Agile Computing Framework
(Web Services & Messaging)

Workgroup members:
Mojgan Amini (UCSD), Lakshmi Dasari (UCLA), Ashish Pandit (UCSD), Mike Kennedy (UCR)
Executive Summary
Use cases
SOAP vs REST
Service Design
  Guiding Design Principles
SOAP
  Interface Definition
    WSDL & XSD Contract Design considerations
    Payload
  Security
  Versioning
REST
  Principles/Constraints
  Architecture Elements
  Security
  Versioning
Service Discovery
Service Governance
Error Handling, Alerting, Monitoring
Recommendations
Checklist
Implications
Appendix
  Guiding EA Principles & Standards
  Web service open standards
Glossary
References
1. Executive Summary

In the context of SaaS Applications, Web services play an important role in exposing application data and functionality to an otherwise black-box use of the application. This enables consumers to integrate their applications with SaaS application and/or consume web APIs to build custom web applications. Certain Web service design considerations are to be given higher importance when building for needs across multiple consumers crossing organizational boundaries. A few that have been identified are -

- **Service Design** -
  - Adherence to architectural and service design principles
  - Adherence to industry standards that promote Interoperability across multiple platforms
  - Meets functional and technical requirements.
  - Contract definitions and conformance.
  - Choosing an appropriate web service implementation - SOAP vs REST

- **Security** -
  - Identifying security requirements
  - Transport layer and Application level security that apply to services that carry sensitive information and provided to authenticated and authorized consumers.
  - Adherence to industry standards in securing services.

- **Versioning** - Web services are built to accommodate change
  - Identifying what constitutes a change
  - Adopting a versioning strategy
  - Communicating service versions

- **Discovery** - Web services are well documented, easily discovered and consumed upon implementation. A central software based repository enables easy discovery.

- **Governance & QoS requirements** - Web services have higher governance, uptime, availability and quality requirements.

- **Error handling, Alerting, Monitoring** - Web services be built with robust error handling techniques.

This document attempts to identify guiding principles, standards, best practices and guidelines for building Web Services that promote reuse and help lower the barrier for application reuse and integration. This document does not attempt to prescribe the implementation details or technology platforms that support development of web services.

2. Use cases

A few common use cases that require usage of web services in a SaaS application:
● Integrating with SAML/SSO Authentication Services
● Integrating with Consumer Campuses’ Authorization Services
● Providing APIs for managing/customizing SaaS application configurations
● Providing and Consuming Data Integration Services for one-time data load or on-going basis
● Providing Data access management APIs - create, retrieve, update, delete, query
● Outbound messaging services such as event notifications

3. **SOAP vs REST**

Two methods have emerged for the way systems exchange data using web services, these are SOAP, an XML based exchange of messages via HTTP that is clearly defined in advanced and enforced via WSDLs and other methods and REST (Representational State Transfer) a more lightweight but loosely defined method utilizing HTTP verbs (POST, GET, PUT, DELETE). Both methods have specific advantages and disadvantages and usage scenarios and can be considered while developing shared services.

Depending on which of the two methods are chosen (SOAP or REST) there are several options for security: two-way SSL via WS-Security for SOAP and several security options for REST such as OAuth, Basic Authentication via HTTP, or another “custom” method all over HTTPS. It should be clearly defined how applications using this framework will secure messages being exchanged.

**SOAP**
- System to System Integration Scenarios
  - Clearer contract definition and translation advantage
  - Security well defined
- Provide restricted services (see EAA-008)
- When business process orchestration must to be done using BPEL like standards
- Requires multiple communication protocols for integration

**REST**
- Built for multi-channel consumption for instance - Mobile and web applications
- Where network bandwidth is limited
- Publishing API platform (along with application). Consumers have a choice to use only APIs or the application as a whole or both.
- Fine grain encryption is not required
- Take advantage of caching opportunities that are available at multiple layers (client side, server side)
- Wider adoption possibility with technical audiences in varying roles (For instance - Developers, System admins)
4. Service Design

Guiding Design Principles

Services designed according to service-orientation principles encourage potential reuse. These design principles apply to both SOAP based web services and REST based web services, however REST based services are additionally governed by specific architectural constraints as defined in Section 7 - Principles/Constraints

- Standardized Service contract
- Loose Coupling
- Abstraction
- Reusability
- Autonomy
- Statelessness
- Discoverability
- Composability
- Interoperability

Look up definitions in Glossary

5. SOAP

Interface Definition

Shared services would be required to pay higher emphasis on web service contract definitions and conformance, as they inform the clients on the purpose and usage of the service and is a binding contract for service design and communication.

WSDL & XSD Contract Design considerations

- Designed to meet business & technical requirements
  - Account for business requirements and non-functional requirements that influence the service design
    - How does the service serve the needs for multiple consumer requirements? Would multiple contracts be needed for the same service for different consumers?
    - What are the usage patterns of the service?
    - What are the expected SLAs of the service?
  Based on requirements, the service could be designed as a synchronous service/asynchronous service/data service/business service/integration service

- Contract first design vs Contract last design vs Meet in the Middle?
Contract first design, Code first design or otherwise also known as the Top down/Bottom up approach describe approaches to building web services.

In Contract first design approach, the web service contract is generated first and code supporting the contract is generated later either manually or using code generating tools. In Code first design, Web Service contract is generated from existing code and operations and data structures map to contract web service operations and schema definition.

Both approaches have specific advantages and disadvantages. Contract first approach facilitates design dialogue between the web service provider and consumer and in ensuring contract correctness before much of the implementation work begins thereby reducing rework. Code first design approach could help in specific scenarios where legacy application functionality is exposed as web services. Generation of contracts from code could however result in type definitions that may not be portable in other programming languages and thus reduces interoperability. The code first approach also does not guarantee that the contract remains the same on subsequent code changes and redeployment thus breaking the service contract.

Note #1: **Best Practice:** Contract-first approach - The best practice is therefore to choose a Contract first approach and in some cases a "Meet in the middle" approach in case of legacy applications, where new code interfaces generated from contracts are bridged with older interfaces mapping to legacy code by code in between.

- **Namespace**
  It is important that web services developed as a “Shared” service do not conflict with other web services in their functional and technical definition, design and implementation. Web service catalog or registry must first be looked upon prior to design and implementation.

Note #2: **Standard:** The namespace must contain a unique shared application name and unique shared service name.

- **Conformance to Web services Standards - WS-I Basic Profile (BP)**
  WS-Interoperability organization provides guidance to create interoperable web services using W3C specifications and additionally provides supporting test verification specifications and tools to verify their compliance with WS-I guidelines. Currently there are three Basic Profile (BP) versions in final status - BP 1.1, BP 1.2 and BP 2.0. Conformance to a BP ensures web services are built using the same set of web service specifications (WSDL 1.1, SOAP 1.1, UDDI 2.0 for instance) and narrower set of specifications particularly identified in the profile that ensure interoperability between web services built using multiple programming languages.

Note #3: **Standard:** For this initiative, It is important to choose a Basic Profile as a standard. For instance, Salesforce.com SOAP APIs are implemented to comply with SOAP 1.1, WSDL 1.1 and Basic Profile 1.1
RPC Style or Document Style
Document style standard is widely adopted. EAA-008 calls out RPC style as a standard for restricted/transactional services. Note: Are we missing anything by not considering RPC style. Are there are any advantages to this style and can Document-Literal style be adopted?

Encoding - encoded or literal
Literal - encoding style has advantages since the the message body conforms to the schema defined in the XSD and can be validated against the schema. WS-I Basic profile prohibits the use of encoding.

XML Schema design
Data constraints in XML schema design typically follow the constraints laid out in a logical or physical data model, but it is not required that the data models be exactly identical. XSD design must additionally follow the data principles defined in Enterprise Architecture Framework. Where applicable, adopting common data models for data interchange between systems, increases interoperability. Look for standards on common data models for common higher education entities such as “Student”, “Faculty”, “Person”. Internet 2, PESC for instance are few organizations that work towards defining common data entities in higher education industry.

Note #4: Guideline: Additional useful resources to guide WSDL/XSD design
○ Best practices, guidelines for WSDL/XSD design.
○ Catalog of industry data models.

WSDL, XSD validation
Web service contract must conform to WSDL and XSD specification. Adopting a WS-I defined Basic profile will help with validation of the contracts.

Payload
Structure of Payload

Header - SOAP header is an optional element in a SOAP message. For restricted services, the SOAP header carries authentication information. Refer to WS-security 1.1 standard.
Note #5: Standard/Guideline - But for the purpose of Shared services, it is recommended to include the SOAP header by default with authentication attributes for restricted services, and additional attributes that can help with auditing, tracing and diagnostics. Care must be taken to process and pass along the header by intermediary consumers. A list of such audit attributes could be -

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Required/Optional</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Originating Source</td>
<td>R</td>
<td>Where does the message originate and/or by whom - User, Application, Campus. What could such an entity</td>
</tr>
<tr>
<td>Attribute</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------------------</td>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Message Timestamp</td>
<td>R</td>
<td>Timestamp when the message is sent. Request timestamp incase of a request message and Response timestamp incase of a response message.</td>
</tr>
<tr>
<td>Request Unique Transaction Id</td>
<td>R</td>
<td>a UUID generator can be used to generate such an id. This attribute would be unaltered if the soap message is a response message.</td>
</tr>
<tr>
<td>Response Unique Transaction Id</td>
<td>O</td>
<td>a UUID generator can be used to generate such an id.</td>
</tr>
<tr>
<td>Previous request Unique Transaction Id</td>
<td>O</td>
<td>In case of an orchestrated process, this attribute would be the previous message id.</td>
</tr>
<tr>
<td>User Id</td>
<td>O</td>
<td>Should there be an additional optional attribute representing a real user, such as a - user Id. This can help with scenarios to diagnose a web service message that is a result of user initiated action.</td>
</tr>
<tr>
<td>Additional debugging information</td>
<td>O?</td>
<td>Any other information to chain multiple web service invocations in an orchestrated process.</td>
</tr>
</tbody>
</table>

**Note #6: Reusable component** - If such audit attributes can be maintained in a common data model that is reused in shared services, it could greatly reduce applications’ effort in having to define custom XSDs and header processing logic.
Request & Response

Request and response messages would need to be designed to meet the business requirements of the service. Response message could be designed to indicate a successful response or an error response.

- Message validation - Web service messages must be validated on server and client side for conformance to the Web Service contract.
- Error message be explicitly defined using a type definition in XSD for a clearer contract definition.
- Soap fault messages and error response messages should contain error codes that allow consuming applications to process errors programmatically.
- Account for business and system errors while defining error codes.

Security

The web service under design must first be evaluated on the level of security it requires. University policies and laws could apply for transmitting sensitive information. For instance: a Web service transmitting sensitive Personally Identifiable Information - such as firstname, lastname, DOB, SSN would need to take into account higher security measures such as transport layer security, message level security and transmitting to authenticated and authorized systems only. For authentication & authorization, look up additional details in IDM Agile workgroup document.

WS-I organization has published two Basic Security profiles - 1.0 and 1.1 which builds on top of W3C's WS-Security 1.1 specification and provides specific guidance and requirements for securing web services that promote interoperability and conformance checks.

Note #7: Standard: For this initiative, it is important to identify which profile to conform to. Standards in Enterprise architecture framework for transport layer security - EAA-009 Two-way SSL and EAA-018 “Last Mile” security for web services can be leveraged to guide development. Additional resources for reference
- NIST's guide to securing web services,
- OWASP’s Web Service security Cheat sheet

Versioning

Web Services for shared services must be designed with the expectation to support more than 1 version of a web service to accommodate requirements from multiple consumers. At the same time versioning is a tedious activity and affects multiple layers of an architecture, deployment and hosting of the service. Supporting multiple versions also raise many important questions -

- How to design a service contract to handle requirements for multiple consumers?
- How many versions should a service support?
  - Industry recommends not more than 2 versions active at any time.
• How to version services?
  ○ Versioning services could imply changes to multiple layers of the architecture.
• How many versions can be active concurrently?
• How to communicate service version in the contract?
  ○ Endpoint, namespace, operations
• Which version strategy to choose from - Date based versioning or Semantic versioning?
• What types of changes are major version changes and minor version changes?
• What is a backward compatible change and non-compatible/breaking change?
• How are new service versions informed to the consumers?
• How are services deprecated and informed?
• What is the release cycle for service versions?

Note #8: Guideline - For Shared Services initiative, it is good to have a common Versioning strategy to guide development.

6. REST

Principles/Constraints

REST based services are bound by specific principles/constraints identified in Roy Fielding’s dissertation on REST. When deciding between SOAP based service or REST based service, it is imperative to choose REST services where the use case fits the requirements of a “resource” and “resource state transfer”. Look up more elaboration on principles in Glossary

• Client-Server
• Uniform Interface
  Uniform interface constraint defines the interface between client and server and is bound by the following four interface constraints:
  ○ Identification of resources
  ○ Manipulation of resources through representations
  ○ Self-descriptive messages
  ○ Hypermedia as the engine of application state (HATEOAS)
• Stateless
• Cacheable
• Layered System
• Code on Demand (Optional)

Architecture Elements

Payload

The REST header or body can contain information for traceability and debugging purpose.
<table>
<thead>
<tr>
<th>Header Field Name</th>
<th>Description</th>
<th>Type</th>
<th>Req’d</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>Time when the message is sent</td>
<td>dateTime</td>
<td>Yes</td>
</tr>
<tr>
<td>From</td>
<td>User that originated the message</td>
<td>string</td>
<td>No</td>
</tr>
<tr>
<td>ApplicationID</td>
<td>Application/System that originated the message</td>
<td>string</td>
<td>No</td>
</tr>
<tr>
<td>CampusCode</td>
<td>Campus that originated the message</td>
<td>string</td>
<td>Yes</td>
</tr>
<tr>
<td>MessageID</td>
<td>Unique identifier for a message.</td>
<td>string</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Security**

Use cases:
A. Service is hosted in the same network as shared application. Strong trust relation between client and service
B. Service is hosted by the consumer of the application – strong trust relation between end user and service
C. Service is hosted by a different campus. E.g. utility service like NetID service
D. Similar to use case A but it exposes services as a platform along with application

Options:
Option 1. Session based authentication
Session-based authentication is much more complex when associated with an API. It requires that the API client keep track of state, and depending on the type of client that can be anything from painful to impossible. Session-based authentication, among other things, makes your API less "RESTful" - an API client can’t just make one HTTP request, but now a minimum of three

Option 2 – Basic authentication
This approach is easy.
Concern:
· Security of the key is a concern
· Hard to manage credentials for large number of clients

Option 3 – OAuth based authentication and authorization.
OAuth is primarily authorization protocol. Various factors that determines which profile to use is based on trust relation and the client type. Below is the summary of various grant types and its suitability to UC Shared service use cases.
<table>
<thead>
<tr>
<th>OAuth Grant Type</th>
<th>Client Profiles supported</th>
<th>Description</th>
<th>UC Shared Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authorization code grant</td>
<td>Web application</td>
<td>The authorization code grant type is used to obtain both access tokens and refresh tokens and is optimized for confidential clients.</td>
<td>Use case B &amp; C.</td>
</tr>
<tr>
<td>Implicit Grant</td>
<td>User Agent based Application</td>
<td>This is less secured compare to other grant types and not recommended.</td>
<td>This is very unlikely scenario as most of SaaS application will be using SSO and user may not be comfortable giving out its own username/pw to SaaS application directly.</td>
</tr>
<tr>
<td>Resource Owner Password Credentials</td>
<td>User Agent based Application, Web application, Native application</td>
<td>High degree of trust between the resource owner and the client.</td>
<td>This is very unlikely scenario as most of SaaS application will be using SSO and user may not be comfortable giving out its own username/pw to SaaS application directly.</td>
</tr>
<tr>
<td>Client Credentials</td>
<td>Web application, Native application</td>
<td>The client credentials (or other forms of client authentication) can be used as an authorization grant when the authorization scope is limited to the protected resources under the control of the client.</td>
<td>Likely scenario where service is hosted by SaaS application provider. Use case A.</td>
</tr>
<tr>
<td>OAuth SAML Assertion</td>
<td>Web application</td>
<td>SAML Assertion can be used to request an access</td>
<td>This could work for Use case A and B. However in the</td>
</tr>
</tbody>
</table>
token when a client wishes to utilize an existing trust relationship, expressed through the semantics of (and digital signature or keyed message digest calculated over) the SAML Assertion, without a direct user approval step at the authorization server.

Recommendation

- **Note #9: Standard: OAuth is the preferred mechanism for securing REST services**
  - In case of use case A, client credential is a strong candidate.
  - In case of use case B, Authorization code grant OR SAML bearer token is recommended. Alternately, should information needed by SaaS can be provided by other integration methods?
  - In case of use case C, SAML bearer grant OR Authorization code grant is recommended.
  - Use case D, use of authorization code grant is recommended.

Reference


Infrastructure need:

There are open source and commercial options available to provide workflow around publishing api, subscription of api, issue OAuth token and validate Oauth token. Workflow to grant permission to api is specific to each campus.

Versioning

The only thing that we know for certain about the future is that things will change. The most common way of handling changes is to have versions of an API. If there are non-compatible changes, they should be introduced in a new version of the API.

What to version
In order to support versioning, various component may be impacted

<table>
<thead>
<tr>
<th>No</th>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Source Code</td>
<td>In order to support enhancements and bug fixes for multiple versions, it is recommended to have separate branch for each version being published</td>
</tr>
<tr>
<td>2</td>
<td>Deployed Artifact (version)</td>
<td>In order to have multiple version coexist in the deployment directory, the deployable artifact should also be versioned or should be stored in version specific repository</td>
</tr>
<tr>
<td>3</td>
<td>URL</td>
<td>This is the URL customer may or may not see depending on the situation.</td>
</tr>
<tr>
<td>4</td>
<td>API Gateway</td>
<td>In case when API Gateway is used for indirection, it should be updated to reflect semantic changes in API interface</td>
</tr>
</tbody>
</table>

Versioning Source Code and Deployed Artifact (#1 and #2)

Semantic Versioning has been on the rise over the last few years. SemVer is a 3-component system in the format of x.y.z where:
- x stands for a major version
- y stands for a minor version
- z stands for a patch

So you have: Major.Minor.Patch

Versioning URL (#3 and #4)

- APIs should be versioned for breaking changes.
- Typical scheme includes single digit version i.e. v1, v2 etc
- The api version can be inserted either in header or URL. The common practice is to insert the version in URL e.g. http://api.ucsd.edu/v1/person/. There are specific advantages and disadvantages for both approaches.
- If different parts of the API need to be versioned separately for some reason, it is recommend that API be split into two separate APIs with their own versioning. In that case the two smaller APIs have to be able to live independent of each other

Note #10: Guideline: Deciding on a common versioning strategy for Restful services is useful for Shared Services initiative.

Version Life Cycle Management

- Typically 2 or 3 versions are supported. Recommendation is to at least support 2 versions
• Provide plenty of time to users/UCs when the next version is coming or when apis are being deprecated.

7. **Service Discovery**

Services exposed as shared services must be documented and discoverable. A central registry or web service catalog with the below information must be available for applications to search, register and consume Web Services.

**SOAP & REST Services**

<table>
<thead>
<tr>
<th>No.</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Service name</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Service business description and function</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Service version</td>
<td>1.0, 2.0.</td>
</tr>
<tr>
<td>4</td>
<td>Service classification</td>
<td>Sync, Async</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data Service</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Business Service</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Integration Service</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Authentication Service</td>
</tr>
<tr>
<td>5</td>
<td>Service Endpoint URL</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Service interface definitions</td>
<td>- WSDLs &amp; XSDs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- WADLs?</td>
</tr>
<tr>
<td>7</td>
<td>Sample Request/Response messages</td>
<td>Sample success &amp; error messages</td>
</tr>
</tbody>
</table>

API Management tools provide many features such as service discovery, access management, request throttling, usage information. Examples of such tools - 3 Scale, apiman.io, Fuse, WS02 support API management. It may worth exploring if a API management tool can provide basic infrastructure that is otherwise needed to be custom built.

**Note #11:** Reusable components: Is a central publishing/discovery system essential for Shared Services or do local campuses host their own?

8. **Service Governance**

- Service lifecycle management
9. Error Handling, Alerting, Monitoring
   ● What are the mechanisms?
   ● Error Handling
     o Error queue
     o Error reporting campus specific endpoint
     o Emails?
     o Error portal (within the application or centralized)
   ● Monitoring
     o Should there be an API endpoint for monitoring?
     o Uptime API?

10. Recommendations
    Note #1: Best Practice: Contract-first design
    Note #2: Standard: Namespace Constraints
    Note #3: Standard: WS-I Basic Profile
    Note #4: Guideline: WSDL/XSD Design
    Note #5: Standard/Guideline - Audit Attributes
    Note #6: Reusable component - Common Data model for Audit Attributes
    Note #7: Standard: WS-I Basic Security Profile
    Note #8: Guideline: Versioning Strategy
    Note #9: Standard: REST Security - OAuth
    Note #10: Guideline: REST - Common Versioning Strategy
    Note #11: Reusable components: Central Service Discovery

11. Checklist
    ☐ Service Design
        ☐ Interface
          ● Contract definition
          ● Request validation against interface
          ● Response Data Format should support XML & JSON
        ☐ Interaction type
          ● Asynchronous communication for data integration
• Use idempotent services for reliable service
  ○ Synchronous
    ● Create – non-Idempotent
    ● Others - Idempotent
  ○ Asynchronous
    ● All should be idempotent
• Data Service/Business Service/ Integration Service/Encryption Service/Authentication Service

❑ Security – authentication and authorization
  ● Transport layer, Message layer
  ● Use SSL for all transactions when using REST
  ● Should be able to revoke API keys in case they are compromised in some way
❑ Alerting and monitoring for technical and business exceptions
  ● Service to support health-check option to check if the service is up
❑ Logging
  ● Log each service invocation
  ● Log minimal information
❑ Versioning – Support at least 2 versions
❑ Support high availability and Resilience
❑ Scalability
❑ Throttling and run time statistics of service
❑ API Web page to describe published services
❑ Signing up for the API and getting issued an API Key

12. Implications

Web service implementation has implications on IT infrastructure and architecture of the application. For e.g.: Usage of an ESB or adopting a Service Oriented Architecture style of application development

13. Appendix

Guiding EA Principles & Standards

<table>
<thead>
<tr>
<th>ID</th>
<th>Category</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAA-020</td>
<td>Application Technology</td>
<td>Interoperability</td>
</tr>
<tr>
<td>EAA-022</td>
<td>Application Technology</td>
<td>Technology Independence and Portability</td>
</tr>
<tr>
<td>EAA-023</td>
<td>Data Principles</td>
<td>Data Naming &amp; Definitions</td>
</tr>
<tr>
<td>EAA-025</td>
<td>Data Principles</td>
<td>Data is an Asset</td>
</tr>
</tbody>
</table>
### Data Principles

<table>
<thead>
<tr>
<th>EAA-026</th>
<th>Data Principles</th>
<th>Data is Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAA-029</td>
<td>Data Principles</td>
<td>Data Access</td>
</tr>
<tr>
<td>EAA-030</td>
<td>Data Principles</td>
<td>Data Stewardship</td>
</tr>
<tr>
<td>EAA-031</td>
<td>Security Principles</td>
<td>Data is Secure</td>
</tr>
<tr>
<td>EAA-008*</td>
<td>Near Real Time Messaging</td>
<td>Transactional (Restricted) Web Service Standards</td>
</tr>
<tr>
<td>EAA-009</td>
<td>Near Real Time Messaging</td>
<td>Two-way SSL (Mutual Authentication)</td>
</tr>
<tr>
<td>EAA-011</td>
<td>Near Real Time Messaging</td>
<td>WS-Security 1.1</td>
</tr>
<tr>
<td>EAA-018</td>
<td>Near Real Time Messaging</td>
<td>“Last Mile” security for Web Services</td>
</tr>
</tbody>
</table>

*Note: EAA-008 refers to SOAP 1.1 instead of 1.2 in web service standards, RPC (literal) instead of Document/Literal. Programming frameworks and platform support would be an important criteria, in choosing a basic profile.

### Web service open standards

The major governing bodies that currently provide specifications around web service technologies are OASIS and W3C – World Wide Web Consortium. Open standards that currently apply to Web Services are -

<table>
<thead>
<tr>
<th>Communication Protocol(s)</th>
<th>Transport Protocol</th>
<th>Interface Definition</th>
<th>Data Exchange Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Object Access Protocol(SOAP)</td>
<td>HTTP(S), SMTP, UDP, JMS</td>
<td>WSDL/XSD</td>
<td>XML</td>
</tr>
<tr>
<td>REST</td>
<td>HTTP(S)</td>
<td>URI</td>
<td>JSON, XML</td>
</tr>
</tbody>
</table>

### 14. Glossary

**Standardized Service contract**  
“Services within the same inventory are in compliance with the same contract design standards”

**Loose Coupling**  
"Service contracts impose low consumer coupling requirements and are themselves decoupled from their surrounding environment."
Loose coupling design principles help services in evolving independently and adapting to service changes independently.

**Abstraction**

"Service contracts only contain essential information and information about services is limited to what is published in service contracts."

**Reusability**

"Services contain and express agnostic logic and can be positioned as reusable enterprise resources."

**Autonomy**

"Services exercise a high level of control over their underlying runtime execution environment."

**Statelessness**

"Services minimize resource consumption by deferring the management of state information when necessary."

**Discoverability**

"Services are supplemented with communicative metadata by which they can be effectively discovered and interpreted."

**Composability**

"Services are effective composition participants, regardless of the size and complexity of the composition."

**Interoperability**

Service Interoperability is fundamental to all the above principles and is a must for Service reuse.

**Client-Server**

Client-server architecture style constraint is bound by the key principle of Separation of concerns. Separation of user-interface concerns from data storage concerns improves portability of user interface across multiple platforms and allows components to evolve independently.

**Uniform Interface**

Uniform interface constraint defines the interface between client and server and is bound by the following four interface constraints:

- **Identification of resources**
  
  Key to REST based services is the concept of a “resource”. A resource is any uniquely identifiable (URI) logical entity that has a state. A resource could be a “student”, “faculty” or “thesis” or “academic paper” or a “news article”.

- **Manipulation of resources through representations**
  
  Representation consists of data, metadata describing the data and at times metadata describing the metadata. A given representation may indicate the current state of the requested resource, desired state, client’s query data or error response.

- **Self-descriptive messages**
  
  Each message includes enough information on how to process the message. The URI, header and the message body contain directives on the resource and operations to perform on the resource. Response in turn provides success, failure status codes and the resource state or representation.

- **Hypermedia as the engine of application state (HATEOAS)**
  
  The application state is communicated via hypertext. An API should be usable and understandable given an initial URI, meaning an API should be navigable via its links to various components of the data. This means that every addressable unit of
information carries an address. This however could increase the chattiness over the network if only links to resources were returned without returning the data itself.

**Stateless**

Communication between the client-server must be stateless in nature. No state is stored at the server side for a conversation style interaction and the client must provide all of the information for the server to understand the request. This simplifies server side design but increases the network overhead by increasing the number of client-server interactions and the size of the payload by passing over repetitive information.

**Cacheable**

This constraints requires that data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. This helps reduce the interactions over the network and also taking advantage of the client side caching which potentially further helps in reducing or negating further server interactions.

**Layered System**

Layered System constraint allows an architecture to be composed of hierarchical layers without adding additional complexity of a client having to know beyond its next “immediate” layer. This helps with scalability of application architecture but could increase overhead and latency which could be addressed by shared cache infrastructure.

**Code on Demand (Optional)**

REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This is an optional constraint as it reduces visibility.

**Web Application**

A web application is an application running on a web server. In reality, a web application typically consists of both a browser part and a server part. If a web application needs access to a resource server (e.g. to Facebook user accounts), then the client password could be stored on the server. The password would thus be confidential.

**User Agent Application**

A user agent application is for instance a JavaScript application running in a browser. The browser is the user agent. A user agent application may be stored on a web server, but the application is only running in the user agent once downloaded. An example could be a little JavaScript game that only runs in the browser.

**Native Application**

A native application is for instance a desktop application or a mobile phone application. Native applications are typically installed on the users computer or device (phone, tablet etc.). Thus, the client password will be stored on the users computer or device too.
15. References

Shared Services Development Framework (SSDF)
Roy Fielding’s dissertation on REST
SOA Principles of Service Design by Thomas Erl
SOAP Usecases

SOAP vs REST

REST API Guidelines
OAuth2


Checklist
https://mathieu.fenniak.net/the-api-checklist/