HARVARD UNIVERSITY

Enterprise Architecture
Advisory

Data Exchange Methods and Considerations

Authors: Greg Charest, Mitch Rogers	**Audience Level:** • Strategy Planning and EA Leader • Solution Architect and Program Manager
Version: 1.0 **Last Revised:** 07Feb2020 **Status:** draft **Document Type:** Single Topic Guidance	**Distribution Scope:** Harvard-wide
Workgroup Members:	**Reviewers:** Raoul Sevier JaZahn Clevenger Mike Thomas Bruce Tikofsky

Table of Contents

1. Purpose of this Document

The intent of this document is to provide general guidance and related criteria for developers and technical managers that can be used to select a data exchange protocol, process and format when more than one option is available.

2. Executive Summary

- Traditionally, data exchange between applications was done using file transfers, today network capacity and reliability have improved to the point where request-response and message-based communication between applications is routine.
- Commoditized web and cloud infrastructures allow for more choices in the design of data exchange, each tuned to the needs of the business context of the interactions.
- The general trend towards synchronous, real-time, web service interactions instead of nightly batch transfers requires re-tooling of development organizations and selection preference for vendors that support these approaches.

2.1. Overview

University data - information about our people, classes, research, and more - are collected from diverse sources and generated using an ever-expanding array of techniques and tools. This collection of people, systems and data can be viewed as a type of ecosystem where flows of information support daily operations and can also drive growth and change. To participate in this environment, individual data systems must support mechanisms to exchange data. Data producers must design and build data exchange interfaces and data consumers must choose from available alternatives. Many times, an individual application plays both the producer and consumer role.

Below, are commonly used data exchange methods, most of them currently in use at Harvard. Although API or 'application programming interface' is a general concept that simply describes software that allows two applications to communicate with each other, the methods categorized as Web Services below are commonly referred to as APIs.

3. Data Exchange Patterns

3.1. Application Programming Interface (API)

An API uses web services to communicate using the HTTP protocol. A web service represents a standardized way of providing interoperability between disparate applications. Common types of web services include:

- SOAP is a standardized protocol that sends messages using HTTP and SMTP. The SOAP specifications are official web standards, maintained and developed by the World Wide Web Consortium (W3C).

- REST is not a protocol but an architectural style. The REST architecture defines a set of guidelines to follow to provide a RESTful web service, for example, stateless existence and the use of HTTP status codes.

- GraphQL and a number of similar tools represent an API design architecture that also includes a query and manipulation language and associated runtime.

The advantages and disadvantages of each of these different web service API styles are beyond the scope of this discussion.

3.2. Extract, Transform, and Load (ETL)

Data is transferred by allowing one application to establish a direct connection to another application's database to read and write data. Extraction, translation and loading (ETL) is an extension to the direct database connection approach that adds data batching, data transformation and scheduling tools.

3.3. File Transfer

An application stores data in a file which is transferred to a destination location, then loaded into the destination system. These might be JSON, XML, CSV, or one of many other text-based or binary file formats.

3.4. Remote Procedure Call

A computer program causes a procedure to execute in a different address space (commonly on another computer on a shared network). [1]

3.5. Event Based/Brokered Messaging

An application creates a message containing data and gives it to a service to deliver. This method often requires various technical components to manage queueing and caching, and a business rules engine to manage publication and subscription services.

3.6. Data Streaming

Multiple data sources transfer data continuously to a receiving process. Stream processing ingests a sequence of data, and incrementally updates metrics in response to each arriving data record. It is well suited to real-time monitoring and response functions.

It is important to note that the data exchange patterns identified above are composed of three elements:

1. an architectural pattern
2. a data format, and
3. a communication protocol.

Examples of data formats and communication protocols are included as appendices. Although these three elements are independent, there are popular and commonly used combinations. For example, the popular RESTful API mechanism typically consists of the Representation State Transfer architectural style, the JavaScript Object Notation (JSON) format and the secure HTTPS protocol. Although certain combinations are common, they are not fixed, and various combinations of elements can be used to create a data exchange method.

[1] A web service is a specific implementation of the Remote Procedure Call pattern. For purposes of this discussion, RPC is used to refer to non-web/HTTP implementations.

Although it is beyond the scope of this advisory, it is also important to consider the advantages and disadvantages of each architectural pattern in light of specific application requirements. Synchronous versus non-synchronous calls, blocking, levels of error handling and service coupling are important considerations in selecting a pattern.

Descriptions of various data formats and network protocols, including some associated advantages and disadvantages are described in the appendices.

4. Considerations in Selecting a Data Exchange Approach

The reasons for selecting a data exchange method are rarely definitive and often will require balancing the advantages and disadvantages of a method as well as local and enterprise needs. There is no 'one size fits all' solution to data exchange. The following considerations may apply.

4.1. Data set characteristics

4.1.1. Data complexity

When the data entity to be transferred includes multiple related elements or the specific components are not known in advance, i.e. the required data elements vary in an ad-hoc manner, direct database access may be the most effective option.

One of the key design principles of a REST API is that it is entity-based. While this has the advantage of a predictable location for each entity (e.g., Plan 123 always lives at /plans/123), it has the disadvantage of being more difficult to string together many related entities. An API approach may require multiple calls and coding to re-assemble the relationships among the various data elements. Note that the use of an integration platform or enterprise service bus may mitigate the data complexity issue.

It is important to remember that from a data formatting point of view, flat files are 'flat' and cannot easily represent hierarchical data. JSON and XML can represent more complex data models, although the REST architecture is specifically designed to avoid complex query and result data.

4.1.2. Frequency of data update

The overhead associated with complete dataset replacement via file transfer or direct database access can be substantial. If the data set is updated extremely frequently (and if the number of updates is very large), these issues are magnified. APIs and Messaging system methods more easily support transactional updates to avoid constant bulk resynchronization and are likely better options in this scenario.

4.1.3. Data set size

The transfer of very large data sets often requires the use of a file transfer or direct database connection for performance reasons. Although there are techniques to improve performance when transferring large data sets or large quantities of large messages via REST or similar APIs, other methods are generally preferable.

4.2. Data environment characteristics

4.2.1. Data flows and breadth of solution

How does data flow from one application to another? An analysis of the various planned and potential data flows will help in selecting optimal data exchange methods

The message broker method typically uses middleware that mediates one-to-one, one-to-many, and many-to-one interactions. Extremely large and highly performant push-based fan-in / fan-out systems can be built based on the Message Broker pattern. This becomes useful for asynchronous communication, unreliable networks, and big data applications.

In scenarios where a large number of data sources continuously send data to a single receiving system, for example log or other instrumentation data, a streaming method is likely the best approach.

4.2.2. Frequency of data usage

Is there a benefit to live data? If access to the most up-to-date version of the data is a requirement, then some form of synchronous remote procedure call or API will be required.

4.2.3. Data versions

When a data provider needs to deliver different versions or schemas to support different consuming applications an API may not be the best choice. An API is, ideally, a single consistent representation of a set of resources. Maintaining multiple schemas or versions in a single API is complex and will often accrue technical debt within a codebase.

4.2.4. Data security

The tools and processes necessary to secure data, both at rest and in transmission, are generally external to the specific data exchange method. For example, an API can be designed to require a key, a direct database connection will be constrained by the database management system security controls, web servers can be configured to protect data files etc. Although not directly part of the data exchange method, it is important to evaluate the needed controls and alternatives when selecting one approach over another.

4.2.5. Data transformation complexity

If the data exchange includes the need for significant processing and data transformation, especially if the transformations are based on complex business rules, then a direct database connection using ETL tools is the preferred approach. This is especially true if the purpose of the transfer is to move data from one place of rest to another place of rest. Less extensive transformations can also be completed in some API management platforms, so the API methods may also be practical depending on the specific requirements.

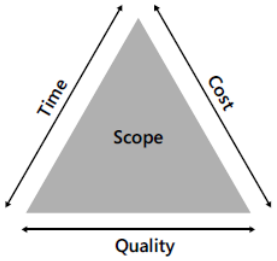
4.2.6. Connection persistence

Long-lived protocols are those for which the connection is intended to remain open indefinitely. An example of such a protocol is ssh. Short-lived protocols are more transactional in nature; a particular action occurs, or series of actions, and then the connection is closed. An example is HTTP POST.

Long-lived connections are particularly useful for streaming data to end-user clients, such as browsers or mobile devices. They may also be useful inside the network when you don’t want to make certain receivers addressable, such as job processors. WebSockets (see appendix 6) is a protocol that can be used to address the need for bi-directional, long-lived connections.

4.3. Scope Constraints

Every project is constrained in some way and selecting a data interchange mechanism is no different. At the highest level the basic 'scope triangle' of time, cost and quality cannot be ignored. Time is the available time to deliver the project, cost represents the amount of money or resources available and quality represents the fit-to-purpose that the project must achieve to be a success. Normally one or more of these factors is fixed and the remaining vary. For example, reducing the time to completion will affect quality and/or costs.



Factors such as available technical skills, business strategies and organizational culture may also represent constraints. In addition, it is unlikely that all, or even many, of the data exchange methods discussed above will be supported in a particular case. This is particularly true in the case of software as a service (SaaS) applications where the customer has no control over the data exchange methods available in the product. However, after taking these larger considerations into account, more than one option may remain. This discussion is focused on those cases.

4.4. Organizational Considerations

Harvard has a large and growing need to clearly understand, easily retrieve and effectively integrate data within and across multiple business units. It is important to view individual project decisions within this enterprise data management framework and to balance project and application specific requirements with broader organizational requirements. Uncoordinated approaches by various segments of the organization can result in data conflicts and quality inconsistencies that reduce efficiency and stifle innovation.

Three of the basic data exchange mechanisms listed above, file transfer, direct database connection and remote procedure calls have traditionally been used to allow dissimilar applications and systems to communicate and exchange data. Unfortunately, because each of these approaches requires detailed knowledge of the operational database or application involved, they are tightly coupled and difficult to change. More importantly, as the number of individual point-to-point exchanges grow, the overall environment becomes increasingly complex and difficult to manage over time. Database links in particular are normally created and maintained by external groups. Wide use of this approach can lead to a substantial access management burden. While there are circumstances in which point-to-point custom integrations are appropriate, they should be carefully considered as they are difficult to evolve based on changing requirements.

Brokered Messaging and Web Services more easily support wider enterprise data integration designs such as the Publish/Subscribe and Gateway patterns. These, and other similar patterns can be used to isolate applications and databases from one another by using a middle service layer to decouple systems. This provide a number of advantages including increased flexibility, better visibility, reduced administration costs, reduced dependencies and the ability to support real time updates.

Web service and messaging methods alone do not necessarily provide increased flexibility. Web services implemented in a point-to-point fashion offer little, if any, improvement over other data exchange methods. It is the combination of these methods with an enterprise integration pattern/platform that reduces integration complexity and provides increased agility.

Absent specific project requirements and within the context of the more detailed criteria discussed below, data exchange designs should favor web service and messaging methods.

4.5. Consumer characteristics

4.5.1. Human beings and front-facing applications

Text files, including the 'comma delimited file' format, are human readable and easily usable by people with commonly available tools. If this form of direct use represents a common use-case, then file transfer is the best choice. Similarly, although APIs are normally used by developers, they generally deliver text or hypermedia. If the receiving system is front-facing, such as a web browsers or similar agent then REST APIs are a reasonable choice. Systems exchanging private data and providing 'back-end' services are more likely to benefit from optimized RPC methods rather than REST APIs.

4.5.2. Receiving system processes

Assumptions built into a receiving system related to the business processes it supports may make one or another exchange method a better choice. For example, the designers of a system oriented to batch processing of transactions may have assumed that that data transfers are always file based. While selecting an alternative data exchange method may be possible, the cost/benefit ratio may not be favorable.

4.5.3. Usage by the receiving system

Is the data being used in support of a feature or as the basis for a platform? If the data is being used to support a 'feature' and supports a specific need, for example a person lookup to retrieve a set of attributes, then an API is likely the most appropriate method. Conversely, if a large dataset is being transferred and used to provide the foundation of a 'platform' or reporting system [2], then a file or database method might be more appropriate.

[2] Localizing data within applications, especially copies of data from systems of record, creates significant data consistency and management problems. The need for large file or database transfer methods may indicate a need for a more maintainable system architecture and design.

5. Summary

The following table summarizes selection criteria and associated data exchange methods.

Typical Use Cases	Method Ranking (High, Medium, Low)			
	API	Messaging	ETL	File
The data is required in multiple formats	H	M	M	L
The data is used in a client front-end	H	M	M	L
The data supports a feature	H	H	M	L
The data is frequently requested	H	H	M	L
The data changes frequently	H	H	M	L
The data is used in a back-end system	L	H	H	M
The data has multiple flows and receivers	M	H	L	L
The data is requested in multiple versions/schemas	L	L	H	L
Assembling the data involves multiple entities and/or variable logic	L	L	H	L
The data set is very large	L	M	H	H
The data forms the basis of a larger platform	L	L	H	H
The data must be human readable	L	L	L	H

6. Appendix

6.1. Data Formats

6.1.1. Text-Based Formats

The primary advantage to text-based codecs is human readability.

6.1.2. XML

XML is A flexible text format for data. The standard for XML document syntax and the many related standards is maintained by the W3C working groups.

Advantages

- Readable and editable by developers
- Error checking by means of Schema and DTDs
- Can represent complex hierarchies of data
- Unicode gives flexibility for international operation
- Plenty of tools in all computer languages for both creation and parsing
- Support Namespace to avoid name conflicts

Disadvantages

- Bulky text with low payload/formatting ratio
- Both creation and client-side parsing are CPU intensive
- Some common word processing characters are illegal
- Images and other binary data require extra encoding

6.1.3. JSON

JSON is a language-independent data format. It was derived from JavaScript, but many programming languages include code to generate and parse JSON-format data.

Advantages

- Readable and editable by developers, easily consumed by web browsers
- Simpler than XML
- Supported by highly developed browser toolkits such as jQuery

Disadvantages

- Bulky text with low payload/formatting ratio, but not as bad as XML
- Client CPU time required to parse
- Not as flexible as XML for some data structures and binary data

6.1.4. Plain Text

Some types of data are easily represented as single elements with a line structure, key value pairs or "comma separated values" or CSV

Advantages

- Readable and editable by developers
- Fairly compact representation for simple types

Disadvantages

- Possible confusion introduced by punctuation in values
- Limited to very simple structures
- Is inherently 'flat' and cannot easily represent hierarchical data

6.1.5. Binary Based Formats

The primary advantage to binary formats is speed. Binary-based encodings are typically 10x to 100x faster than text-based codecs.

6.1.6. CORBA

The Common Object Request Broker Architecture or CORBA was designed to provide for communication of complex data objects between different systems. CORBA is more than just a binary format and includes protocol and architectural standards.

Advantages

- Language and operating system independent
- Compact data representation
- Built in mapping in Java covers almost all features
- Open-source versions are available

Disadvantages

- Complex, difficult learning curve
- Not well supported by OS vendors
- Difficult to use if a server and/or client is behind a firewall or if network address translation is being used

6.1.7. Google Protocol Buffers, Avro, Thrift

Protocol buffers and similar products are a language-neutral, platform-neutral, extensible mechanisms for serializing structured data.

Advantages

- Very compact representation, approaching theoretical maximum
- Tools for many languages
- Not sensitive to version changes
- Include schemas and generated documentation

Disadvantages

- Not readable or editable by developers
- Yet another data definition syntax to learn

6.2. Transfer Protocols

6.2.1. FTP (File Transfer Protocol)

FTP is built for both single file and bulk file transfers. Its weak security model precludes its use for applications with data security requirements. In particular it should not be used with HIPAA, PCI-DSS, SOX, GLBA, and EU Data Protection Directive related data exchanges.

6.2.2. FTPS (FTP over SSL)

FTP and HTTP now have secure versions. FTP has FTPS, while HTTP has HTTPS. Both are protected through SSL. If you use FTPS, you retain the benefits of FTP but gain the security features that come with SSL, including data-in-motion encryption as well as server and client authentication. Because FTPS is based on FTP, you'll still be subjected to the same firewall issues that come with FTP.

6.2.3. HTTP (Hypertext Transfer Protocol)

HTTP is an extensible protocol and is the underlying protocol of the Internet. It is an application layer protocol that is sent over TCP, though any reliable transport protocol could theoretically be used. HTTP is less prone to firewall issues than FTP. However, like FTP, HTTP by itself is inherently insecure and incapable of meeting regulatory compliance or securing data.

6.2.4. HTTPS (HTTP over SSL)

HTTPS is an extension of the Hypertext Transfer Protocol (HTTP). The communication protocol is encrypted using Transport Layer Security (TLS), or, formerly, its predecessor, Secure Sockets Layer (SSL). The protocol is therefore also often referred to as HTTP over TLS,[3] or HTTP over SSL. HTTPS is required for all US Government web sites and is becoming the standard for all web traffic.

6.2.5. WebSocket

WebSocket provides full-duplex communication channels over a single TCP connection. WebSocket is distinct from HTTP. Both protocols are located at layer 7 in the OSI model and depend on TCP at layer 4. Although they are different, RFC 6455 states that WebSocket "is designed to work over HTTP ports 80 and 443 as well as to support HTTP proxies and intermediaries," thus making it compatible with the HTTP protocol.

The WebSocket protocol enables interaction between a web browser (or other client application) and a web server with lower overhead than half-duplex alternatives such as HTTP polling, facilitating real-time data transfer from and to the server. This is made possible by providing a standardized way for the server to send content to the client without being first requested by the client and allowing messages to be passed back and forth while keeping the connection open. In this way, a two-way ongoing conversation can take place between the client and the server. The communications are done over TCP port number 80 (or 443 in the case of TLS-encrypted connections), which is of benefit for those environments which block non-web Internet connections using a firewall.

6.2.6. SFTP (SSH File Transfer Protocol)

While FTPS adds a layer to the FTP protocol, SFTP is an entirely different protocol based on the network protocol SSH (Secure Shell). Unlike both FTP and FTPS, SFTP uses only one connection and encrypts both authentication information and data files being transferred. The main advantage of SFTP over FTPS is that it's more firewall-friendly.

6.2.7. SCP (Secure Copy)

This is an older, more primitive version of SFTP. It also runs on SSH, so it comes with the same security features. However, if you're using a recent version of SSH, you'll already have access to both SCP and SFTP. The only instance you'll probably need SCP is if you'll be exchanging files with an organization that only has a legacy SSH server.

6.2.8. File sharing protocols (CIFS/SMB and NFS)

The Server Message Block (SMB) Protocol is a network file sharing protocol, and as implemented in Microsoft Windows is known as Microsoft SMB Protocol. The set of message packets that defines a particular version of the protocol is called a dialect. The Common Internet File System (CIFS) Protocol is a dialect of SMB.

The NFS protocol was developed by Sun Microsystems and serves essentially the same purpose as SMB (i.e., to access files systems over a network as if they were local) but is incompatible with CIFS/SMB. NFS clients can't speak directly to SMB servers.

6.2.9. AMQP

The Advanced Message Queuing Protocol (AMQP) is an open standard for passing messages between applications or organizations. AMQP supports, queuing and routing (including point-to-point and publish-and-subscribe) and offers authentication and encryption by way of SASL or TLS, relying on a transport protocol such as TCP.

6.2.10. LDAP

Lightweight Directory Access Protocol (LDAP) is a standards-based protocol used to access and manage directory information. It reads and edits directories over IP networks and runs directly over TCP/IP using simple string formats for data transfer. The LDAP protocol is independent of any particular LDAP server implementation.

6.2.11. AS2 (Applicability Statement 2)

Although nearly all of the protocols discussed earlier are capable of supporting B2B exchanges, there are a few protocols that are really designed specifically for such tasks. One of them is AS2.

AS2 is built for EDI (Electronic Data Interchange) transactions, the automated information exchanges normally seen in the manufacturing and retail industries. EDI is now also used in healthcare, as a result of the HIPAA legislation (read Securing HIPAA EDI Transactions with AS2).

6.2.12. AFTP (Accelerated File Transfer Protocol)

WAN file transfers, especially those carried out over great distances, are easily affected by poor network conditions like latency and packet loss, which result in considerably degraded throughputs. AFTP is a TCP-UDP hybrid that makes file transfers virtually immune to these network conditions.

6.2.13. APIs that are confused with protocols

Finally, note that certain tools that are sometimes mistakenly conflated with protocols. Good examples are JDBC (Java database connectivity) and ODBC (open database connectivity). JDBC and ODBC are more properly described as APIs (in the generic sense) to access database servers. The RDBMS vendors provide ODBC or JDBC drivers

so that their database can be accessed by the application. JDBC is language dependent and it is Java specific whereas, the ODBC is a language independent. Another example is Amazon S3. S3 is a service that offers object storage through a web service interface.