and Data Space Query Language (DSQL) syntax and command verbs are expressed in **BOLD UPPER CASE**, for example:

INSERT INTO and LIST COMPONENTS

When describing command syntax and language elements the following conventions will be used based on a simplified EBNF notation:

| | |
|-------------------------------------|--|
| <code><Token></code> | Required value or parameter that is an identifier |
| <code>[Token]</code> | Optional value or parameter that is an identifier |
| <code>'<Token>'</code> | Required alphanumeric string |
| <code>'[Token]'</code> | Optional alphanumeric string |
| <code>{Token1 Token2}</code> | Required value that may be one of the specified choices |
| <code><Expr></code> | A substitution expression that resolves to a single value of the specified data type |
| <code><Token = Value></code> | Assignment |
| <code><Token == Value></code> | Equality |

REST API Overview

Representational State Transfer (REST)

Representational State Transfer (REST) is a style of data exchange between two software components within a distributed system, such as the [World Wide Web](#). Although technically it is just a data exchange pattern, REST has emerged over the past few years as a predominant [Web Service](#) architecture model. As of this writing, thousands of applications, Cloud System developers, Web and mobile application designers have used the REST design model to offer full service platforms, [mash-up](#) and [widget](#) capabilities to their users; resulting the emergence of a so-called API Economy.

By strict definition of REST constraints, a [client](#) application makes a synchronous request to a [resource](#) provider and receives back some data. This information now represents the new state of the [client](#) application. For example an application may request authentication with a back-end system by passing a security credential to the [resource](#) provider. The result of such a request may be an *authenticated session identifier*. This allows the client to transition into a trusted (authenticated) mode, ready to transmit secure work requests. A subsequent request may return a list of available resource providers or services that a client application may make use of.

While this concept is not new (mainframe systems used to call such exchanges *conversational programs*), its application to distributed system design is a relatively recent development. Whereas legacy systems focused on conversational interactions between a single client and a [server](#), REST –based systems allow the same client application to access a broad range of resource providers. This model offers developers the ability to compose applications from a variety of resources and capabilities, leading to new synergies and innovation.

The term *representational state transfer* was introduced in 2000 by [Roy Fielding](#), one of the principal authors of the [Hypertext Transfer Protocol](#) (HTTP) specification. REST as an architectural style was developed in parallel with [HTTP/1.1](#). The largest implementation of a system conforming to the REST architectural style is the [World Wide Web](#) itself. REST exemplifies how the Web's architecture and application design evolved based on the interactions of four key components of the Web: [origin servers](#), [gateways](#), [proxies](#) and [clients](#), without imposing limitations on individual participants.

As an architectural approach REST –based applications are complementary to Collaborative Computing systems because they promote self-governance and proper behavior of participants. From a technology perspective, REST constraints limit the communication model, thereby simplifying implementation and ultimately, reducing cost. Conforming to the [REST Constraints](#) outlined below is generally referred to as being "RESTful".

REST has increasingly displaced other interface design models such [Messaging](#), [SOAP](#) and [WSDL](#) due to its simpler style and flexibility of *format* and *protocol*. Most REST –based implementations conform to some of the constraints but do not enforce the strict definition of application state transitions described above. As such, RESTful applications make use of the concepts to facilitate a simpler and more flexible RPC mechanism. The Service Application Engine™ supports all REST constraints including *Layering* and *On-Demand Code* facilities.

REST Architecture

REST-style architectures consist of [clients](#) and [servers](#). Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of resource representations. A [resource](#) is any structured and meaningful concept that may be located by a discrete resource address, a so-called Uniform Resource Identifier (URI). The [representation](#) of a resource is typically a document or similar structured data unit that captures the current or intended state of a resource.

What is particularly interesting about the REST architecture is that resource state may represent conversational sessions, application services or transactions, or a combination of such elements without restrictions. Consider the

example of a well-known Shopping Cart application when re-implemented as a REST –based system: A consumer must enter the Store by providing a security credential and identify themselves. As a result of this operation the REST resource provider validates a consumer and puts the consumer application into a *ready state* by returning a session token and the address of the next resource manager. An application communicates its readiness by accessing the consumer’s Profile and Preference information from a resource at the specified location (URI).

The profile document returned to a consumer may be (and frequently is) augmented by promotional advertising targeted at the individual. If this is a returning customer the system may offer additional incentives by including links to other store resources. Adding items to a Shopping Cart begins an actual purchasing session and may potentially result in reservation of funds on behalf of the consumer, for example by contacting PayPal or a similar online payment system.

If a Shopping Session is interrupted for some reason and the consumer application remains functioning it may potentially cache session information and credentials. When communication is re-established the application can request to be re-directed to the same resource provide it was using and complete the transaction without the need to re-enter information. REST facilitates transactions between resources by allowing [loose coupling](#) of services and their consumers, potentially without sacrificing control over session state.

This approach to application state management allows server-side implementations to be essentially Stateless and provides for a high degree of *fault tolerance* and *scalability* without the need for developing complex back-end systems to manage session state. REST resource managers are abstracted, resulting in a Layered architecture that easily facilitates load balancing and caching of critical data.

The Service Application Engine™ is purpose-built to support REST application and provides session control, load balancing and data caching as well as Asynchronous HTTP Communication extensions in addition to transaction access to in-memory resources making it an ideal intermediary (proxy) for RESTful client applications.

REST is weak-typed when compared to its [SOAP](#) counterpart. There is no schema enforcement and no standard for data exchange beyond Standard General Markup Language (SGML). REST commands are based on the use of nouns and verbs typically found in the HTTP protocol (such as GET, PUT, POST and DELETE) with emphasis on readability and data access. This makes the REST operation set closer the [CRUD Matrix operations](#) of a database engine. Unlike SOAP, REST is not completely dependent on [XML](#) parsing or rendering technologies and does not have a protocol specification or a formal interface definition language, thus making more compact and versatile.

The REST model describes the following six constraints applied to an architecture model that make it RESTful:

Client-Server

A uniform interface separates clients from servers. This [separation of function](#) means clients have no dependency on data storage, language or the specifics of a resource server implementation; thus improving client the [portability](#) and making it server-agnostic. Resource servers are not concerned with the applications overall interface or state, making server implementations simple and more [scalable](#). Format of the client-server interaction is user defined allowing system components to be developed in a decoupled fashion, replaced or developed independently.

Stateless

Client context does not have to be stored or represented on the server between requests. While this capability may be provided as a combination of client-side caching and intermediate proxy technologies a resource server is not aware of client application state in contrast to traditional database and application server technologies.

Each request from a REST client contains all of the information necessary to service the request, and any session state is maintained by a client or intermediary as necessary. Although a resource server may be [stateful](#) this constraint merely implies that the client request (URL) provide enough information to address a server-side state manager (for example by providing a *session id*). This option makes servers more [visible for monitoring](#), and also makes them more [reliable](#) and [scalable](#) in the face of partial network failures.

Cacheable

Clients or REST intermediaries may cache responses. Therefore responses must implicitly or explicitly, define themselves as cacheable, or not, to prevent clients from using stale or inappropriate data as the basis for further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.

Layered System

A client is abstracted from the resource server and cannot typically determine if it is connected directly to the end server, or to an intermediary along the way. REST intermediaries may improve system scalability by enabling load-balancing, providing shared caches, session management or enforcing security policies.

Code on Demand (optional)

Servers are able to extend or customize the functionality of a client by the transfer (download) of executable code into the client application’s runtime environment. Examples of this may include compiled components such as [Java Applets](#), Semantic Type objects, Event Prototype definitions used by the Service Application Engine™ or client-side scripts such as [Java Script](#).

Uniform Interface

A uniform interface between clients and servers simplifies and decouples the architecture, enabling each component to evolve independently. The guiding principles of a REST interface are detailed below. While there is no format specification for an interface definition in REST–based systems it is expected that data exchange between participants occurs using structured data elements that are self-describing over the HTTP protocol or one of its secure variants. Complying with the so-called REST Principles and architectural style, enables a variety of distributed hypermedia systems with desirable [emergent properties](#), such as performance, scalability, simplicity, modifiability, visibility, portability and reliability.

REST Interface Principles

A key concept in RESTful systems is the existence of [resources](#) (sources of specific information), each of which is referenced with a global identifier (such as a [URI](#) in HTTP). To manipulate these resources, *components* of the REST system communicate via a standardized protocol (such as HTTP) and exchange *representations* of resources (potentially documents or objects conveying information). For example, a resource that represents a Purchase Order may accept a representation that specifies a *list of items to purchase and billing information*, and return a Purchase Conformation or Receipt formatted as a [PDF](#) document or a URL pointing to the resource.

As such, a REST client application can interact with a resource by knowing only two things: the identifier of the resource and the action required—it does not need to know whether there are caches, proxies, gateways, firewalls, tunnels, or other intermediaries between it and the resource server delivering the information. However, the application does need to understand the format of the information (*representation*) returned, which is typically an [HTML](#), [XML](#) or [JSON](#) document, although it may be image, plain text, or any other content. The uniform interface that any REST resource provides is considered fundamental to design of any REST service.

Identification of Resources

Resources are identified in requests, for example using [URIs](#) in web-based REST systems. The resources and their location are conceptually separate from the representations that are returned to the client. For example, a data server may return [HTML](#), [XML](#) or [JSON](#) that represents a set of information requested by the user application and may return such results in different formats or languages based on request URL.

Representation-Driven Data Modifications

A client holding a representation of a resource, including any [metadata](#) attached, should have enough information to modify or delete the resource on the server, provided it has permission to do so. Depending on the type of data being modified, resource representation may vary. For example a `Key-Value` data collection such as a `MAP` may simply require a `Key` in order to modify or delete the resource.

The Service Application Engine™ provides a variety of representational state objects including a `Query` object for working with SQL Queries, `Row` and `Row Set` objects.

Self-Descriptive Data Exchange

Exchange of information between REST system participants should occur using self-describing data. Each datum should include enough information to describe how to process the message. For example, a data element may include information about which parser to invoke and may specify an [Internet Media Type](#) (previously known as a [MIME](#) type). Data elements may also explicitly indicate their cache-ability to indicate whether their content is ‘sticky’ and will be retained by the client or the intermediary.

For example the service engine uses Event Datagrams as a mechanism for structured data exchange and allows the users to declare data content as *durable* instructing the engine intermediary to cache data produced by event publishers or create *data collections* that explicitly stored data in memory.

[Hypermedia as the Engine of Application State](#)

In a REST –based system client applications make state transitions as a result of actions represented as hypermedia to the resource server (ie. by [hyperlinks](#) within [hypertext](#)). The client application is aware of general application access points (as specified by the URI), but does not otherwise assume that any particular actions or operation will be available for a given resource beyond those described in representations previously received from the server. In other words, the state of an application is the sum total of data received thru representation objects.

While this style of application design may be less applicable to traditional client applications that make use of application-local data, it is ideal for composite application design, self –provisioning portals and other systems wherein interaction occurs thru discovery of resources and navigation between resource servers.

RESTful Web Services and API

SOA Services implemented using HTTP and the principles of REST are referred to as RESTful Web Services. This is also referred to as a RESTful [Web API](#); and provides a structured programming interface for exposing complete business functionality to application developers in a controlled, cost-effective fashion. REST –based Web Services allow users to access a collection of resources in a structured fashion, using the following guidelines:

- Server *resources* must be addressable by a base URI, such as `http://example.com/resources/`
- Resource data must be represented via some [Internet Media Type](#) supported by the Web Service. This is often [JSON](#), [XML](#) or [YAML](#) but can be any other valid media type including binary (`Base64` encoded) type.
- Command verbs, modifiers and data must be presentable as URL content and contain a complete set of information that allows a client application to query, modify or delete REST resources.
- Web Service operations typically map to [HTTP methods](#) (e.g., GET, PUT, POST, or DELETE).
- The API must be Hypertext –driven, allowing HTTP and Browser applications to access resources .

Using the REST API

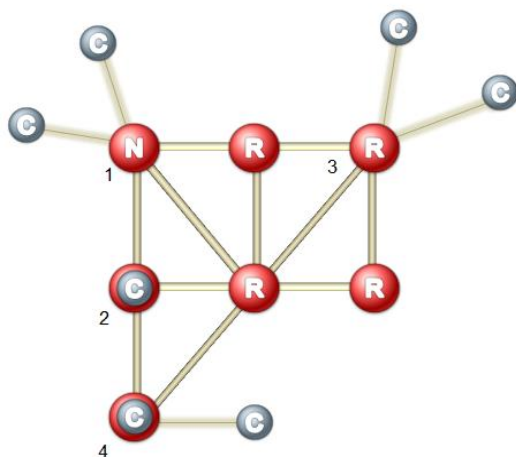
The Service Application Engine™ provides full support for REST –based Web Services and allows users to expose all *application fabric components* using a RESTful Web API. Web Service developers have access to the following resources that may be invoked thru a REST API:

- Java classes deployed as Application Fabric Services
- Event Fabric™ facilities for raising and consuming Events
- Application Data Spaces™ for query and modification of Data Collections
- Back-end (pass-thru) resources such as Databases, FTP and legacy systems
- Files and Unstructured Data resources accessible via Fabric Services
- Micro-flows for automating Business Processes composed from Fabric Services
- User Profiles and Identity information

Accessing Service Engine Resources

By default an application engine disables HTTP access to its resources. This is done to limit the number of initial Access Points into an application fabric *SYSPLEX* and simplify configuration. Core topology of the overlay network formed by service engine nodes is configured thru the Directory Table that is managed by an Exchange Discovery Manager. The *SYSPLEX* establishes connections between nodes using the TruLink Protocol, potentially allowing all fabric participants to see every resource within the application domain.

Specific resource visibility is dependent on the resource component's event scope, providing a way to secure and limit access to application fabric resources. However the inherent nature of service engine communications means that any resource may be accessed from an HTTP client on any node. In the diagram below participants labeled **N** are fabric nodes that do not host resources, **C** represents *client components* such as HTTP clients. Participants labeled **R** represent resources such as *service methods* that may be invoked, *data collections* that may be queried or modified and *events* that may be raised or consumed. All topology combinations described below are possible:



¹ *Access by Proxy* allowing a client to connect to a proxy node and access remote resources.

² *Embedded Client* access allows clients to access resources by opening an Accessor to a Service or Data Space, or to raise Events.

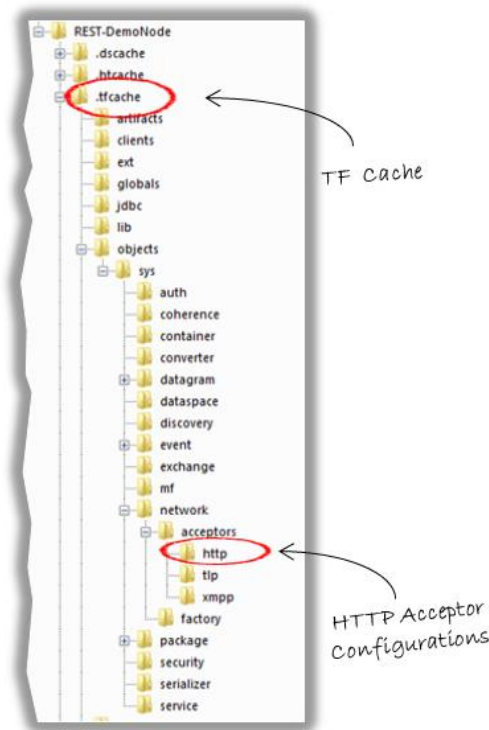
³ *Direct Access* allowing a client to connect to a fabric node and access its Services and Data Spaces or to raise Events.

⁴ *Access by Service Proxy* allows users to create Service resources that are themselves proxy clients to other resources, accessible from other client participants.

Configuring HTTP Acceptors

An acceptor is a socket listener that supports a specific network protocol. Protocol acceptors are configured by editing their XML Data Object (XDO) files located in the fabric cache directory (`.tfcache`). The object type of the data object is always `HTTPAcceptor` and the acceptor's name must be the *instance name*. In the example below the name of the acceptor file resolves to `HTTPAcceptor.Default.xdo`

Acceptors may be auto-started by the engine or they may be controlled manually thru the SLANG language environment. Like any other repository artifact an acceptor configuration object is locked by the runtime to prevent unauthorized editing. An acceptor that is defined with a host name of `localhost` will not be accessible by remote clients. Also note that on certain platforms such as Windows, new port allocations may be automatically blocked by Firewall software.



In addition to standard set of information regarding host and port names the HTTP acceptor supports the `anonymousRegistration` parameter that indicates whether or not the acceptor supports ability of users to self-register with the fabric; a setting that is potentially unsecure.

The XML snippet below shows a typical HTTP acceptor configuration. It is incomplete and does not include security information, `REALM` definitions or URL path mappings. It is provided for sample purposes and should not be used as an actual acceptor configuration object:

```
<?xml version="1.0"?>
<HTTPAcceptor>
  <configuration>
    <name>Default</name>
    <description>Default HTTP acceptor</description>
    <isAutoStart>true</isAutoStart>
    <host>localhost</host>
    <port>8099</port>
    <anonymousRegistration>true</anonymousRegistration>
    <logEachRequest>false</logEachRequest>
```

```

<logUserAgent>false</logUserAgent>
<logReferer>false</logReferer>
<compressResponse>false</compressResponse>
<storeSessions>false</storeSessions>
<keepAlive>true</keepAlive>
<requestLogFormat>{0}:{9} {1} {2} &quot;{4} {5} {6}&quot; {7,number,#}
{8,number} {10} {11}</requestLogFormat>
<webApplicationDir>webapps</webApplicationDir>
<webArchiveDir>war</webArchiveDir>
<backlogSize>50</backlogSize>
<keepAliveTimeout>15</keepAliveTimeout>
<sessionTimeout>5</sessionTimeout>
<checkDeployPeriod>-1</checkDeployPeriod>
<maxActiveSessionsNumber>-1</maxActiveSessionsNumber>
<maxRequestsPerConnection>100</maxRequestsPerConnection>
<maxThreadsInPool>100</maxThreadsInPool>
<authenticationType>BASIC</authenticationType>
<sessionAuthentication>false</sessionAuthentication>
..

```

The table below outlines HTTP Acceptor property settings. Not all settings are currently in use:

| Property Name | Description |
|--------------------------|---|
| name | Distinguished name of an HTTP acceptor |
| description | Description of the acceptor |
| isAutoStart | When TRUE the acceptor is automatically started and bound by the runtime |
| host | Name of the network interface (localhost by default) |
| port | Port number of HTTP listener |
| anonymousRegistration | When TRUE allows new users to dynamically register their ID and Password |
| logEachRequest | When TRUE the acceptor trace file logs all HTTP request |
| logUserAgent | When TRUE the acceptor trace file logs all User Agent requests |
| logReferer | When TRUE the acceptor trace file logs all Domain Referrals |
| compressResponse | When TRUE replies are compressed using GZIP compression |
| storeSessions | When TRUE the acceptor stores HTTP Session information |
| keepAlive | When TRUE then Keepalive checks are performed |
| requestLogFormat | Specifies the request log format using positional notation |
| webApplicationDir | Specifies application directory for this acceptor in .htcache |
| webArchiveDir | Specifies WAR file pickup directory for this acceptor in .htcache |
| backlogSize | Specified connection back-log size (currently unused) |
| keepAliveTimeout | When Keepalive is used specifies the length of check timeout |
| sessionTimeout | Specifies in minutes how long an inactive Session may remain |
| checkDeployPeriod | Specifies how often to check the WAR deployment directory for new content |
| maxActiveSessionsNumber | Specifies how many HTTP Sessions may be active concurrently |
| maxRequestsPerConnection | Specifies maximum concurrent requests per user Connection |
| maxThreadsInPool | Sets the maximum number of HTTP acceptor threads in pool |
| authenticationType | Indicates BASIC or DIGEST authentication type (see Digest Authentication) |
| sessionAuthentication | Enables or disables x-session-token support (see Delegated Authorization) |

Base Resource URI

All application fabric *resource types* are accessible thru a set of *base resource URI* that are present on every node that has an HTTP acceptor. However as noted previously the actual resources do not have to be hosted on the specific node that an HTTP client has connected to. The following resource URI are provided:

| Resource | Base URI |
|-----------------------|--|
| Service Access | http://< Host >[:< Port >]/service/invoke |
| Event Dispatcher | http://< Host >[:< Port >]/exchange/< Operand > |
| Service Configuration | http://< Host >[:< Port >]/sor/service/< Service Name >/ |
| Node Repository | http://< Host >[:< Port >]/repository |

Base URI are associated internally with Servlet entities that handle individual requests as specified in the acceptor configuration object. Users should not change or alter default implementations. The REST API is session –based. Within the application engine user authorization in the HTTP header is validated against a Client Context that is created on behalf of an individual user and used to control the HTTP Session.

Request and Response Objects

RESTful API clients exchange information with the application fabric by using request objects and response *object representations*. Objects are defined and modeled using the standard Structured Data Object interface. They are stored by the engine as standard Java classes. Users may model any arbitrary object and use it for data exchange.

The service engine allows use of both Objects and Event Datagrams (that wrap objects) for data exchange. The former is useful when the RESTful API is used for *remoting* (remote service method invocation); the latter is more applicable to situations where users want to *raise or consume events* that have been raised by other application fabric components. This allows REST –based applications to seamlessly interoperate with *event-driven* systems.

The data *representational format* may be XML or JSON as required by the application. JSON responses are typically returned in *condensed* JSON format which is a single text string with all non-essential space characters removed. It is expected that such data are used to initialize application-side JSON objects. Results may also be compressed using GZIP compression to reduce the size of a response object. This option is set at the level of the HTTP acceptor and applies to all responses. It is up to the client application to handle compressed data properly.

The API provides a general set of verbs that may be used to specify request and response data formats. The `&requestFormat` verb specifies request format and `&responseFormat` verb specifies the reply format. Object serialization and deserialization is carried out by the engine's Object Mediation Framework. For XML data use of *namespace identifiers* is currently not allowed and generally discouraged.

Semantic References

All data structures used by the application fabric are expressed as Semantic Data Types (or simply *semantic types*). In StreamScape parlance a semantic type is a data structure that is (i) *defined by a short name* (ii) *resolves to a real Java class* and (iii) *includes meta-data pointing to an ancestor type*. Ancestors and semantic names do not need to be structurally related. The Object Mediation Framework allows for registering data structure aliases and arbitrary ancestor names, allowing users to define *application-specific data ontologies* that may be queried via the *Repository API*. Semantic Types are automatically synchronized between all SYSPLEX members.

The RESTful interface provides access to data object definition, event handler declarations and configuration objects of services that implement such handlers. This allows API users to query service meta-data, obtain template representations of request and response objects and discover relationships between semantic types.

Access to such information is achieved by using the Semantic Object Reference URL of the appropriate services. This set of links provides a read-only interface into the global repository that is best accessed via a browser client because results are returned as HTML content. The interface currently supports basic service repository browsing and will be expanded in future versions. For more information on Semantic Types see general documentation.

Application Engine Clients

The service engine's HTTP interface provides a full-capability Web Server, complete with an integrated, realm-based security mechanism, ability to serve web pages, complete applications packaged as WAR files and support for popular Web programming tools such as JQuery, AJAX libraries and JavaScript. Web clients may develop composite applications that make use of the RESTful API, Java Script client or use the engine to develop mash-up systems that aggregate data from multiple back-end resources.

The REST API may be accessed directly from Web Browsers and Browser-resident tools for API authoring and testing. HTTP clients such as those from Apache, Jersey Client API for Java, .Net framework and all popular micro-browsers for mobile devices are currently supported without restriction. For cross-domain browser requests user may implement standard techniques such as embedded I-Frame or use the engine-supplied HTTP client utility classes, downloaded to the browser automatically as *server-side includes*.



Note

It should be noted that often, a better alternative to cumbersome browser-based mash-ups or server-centric UI frameworks such as Struts is use of the Service Application Engine™ as a Web Intermediary.

The service engine provides a single, secure entry point into a system, yet allows users to aggregate data from many sources using Service Calls, Event Processing or Application Data Spaces™. Resources may be accessed in a location-transparent manner without concern for cross-domain browser issues. Visual aspects of a Web application can be truly separated from back-end processing without sacrificing functionality or control.

Security Considerations

The service engine implements standard HTTP Security by allowing developers to provide user and credential information in the HTTP header of client requests. Often this is an acceptable solution as in most cases only the login credentials need to be protected. Acquiring a costly SSL server certificate from a certificate authority and writing the web app to ensure it only uses it for authentication is overkill. Even with an SSL infrastructure in place, many web applications store usernames and passwords in a database in clear text. In such a case, an attacker compromising the server has access to all user authentication credentials.

REST –based interfaces are particularly vulnerable because by default a REST application is stateless and does not handle session-based authentication. In contrast to default behaviors the Service Application Engine™ provides a flexible and extensible security framework that can protect a system's access points and provide an independent authorization layer for back-end resources such as Files, Databases and Web Services.

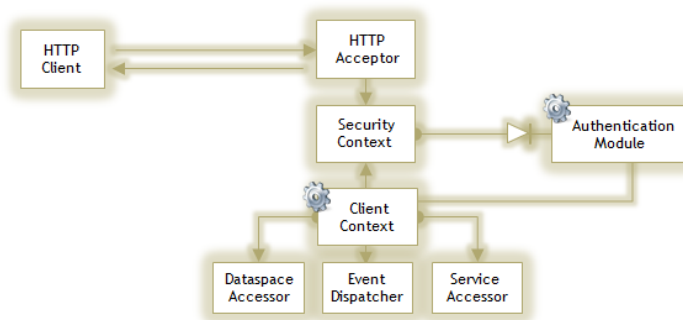
Authentication Framework

The application fabric provides a comprehensive framework for handling security and user authentication that focuses on user credential protection and active policy enforcement. Authentication extends to the HTTP protocol as well as StreamScape TLP, XMPP and the XMPP –over –HTTP variants supported by the service engine.

Credential information is *never stored as open text*. Each node has a local copy of the security database which stores credential information in encrypted format. Security information of a node is strictly overlaid when the node joins an active SYSPLEX, preventing injection of users and roles into an existing system. Changes in security status are replicated across the SYSPLEX and have a *CASCADE* effect on permissions and security context. Users are mapped to *GROUPS* and *REALMS* independently. Group entities are simply a way to organize users.

REALM authority applies to base URI or URL used to protect Fabric Services or specific resources. For example by including the base URI `http://< Host >[:< Port >]/exchange/` in a *REALM* mapping of an HTTP acceptor and then excluding specific *users* and *groups* from the authorization list it is possible to disable access to the fabric's Event Dispatcher for such users.

All participant components in a SYSPLEX have a so-called Security Context. This is a set of security credentials that identify a component to the application engine. The security context is particularly meaningful for clients that connect to the application engine. A client context is associated with its security context that is checked for permissions whenever engine resources are accessed. This is especially important when clients attempt to access services or elements in a data space. Internally the client context typically creates a resource Accessor for every resource it intends to use. Accessors are light-weight fabric constructs (similar to semaphores or session objects) that manage access to fabric resources; they work together with the security context to enforce security policies. In general, the following diagram illustrates all relevant components of an HTTP client interaction with the engine:



When an HTTP Client establishes a connection for the first time it passes security information in the HTTP header and establishes a Client Context in the application engine. The client's credentials are then verified by the engine's Authentication Module. If the client passes authentication a *security context* is established and a fully functional Client Context is then created. The Web client is passed back a valid response with HTTP header information that must be then re-used by the client application in order to maintain a *virtual* HTTP session. Listing components in the engine will display the names and types of client participants.

```
REST-DemoNode>list components
```

| Type | Name | Description | Model | User |
|-----------------|-----------------------|-----------------|---------------|----------|
| .. DemoService | Default | | Service | .. |
| .. Client_TLP | Client3_REST-DemoNode | TLP Connection | Local Client | .. Admin |
| .. TSPACE | RESTful | Table Space | Dataspace | |
| .. Client_REST | Client1_REST-DemoNode | REST Connection | Local Client | .. Admin |
| .. FSPACE | RestFS | File Space | Dataspace | |
| .. Client_SLANG | Client1_REST-DemoNode | TLP Connection | Remote Client | .. Admin |

An HTTP *client context* is lease –based, meaning that it will exist for some time and will eventually be closed by the engine if no further activity on that connection occurs. This is done to optimize resource use and prevent certain intrusion attacks on the Web Server. Each time the client sends a REST request to the engine the HTTP header information is compared to that of the existing *client context* for the user. If there is a match, the *lease* is extended and no further authentication occurs. If there is no match a new context is created and the user is authenticated.

again. To prevent denial of service and flood attacks, the number of *concurrent connections* may be limited by Acceptor configuration. Enabling `INFO` trace for `com.streamscape.sef.*` will show the following in the error log, confirming the described behavior:

```
.. Component 'REST-DemoNode://Client_SLANG.Client1_REST-DemoNode' added.
.. Remote client 'REST-DemoNode://Client_SLANG.Client1_REST-DemoNode'
   [127.0.0.1:59833] connected.
.. Component 'REST-DemoNode://Client_REST.Client1_REST-DemoNode' removed.
.. Fabric connection 'Client_REST.Client1_REST-DemoNode' closed.
```

It should be noted that a Client Context is established regardless of the HTTP connection mode that is used. Context objects represent active user sessions and may be used to *view* and *analyze client interactions* with the Web Server and its requested resources, allowing developers to track mouse clicks, and other HTTP requests.

The Authentication Module is a pluggable component that handles *client identity validation*. It is configured in the engine's Deployment Descriptor. Developers may create their own modules to extend the application fabric's security model as needed. The default module validates users against the engine's security database. It has been tested to support hundreds of thousands of users without significant performance impact. Users developing their own authentication modules should consider performance as part of their design. See general documentation for more information on authentication Module development and security.

Basic Authentication

In the context of an HTTP interface, *basic access authentication* is a method for a Web Browser or other client program to provide credentials such as *user name* and *password* to the Web Server when making a request. Basic authentication does not make attempts to transmit credentials in a secure fashion. It implies that credentials are included with every HTTP request.

Before a request is transmitted, the *user name* is appended with a *colon* and with a *password*. The resulting string is then encoded using a [Base64](#) algorithm. For example, given the user name `Bob` and password `MySecret`, the string `Bob:MySecret` is Base64 encoded, resulting in a string similar to `QWxhZGRpbjpvcGVuIHNlc2FtZQ==`. The Base64-encoded string is transmitted in the [HTTP header](#) and decoded by the server, resulting in a colon-separated *user name* and *password* string. The service engine then performs standard *authentication* for a Client Context.

While encoding the *user name* and *password* with the Base64 algorithm makes them unreadable to the unaided eye, they are trivially decoded by software and as such, are susceptible to traffic sniffers and Man-in-the-Middle attacks. Confidentiality is *not* the intent of the encoding step. HTTP in general does not provide such guarantees (see [HTTPS](#)). Rather, the intent is to encode characters not compatible with HTTP that may be in the *user name* or *password* into those that are HTTP-compatible.

Basic authentication does implement a challenge-response exchange in Web Browser applications. If credentials are not specified, the service engine will reply with a typical HTTP 401 `Unauthorized` response to the sender. This will allow the client application to automatically retry the transmission once credentials have been supplied. Most browsers react by displaying a pop-up window for entering User and Password credentials and automatically resubmitting the request. Basic authentication is useful for testing and in situations where extra-net security is not a prevailing issue.

To enable BASIC Authentication a runtime acceptor's `authenticationType` parameter must be set to `BASIC`. The samples directory provides additional examples of JavaScript –based *basic authentication*.

Java Example:

```
import com.streamscape.lib.http.*;
..
public static final int    DEFAULT_TIMEOUT = 30000;
public static final String SERVICE_REALM   = "Service";
..
HTTPConnection connection = null;
HTTPResponse  response  = null;
String        requestURL = null;
..
// Build a request URL here
..
connection = new HTTPConnection("localhost", 8099);
connection.setTimeout(DEFAULT_TIMEOUT);
connection.setAllowUserInteraction(false);
connection.addBasicAuthorization(SERVICE_REALM, "Admin", "admin");
..
response = httpConnection.Get(requestURL);
```

In the sample code above a StreamScape –supplied HTTP client is used to construct the URL for issuing an HTTP request. Base64 encoding occurs *implicitly*. Note the use of a `REALM` identifier to specify the security domain.

Digest Authentication

HTTP [digest authentication](#) is a little-known standard built into *all* major web servers and web browsers. With it, login credentials are not directly transmitted across the network, nor are they stored in the engine as plain text. There is no need to purchase an expensive SSL host certificate each year, nor worry about CPU load due to cryptographic processing. Also, the service engine performs *authentication*, removing the need to develop such code in the web application itself. An application only has to verify that *authentication* has taken place.

Digest authentication is not the same as [Basic Authentication](#). Web servers and browsers have always supported *basic authentication* and many servers (including the service engine) implicitly support digest authentication as well. *Digest authentication* looks similar from the user's perspective. The browser pops up a dialog box asking for User and Password credentials. However, under the hood the protocol uses hashes instead of encoded text.

How Digest Authentication Works

Digest authentication is a challenge-response mechanism that performs implicit negotiation of credentials:

- A browser or client application sends an HTTP request (e.g. a GET) to the service engine
- The engine sees the URL being accessed has been configured to use Digest Authentication and replies with a 401 `Authentication Required` status plus a `nonce`, which is a unique hash of several data items, one of which is a secret *key* known only to the engine. Validation and nonce generation are handled by the Authentication Module and may be customized by a user –defined module.
- The browser pops up a dialog box requesting username and password. A client application may respond by supplying user credentials. Once the credential information is supplied, an MD5 hash of the username, password, `nonce` and URL are computed and the client re-sends the original request along with the hash.
- The service engine then compares that hash with it's own computation of the same values. If they match, the original HTTP request is allowed to complete.

Since hashes are used instead of the actual data, an eavesdropper intercepting the communications never sees the user name and password. *Digest authentication* is appropriate in situations where secure transmission of user

credentials is needed. It is a good, light –weight mechanism for secure authentication that is supported by all major browsers. Client application change should be fairly transparent as indicated by example below. A variety of browser forms, Java Script and tool kits such as DigestJ also provide support for *digest authentication*.

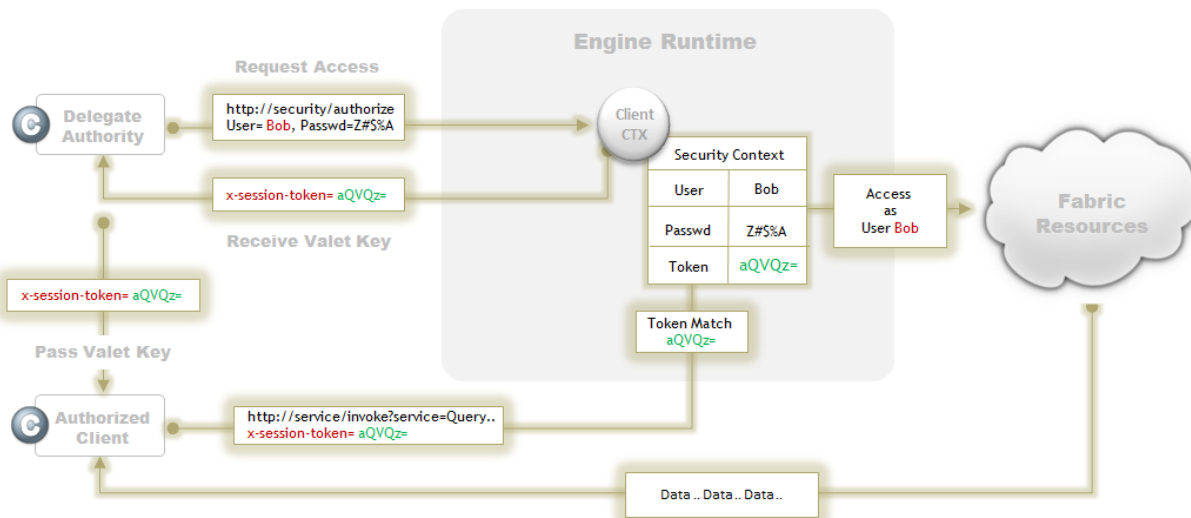
```
..
connection.addDigestAuthorization (SERVICE_REALM, "Admin", "admin");
..
```

To enable DIGEST Authentication a runtime acceptor's `authenticationType` parameter must be set to `DIGEST`. The samples directory provides additional examples of JavaScript –based *digest authentication*.

Delegated Authorization

The service application engine supports session –level authorization tokens, also referred to as a Delegate or Valet Key authorization. Delegated authorization is the granting of *access* to another person or application in order to perform actions on your behalf.

By way of example, when you drive your car to a classy hotel, they may offer valet parking. You then authorize the valet attendant to drive your car by handing them the key in order to let them perform actions on your behalf. Delegated authorization works the same way. A user grants access to application fabric resources in order to have other applications perform actions on their behalf by handing them a *Session Token*, also referred to as a *Valet Key*. Such applications can only perform the authorized actions allowed by the associate Security Context.



The diagram above illustrates the relationship between a Delegate Authority, a Valet Key (session token) and the client application that is performing a service invocation. An authorization delegate may be an application that self –authorizes, or may be a different node that assumes the role of an Identity Manager. The application engine stores user profiles in a vCARD –compliant structure that may be used to identify and authenticate the delegate. In the current version application fabric clients may only authenticate against the application engine, although the engine may further authenticate against any 3rd party application. The delegated authorization model is used by REST –based applications, allowing them to use a single authorization technology across hundreds of APIs on the Web. It is compatible with similar Web standards such as Open ID and the OAuth specification.

Delegate authorization is enabled on a runtime acceptor by setting the `sessionAuthentication` parameter to `TRUE`. It may be used in combination with `BASIC` or `DIGEST` authentication. Once authorization is enabled clients may authenticate by using the `http://<host:port>/security/authorize` URL. The engine returns a Valet Key that can then be passed as part of the HTTP header using the `x-session-token` property. Session tokens

allow authorized clients to access protected resources using permissions associated with the delegate Security Context. Sessions can be configured to expire using the `sessionTimeout` parameter in the HTTP acceptor, providing additional security. Session timeout follows a leased resource model. Each time a session is used its timeout lease is extended. Session tokens may also be invalidated immediately by using the `http://<host:port>/security/unauthorize` URL.

SSL Encryption

Secure Sockets Layer (SSL) is the only choice to protect web data in transit and simultaneously verify the identity of the server. However, SSL is often used when the only data that must be protected are authentication credentials such as *user name* and *password*, leading to excessive processing overhead and CPU consumption. Encryption routines of SSL seriously impact a web server when under load.

SSL is not enabled by default in the service engine or the HTTP client, however both may be configured to use a specific SSL provider to affect secure end-to-end communications over HTTP. Please contact StreamScape support at support@streamscape.com for additional information.

RESTful API Guide

The following section describes all supported interface calls and provides syntax and examples for accessing the application fabric's resources using REST –based clients. Where appropriate both XML and JSON format are used. Standard URL syntax separates the base URI from the *query string* with the ? symbol. Query parameters are presented as a subject and modifier verbs separated by the & symbol. For syntax clarity modifier verbs are presented as &< verb >, for example &data or &event. Default values are presented underlined>.

Authorize

```
http://< Host >[:< Port >]/security/authorize
```

An *authorize* request allows applications or external delegate systems to authenticate with the application fabric and request session tokens that maybe used as Valet Keys by trusted applications. This URL is only valid if the `sessionAuthentication` parameter is set to `TRUE`. The call takes standard authentication credentials that are passed in as HTTP header elements and may use `BASIC` or `DIGEST` authentication to pass credentials to the acceptor. The following header elements are suggested as a minimum:

| HTTP Header Element | Description |
|-----------------------|--|
| <code>url</code> | The URL that an HTTP client is connecting to. Must be <code>/security/authorize</code> . |
| <code>type</code> | The type of HTTP request. GET and POST are supported. |
| <code>timeout</code> | The time in milliseconds that a connection waits for a response on. |
| <code>username</code> | Security principal that may be encoded or encrypted based on authentication model. |
| <code>password</code> | Security credential. May be encoded or encrypted based on authentication model. |

When an authorization request is submitted the application fabric reacts by first checking the acceptor's `authenticationType` parameter. If this is set to `BASIC` the acceptor will decode the `username` and `password` and attempt an authentication by calling the node's Authentication Module. If successful, a Client Context is established and a session token is returned. If the acceptor's `authenticationType` parameter is set to `DIGEST` a standard challenge/response exchange occurs. The application fabric receives and decrypts the username and password and attempts authentication.

Note that the Authentication Module is a *system extension point*. The `Default` implementation will simply validate credentials against the fabric's security database. However users may create their own implementations of Authentication Module and pass additional information in the HTTP header such as those required by identity managers like OAuth and Open ID. This information can then be verified against the user's profile that is stored in the security database and follows the general vCARD format as outlined in IETF RFC 6350. As such, custom modules may verify identity as part of general authentication, before authorizing an Valet Key.

A failed authentication will return `401`, whereas if the feature is not enabled a `404` will be returned indicating that the URL is not valid.

Examples:

The following illustrates an authorization request using jQuery:

```
<script type="text/javascript" src="jquery/jquery-1.7.2.min.js"></script>

token:<input type="text" name="x-session-token" id="x-session-
token"  readonly="readonly" disabled="true" width="100"/><br>

..

<script>

$("#Authorize").click(function() {
    $.ajax({
        url      : "/security/authorize",
        type     : "GET",
        timeout  : "60000",
        username : "http",
        password : "http",
        success: function(data) {
            $("#x-session-token").val(data);
            $("#x-session-token").attr("disabled", false);
        },
        statusCode : {
            401: function() {
                alert("Authentication failed.");
            },
            404: function() {
                alert("Security authentication disabled.");
            }
        },
        error : function(XMLHttpRequest, error) {
            alert("Authorize request failed, status code: " +
XMLHttpRequest.status);
        }
    });
});

</script>
```

This request returns a session token such as: zQW3454500aaf=. This session token may then be passed into subsequent HTTP requests thru the use of the `x-session-token` element in the HTTP header. Session tokens are potentially temporary and will expire. When this occurs the user will be returned a 401 error and must then re-authorize.

Unauthorized

`http://< Host >[:< Port >]/security/unauthorize`

An *unauthorize* request invalidates a specific session token and removes the Client Context associated with the token from the fabric runtime. To issue the request a client must specify the `x-session-token` element in the HTTP header. The operation returns a 401 if the session token is not valid or expired. A 404 error is returned if security authentication is disabled.

| HTTP Header Element | Description |
|------------------------------|--|
| <code>url</code> | The URL that an HTTP client is connecting to. Must be <code>/security/authorize</code> . |
| <code>type</code> | The type of HTTP request. GET and POST are supported. |
| <code>timeout</code> | The time in milliseconds that a connection waits for a response on. |
| <code>x-session-token</code> | Session token to be invalidated. |

Examples:

The following illustrates how to revoke an authorization request using jQuery:

```
<script type="text/javascript" src="jquery/jquery-1.7.2.min.js"></script>

token:<input type="text" name="x-session-token" id="x-session-
token" readonly="readonly" disabled="true" width="100"/><br>

..

<script>
$("#Unauthorize").click(function() {
    $.ajax({
        url      : "/security/unauthorize",
        type     : "GET",
        timeout  : "60000",
        username : "http",
        password : "http",
        headers  : {"x-session-token" : $("#x-session-token").val()},
        success: function(data) {
            $("#x-session-token").val("");
            $("#x-session-token").attr("disabled", true);
        },
        statusCode : {
            401: function() {
                alert("Invalid session token.");
            },
            404: function() {
                alert("Security authentication disabled.");
            }
        },
        error : function(XMLHttpRequest, error) {
            alert("Unauthorize request failed, status code: " +
XMLHttpRequest.status);
        }
    });
});
</script>
```

Invoke Service

```
http://< Host>[:< Port > ]/service/invoke?service=< Service Name >
{
  &eventId=< EventId > | &eventHandler=< Event Handler Name > }
  [ &correlationId=< CorrelationId > ]
  [ &eventGroup=< Event Group > ]
  [ &eventKey=< Event Key > ]
  [ &durable={ true | false } ]
  [ &eventProperties=< Property1=Value1; ... PropertyN=ValueN > ]
  [ &requestFormat={ XML | JSON } ]
  [ &responseFormat={ XML | JSON } ]
{
  &data={ < String Value > | < JSON Object > | < XML Object > } |
  &event={ < JSON Event Object > | < XML Event Object > }
}
```

A *service invoke* allows REST clients to directly call methods of a specified Application Fabric Service by referencing the Event Id or Event Handler associated with the service method. The base resource identifier (URI) followed by the subject *service* uniquely identifies a REST resource and host. The complete URL specifies a reference to the service method and additional options that allow users to specify identity, format and content of the request.



Note

All service calls may be invoked using GET or POST. The REST Service API does not currently support a specific mapping to standard HTTP protocol verbs such as GET, PUT, POST or DELETE. Each service class may expose several methods and therefore does not map directly to the HTTP verbs. Future versions are intended to present mappings that allow users to overload method invocations to elicit specific behavior.

Invoking a service follows the standard Web Server communication pattern. Clients may choose how they access a service based on the type of Semantic Type that a service *event handler* supports. Methods that are implemented to accept Structured Data Objects (classes) will require users to declare the *&eventHandler* verb in order to identify the event handler that accepts the object. When this option is specified the user must also declare a *&data* verb followed by an object *representation* in XML, JSON or STRING format.

Declaring an *&eventHandler* verb may also be used with methods that accept Event Datagram objects. Various verb combinations may be used to provide maximum flexibility for application developers, allowing them to supply *representations* as data objects, datagram objects or dynamically construct Event Datagrams on the fly.

The URL syntax allows users to declare request formats as JSON or XML. If neither option is specified the request *representation* defaults to STRING. A response may also be declared as JSON or XML and defaults to XML. When REST requests are specified using the *&data* verb users will need to specify object *representations* using the appropriate format based on their *semantic type*. When requests are specified using the *&event* verb the user must supply a complete, well-formed Event Datagram object including header information using XML or JSON.

Note that for JSON objects the primitive types are limited to STRING and NUMERIC as dictated by Java Script and the ECMA standards. As such text strings do not require explicit type declaration unless presented as part of a datagram *representation*.

The following verbs are supported by the Service Invoke operation:

| URL Element | Description |
|--------------|---|
| Host | HTTP host name of the node that a client is connecting to. |
| Port | HTTP port of the node (80 by default). |
| Service Name | Name of the service running on this node in <Type>.<Instance> format. |

| | |
|----------------|---|
| eventId | EventId of the <i>event</i> to be passed to the <i>service</i> . Only valid if <i>service event handler</i> accepts an Event Datagram as a parameter. Otherwise an <i>exception</i> is raised. If this option is set and an <i>event</i> option is provided the properties of the <i>event</i> object override any URL specified properties. This option is mutually exclusive to the <i>eventHandler</i> option. |
| durable | If an <i>event</i> is being <i>dynamically created</i> (by specifying <code>&eventId</code> and <code>&data</code> elements), this option specifies whether or not the new <i>event</i> is to be <i>durable</i> or not. Durable <i>events</i> are cached if an Event Cache is defined for this Event Id. |
| correlationId | If an <i>event</i> is being <i>dynamically created</i> (by specifying <code>&eventId</code> and <code>&data</code> properties), sets the Correlation Id of the <i>event</i> . |
| eventGroup | If an <i>event</i> is being <i>dynamically created</i> (by specifying <code>&eventId</code> and <code>&data</code> properties), sets the EventGroup for the <i>event</i> . This property is part of Event Identity Management. |
| eventKey | If an <i>event</i> is being <i>dynamically created</i> (by specifying <code>&eventId</code> and <code>&data</code> properties), specifies the EventKey for the <i>event</i> . This property is part of Event Identity Management. |
| property | If an <i>event</i> is being <i>dynamically created</i> (by specifying <code>&eventId</code> and <code>&data</code> properties), specifies one or more user defined properties for the <i>event</i> . |
| eventHandler | Name of the Event Handler to invoke. This option is mutually exclusive to the <code>&eventId</code> option and valid for all cases, allowing users to specify the handler regardless of the type of object it accepts. The type is specified by either <code>&data</code> or <code>&event</code> content. |
| event | Specifies an <i>event object representation</i> in XML or JSON format to be passed to the <i>service</i> . If <code>&event</code> is specified a <i>service event handler</i> must accept an Event Datagram as parameter, else an <i>exception</i> is raised. If an <code>&eventId</code> is specified in the URL, the object <code>&eventId</code> takes precedence. If this option is set, adding <code>&data</code> raises an exception. |
| data | Specifies a <i>data object representation</i> in String, XML or JSON format to be passed to the <i>service event handler</i> . The Semantic Type of the data element must match that of the event handler, otherwise an <i>exception</i> is raised. |
| requestFormat | Specifies whether the format of a request is XML or JSON. This option controls how the engine interprets the <code>&data</code> or <code>&event</code> parameters being passed to the <i>service</i> . If this option is omitted the <code>&data</code> contents are interpreted as a <code>STRING</code> . |
| responseFormat | Specifies whether the format of a response is XML or JSON. This option controls how the engine returns the representation resulting object from a <i>services</i> execution. This includes <i>exceptions</i> or other status messages returned by the <i>service</i> . |

Examples:

The following illustrates invoking a service called `Process.ListRequest` using a JSON object:

```
http://streamscape.com:8099/service/invoke?service=Process.ListRequest
&requestFormat=json&eventHandler=processList
&data={"SemanticType":"ProcessListRequest",
      "processType":"TestProcess","processNumber":"3"}
```

A simple ‘hello world’ service that accepts and returns a `STRING` may be invoked as:

```
http://localhost:8055/service/invoke?service=DemoService.Default&responseFormat=json
&eventHandler=helloWorld&data=Jimmy
```

and results in the following *representation* response when the response format is set to `JSON`:

```
{ "data" : "Hello, Jimmy" }
```

and the following *representation* response if set to XML:

```
<data SemanticType="string">Hello, Jimmy</data>
```

Note that syntax does not require `STRING` data to be presented as XML or JSON.

The following example invokes a service that accepts an event datagram. The response type is also specified and the resulting `EVENT` is marked *durable* so that it may be cached by the application fabric if there is an event cache defined. A complete set of event properties may be defined as part of the payload. See the documentation of the Service Engine regarding what properties may be set and their possible values. The URL is formatted for legibility:

```
http://localhost/service/invoke?service=EventService.Sample&requestFormat=xml
&responseFormat=xml
  &eventId=event.DemoService.request
  &correlationId=Test0001
  &durable=true
  &event=
<?xml version='1.0'?>
  <DataEvent>
    <serialVersionUID>9969000000302071</serialVersionUID>
    <timeStamp>0</timeStamp>
    <eventExpiration>0</eventExpiration>
    <dataProtected>Rw==</dataProtected>
    <acl>
      <ACL>GyUs</ACL>
    </acl>
    <coalesced>false</coalesced>
    <data SemanticType="string">Sample Data</data>
  </DataEvent>
```

The service is similar to our *hello world* example and returns the following *representation* response:

```
<?xml version='1.0'?>
  <Response>
    <data SemanticType="string">Sample Data</data>
  </ Response >
```

Note that when specifying *XML representations* the XML prolog `<?xml version='1.0'?>` is optional. For example a data object may be specified as `<data SemanticType="string">Hello, Jimmy</data>`. Responses always return well-formed XML however and will always contain a prolog.

Raise Event

```
http://< Host>[:< Port > ]/exchange/raiseEvent?eventId=< Event Id >
[ &corrleationId=< Correlation Id > ]
[ &eventGroup=< Event Group > ]
[ &eventKey=< Event Key > ]
[ &durable={ true | false } ]
[ &eventProperties=< Property1=Value1; ... PropertyN=ValueN > ]
[ &requestFormat={ XML | JSON } ]
[ &responseFormat={ XML | JSON } ]
{
  &data={ < String Value > | < JSON Object > | < XML Object > } |
  &event={ < JSON Event Object > | < XML Event Object > }
}
```

The *raise event* allows REST clients to raise events within the application fabric the same way that standard TLP clients and services may do. A full set of event types is supported (ie. `XMLEvent`, `DataEvent`, `BytesEvent`). The URL syntax allows users to specify event formats as `JSON` or `XML`. If neither option is specified the event *representation* defaults to `STRING` and the event is assumed to be a `TextEvent`. A prototype mismatch will raise an exception. The *raise event* operation is fire-and-forget and returns a response indicating operation success.

The following verbs are supported by the Raise Event operation:

| URL Element | Description |
|----------------|--|
| Host | HTTP host name of the node that a client is connecting to. |
| Port | HTTP port of the node (80 by default). |
| eventId | Event Id of the <i>event</i> to be raised. An <code>&eventId</code> is mandatory and will always take precedence over the properties of the <i>event</i> object. |
| durable | If an <i>event</i> is being <i>dynamically created</i> (by specifying <code>&eventId</code> and <code>&data</code> elements), this option specifies whether or not the new <i>event</i> is to be <i>durable</i> or not. Durable <i>events</i> are cached if an Event Cache is defined for this Event Id. |
| correlationId | If an <i>event</i> is being <i>dynamically created</i> (by specifying <code>&eventId</code> and <code>&data</code> properties), sets the Correlation Id of the <i>event</i> . |
| eventGroup | If an <i>event</i> is being <i>dynamically created</i> (by specifying <code>&eventId</code> and <code>&data</code> properties), sets the EventGroup for an <i>event</i> . The property is part of Event Identity Management. |
| eventKey | If an <i>event</i> is being <i>dynamically created</i> (by specifying <code>&eventId</code> and <code>&data</code> properties), sets the EventKey for the <i>event</i> . This property is part of Event Identity Management. |
| property | If an <i>event</i> is being <i>dynamically created</i> (by specifying <code>&eventId</code> and <code>&data</code> properties), specifies one or more user defined properties for the <i>event</i> . |
| event | Specifies an <i>event object representation</i> as XML or JSON. The <code>eventId</code> specified in the URL takes precedence. If this option is set, adding a <code>&data</code> option raises an exception. |
| data | Specifies a <i>Data Event representation</i> in STRING, XML or JSON format. Semantic Type of data must match that of the event prototype, else an <i>exception</i> is raised. When specified the event is implicitly created as a <code>DataEvent</code> . |
| requestFormat | Specifies whether the format of an <i>event</i> is XML or JSON. This option controls how the engine interprets the <code>&data</code> or <code>&event</code> parameters of the raise event. If this option is omitted the <code>&data</code> contents are interpreted as a <code>STRING</code> . |
| responseFormat | Specifies whether the format of a response is XML or JSON. This option controls how the engine returns the <i>representation</i> of a <i>raise event</i> confirmation or an <i>exception</i> . |

A *raise event* response may be declared as `JSON` or `XML` and defaults to `XML`. When REST events are specified using the `&data` verb users will need to specify object *representations* using the appropriate format based on their *semantic type*. If a *semantic type* does not match the *event prototype* an *exception* response is returned. Events described using the `&data` verb must be of `DataEvent` type or an *exception* response is returned. When events are specified using the `&event` verb the user must supply a complete, well-formed Event Datagram object including header information using `XML` or `JSON`.

Note that when specifying `XML representations` the XML prolog `<?xml version='1.0'?>` is optional. For example data may be specified as `<data SemanticType="string">Hello, Jimmy</data>` without the prolog when compared to a well-formed XML document such as:

```
<?xml version='1.0'?>
  <data SemanticType="string">
    Hello, Jimmy
  </data>
```

Examples:

An example of a Text Event being raised using `JSON representation`:

```
http://streamscape.com/exchange/raiseEvent?eventId=event.sample.Text
&correlationId=10583
&eventKey=Audit Text
&requestFormat=json
&data={"SemanticType":"Text", "text":"Hello world."}
```

An acknowledgement response is returned as `XML`:

```
<?xml version='1.0'?>
  <Void/>
```

Here is another example that raises a notification event with user-defined properties. Note that REST interactions with the dispatcher are the same as any other fabric client. Events that are raised must have a valid prototype defined. It is expected in the example below that a prototype for `event.notification` must exist and must also contain the properties `severity` and `date`:

```
http://streamscape.com/exchange/raiseEvent?eventId=event.notification
&eventProperties=severity=Critical;date=04 Jan 2011 00:57:13.274
&durable=true
&requestFormat=xml
&data=<data SemanticType="Text">System is active!</data>
```

Raise Request

```
http://< Host>[:< Port > ]/exchange/raiseRequest?eventId=< Event Id >
[ &corrleationId=< Correlation Id > ]
[ &eventGroup=< Event Group > ]
[ &eventKey=< Event Key > ]
[ &durable={ true | false } ]
[ &eventProperties=< Property1=Value1; ... PropertyN=ValueN > ]
[ &requestFormat={ XML | JSON } ]
[ &responseFormat={ XML | JSON } ]
[ &replyTo=< Event Id > ]
[ &timeout=< Milliseconds > ]
{
  &data={ < String Value > | < JSON Object > | < XML Object > } |
  &event={ < JSON Event Object > | < XML Event Object > }
}
```

The *raise request* allows REST clients to raise request events within the application fabric the same way that standard TLP clients, services and triggers may do. A full set of event types is supported (ie. `XMLEvent`, `DataEvent`, `BytesEvent`). A *raise request* operation is a blocking call. The client application blocks waiting for the service engine to respond with a proper HTTP reply. However the actual retrieval and building of a response object may be done in a variety of ways driven by the service logic of the application engine.

Requests are raised in an *asynchronous* fashion, as events with a specific event id. As such participant components that process *request events* may exist anywhere within the application fabric and handle incoming requests in a location-transparent fashion. To match requests to responses the request should supply a `&replyTo` value which is an *event id* that is used to identify the *response object*. When a REST request event is raised the Client Context proxies the request and dynamically creates a listener that waits for replies using the *event id* specified by the `&replyTo` verb.

The URL syntax allows users to specify *request event* formats as `JSON` or `XML`. If neither option is specified the event *representation* defaults to `STRING` and the event is assumed to be a `TextEvent`. A prototype mismatch will raise an exception.

Replies are any events that are raised using the `&replyTo` event id. As such any service, client application or data collection may respond to requests. The engine makes use of the so called *edge-format processing* technique, wherein all data objects used for communications are *semantic types* based on Java objects; however they can be presented as XML or JSON to client applications (at the edge). For REST –based requests the response format is specified using `&responseFormat`.

A *raise request* response may be declared as `JSON` or `XML` and defaults to `XML`. When REST events are specified using the `&data` verb users will need to specify object *representations* using the appropriate format based on their *semantic type*. If a *semantic type* does not match the *event prototype* an *exception* response is returned. Events described using the `&data` verb must be of `DataEvent` type or an *exception* response is returned. When events are specified using the `&event` verb the user must supply a complete, well-formed Event Datagram object including header information using XML or JSON.

Note that when specifying XML *representations* the XML prolog `<?xml version='1.0'?>` is optional. For example data may be specified as `<data SemanticType="string">Hello, Jimmy</data>` without the prolog. This is true for all XML representations.

Because requests are blocking operations and participants may not always reply to requests properly (especially during testing), the *raise request* operation allows users to specify a request timeout by using the `&timeout` verb. Timeout is specified in milliseconds. A request that times out returns an `HTTPClientException` indicating that the raising of a request failed. It should be noted that the timeout is superseded by HTTP acceptor settings such as `keepAliveTimeout`, `sessionTimeout` and `clientTimeout` which will cause long running, blocking requests to be aborted due to inactivity.

The following verbs are supported by the Raise Request operation:

| URL Element | Description |
|----------------|--|
| Host | HTTP host name of the node that a client is connecting to. |
| Port | HTTP port of the node (80 by default). |
| eventId | Event Id of the request <i>event</i> to be raised. An <code>&eventId</code> is mandatory and will always take precedence over the properties of an <i>event</i> object. |
| durable | If an <i>event</i> is being <i>dynamically created</i> (by specifying <code>&eventId</code> and <code>&data</code> elements), this option specifies whether or not the new <i>event</i> is to be <i>durable</i> or not. Durable <i>events</i> are cached if an Event Cache is defined for this Event Id. |
| correlationId | If an <i>event</i> is being <i>dynamically created</i> (by specifying <code>&eventId</code> and <code>&data</code> properties), sets the Correlation Id of the <i>event</i> . |
| eventGroup | If an <i>event</i> is being <i>dynamically created</i> (by specifying <code>&eventId</code> and <code>&data</code> properties), sets the EventGroup for an <i>event</i> . The property is part of Event Identity Management. |
| eventKey | If an <i>event</i> is being <i>dynamically created</i> (by specifying <code>&eventId</code> and <code>&data</code> properties), sets the EventKey for the <i>event</i> . This property is part of Event Identity Management. |
| property | If an <i>event</i> is being <i>dynamically created</i> (by specifying <code>&eventId</code> and <code>&data</code> properties), specifies one or more user defined properties for the <i>event</i> . |
| event | Specifies an <i>event object representation</i> as XML or JSON. The <code>eventId</code> specified in the URL takes precedence. If this option is set, adding a <code>&data</code> option raises an exception. |
| data | Specifies a <i>Data Event representation</i> in STRING, XML or JSON format. Semantic Type of data must match that of the event prototype, else an <i>exception</i> is raised. When specified the event is implicitly created as a DataEvent. |
| requestFormat | Specifies whether the format of an <i>event</i> is XML or JSON. This option controls how the engine interprets the <code>&data</code> or <code>&event</code> parameters of the raise event. If this option is omitted the <code>&data</code> contents are interpreted as a STRING. |
| responseFormat | Specifies whether the format of a response is XML or JSON. This option controls how the engine returns the <i>representation</i> of a <i>raise request</i> response or an <i>exception</i> . |
| replyTo | Specifies the <code>eventId</code> of the response. If an event supplied as <code>&event</code> representation contains a <code>replyTo</code> value it takes precedence. |
| timeout | Specifies how long a request remains outstanding before it is aborted by the framework. The timeout is specified in milliseconds. A setting of 0 indicates no timeout. Note that this setting may be superseded by HTTP acceptor settings. |

Examples:

An example of a request URL being constructed as an implicit DataEvent using JSON *representation*:

```
String request = "{\"SemanticType\":\"Employee\",\"name\":\"James Franco\",
    \"empId\":101223,\"action\":\"NEW_HIRE\",\"dept\":\"SYSOPS\",
    \"timestampValue\":{\"SemanticType\":\"sql-timestamp\",
    \"millis\":1338415300439}}";

String URL = "/exchange/raiseRequest?responseFormat=json&requestFormat=json
    &eventId=event.CRM.Employee&replyTo=event.CRM.reply
    &data=" + request + "&timeout=1000";
```

The reply of such a request could be an `AcknowledgementEvent` and would be matched using the *event id* specified by the `&replyTo` verb:

```
{
  "SemanticType" : "AcknowledgementEvent",
  "serialVersionUID" : 9969000000302071,
  "eventSource" : "AAAAAQAM",
  "eventId" : " event.CRM.reply",
  "durable" : false,
  "timeStamp" : 1338583100244,
  "eventExpiration" : 0,
  "dataProtected" : "Rw\u003d\u003d",
  "acl" :
    {
      "ACL" : "GyUs"
    },
  "coalesced" : true,
  "correlationEventTimeStamp" : 1338583100244,
  "correlationEventExpiration" : 0,
  "correlationEventSource" : "AAAAQAO",
  "correlationEventId" : "event.CRM.Employee",
  "onAcknowledgeAction" :
    {
      "SemanticType" : "AcknowledgeAction",
      "value" : "ACKNOWLEDGE"
    }
}
```

When a timeout or other error occurs a complete `ExceptionEvent` is returned including a stack trace. For example, here is an abridged XML *representation*:

```
<?xml version="1.0"?>
<HTTPClientException>
  <detailMessage>Raising of request &apos;event.http.test&apos;
  failed.</detailMessage>
  <cause SemanticType="ServletException">
    <detailMessage>Raising of request &apos;event.http.test&apos;
    failed.</detailMessage>
    <stackTrace>
      <trace>com.streamscape.sef.network.http.server.servlet.ExchangeServlet.proces
      sRaiseOperation(ExchangeServlet.java:454)</trace>
    ..
  </rootCause SemanticType="FabricEventSourceException">
    <detailMessage>Raising of request &apos;event.http.test&apos;
    failed.</detailMessage>
    <cause SemanticType="FabricEventSourceException">
      <detailMessage>Reply timeout expired.</detailMessage>
      <stackTrace>
    ..
      </stackTrace>
      <serialVersionUID>9969000000302071</serialVersionUID>
      <eventId>exception.fabric.EventSource</eventId>
      <errPrefix>SEF</errPrefix>
      <errorCode>6034</errorCode>
      <severity>GENERIC</severity>
      <durable>>false</durable>
      <timeStamp>0</timeStamp>
      <coalesced>>false</coalesced>
    </cause>
```

Receive Event

```
http://< Host>[:< Port > ]/exchange/receiveEvent?eventId=< Event Id >
[ &responseFormat={ XML | JSON } ]
[ &timeout=< Milliseconds > ]
```

The *receive event* operation allows REST clients to receive events from an event stream within the application fabric the same way that standard TLP clients, services and triggers may do. A full set of event types is supported (ie. `XMLEvent`, `DataEvent`, `BytesEvent`). A *receive event* operation is a blocking call. The client application blocks waiting for the service engine to respond with a proper HTTP reply.

A receive operation retrieves the most recent available event in the stream. Normally this type of stream sampling does not return an event because the likelihood of an event being available in the dispatcher at the exact moment a receive occurs is unlikely. However specifying a `&timeout` allows the *receive* to wait on arriving events. For the duration of a receive operation REST –based clients block. Browser applications may use standard AJAX programming techniques such as hidden i-frame, tools such as jQuery or application fabric Java Script utilities that wrap i-frame implementations to affect asynchronous receive operations.

Alternatively, events that have an *event cache* declared and raised as *durable* will be buffered by the event fabric and delivered to a REST –based receiver. Users may control cache depth and set it to buffer a significant amount of events allowing receivers to continuously poll the *event stream* without missing any events and preserving the delivery order.

The URL syntax allows users to specify response *event* formats as `JSON` or `XML` making use of the so called *edge-format processing* technique, wherein all data objects used for communications are *semantic types* based on Java objects; but may be presented as `XML` or `JSON` to client applications (at the edge). Response format is specified using `&responseFormat` verb.

Timeout is specified in milliseconds. A receive that times out returns a `Null` response and not an exception because strictly speaking a receive operation does not guarantee results and there may not be any events available in the stream during the operation’s cycle. This is not considered an error.

If a receive operation times out the following is returned as `JSON` format:

```
{ "SemanticType" : "Null" }
```

For `XML` format the following is returned:

```
<?xml version="1.0"?>
<Null/>
```

It should be noted that the timeout is superseded by HTTP acceptor settings such as `keepAliveTimeout`, `sessionTimeout` and `clientTimeout` which will cause long running, blocking requests to be aborted due to inactivity. Abortive behavior may result in exceptions being returned to the HTTP client.

Note that a *receive event* operation must be called with a timeout value and the operation always blocks for some period of time. Small timeout settings maybe combined with event cache to facilitate asynchronous event processing behavior using RESTful client interactions.

The following verbs are supported by the Receive Event operation:

| URL Element | Description |
|----------------|--|
| Host | HTTP host name of the node that a client is connecting to. |
| Port | HTTP port of the node (80 by default). |
| eventId | Event Id of the <i>event</i> to be received. |
| responseFormat | Specifies whether the format of a response is XML or JSON. This option controls how the engine returns the <i>representation</i> of a <i>receive event</i> or an <i>exception</i> . |
| timeout | Specifies how long a request remains outstanding before it is aborted by the framework. The timeout is specified in milliseconds. A setting of 0 indicates no timeout. Note that this setting may be superseded by HTTP acceptor settings. |

Examples:

An example of a request URL being constructed for receiving an event using *JSON representation*:

```
String URL = "/exchange/receiveEvent?responseFormat=json
&eventId=event.http.test
&timeout=5000";
```

The result of this request may be:

```
{
  "SemanticType" : "DataEvent",
  "serialVersionUID" : 9969000000302071,
  "eventSource" : "AAAAAQAM",
  "eventId" : "event.http.test",
  "durable" : false,
  "timeStamp" : 1338584733227,
  "eventExpiration" : 0,
  "dataProtected" : "Rw\u003d\u003d",
  "acl" :
  {
    "ACL" : "GyUs"
  },
  "coalesced" : true,
  "data" :
  {
    "SemanticType" : "SimpleHttpObject",
    "stringValue" : "Test Receive Operation",
    "longValue" : 654321,
    "timestampValue" :
    {
      "SemanticType" : "sql-timestamp",
      "millis" : 1338584732727
    }
  }
}
```

Receive Event with NoWait

```
http://< Host>[:< Port > ]/exchange/receiveEventNoWait?eventId=< Event Id >
[ &responseFormat={ XML | JSON } ]
```

A *receive event* with *no wait* operation performs the same operation as a *receive event*. The main difference is that this version of the call *returns immediately* without waiting on events in the stream. It is a non-blocking call. As such, this version of the call may only function properly with events that are *cached* and raised as *durable*. It is equivalent to the standard TLP operation of with the same name. A full set of event types is supported (ie. `XMLEvent`, `DataEvent`, `BytesEvent`).

If a *no wait* operation does not return an event it returns a `Null` response. Here is a `JSON` format example:

```
{ "SemanticType" : "Null" }
```

For `XML` format the following is returned:

```
<?xml version="1.0"?>
  <Null/>
```

The following verbs are supported by the Receive Event NoWait operation:

| URL Element | Description |
|----------------|---|
| Host | HTTP host name of the node that a client is connecting to. |
| Port | HTTP port of the node (80 by default). |
| eventId | Event Id of the <i>event</i> to be received. |
| responseFormat | Specifies whether the format of a response is XML or JSON. This option controls how the engine returns the <i>representation</i> of a response or an <i>exception</i> . |

Examples:

An example of a request URL being constructed for receiving an event using `JSON representation`:

```
String URL = "/exchange/receiveEvent?responseFormat=json&eventId=event.http.test";
```


Querying Service Configuration

The following section describes all supported interface URLs for browsing service configurations. Results are presented as HTML, XML or JSON format allowing users to view fabric service configurations (SCO) or obtain *representations* for *service access objects* and *event handlers*. Query requests only support the GET HTTP verb.

Get Service List

```
http://< Host>[:< Port > ]/sor/service/list
```

A *service list* is part of the Semantic Object Reference interface that allows users to work with service definitions. The request returns a list of service that are deployed in a given application fabric node as HTML.

The following elements are supported by the Service List operation:

| URL Element | Description |
|-------------|--|
| Host | HTTP host name of the node that a client is connecting to. |
| Port | HTTP port of the node (80 by default). |

Examples:

An example of a Service List URL:

```
http://localhost:5099/sor/service/list
```

Get Service Reference

```
http://< Host>[:< Port > ]/sor/service/< Service Name >
```

A *service reference* is part of the Semantic Object Reference interface that allows users to retrieve a service reference as an HTML document. The response presents service information and access object *reference* links as well as event trigger and event handler definitions, allowing users to browse the service configuration.

The following elements are supported by the Service Reference operation:

| URL Element | Description |
|--------------|---|
| Host | HTTP host name of the node that a client is connecting to. |
| Port | HTTP port of the node (80 by default). |
| Service Name | Name of the service running on this node in <Type>.<Instance> format. |

Examples:

An example of a Service Reference URL:

```
http://localhost:5099/sor/service/DemoService.Default
```

Get Service Actionable Events

```
http://< Host>[:< Port > ]/sor/service/< Service Name >/ActionableEvents/list
```

An *actionable events list* is part of the Semantic Object Reference interface that allows users to retrieve a list of actionable events that may be raised by a service as an HTML document. The response presents a list of *event id* that will include exceptions, notifications and service events, allowing users to browse the service configuration.

The following elements are supported by the Service Actionable Event List operation:

| URL Element | Description |
|--------------|---|
| Host | HTTP host name of the node that a client is connecting to. |
| Port | HTTP port of the node (80 by default). |
| Service Name | Name of the service running on this node in <Type>.<Instance> format. |

Examples:

An example of a Service Actionable Event List URL:

```
http://localhost:5099/sor/service/DemoService.Default/ActionableEvents/list
```

Get Service Event Handlers

```
http://< Host>[:< Port > ]/sor/service/< Service Name >/EventHandlers/list
```

An *event handler list* is part of the Semantic Object Reference interface that allows users to retrieve a list of event handlers that expose callable service methods as an HTML document. The response presents a list of *event handler* names, allowing users to browse the service configuration. The document includes links to service event handler definitions and access object representations as well as other relevant event handler information.

The following elements are supported by the Service Event Handler Names List operation:

| URL Element | Description |
|--------------|---|
| Host | HTTP host name of the node that a client is connecting to. |
| Port | HTTP port of the node (80 by default). |
| Service Name | Name of the service running on this node in <Type>.<Instance> format. |

Examples:

An example of a Service Event Handler Names List URL:

```
http://localhost:5099/sor/service/DemoService.Default/EventHandlers/list
```

Get Service Event Handler

```
http://< Host>[:< Port > ]/sor/service/< Service Name >/EventHandler
{ ?handlerName=< Handler Name > | ?methodName=< Method Name > }
```

An *event handler query* is part of the Semantic Object Reference interface that allows users to retrieve the definition of a particular event handler as an HTML document. The response document includes links to service access object representations as well as other relevant event handler information. A handler may be located by its name or by the method name that it maps to in the service configuration.

The following elements are supported by the Service Event Handler query operation:

| URL Element | Description |
|--------------|--|
| Host | HTTP host name of the node that a client is connecting to. |
| Port | HTTP port of the node (80 by default). |
| Service Name | Name of the service running on this node in <code><Type>.<Instance></code> format. |
| handlerName | The name of the event handler to retrieve. This is mutually exclusive to <code>methodName</code> . |
| methodName | The name of the handler method to retrieve. This is mutually exclusive to <code>handlerName</code> . |

Examples:

An example of a Service Event Handler query URL:

```
http://localhost:5099/sor/service/DemoService.Default/EventHandler
?handlerName=getText
```

This query may return HTML content such as:

```
requestEventId = event.DemoService.request
  SemanticType = String
responseEventId = event.DemoService.reply
  SemanticType = String
```

Get Service Configuration Object

```
http://< Host>[:< Port > ]/sor/service/< Service Name >/Configuration
[ ?responseFormat={ XML | JSON } ]
```

A *service configuration query* is part of the Semantic Object Reference interface that allows users to retrieve the definition of a particular service as an `XML` or `JSON` *representation*. This is in contrast to returning a service *reference* which is provided as an HTML document.

The Service Configuration Object contains all information that is pertinent to running a SOA service. This includes information about the service's parameters, actionable events, potential exceptions, metrics and alerts that may be raised by a service as well as service methods that have been exposed as event handlers. A service configuration query returns the entire object and may be useful for debugging purposes or as a mechanism for versioning and comparing configurations.

Developers may use the Repository Browser to *access* and *modify* node configuration artifacts including Java Archives, Connection Factories and system configuration objects. This is a separate interface described below.

The following elements are supported by the Service Configuration query operation:

| URL Element | Description |
|----------------|--|
| Host | HTTP host name of the node that a client is connecting to. |
| Port | HTTP port of the node (80 by default). |
| Service Name | Name of the service running on this node in <code><Type>.<Instance></code> format. |
| responseFormat | Specifies whether the format of a response is XML or JSON. |

Examples:

An example of a Service Configuration query URL:

```
http://localhost:5099/sor/service/DemoService.Default/Configuration
?responseFormat=XML
```

Get Service Request Object

```
http://< Host>[:< Port > ]/sor/service/< Service Name >/RequestObject
{ ?eventHandler=< Handler Name > | ?eventId=< Event Id > }
[ &responseFormat={ XML | JSON } ]
```

The *service request object* query allows users to retrieve *representations of service request objects* as either XML or JSON format documents. Users can specify a *handler name* or *event id* and query its request object; or navigate the Service Reference HTML to access the request object. Note that by default an SOR *reference* presents results in XML format. To generate *data objects* in JSON format this API call must be used.

The following elements are supported by the Service request Object query operation:

| URL Element | Description |
|----------------|--|
| Host | HTTP host name of the node that a client is connecting to. |
| Port | HTTP port of the node (80 by default). |
| Service Name | Name of the service running on this node in <code><Type>.<Instance></code> format. |
| eventHandler | Specifies name of the <i>event handler</i> for which the <i>request object</i> is queried. This parameter is mutually exclusive to <code>&eventId</code> . |
| eventId | Specifies the name of the <i>event id</i> for which the <i>request object</i> is queried. This parameter is mutually exclusive to <code>&eventHandler</code> . |
| responseFormat | Specifies whether the format of a response is XML or JSON. |

Examples:

An example of a Service Request Object query URL:

```
http://localhost:5099/sor/service/DemoService.Default/RequestObject
?eventId=event.DemoService.request
&responseFormat=XML
```

This may return a result such as the one below, representing the object as a *semantic type*:

```
</string>
```



Note

There is a subtle difference between querying Request Object and Request Event. It is expected that users developing interfaces and services will want to know what the Object looks like and to obtain its representation as XML or JSON so that they can construct proper URLs that invoke the service methods. As such service developers are likely interested in the DATA element being passed to the service. However all services are invoked internally thru raising of events. So every service call has an implicit Event that is used to call it and an Event that is returned as result. To get access to the complete Event users should use the requestEvent query described below.

Get Service Response Object

```
http://< Host>[:< Port > ]/sor/service/< Service Name >/ResponseObject
{ ?eventHandler=< Handler Name > | ?eventId=< Event Id > }
[ &responseFormat={ XML | JSON } ]
```

The *service response object* query allows users to retrieve *representations* of *service response objects* as either XML or JSON format documents. Users can specify a *handler name* or *event id* and query its response object; or navigate the Service Reference HTML to access the request object. Note that by default an SOR *reference* presents results in XML format. To generate *data objects* in JSON format this API call must be used.

The following elements are supported by the Service Response Object query operation:

| URL Element | Description |
|----------------|--|
| Host | HTTP host name of the node that a client is connecting to. |
| Port | HTTP port of the node (80 by default). |
| Service Name | Name of the service running on this node in <Type>.<Instance> format. |
| eventHandler | Specifies name of the <i>event handler</i> for which the <i>request object</i> is queried. This parameter is mutually exclusive to &eventId. |
| eventId | Specifies the name of the <i>event id</i> for which the <i>request object</i> is queried. This parameter is mutually exclusive to &eventHandler. |
| responseFormat | Specifies whether the format of a response is XML or JSON. |

Examples:

An example of a Service Response Object query URL:

```
http://localhost:5099/sor/service/DemoService.Default/ResponseObject
?eventHandler=RequestHandler
&responseFormat=XML
```

The result of this call will be similar to the example above and will return the data element *representation*.

Get Service Request Event

```
http://< Host>[:< Port > ]/sor/service/< Service Name >/RequestEvent
{ ?eventHandler=< Handler Name > | ?eventId=< Event Id > }
[ &responseFormat={ XML | JSON } ]
```

The *service request event* query allows users to retrieve *representations* of *service request event* as either XML or JSON format documents. Users can specify a *handler name* or *event id* and query its request event; or navigate the Service Reference HTML to access the request event. Note that by default an SOR *reference* presents results in XML format. To generate event objects in JSON format this API call must be used.

The following elements are supported by the Service request Object query operation:

| URL Element | Description |
|----------------|--|
| Host | HTTP host name of the node that a client is connecting to. |
| Port | HTTP port of the node (80 by default). |
| Service Name | Name of the service running on this node in <Type>.<Instance> format. |
| eventHandler | Specifies name of the <i>event handler</i> for which the <i>request object</i> is queried. This parameter is mutually exclusive to &eventId. |
| eventId | Specifies the name of the <i>event id</i> for which the <i>request object</i> is queried. This parameter is mutually exclusive to &eventHandler. |
| responseFormat | Specifies whether the format of a response is XML or JSON. |

Examples:

An example of a Service Request Event query URL:

```
http://localhost:5099/sor/service/DemoService.Default/RequestEvent
?eventId=event.DemoService.request
&responseFormat=XML
```

This may return a result such as the one below, representing a complete Data Event:

```
<DataEvent>
  <serialVersionUID>9969000000302071</serialVersionUID>
  <eventId>event.DemoService.request</eventId>
  <durable>>false</durable>
  <timestamp>0</timestamp>
  <eventExpiration>0</eventExpiration>
  <dataProtected>Rw==</dataProtected>
  <acl>
    <ACL>GyUs</ACL>
  </acl>
  <coalesced>>false</coalesced>
  <data SemanticType="string"/>
</DataEvent>
```

Get Service Response Event

```
http://< Host>[:< Port > ]/sor/service/< Service Name >/ResponseEvent
{ ?eventHandler=< Handler Name > | ?eventId=< Event Id > }
[ &responseFormat={ XML | JSON } ]
```

The *service response event* query allows users to retrieve *representations* of *service response events* as either XML or JSON format documents. Users can specify a *handler name* or *event id* and query its response event; or navigate the Service Reference HTML to access the request event. Note that by default an SOR *reference* presents results in XML format. To generate *event objects* in JSON format this API call must be used.

The following elements are supported by the Service Response Event query operation:

| URL Element | Description |
|----------------|--|
| Host | HTTP host name of the node that a client is connecting to. |
| Port | HTTP port of the node (80 by default). |
| Service Name | Name of the service running on this node in <Type>.<Instance> format. |
| eventHandler | Specifies name of the <i>event handler</i> for which the <i>request object</i> is queried. This parameter is mutually exclusive to &eventId. |
| eventId | Specifies the name of the <i>event id</i> for which the <i>request object</i> is queried. This parameter is mutually exclusive to &eventHandler. |
| responseFormat | Specifies whether the format of a response is XML or JSON. |

Examples:

An example of a Service Response Event query URL:

```
http://localhost:5099/sor/service/DemoService.Default/ResponseEvent
?eventHandler=RequestHandler
&responseFormat=XML
```

The result of this call will be similar to the example above and will return the data element *representation*.

Browsing the Entity Repository

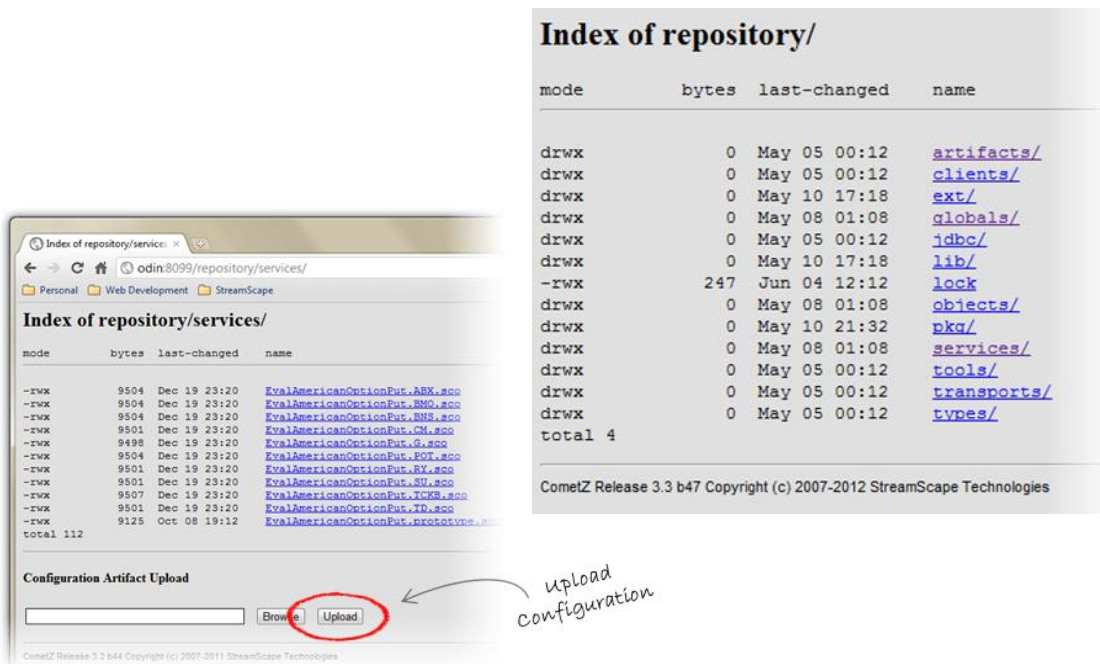
The entity repository may also be accessed thru a standard web browser interface. All major browsers are supported and allow users to work with service configuration artifacts and view configuration elements as XML documents.

Repository access is only available to users with administrative rights. A browser based interface provides facilities for uploading configuration files in order to update an existing configuration or create a new one. New artifacts are validated and processed by the house keeper thread. Service interfaces may also be inspected using the browser, allowing users to query and download service interface artifacts. See the platform's [Java Documentation](#) for examples.

Accessing the repository is done via the following URL:

```
http://localhost:5099/repository
```

By default the service engine implements `REALM` authentication to ensure that only users in the Admin GROUP are able to access Repository content. This security can be configured to extend to other groups but may not be disabled. Once valid credentials are supplied the user may access repository content and even upload configuration artifacts for Services, Java Archives and Connection Factories.



Index of repository/

| mode | bytes | last-changed | name |
|---------|-------|--------------|-----------------------------|
| drwx | 0 | May 05 00:12 | artifacts/ |
| drwx | 0 | May 05 00:12 | clients/ |
| drwx | 0 | May 10 17:18 | ext/ |
| drwx | 0 | May 08 01:08 | globals/ |
| drwx | 0 | May 05 00:12 | jdbc/ |
| drwx | 0 | May 10 17:18 | lib/ |
| -rwx | 247 | Jun 04 12:12 | lock |
| drwx | 0 | May 08 01:08 | objects/ |
| drwx | 0 | May 10 21:32 | pkg/ |
| drwx | 0 | May 08 01:08 | services/ |
| drwx | 0 | May 05 00:12 | tools/ |
| drwx | 0 | May 05 00:12 | transports/ |
| drwx | 0 | May 05 00:12 | types/ |
| total 4 | | | |

CometZ Release 3.3 b47 Copyright (c) 2007-2012 StreamScape Technologies

Configuration Artifact Upload

upload configuration